<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2  What’s new in IPython</td>
<td>7</td>
</tr>
<tr>
<td>3  Installation</td>
<td>275</td>
</tr>
<tr>
<td>4  Using IPython for interactive work</td>
<td>279</td>
</tr>
<tr>
<td>5  Configuration and customization</td>
<td>345</td>
</tr>
<tr>
<td>6  Developer’s guide for third party tools and libraries</td>
<td>379</td>
</tr>
<tr>
<td>7  Developer’s guide to core IPython</td>
<td>395</td>
</tr>
<tr>
<td>8  The IPython API</td>
<td>399</td>
</tr>
<tr>
<td>9  About IPython</td>
<td>579</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>589</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1 Overview

One of Python’s most useful features is its interactive interpreter. It allows for very fast testing of ideas without the overhead of creating test files as is typical in most programming languages. However, the interpreter supplied with the standard Python distribution is somewhat limited for extended interactive use.

The goal of IPython is to create a comprehensive environment for interactive and exploratory computing. To support this goal, IPython has three main components:

- An enhanced interactive Python shell.
- A decoupled two-process communication model, which allows for multiple clients to connect to a computation kernel, most notably the web-based notebook
- An architecture for interactive parallel computing.

All of IPython is open source (released under the revised BSD license).

1.2 Enhanced interactive Python shell

IPython’s interactive shell (ipython), has the following goals, amongst others:

1. Provide an interactive shell superior to Python’s default. IPython has many features for tab-completion, object introspection, system shell access, command history retrieval across sessions, and its own special command system for adding functionality when working interactively. It tries to be a very efficient environment both for Python code development and for exploration of problems using Python objects (in situations like data analysis).

2. Serve as an embeddable, ready to use interpreter for your own programs. An interactive IPython shell can be started with a single call from inside another program, providing access to the current namespace. This can be very useful both for debugging purposes and for situations where a blend of batch-processing and interactive exploration are needed.

3. Offer a flexible framework which can be used as the base environment for working with other systems, with Python as the underlying bridge language. Specifically scientific environments like Mathematica, IDL and Matlab inspired its design, but similar ideas can be useful in many fields.
4. Allow interactive testing of threaded graphical toolkits. IPython has support for interactive, non-blocking control of GTK, Qt, WX, GLUT, and OS X applications via special threading flags. The normal Python shell can only do this for Tkinter applications.

1.2.1 Main features of the interactive shell

- Dynamic object introspection. One can access docstrings, function definition prototypes, source code, source files and other details of any object accessible to the interpreter with a single keystroke (?), and using ?? provides additional detail.

- Searching through modules and namespaces with * wildcards, both when using the ? system and via the %psearch command.

- Completion in the local namespace, by typing TAB at the prompt. This works for keywords, modules, methods, variables and files in the current directory. This is supported via the readline library, and full access to configuring readline’s behavior is provided. Custom completers can be implemented easily for different purposes (system commands, magic arguments etc.)

- Numbered input/output prompts with command history (persistent across sessions and tied to each profile), full searching in this history and caching of all input and output.

- User-extensible ‘magic’ commands. A set of commands prefixed with % is available for controlling IPython itself and provides directory control, namespace information and many aliases to common system shell commands.

- Alias facility for defining your own system aliases.

- Complete system shell access. Lines starting with ! are passed directly to the system shell, and using ! or var = !cmd captures shell output into python variables for further use.

- The ability to expand python variables when calling the system shell. In a shell command, any python variable prefixed with $ is expanded. A double $$ allows passing a literal $ to the shell (for access to shell and environment variables like PATH).

- Filesystem navigation, via a magic %cd command, along with a persistent bookmark system (using %bookmark) for fast access to frequently visited directories.

- A lightweight persistence framework via the %store command, which allows you to save arbitrary Python variables. These get restored when you run the %store -r command.

- Automatic indentation (optional) of code as you type (through the readline library).

- Macro system for quickly re-executing multiple lines of previous input with a single name via the %macro command. Macros can be stored persistently via %store and edited via %edit.

- Session logging (you can then later use these logs as code in your programs). Logs can optionally timestamp all input, and also store session output (marked as comments, so the log remains valid Python source code).

- Session restoring: logs can be replayed to restore a previous session to the state where you left it.

- Verbose and colored exception traceback printouts. Easier to parse visually, and in verbose mode they produce a lot of useful debugging information (basically a terminal version of the cgitb module).
• Auto-parentheses via the %autocall command: callable objects can be executed without parentheses: `sin 3` is automatically converted to `sin(3)`

• Auto-quoting: using ` `, `;`, or `;` as the first character forces auto-quoting of the rest of the line: `,my_function a b` becomes automatically `my_function("a","b")`, while `;my_function a b` becomes `my_function("a b")`.

• Extensible input syntax. You can define filters that pre-process user input to simplify input in special situations. This allows for example pasting multi-line code fragments which start with `>>>` or `...` such as those from other python sessions or the standard Python documentation.

• Flexible `configuration system`. It uses a configuration file which allows permanent setting of all command-line options, module loading, code and file execution. The system allows recursive file inclusion, so you can have a base file with defaults and layers which load other customizations for particular projects.

• Embeddable. You can call IPython as a python shell inside your own python programs. This can be used both for debugging code or for providing interactive abilities to your programs with knowledge about the local namespaces (very useful in debugging and data analysis situations).

• Easy debugger access. You can set IPython to call up an enhanced version of the Python debugger (pdb) every time there is an uncaught exception. This drops you inside the code which triggered the exception with all the data live and it is possible to navigate the stack to rapidly isolate the source of a bug. The `%run` magic command (with the `-d` option) can run any script under pdb’s control, automatically setting initial breakpoints for you. This version of pdb has IPython-specific improvements, including tab-completion and traceback coloring support. For even easier debugger access, try `%debug` after seeing an exception.

• Profiler support. You can run single statements (similar to `profile.run()`) or complete programs under the profiler’s control. While this is possible with standard cProfile or profile modules, IPython wraps this functionality with magic commands (see `%prun` and `%run -p`) convenient for rapid interactive work.

• Simple timing information. You can use the `%timeit` command to get the execution time of a Python statement or expression. This machinery is intelligent enough to do more repetitions for commands that finish very quickly in order to get a better estimate of their running time.

```
In [1]: %timeit 1+1
10000000 loops, best of 3: 25.5 ns per loop

In [2]: %timeit [math.sin(x) for x in range(5000)]
1000 loops, best of 3: 719 µs per loop
```

To get the timing information for more than one expression, use the `%%timeit` cell magic command.

• Doctest support. The special `%doctest_mode` command toggles a mode to use doctest-compatible prompts, so you can use IPython sessions as doctest code. By default, IPython also allows you to paste existing doctests, and strips out the leading `>>>` and `...` prompts in them.

1.2. Enhanced interactive Python shell
1.3 Decoupled two-process model

IPython has abstracted and extended the notion of a traditional Read-Evaluate-Print Loop (REPL) environment by decoupling the evaluation into its own process. We call this process a kernel: it receives execution instructions from clients and communicates the results back to them.

This decoupling allows us to have several clients connected to the same kernel, and even allows clients and kernels to live on different machines. With the exclusion of the traditional single process terminal-based IPython (what you start if you run ipython without any subcommands), all other IPython machinery uses this two-process model. This includes ipython console, ipython qtconsole, and ipython notebook.

As an example, this means that when you start ipython qtconsole, you’re really starting two processes, a kernel and a Qt-based client can send commands to and receive results from that kernel. If there is already a kernel running that you want to connect to, you can pass the --existing flag which will skip initiating a new kernel and connect to the most recent kernel, instead. To connect to a specific kernel once you have several kernels running, use the %connect_info magic to get the unique connection file, which will be something like --existing kernel-19732.json but with different numbers which correspond to the Process ID of the kernel.

You can read more about using ipython qtconsole, and ipython notebook. There is also a message spec which documents the protocol for communication between kernels and clients.

See also:
Frontend/Kernel Model example notebook

1.4 Interactive parallel computing

Increasingly, parallel computer hardware, such as multicore CPUs, clusters and supercomputers, is becoming ubiquitous. Over the last several years, we have developed an architecture within IPython that allows such hardware to be used quickly and easily from Python. Moreover, this architecture is designed to support interactive and collaborative parallel computing.

The main features of this system are:

• Quickly parallelize Python code from an interactive Python/IPython session.

• A flexible and dynamic process model that be deployed on anything from multicore workstations to supercomputers.

• An architecture that supports many different styles of parallelism, from message passing to task farming. And all of these styles can be handled interactively.

• Both blocking and fully asynchronous interfaces.

• High level APIs that enable many things to be parallelized in a few lines of code.

• Write parallel code that will run unchanged on everything from multicore workstations to supercomputers.

• Full integration with Message Passing libraries (MPI).
• Capabilities based security model with full encryption of network connections.
• Share live parallel jobs with other users securely. We call this collaborative parallel computing.
• Dynamically load balanced task farming system.
• Robust error handling. Python exceptions raised in parallel execution are gathered and presented to the top-level code.

For more information, see our overview of using IPython for parallel computing.

1.4.1 Portability and Python requirements

As of the 2.0 release, IPython works with Python 2.7 and 3.3 or above. Version 1.0 additionally worked with Python 2.6 and 3.2. Version 0.12 was the first version to fully support Python 3.

IPython is known to work on the following operating systems:

• Linux
• Most other Unix-like OSs (AIX, Solaris, BSD, etc.)
• Mac OS X
• Windows (CygWin, XP, Vista, etc.)

See here for instructions on how to install IPython.
What’s new in IPython

This section documents the changes that have been made in various versions of IPython. Users should consult these pages to learn about new features, bug fixes and backwards incompatibilities. Developers should summarize the development work they do here in a user friendly format.

2.1 Development version

This document describes in-flight development work.

**Warning:** Please do not edit this file by hand (doing so will likely cause merge conflicts for other Pull Requests). Instead, create a new file in the `docs/source/whatsnew/pr` folder

2.1.1 Backwards incompatible changes

2.2 4.x Series

2.2.1 IPython 4.2.1

IPython 4.2.1 (July, 2016) includes various bugfixes and improvements over 4.2.0

- Only requires backports packages on Python 2.
- Bugfix(feature regression): Configuration options on CLI get precedence on configuration options.

2.2.2 IPython 4.2

IPython 4.2 (April, 2016) includes various bugfixes and improvements over 4.1.

- Fix `ipython -i` on errors, which was broken in 4.1.
- The delay meant to highlight deprecated commands that have moved to jupyter has been removed.
- Improve compatibility with future versions of traitlets and matplotlib.
• Use stdlib `shutil.get_terminal_size()` to measure terminal width when displaying tracebacks (provided by `backports.shutil_get_terminal_size` on Python 2).

You can see the rest on GitHub.

### 2.2.3 IPython 4.1

IPython 4.1.2 (March, 2016) fixes installation issues with some versions of setuptools. Released February, 2016. IPython 4.1 contains mostly bug fixes, though there are a few improvements.

• IPython debugger (IPdb) now supports the number of context lines for the `where` (and `w`) commands. The `context` keyword is also available in various APIs. See PR PR #9097

• YouTube video will now show thumbnail when exported to a media that do not support video. (PR #9086)

• Add warning when running `ipython <subcommand>` when subcommand is deprecated. `jupyter` should now be used.

• Code in `%pinfo` (also known as `??`) are now highlighter (PR #8947)

• `%aimport` now support module completion. (PR #8884)

• `ipdb` output is now colored ! (PR #8842)

• Add ability to transpose columns for completion. (PR #8748)

Many many docs improvements and bug fixes, you can see the list of changes

### 2.2.4 IPython 4.0

Released August, 2015

IPython 4.0 is the first major release after the Big Split. IPython no longer contains the notebook, qtconsole, etc. which have moved to jupyter. IPython subprojects, such as IPython.parallel and widgets have moved to their own repos as well.

The following subpackages are deprecated:

• IPython.kernel (now jupyter_client and ipykernel)

• IPython.consoleapp (now jupyter_client.consoleapp)

• IPython.nbformat (now nbformat)

• IPython.nbconvert (now nbconvert)

• IPython.html (now notebook)

• IPython.parallel (now ipyparallel)

• IPython.utils.traitlets (now traitlets)

• IPython.config (now traitlets.config)

• IPython.qt (now qtconsole)
• IPython.termial.console (now jupyter_console)

and a few other utilities.

Shims for the deprecated subpackages have been added, so existing code should continue to work with a
warning about the new home.

There are few changes to the code beyond the reorganization and some bugfixes.

IPython highlights:

• Public APIs for discovering IPython paths is moved from IPython.utils.path to
IPython.paths. The old function locations continue to work with deprecation warnings.

• Code raising DeprecationWarning entered by the user in an interactive session will now display
the warning by default. See PR #8480 an #8478.

• The --deep-reload flag and the corresponding options to inject dreload or reload into the
interactive namespace have been deprecated, and will be removed in future versions. You should now
explicitly import reload from IPython.lib.deepreload to use it.

2.3 Issues closed in the 4.x development cycle

2.3.1 Issues closed in 4.2

GitHub stats for 2015/02/02 - 2016/04/20 (since 4.1)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 10 issues and merged 22 pull requests. The full list can be seen on GitHub

The following 10 authors contributed 27 commits.

• Benjamin Ragan-Kelley
• Carlos Cordoba
• Gökhan Karabulut
• Jonas Rauber
• Matthias Bussonnier
• Paul Ivanov
• Sebastian Bank
• Thomas A Caswell
• Thomas Kluyver
• Vincent Woo
2.3.2 Issues closed in 4.1

GitHub stats for 2015/08/12 - 2016/02/02 (since 4.0.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 60 issues and merged 148 pull requests. The full list can be seen on GitHub

The following 52 authors contributed 468 commits.

- Aaron Meurer
- Alexandre Avanian
- Anthony Sottile
- Antony Lee
- Arthur Loder
- Ben Kasel
- Ben Rousch
- Benjamin Ragan-Kelley
- bollwyvl
- Carol Willing
- Christopher Roach
- Douglas La Rocca
- Fairly
- Fernando Perez
- Frank Sachsenheim
- Guillaume DOUMENC
- Gábor Luk
- Hoyt Koepke
- Ivan Timokhin
- Jacob Niehus
- JamshedVesuna
- Jan Schulz
- Jan-Philip Gehrcke
- jc
- Jessica B. Hamrick
- jferara
• John Bohannon
• John Kirkham
• Jonathan Frederic
• Kyle Kelley
• Lev Givon
• Lilian Besson
• lingxz
• Matthias Bussonnier
• memeplex
• Michael Droettboom
• naught101
• Peter Waller
• Pierre Gerold
• Rémy Léone
• Scott Sanderson
• Shanzhuo Zhang
• Sylvain Corlay
• Tayfun Sen
• Thomas A Caswell
• Thomas Ballinger
• Thomas Kluyver
• Vincent Legoll
• Wouter Bolsterlee
• xconverge
• Yuri Numerov
• Zachary Pincus

2.3.3 Issues closed in 4.0

GitHub stats for 2015/02/27 - 2015/08/11 (since 3.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 35 issues and merged 125 pull requests. The full list can be seen [on GitHub](https://github.com/ipython/ipython/issues)

The following 69 authors contributed 1186 commits.
2.3. Issues closed in the 4.x development cycle
2.4 3.x Series

2.4.1 IPython 3.1

Released April 3, 2015

The first 3.x bugfix release, with 33 contributors and 344 commits. This primarily includes bugfixes to notebook layout and focus problems.

Highlights:

- Various focus jumping and scrolling fixes in the notebook.
- Various message ordering and widget fixes in the notebook.
- Images in markdown and output are confined to the notebook width. An .unconfined CSS class is added to disable this behavior per-image. The resize handle on output images is removed.
- Improved ordering of tooltip content for Python functions, putting the signature at the top.
- Fix UnicodeErrors when displaying some objects with unicode reprs on Python 2.
- Set the kernel’s working directory to the notebook directory when running nbconvert --execute, so that behavior matches the live notebook.
- Allow setting custom SSL options for the tornado server with NotebookApp.ssl_options, and protect against POODLE with default settings by disabling SSLv3.
- Fix memory leak in the IPython.parallel Controller on Python 3.

See Issues closed in the 3.x development cycle for details.

2.4.2 Release 3.0

Released February 27, 2015

This is a really big release. Over 150 contributors, and almost 6000 commits in a bit under a year. Support for languages other than Python is greatly improved, notebook UI has been significantly redesigned, and a lot of improvement has happened in the experimental interactive widgets. The message protocol and document format have both been updated, while maintaining better compatibility with previous versions than prior updates. The notebook webapp now enables editing of any text file, and even a web-based terminal (on Unix platforms).
3.x will be the last monolithic release of IPython, as the next release cycle will see the growing project split into its Python-specific and language-agnostic components. Language-agnostic projects (notebook, qtconsole, etc.) will move under the umbrella of the new Project Jupyter name, while Python-specific projects (interactive Python shell, Python kernel, IPython.parallel) will remain under IPython, and be split into a few smaller packages. To reflect this, IPython is in a bit of a transition state. The logo on the notebook is now the Jupyter logo. When installing kernels system-wide, they go in a `jupyter` directory. We are going to do our best to ease this transition for users and developers.

Big changes are ahead.

**Using different kernels**

You can now choose a kernel for a notebook within the user interface, rather than starting up a separate notebook server for each kernel you want to use. The syntax highlighting adapts to match the language you’re working in.

Information about the kernel is stored in the notebook file, so when you open a notebook, it will automatically start the correct kernel.

It is also easier to use the Qt console and the terminal console with other kernels, using the `--kernel` flag:

```bash
ipython qtconsole --kernel bash
ipython console --kernel bash
```

# To list available kernels
```bash
ipython kernelspec list
```
Kernel authors should see Kernel specs for how to register their kernels with IPython so that these mechanisms work.

**Typing unicode identifiers**

Complex expressions can be much cleaner when written with a wider choice of characters. Python 3 allows unicode identifiers, and IPython 3 makes it easier to type those, using a feature from Julia. Type a backslash followed by a LaTeX style short name, such as `\alpha`. Press tab, and it will turn into $\alpha$.

**Widget migration guide**

The widget framework has a lot of backwards incompatible changes. For information about migrating widget notebooks and custom widgets to 3.0 refer to the widget migration guide.

**Other new features**

- TextWidget and TextareaWidget objects now include a placeholder attribute, for displaying placeholder text before the user has typed anything.

- The `%load` magic can now find the source for objects in the user namespace. To enable searching the namespace, use the `--n` option.

  ```
  In [1]: %load --n my_module.some_function
  ```

- DirectView objects have a new `use_cloudpickle()` method, which works like `view.use_dill()`, but causes the cloudpickle module from PiCloud’s cloud library to be used rather than dill or the built-in pickle module.

- Added a .ipynb exporter to nbconvert. It can be used by passing `--to notebook` as a command line argument to nbconvert.

- New nbconvert preprocessor called ClearOutputPreprocessor. This clears the output from IPython notebooks.

- New preprocessor for nbconvert that executes all the code cells in a notebook. To run a notebook and save its output in a new notebook:

  ```
  ipython nbconvert InputNotebook --ExecutePreprocessor.enabled=True --to notebook --output Executed
  ```
In Ipython Documentation, Release 4.2.1

- Consecutive stream (stdout/stderr) output is merged into a single output in the notebook document. Previously, all output messages were preserved as separate output fields in the JSON. Now, the same merge is applied to the stored output as the displayed output, improving document load time for notebooks with many small outputs.

- NotebookApp.webapp_settings is deprecated and replaced with the more informatively named NotebookApp.tornado_settings.

- Using %timeit prints warnings if there is at least a 4x difference in timings between the slowest and fastest runs, since this might mean that the multiple runs are not independent of one another.

- It’s now possible to provide mechanisms to integrate IPython with other event loops, in addition to the ones we already support. This lets you run GUI code in IPython with an interactive prompt, and to embed the IPython kernel in GUI applications. See Integrating with GUI event loops for details. As part of this, the direct enable_* and disable_* functions for various GUIs in IPython.lib.inputhook have been deprecated in favour of enable_gui() and disable_gui().

- A ScrollManager was added to the notebook. The ScrollManager controls how the notebook document is scrolled using keyboard. Users can inherit from the ScrollManager or TargetScrollManager to customize how their notebook scrolls. The default ScrollManager is the SlideScrollManager, which tries to scroll to the nearest slide or sub-slide cell.

- The function interact_manual() has been added which behaves similarly to interact(), but adds a button to explicitly run the interacted-with function, rather than doing it automatically for every change of the parameter widgets. This should be useful for long-running functions.

- The %cython magic is now part of the Cython module. Use %load_ext Cython with a version of Cython >= 0.21 to have access to the magic now.

- The Notebook application now offers integrated terminals on Unix platforms, intended for when it is used on a remote server. To enable these, install the terminado Python package.

- The Notebook application can now edit any plain text files, via a full-page CodeMirror instance.

- Setting the default highlighting language for nbconvert with the config option NbConvertBase.default_language is deprecated. Nbconvert now respects metadata stored in the kernel spec.

- IPython can now be configured systemwide, with files in /etc/ipython or /usr/local/etc/ipython on Unix systems, or %PROGRAMDATA%\ipython on Windows.

- Added support for configurable user-supplied Jinja HTML templates for the notebook. Paths to directories containing template files can be specified via NotebookApp.extra_template_paths. User-supplied template directories searched first by the notebook, making it possible to replace existing templates with your own files.

For example, to replace the notebook’s built-in error.html with your own, create a directory like /home/my_templates and put your override template at /home/my_templates/error.html. To start the notebook with your custom error page enabled, you would run:
It’s also possible to override a template while also inheriting from that template, by prepending `templates/` to the `{% extends %}` target of your child template. This is useful when you only want to override a specific block of a template. For example, to add additional CSS to the built-in `error.html`, you might create an override that looks like:

```html
{% extends "templates/error.html" %}

{% block stylesheet %}
{{super()}}
<style type="text/css">
    /* My Awesome CSS */
</style>
{% endblock %}
```

- **Added a widget persistence API.** This allows you to persist your notebooks interactive widgets. Two levels of control are provided: 1. Higher level: `WidgetManager.set_state_callbacks` allows you to register callbacks for loading and saving widget state. The callbacks you register are automatically called when necessary. 2. Lower level: the `WidgetManager` Javascript class now has `get_state` and `set_state` methods that allow you to get and set the state of the widget runtime.

Example code for persisting your widget state to session data:

```javascript
%%javascript
require(['widgets/js/manager'], function(manager) {
    manager.WidgetManager.set_state_callbacks(function() { // Load
        return JSON.parse(sessionStorage.widgets_state || '{}');
    }, function(state) { // Save
        sessionStorage.widgets_state = JSON.stringify(state);
    });
});
```

- **Enhanced support for `%env` magic.** As before, `%env` with no arguments displays all environment variables and values. Additionally, `%env` can be used to get or set individual environment variables. To display an individual value, use the `%env var` syntax. To set a value, use `env var=val` or `env var=val`. Python value expansion using `$` works as usual.

**Backwards incompatible changes**

- The message protocol has been updated from version 4 to version 5. Adapters are included, so IPython frontends can still talk to kernels that implement protocol version 4.

- The notebook format has been updated from version 3 to version 4. Read-only support for v4 notebooks has been backported to IPython 2.4. Notable changes:
  - heading cells are removed in favor of markdown headings
  - notebook outputs and output messages are more consistent with each other
  - use `IPython.nbformat.read()` and `write()` to read and write notebook files instead of the deprecated `IPython.nbformat.current` APIs.
You can downgrade a notebook to v3 via nbconvert:

```bash
ipython nbconvert --to notebook --nbformat 3 <notebook>
```

which will create `notebook.v3.ipynb`, a copy of the notebook in v3 format.

- **IPython.core.oinspect.getsource()** call specification has changed:
  - `oname` keyword argument has been added for property source formatting
  - `is_binary` keyword argument has been dropped, passing `True` had previously short-circuited
    the function to return `None` unconditionally

- Removed the octavemagic extension: it is now available as `oct2py.ipython`.

- Creating PDFs with LaTeX no longer uses a post processor. Use `nbconvert --to pdf` instead
  of `nbconvert --to latex --post pdf`.

- Used `https://github.com/jdfreder/bootstrap2to3` to migrate the Notebook to Bootstrap 3.

Additionally:

- Set `.tab-content .row 0px;` left and right margin (bootstrap default is `-15px;`)
- Removed `height: @btn_mini_height;` from `.list_header>div, .list_item>div` in `tree.less`
- Set `#header div margin-bottom: 0px;`
- Set `#menus to float: left;`
- Set `#maintoolbar .navbar-text to float: none;`
- Added no-padding convenience class.
- Set border of #maintoolbar to 0px

- Accessing the container DOM object when displaying javascript has been deprecated in IPython
  2.0 in favor of accessing element. Starting with IPython 3.0 trying to access container will raise
  an error in browser javascript console.

- **IPython.utils.py3compat.open** was removed: `io.open()` provides all the same functionality.

- The NotebookManager and `/api/notebooks` service has been replaced by a more generic ContentsManager
  and `/api/contents` service, which supports all kinds of files.

- The Dashboard now lists all files, not just notebooks and directories.

- The `--script` hook for saving notebooks to Python scripts is removed, use `ipython nbconvert --to python notebook` instead.

- The rmagic extension is deprecated, as it is now part of rpy2. See `rpy2.ipython.rmagic`.

- `start_kernel()` and `format_kernel_cmd()` no longer accept a executable parameter. Use the kernelspec machinery instead.

- The widget classes have been renamed from `*Widget` to `*`. The old names are still functional, but
  are deprecated. i.e. `IntSliderWidget` has been renamed to `IntSlider`.

---

2.4. 3.x Series
• The ContainerWidget was renamed to Box and no longer defaults as a flexible box in the web browser. A new FlexBox widget was added, which allows you to use the flexible box model.

• The notebook now uses a single websocket at /kernels/<kernel-id>/channels instead of separate /kernels/<kernel-id>/{shell|iopub|stdin} channels. Messages on each channel are identified by a channel key in the message dict, for both send and recv.

Content Security Policy

The Content Security Policy is a web standard for adding a layer of security to detect and mitigate certain classes of attacks, including Cross Site Scripting (XSS) and data injection attacks. This was introduced into the notebook to ensure that the IPython Notebook and its APIs (by default) can only be embedded in an iframe on the same origin.

Override headers['Content-Security-Policy'] within your notebook configuration to extend for alternate domains and security settings:

```python
c.NotebookApp.tornado_settings = {
    'headers': {
        'Content-Security-Policy': "frame-ancestors 'self'"
    }
}
```

Example policies:

```python
Content-Security-Policy: default-src 'self' https://*.jupyter.org
```

Matches embeddings on any subdomain of jupyter.org, so long as they are served over SSL.

There is a report-uri endpoint available for logging CSP violations, located at /api/security/csp-report. To use it, set report-uri as part of the CSP:

```python
c.NotebookApp.tornado_settings = {
    'headers': {
        'Content-Security-Policy': "frame-ancestors 'self'; report-uri /api/security/csp-report"
    }
}
```

It simply provides the CSP report as a warning in IPython’s logs. The default CSP sets this report-uri relative to the base_url (not shown above).

For a more thorough and accurate guide on Content Security Policies, check out MDN’s Using Content Security Policy for more examples.

2.5 Issues closed in the 3.x development cycle

2.5.1 Issues closed in 3.1

GitHub stats for 2015/02/27 - 2015/04/03 (since 3.0)

These lists are automatically generated, and may be incomplete or contain duplicates.
We closed 46 issues and merged 133 pull requests. The full list can be seen on GitHub.

The following 33 authors contributed 344 commits:

- Abe Guerra
- Adal Chiriliuc
- Benjamin Ragan-Kelley
- Brian Drawert
- Fernando Perez
- Gareth Elston
- Gert-Ludwig Ingold
- Giuseppe Venturini
- Jakob Gager
- Jan Schulz
- Jason Grout
- Jessica B. Hamrick
- Jonathan Frederic
- Justin Tyberg
- Lorena Pantano
- mashenjun
- Mathieu
- Matthias Bussonnier
- Morten Enemark Lund
- Naveen Nathan
- Nicholas Bollweg
- onesandzeroes
- Patrick Snape
- Peter Parente
- RickWinter
- Robert Smith
- Ryan Nelson
- Scott Sanderson
- Sylvain Corlay
- Thomas Kluyver
2.5.2 Issues closed in 3.0

GitHub stats for 2014/04/02 - 2015/02/13 (since 2.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

We closed 469 issues and merged 925 pull requests. The full list can be seen on GitHub.

The following 155 authors contributed 5975 commits.

- A.J. Holyoake
- abalkin
- Adam Hodgen
- Adrian Price-Whelan
- Amin Bandali
- Andreas Amann
- Andrew Dawes
- Andrew Jesaitis
- Andrew Payne
- AnneTheAgile
- Aron Ahmadia
- Ben Duffield
- Benjamin ABEL
- Benjamin Ragan-Kelley
- Benjamin Schultz
- Björn Grüning
- Björn Linse
- Blake Griffith
- Boris Egorov
- Brian E. Granger
- bsvh
- Carlos Cordoba
- Cedric GESTES
2.5. Issues closed in the 3.x development cycle
• Kester Tong
• Kyle Kelley
• Kyle Rawlins
• Lev Abalkin
• Manuel Riel
• Martin Bergtholdt
• Martin Spacek
• Mateusz Paprocki
• Mathieu
• Matthias Bussonnier
• Maximilian Albert
• mbyt
• MechCoder
• Mohan Raj Rajamanickam
• mvr
• Narahari
• Nathan Goldbaum
• Nathan Heijermans
• Nathaniel J. Smith
• ncornette
• Nicholas Bollweg
• Nick White
• Nikolay Koldunov
• Nile Geisinger
• Olga Botvinnik
• Osada Paranaliyanage
• Pankaj Pandey
• Pascal Bugnion
• patricktokeeffe
• Paul Ivanov
• Peter Odding
• Peter Parente

2.5. Issues closed in the 3.x development cycle 25
2.6 Migrating Widgets to IPython 3

2.6.1 Upgrading Notebooks

1. The first thing you’ll notice when upgrading an IPython 2.0 widget notebook to IPython 3.0 is the “Notebook converted” dialog. Click “ok”.

2. All of the widgets distributed with IPython have been renamed. The “Widget” suffix was removed from the end of the class name. i.e. ButtonWidget is now Button.

3. ContainerWidget was renamed to Box.

4. PopupWidget was removed from IPython. If you use the PopupWidget, try using a Box widget instead. If your notebook can’t live without the popup functionality, subclass the Box widget (both in Python and JS) and use JQuery UI’s draggable() and resizable() methods to mimic the behavior.

5. add_class and remove_class were removed. More often than not a new attribute exists on the widget that allows you to achieve the same explicitly. i.e. the Button widget now has a button_style attribute which you can set to ‘primary’, ‘success’, ‘info’, ‘warning’, ‘danger’, or ‘’ instead of using add_class to add the bootstrap class. VBox and HBox classes (flexible Box subclasses) were added that allow you to avoid using add_class and remove_class to make flexible box model layouts. As a last resort, if you can’t find a built in attribute for the class you want to use, a new _dom_classes list trait was added, which combines add_class and remove_class into one stateful list.

6. set_css and get_css were removed in favor of explicit style attributes - visible, width, height, padding, margin, color, background_color, border_color, border_width, border_radius, border_style, font_style, font_weight, font_size, and font_family are a few. If you can’t find a trait to see the css attribute you need, you can, in order of preference, (A) subclass to create your own custom widget, (B) use CSS and the _dom_classes trait to set _dom_classes, or (C) use the _css dictionary to set CSS styling like set_css and get_css.

7. For selection widgets, such as Dropdown, the values argument has been renamed to options.
2.6.2 Upgrading Custom Widgets

Javascript

1. If you are distributing your widget and decide to use the deferred loading technique (preferred), you can remove all references to the WidgetManager and the register model/view calls (see the Python section below for more information).

2. In 2.0 require.js was used incorrectly, that has been fixed and now loading works more like Python’s import. Requiring widgets/js/widget doesn’t import the WidgetManager class, instead it imports a dictionary that exposes the classes within that module:

```javascript
{
  'WidgetModel': WidgetModel,
  'WidgetView': WidgetView,
  'DOMWidgetView': DOMWidgetView,
  'ViewList': ViewList,
}
```

If you decide to continue to use the widget registry (by registering your widgets with the manager), you can import a dictionary with a handle to the WidgetManager class by requiring widgets/js/manager. Doing so will import:

```javascript
{'WidgetManager': WidgetManager}
```

3. Don’t rely on the IPython namespace for anything. To inherit from the DOMWidgetView, WidgetView, or WidgetModel, require widgets/js/widget as widget. If you were inheriting from DOMWidgetView, and the code looked like this:

```python
IPython.DOMWidgetView.extend({...})
```

It would become this:

```python
widget.DOMWidgetView.extend({...})
```

4. Custom models are encouraged. When possible, it’s recommended to move your code into a custom model, so actions are performed 1 time, instead of N times where N is the number of displayed views.

Python

Generally, custom widget Python code can remain unchanged. If you distribute your custom widget, you may be using display and Javascript to publish the widget’s Javascript to the front-end. That is no longer the recommended way of distributing widget Javascript. Instead have the user install the Javascript to his/her nbextension directory or their profile’s static directory. Then use the new _view_module and _model_module traitlets in combination with _view_name and _model_name to instruct require.js on how to load the widget’s Javascript. The Javascript is then loaded when the widget is used for the first time.
2.6.3 Details

Asynchronous

In the IPython 2.x series the only way to register custom widget views and models was to use the registry in the widget manager. Unfortunately, using this method made distributing and running custom widgets difficult. The widget maintainer had to either use the rich display framework to push the widget’s Javascript to the notebook or instruct the users to install the Javascript by hand in a custom profile. With the first method, the maintainer would have to be careful about when the Javascript was pushed to the front-end. If the Javascript was pushed on Python widget import, the widgets wouldn’t work after page refresh. This is because refreshing the page does not restart the kernel, and the Python import statement only runs once in a given kernel instance (unless you reload the Python modules, which isn’t straight forward). This meant the maintainer would have to have a separate push_js() method that the user would have to call after importing the widget’s Python code.

Our solution was to add support for loading widget views and models using require.js paths. Thus the comm and widget frameworks now support lazy loading. To do so, everything had to be converted to asynchronous code. HTML5 promises are used to accomplish that (#6818, #6914).

Symmetry

In IPython 3.0, widgets can be instantiated from the front-end (#6664). On top of this, a widget persistence API was added (#7163, #7227). With the widget persistence API, you can persist your widget instances using Javascript. This makes it easy to persist your widgets to your notebook document (with a small amount of custom JS). By default, the widgets are persisted to your web browsers local storage which makes them reappear when you refresh the page.

Smaller Changes

- Latex math is supported in widget descriptions (#5937).
- Widgets can be display more than once within a single container widget (#5963, #6990).
- FloatRangeSlider and IntRangeSlider were added (#6050).
- “Widget” was removed from the ends of all of the widget class names (#6125).
- ContainerWidget was renamed to Box (#6125).
- HBox and VBox widgets were added (#6125).
- add\_class and remove\_class were removed in favor of a \_dom\_classes list (#6235).
- get\_css and set\_css were removed in favor of explicit traits for widget styling (#6235).
- jslink and jsdlink were added (#6454, #7468).
- An Output widget was added, which allows you to print and display within widgets (#6670).
- PopupWidget was removed (#7341).
- A visual cue was added for widgets with ‘dead’ comms (#7227).

2.6. Migrating Widgets to IPython 3
• A SelectMultiple widget was added (a Select widget that allows multiple things to be selected at once) (#6890).

• A class was added to help manage children views (#6990).

• A warning was added that shows on widget import because it’s expected that the API will change again by IPython 4.0. This warning can be supressed (#7107, #7200, #7201, #7204).

2.6.4 Comm and Widget PR Index

Here is a chronological list of PRs affecting the widget and comm frameworks for IPython 3.0. Note that later PRs may revert changes made in earlier PRs:

• Add placeholder attribute to text widgets #5652
• Add latex support in widget labels, #5937
• Allow widgets to display more than once within container widgets. #5963
• use require.js, #5980
• Range widgets #6050
• Interact on_demand option #6051
• Allow text input on slider widgets #6106
• support binary buffers in comm messages #6110
• Embrace the flexible box model in the widgets #6125
• Widget trait serialization #6128
• Make Container widgets take children as the first positional argument #6153
• once-displayed #6168
• Validate slider value, when limits change #6171
• Unregistering comms in Comm Manager #6216
• Add EventfulList and EventfulDict trait types. #6228
• Remove add/remove_class and set/get_css. #6235
• avoid unregistering widget model twice #6250
• Widget property lock should compare json states, not python states #6332
• Strip the IPY_MODEL_ prefix from widget IDs before referencing them. #6377
• “event” is not defined error in Firefox #6437
• Javascript link #6454
• Bulk update of widget attributes #6463
• Creating a widget registry on the Python side. #6493
• Allow widget views to be loaded from require modules #6494
• Fix Issue #6530 #6532
• Make comm manager (mostly) independent of InteractiveShell #6540
• Add semantic classes to top-level containers for single widgets #6609
• Selection Widgets: forcing ‘value’ to be in ‘values’ #6617
• Allow widgets to be constructed from Javascript #6664
• Output widget #6670
• Minor change in widgets.less to fix alignment issue #6681
• Make Selection widgets respect values order. #6747
• Widget persistence API #6789
• Add promises to the widget framework. #6818
• SelectMultiple widget #6890
• Tooltip on toggle button #6923
• Allow empty text box *while typing* for numeric widgets #6943
• Ignore failure of widget MathJax typesetting #6948
• Refactor the do_diff and manual child view lists into a separate ViewList object #6990
• Add warning to widget namespace import. #7107
• lazy load widgets #7120
• Fix padding of widgets. #7139
• Persist widgets across page refresh #7163
• Make the widget experimental error a real python warning #7200
• Make the widget error message shorter and more understandable. #7201
• Make the widget warning brief and easy to filter #7204
• Add visual cue for widgets with dead comms #7227
• Widget values as positional arguments #7260
• Remove the popup widget #7341
• document and validate link, dlink #7468
• Document interact 5637 #7525
• Update some broken examples of using widgets #7547
• Use Output widget with Interact #7554
• don’t send empty execute_result messages #7560
• Validation on the python side #7602
• only show prompt overlay if there’s a prompt #7661
• Allow predictate to be used for comparison in selection widgets #7674
• Fix widget view persistence. #7680
• Revert “Use Output widget with Interact” #7703

2.7 2.x Series

2.7.1 Release 2.4

January, 2015

• backport read support for nbformat v4 from IPython 3
• support for PyQt5
• support for Pygments 2.0

For more information on what fixes have been backported to 2.4, see our detailed release info.

2.7.2 Release 2.3.1

November, 2014

• Fix CRCRLF line-ending bug in notebooks on Windows

For more information on what fixes have been backported to 2.3.1, see our detailed release info.

2.7.3 Release 2.3.0

October, 2014

• improve qt5 support
• prevent notebook data loss with atomic writes

For more information on what fixes have been backported to 2.3, see our detailed release info.

2.7.4 Release 2.2.0

August, 2014

• Add CORS configuration

For more information on what fixes have been backported to 2.2, see our detailed release info.
2.7.5 Release 2.1.0

May, 2014

IPython 2.1 is the first bugfix release for 2.0. For more information on what fixes have been backported to 2.1, see our detailed release info.

2.7.6 Release 2.0.0

April, 2014

IPython 2.0 requires Python 2.7.2 or 3.3.0. It does not support Python 3.0, 3.1, 3.2, 2.5, or 2.6.

The principal milestones of 2.0 are:

• interactive widgets for the notebook
• directory navigation in the notebook dashboard
• persistent URLs for notebooks
• a new modal user interface in the notebook
• a security model for notebooks

Contribution summary since IPython 1.0 in August, 2013:

• ~8 months of work
• ~650 pull requests merged
• ~400 issues closed (non-pull requests)
• contributions from ~100 authors
• ~4000 commits

The amount of work included in this release is so large that we can only cover here the main highlights; please see our detailed release statistics for links to every issue and pull request closed on GitHub as well as a full list of individual contributors.
New stuff in the IPython notebook

Directory navigation

The IPython notebook dashboard allows navigation into subdirectories. URLs are persistent based on the notebook’s path and name, so no more random UUID URLs.

Serving local files no longer needs the `files` prefix. Relative links across notebooks and other files should work just as if notebooks were regular HTML files.
Interactive widgets

IPython 2.0 adds IPython.html.widgets, for manipulating Python objects in the kernel with GUI controls in the notebook. IPython comes with a few built-in widgets for simple data types, and an API designed for developers to build more complex widgets. See the widget docs for more information.

Modal user interface

The notebook has added separate Edit and Command modes, allowing easier keyboard commands and making keyboard shortcut customization possible. See the new User Interface notebook for more information.

You can familiarize yourself with the updated notebook user interface, including an explanation of Edit and Command modes, by going through the short guided tour which can be started from the Help menu.
Security

2.0 introduces a security model for notebooks, to prevent untrusted code from executing on users’ behalf when notebooks open. A quick summary of the model:

- Trust is determined by signing notebooks.
- Untrusted HTML output is sanitized.
- Untrusted Javascript is never executed.
- HTML and Javascript in Markdown are never trusted.
Dashboard “Running” tab

The dashboard now has a “Running” tab which shows all of the running notebooks.

Single codebase Python 3 support

IPython previously supported Python 3 by running 2to3 during setup. We have now switched to a single codebase which runs natively on Python 2.7 and 3.3.

For notes on how to maintain this, see Writing code for Python 2 and 3.

Selecting matplotlib figure formats

Deprecate single-format InlineBackend.figure_format configurable in favor of InlineBackend.figure_formats, which is a set, supporting multiple simultaneous figure formats (e.g. png, pdf).

This is available at runtime with the new API function IPython.display.set_matplotlib_formats().

clear_output changes

- There is no longer a 500ms delay when calling clear_output.
- The ability to clear stderr and stdout individually was removed.
- A new wait flag that prevents clear_output from being executed until new output is available. This eliminates animation flickering by allowing the user to double buffer the output.
- The output div height is remembered when the wait=True flag is used.
Extending configurable containers

Some configurable traits are containers (list, dict, set) Config objects now support calling extend, update, insert, etc. on traits in config files, which will ultimately result in calling those methods on the original object.

The effect being that you can now add to containers without having to copy/paste the initial value:

```python
c = get_config()
c.InlineBackend.rc.update({ 'figure.figsize' : (6, 4) })
```

Changes to hidden namespace on startup

Previously, all names declared in code run at startup (startup files, ipython -i script.py, etc.) were added to the hidden namespace, which hides the names from tools like %whos. There are two changes to this behavior:

1. Scripts run on the command-line ipython -i script.py now behave the same as if they were passed to `%run`, so their variables are never hidden.
2. A boolean config flag InteractiveShellApp.hide_initial_ns has been added to optionally disable the hidden behavior altogether. The default behavior is unchanged.

Using dill to expand serialization support

The new function use_dill() allows dill to extend serialization support in IPython.parallel (closures, etc.). A DirectView.use_dill() convenience method was also added, to enable dill locally and on all engines with one call.

New IPython console lexer

The IPython console lexer has been rewritten and now supports tracebacks and customized input/output prompts. See the new lexer docs for details.

DisplayFormatter changes

There was no official way to query or remove callbacks in the Formatter API. To remedy this, the following methods are added to BaseFormatter:

- `lookup(instance)` - return appropriate callback or a given object
- `lookup_by_type(type_or_str)` - return appropriate callback for a given type or ‘mod.name’ type string
- `pop(type_or_str)` - remove a type (by type or string). Pass a second argument to avoid KeyError (like dict).
All of the above methods raise a KeyError if no match is found.

And the following methods are changed:

- `for_type(type_or_str)` - behaves the same as before, only adding support for `mod.name` type strings in addition to plain types. This removes the need for `for_type_by_name()`, but it remains for backward compatibility.

Formatters can now raise `NotImplementedError` in addition to returning None to indicate that they cannot format a given object.

**Exceptions and Warnings**

Exceptions are no longer silenced when formatters fail. Instead, these are turned into a `FormatterWarning`. A `FormatterWarning` will also be issued if a formatter returns data of an invalid type (e.g. an integer for `image/png`).

**Other changes**

- `%%capture` cell magic now captures the rich display output, not just stdout/stderr.
- In notebook, Showing tooltip on tab has been disables to avoid conflict with completion. Shift-Tab could still be used to invoke tooltip when inside function signature and/or on selection.
- `object_info_request` has been replaced by `object_info` for consistency in the javascript API. `object_info` is a simpler interface to register callback that is incompatible with `object_info_request`.
- Previous versions of IPython on Linux would use the XDG config directory, creating `~/.config/ipython` by default. We have decided to go back to `~/.ipython` for consistency among systems. IPython will issue a warning if it finds the XDG location, and will move it to the new location if there isn’t already a directory there.
- Equations, images and tables are now centered in Markdown cells.
- Multiline equations are now centered in output areas; single line equations remain left justified.
- IPython config objects can be loaded from and serialized to JSON. JSON config file have the same base name as their .py counterpart, and will be loaded with higher priority if found.
- Bash completion updated with support for all ipython subcommands and flags, including nbconvert.
- `ipython history trim` added --keep=<N> as an alias for the more verbose --HistoryTrim.keep=<N>
- New `ipython history clear` subcommand, which is the same as the newly supported `ipython history trim --keep=0`
- You can now run notebooks in an interactive session via `%run notebook.ipynb`.
- Print preview is back in the notebook menus, along with options to download the open notebook in various formats. This is powered by nbconvert.
- PandocMissing exceptions will be raised if Pandoc is unavailable, and warnings will be printed if the version found is too old. The recommended Pandoc version for use with nbconvert is 1.12.1.

- The InlineBackend.figure_format now supports JPEG output if PIL/Pillow is available.

- Input transformers (see Custom input transformation) may now raise SyntaxError if they determine that input is invalid. The input transformation machinery in IPython will handle displaying the exception to the user and resetting state.

- Calling container.show() on javascript display is deprecated and will trigger errors on future IPython notebook versions. container now show itself as soon as non-empty

- Added InlineBackend.print_figure_kwargs to allow passing keyword arguments to matplotlib's Canvas.print_figure. This can be used to change the value of bbox_inches, which is ‘tight’ by default, or set the quality of JPEG figures.

- A new callback system has been introduced. For details, see IPython Events.

- jQuery and require.js are loaded from CDNs in the default HTML template, so javascript is available in static HTML export (e.g. nbviewer).

**Backwards incompatible changes**

- Python 2.6 and 3.2 are no longer supported: the minimum required Python versions are now 2.7 and 3.3.

- The Transformer classes have been renamed to Preprocessor in nbconvert and their call methods have been renamed to preprocess.

- The call methods of nbconvert post-processors have been renamed to postprocess.

- The module IPython.core.fakemodule has been removed.

- The alias system has been reimplemented to use magic functions. There should be little visible difference while automagics are enabled, as they are by default, but parts of the AliasManager API have been removed.

- We fixed an issue with switching between matplotlib inline and GUI backends, but the fix requires matplotlib 1.1 or newer. So from now on, we consider matplotlib 1.1 to be the minimally supported version for IPython. Older versions for the most part will work, but we make no guarantees about it.

- The pycolor command has been removed. We recommend the much more capable `pygmentize` command from the Pygments project. If you need to keep the exact output of pycolor, you can still use python -m IPython.utils.PyColorize foo.py.

- IPython.lib.irunner and its command-line entry point have been removed. It had fallen out of use long ago.

- The input_prefilter hook has been removed, as it was never actually used by the code. The input transformer system offers much more powerful APIs to work with input code. See Custom input transformation for details.
- `IPython.core.inputsplitter.IPythonInputSplitter` no longer has a method `source_raw_reset()`, but gains `raw_reset()` instead. Use of `source_raw_reset` can be replaced with:

```python
raw = isp.source_raw
transformed = isp.source_reset()
```

- The Azure notebook manager was removed as it was no longer compatible with the notebook storage scheme.
- Simplifying configurable URLs
  - `base_project_url` is renamed to `base_url` (`base_project_url` is kept as a deprecated alias, for now)
  - `base_kernel_url` configurable is removed (use `base_url`)
  - `websocket_url` configurable is removed (use `base_url`)

## 2.8 Issues closed in the 2.x development cycle

### 2.8.1 Issues closed in 2.4.0

GitHub stats for 2014/11/01 - 2015/01/30

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 7 authors contributed 35 commits.

- Benjamin Ragan-Kelley
- Carlos Cordoba
- Damon Allen
- Jessica B. Hamrick
- Mateusz Paprocki
- Peter Würtz
- Thomas Kluyver

We closed 10 issues and merged 6 pull requests; this is the full list (generated with the script `tools/github_stats.py`):

Pull Requests (10):

- **PR #7106**: Changed the display order of rich output in the live notebook.
- **PR #6878**: Update pygments monkeypatch for compatibility with Pygments 2.0
- **PR #6778**: backport nbformat v4 to 2.x
- **PR #6761**: object_info_reply field is oname, not name
- **PR #6653**: Fix IPython.utils.ansispan() to ignore stray [0m
• PR #6706: Correctly display prompt numbers that are ‘None’
• PR #6634: don’t use contains in SelectWidget item_query
• PR #6593: note how to start the qtconsole
• PR #6281: more minor fixes to release scripts
• PR #5458: Add support for PyQt5.

Issues (6):
• #7272: qtconsole problems with pygments
• #7049: Cause TypeError: ‘NoneType’ object is not callable in qtconsole
• #6877: Qt console doesn’t work with pygments 2.0rc1
• #6689: Problem with string containing two or more question marks
• #6702: Cell numbering after ClearOutput preprocessor
• #6633: selectwidget doesn’t display 1 as a selection choice when passed in as a member of values list

2.8.2 Issues closed in 2.3.1

Just one bugfix: fixed bad CRCRLF line-endings in notebooks on Windows

Pull Requests (1):
• PR #6911: don’t use text mode in mkstemp

Issues (1):
• #6599: Notebook.ipynb CR+LF turned into CR+CR+LF

2.8.3 Issues closed in 2.3.0

GitHub stats for 2014/08/06 - 2014/10/01

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 6 authors contributed 31 commits.
• Benjamin Ragan-Kelley
• David Hirschfeld
• Eric Firing
• Jessica B. Hamrick
• Matthias Bussonnier
• Thomas Kluyver
We closed 16 issues and merged 9 pull requests; this is the full list (generated with the script 
`tools/github_stats.py`):

Pull Requests (16):

- PR #6587: support `%matplotlib qt5` and `%matplotlib nbagg`
- PR #6583: Windows symlink test fixes
- PR #6585: fixes #6473
- PR #6581: Properly mock winreg functions for test
- PR #6556: Use some more informative asserts in inprocess kernel tests
- PR #6514: Fix for copying metadata flags
- PR #6453: Copy file metadata in atomic save
- PR #6480: only compare host:port in Websocket.check_origin
- PR #6483: Trim anchor link in heading cells, fixes #6324
- PR #6410: Fix relative import in appnope
- PR #6395: update mathjax CDN url in nbconvert template
- PR #6269: Implement atomic save
- PR #6374: Rename `abort_queues` → `_abort_queues`
- PR #6321: Use appnope in qt and wx gui support from the terminal; closes #6189
- PR #6318: use write_error instead of get_error_html
- PR #6303: Fix error message when failing to load a notebook

Issues (9):

- #6057: `%matplotlib` + qt5
- #6518: Test failure in atomic save on Windows
- #6473: Switching between “Raw Cell Format” and “Edit Metadata” does not work
- #6405: Creating a notebook should respect directory permissions; saving should respect prior permissions
- #6324: Anchors in Heading don’t work.
- #6409: No module named ‘_dummy’
- #6392: Mathjax library link broken
- #6329: IPython Notebook Server URL now requires “tree” at the end of the URL? (version 2.2)
- #6189: ipython console freezes for increasing no of seconds in %pylab mode

2.8. Issues closed in the 2.x development cycle
2.8.4 Issues closed in 2.2.0

GitHub stats for 2014/05/21 - 2014/08/06 (tag: rel-2.1.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 13 authors contributed 36 commits.

- Adam Hodgen
- Benjamin Ragan-Kelley
- Björn Grüning
- Dara Adib
- Eric Galloway
- Jonathan Frederic
- Kyle Kelley
- Matthias Bussonnier
- Paul Ivanov
- Shayne Hodge
- Steven Anton
- Thomas Kluyver
- Zahari

We closed 23 issues and merged 11 pull requests; this is the full list (generated with the script tools/github_stats.py):

Pull Requests (23):

- PR #6279: minor updates to release scripts
- PR #6273: Upgrade default mathjax version.
- PR #6249: always use HTTPS getting mathjax from CDN
- PR #6114: update hmac signature comparison
- PR #6195: Close handle on new temporary files before returning filename
- PR #6143: pin tornado to < 4 on travis js tests
- PR #6134: remove rackcdn https workaround for mathjax cdn
- PR #6120: Only allow iframe embedding on same origin.
- PR #6117: Remove / from route of TreeRedirectHandler.
- PR #6105: only set allow_origin_pat if defined
- PR #6102: Add newline if missing to end of script magic cell
- PR #6077: allow unicode keys in dicts in json_clean
• PR #6061: make CORS configurable
• PR #6081: don’t modify dict keys while iterating through them
• PR #5803: unify visual line handling
• PR #6005: Changed right arrow key movement function to mirror left arrow key
• PR #6029: add pickleutil.PICKLE_PROTOCOL
• PR #6003: Set kernel_id before checking websocket
• PR #5994: Fix ssh tunnel for Python3
• PR #5973: Do not create checkpoint_dir relative to current dir
• PR #5933: fix qt_loader import hook signature
• PR #5944: Markdown rendering bug fix.
• PR #5917: use shutil.move instead of os.rename

Issues (11):

• #6246: Include MathJax by default or access the CDN over a secure connection
• #5525: Websocket origin check fails when used with Apache WS proxy
• #5901: 2 test failures in Python 3.4 in parallel group
• #5926: QT console: text selection cannot be made from left to right with keyboard
• #5998: use_dill does not work in Python 3.4
• #5964: Traceback on Qt console exit
• #5787: Error in Notebook-Generated latex (nbconvert)
• #5950: qtconsole truncates help
• #5943: 2.x: notebook fails to load when using HTML comments
• #5932: Qt ImportDenier Does Not Adhere to PEP302
• #5898: OSError when moving configuration file

2.8.5 Issues closed in 2.1.0

GitHub stats for 2014/04/02 - 2014/05/21 (since 2.0.0)
These lists are automatically generated, and may be incomplete or contain duplicates.
The following 35 authors contributed 145 commits.
• Adrian Price-Whelan
• Aron Ahmadia
• Benjamin Ragan-Kelley
• Benjamin Schultz
• Björn Linse
• Blake Griffith
• chebee7i
• Damián Avila
• Dav Clark
• dexterdev
• Erik Tollerud
• Grzegorz Rożniecki
• Jakob Gager
• jdavidheiser
• Jessica B. Hamrick
• Jim Garrison
• Jonathan Frederic
• Matthias Bussonnier
• Maximilian Albert
• Mohan Raj Rajamanickam
• ncornette
• Nikolay Koldunov
• Nile Geisinger
• Pankaj Pandey
• Paul Ivanov
• Pierre Haessig
• Raffaele De Feo
• Renaud Richardet
• Spencer Nelson
• Steve Chan
• sunny
• Susan Tan
• Thomas Kluyver
• Yaroslav Halchenko
• zah
We closed a total of 129 issues, 92 pull requests and 37 regular issues; this is the full list (generated with the script `tools/github_stats.py --milestone 2.1`):

Pull Requests (92):

- PR #5871: specify encoding in msgpack.unpackb
- PR #5869: Catch more errors from clipboard access on Windows
- PR #5866: Make test robust against differences in line endings
- PR #5605: Two cell toolbar fixes.
- PR #5843: remove Firefox-specific CSS workaround
- PR #5845: Pass Windows interrupt event to kernels as an environment variable
- PR #5835: fix typo in v2 convert
- PR #5841: Fix writing history with output to a file in Python 2
- PR #5842: fix typo in nbconvert help
- PR #5846: Fix typos in Cython example
- PR #5839: Close graphics dev in finally clause
- PR #5837: pass on install docs
- PR #5832: Fixed example to work with python3
- PR #5826: allow notebook tour instantiation to fail
- PR #5560: Minor expansion of Cython example
- PR #5818: interpret any exception in getcallargs as not callable
- PR #5816: Add output to IPython directive when in verbatim mode.
- PR #5822: Don’t overwrite widget description in interact
- PR #5782: Silence exception thrown by completer when dir() does not return a list
- PR #5807: Drop log level to info for Qt console shutdown
- PR #5814: Remove -i options from mv, rm and cp aliases
- PR #5812: Fix application name when printing subcommand help.
- PR #5804: remove an inappropriate !
- PR #5805: fix engine startup files
- PR #5806: Don’t auto-move .config/ipython if symbolic link
- PR #5716: Add booktabs package to latex base.tplx
- PR #5669: allows threadsafe sys.stdout.flush from background threads
- PR #5668: allow async output on the most recent request
- PR #5768: fix cursor keys in long lines wrapped in markdown
• PR #5788: run cells with silent=True in %run nb.ipynb
• PR #5715: log all failed ajax API requests
• PR #5769: Don’t urlescape the text that goes into a title tag
• PR #5762: Fix check for pickling closures
• PR #5766: View.map with empty sequence should return empty list
• PR #5758: Applied bug fix: using fc and ec did not properly set the figure canvas ...
• PR #5754: Format command name into subcommand_description at run time, not import
• PR #5744: Describe using PyPI/pip to distribute & install extensions
• PR #5712: monkeypatch inspect.findsource only when we use it
• PR #5708: create checkpoints dir in notebook subdirectories
• PR #5714: log error message when API requests fail
• PR #5732: Quick typo fix in nbformat/convert.py
• PR #5713: Fix a NameError in IPython.parallel
• PR #5704: Update nbconvertapp.py
• PR #5534: cleanup some pre css inheritance
• PR #5699: don’t use common names in require decorators
• PR #5692: Update notebook.rst fixing broken reference to notebook examples readme
• PR #5693: Update parallel_intro.rst to fix a broken link to examples
• PR #5486: disambiguate to location when no IPs can be determined
• PR #5574: Remove the outdated keyboard shortcuts from notebook docs
• PR #5568: Use __qualname__ in pretty reprs for Python 3
• PR #5678: Fix copy & paste error in docstring of ImageWidget class
• PR #5677: Fix %bookmark -l for Python 3
• PR #5670: nbconvert: Fix CWD imports
• PR #5647: Mention git hooks in install documentation
• PR #5671: Fix blank slides issue in Reveal slideshow pdf export
• PR #5657: use ‘localhost’ as default for the notebook server
• PR #5584: more semantic icons
• PR #5594: update components with marked-0.3.2
• PR #5500: check for Python 3.2
• PR #5582: reset readline after running PYTHONSTARTUP
• PR #5630: Fixed Issue #4012 Added Help menubar link to Github markdown doc
• PR #5613: Fixing bug #5607
• PR #5633: Provide more help if lessc is not found.
• PR #5620: fixed a typo in IPython.core.formatters
• PR #5619: Fix typo in storemagic module docstring
• PR #5592: add missing browser to notebook_aliases list
• PR #5506: Fix ipconfig regex pattern
• PR #5581: Fix rmagic for cells ending in comment.
• PR #5576: only process cr if it’s found
• PR #5478: Add git-hooks install script. Update README.md
• PR #5546: do not shutdown notebook if ‘n’ is part of answer
• PR #5527: Don’t remove upload items from nav tree unless explicitly requested.
• PR #5501: remove inappropriate wheel tag override
• PR #5548: FileNotebookManager: Use shutil.move() instead of os.rename()
• PR #5524: never use for (var i in array)
• PR #5459: Fix interact animation page jump FF
• PR #5559: Minor typo fix in “Cython Magics.ipynb”
• PR #5507: Fix typo in interactive widgets examples index notebook
• PR #5554: Make HasTraits pickleable
• PR #5535: fix n^2 performance issue in coalesce_streams preprocessor
• PR #5522: fix iteration over Client
• PR #5488: Added missing require and jquery from cdn.
• PR #5516: ENH: list generated config files in generated, and rm them upon clean
• PR #5493: made a minor fix to one of the widget examples
• PR #5512: Update tooltips to refer to shift-tab
• PR #5505: Make backport_pr work on Python 3
• PR #5503: check explicitly for ‘dev’ before adding the note to docs
• PR #5498: use milestones to indicate backport
• PR #5492: Polish whatsnew docs
• PR #5495: Fix various broken things in docs
• PR #5496: Exclude whatsnew/pr directory from docs builds
• PR #5489: Fix required Python versions

Issues (37):
• #5364: Horizontal scrollbar hides cell’s last line on Firefox
• #5192: horizontal scrollbar overlaps output or touches next cell
• #5840: Third-party Windows kernels don’t get interrupt signal
• #2412: print history to file using qtconsole and notebook
• #5703: Notebook doesn’t render with “ask me every time” cookie setting in Firefox
• #5817: calling mock object in IPython 2.0.0 under Python 3.4.0 raises AttributeError
• #5499: Error running widgets nbconvert example
• #5654: Broken links from ipython documentation
• #5019: print in QT event callback doesn’t show up in ipython notebook.
• #5800: Only last In prompt number set?
• #5801: startup_command specified in ipengine_config.py is not executed
• #5690: ipython 2.0.0 and pandoc 1.12.2.1 problem
• #5408: Add checking/flushing of background output from kernel in mainloop
• #5407: clearing message handlers on status=idle loses async output
• #5467: Incorrect behavior of up/down keyboard arrows in code cells on wrapped lines
• #3085: nicer notebook error message when lacking permissions
• #5765: map_sync over empty list raises IndexError
• #5553: Notebook matplotlib inline backend: can’t set figure facecolor
• #5710: inspect.findsource monkeypatch raises wrong exception for C extensions
• #5706: Multi-Directory notebooks overwrite each other’s checkpoints
• #5698: can’t require a function named f
• #5569: Keyboard shortcuts in documentation are out of date
• #5566: Function name printing should use __qualname__ instead of __name__ (Python 3)
• #5676: “bookmark -l” not working in ipython 2.0
• #5555: Differentiate more clearly between Notebooks and Folders in new UI
• #5590: Marked double escape
• #5514: import tab-complete fail with ipython 2.0 shell
• #4012: Notebook: link to markdown formatting reference
• #5611: Typo in ‘storemagic’ documentation
• #5589: Kernel start fails when using --browser argument
• #5491: Bug in Windows ipconfig ip address regular expression
• #5579: rmagic extension throws ‘Error while parsing the string.’ when last line is comment
• #5518: Ipython2 will not open ipynb in example directory
• #5561: New widget documentation has missing notebook link
• #5128: Page jumping when output from widget interaction replaced
• #5519: IPython.parallel.Client behavior as iterator
• #5510: Tab-completion for function argument list

2.8.6 Issues closed in 2.0.0

GitHub stats for 2013/08/09 - 2014/04/01 (since 1.0.0)
These lists are automatically generated, and may be incomplete or contain duplicates.

The following 94 authors contributed 3949 commits.

• Aaron Meurer
• Abhinav Upadhyay
• Adam Riggall
• Alex Rudy
• Andrew Mark
• Angus Griffith
• Antony Lee
• Aron Ahmadia
• Arun Persaud
• Benjamin Ragan-Kelley
• Bing Xia
• Blake Griffith
• Bouke van der Bijl
• Bradley M. Froehle
• Brian E. Granger
• Carlos Cordoba
• chapmanb
• chebee7i
• Christoph Gohlke
• Christophe Pradal
• Cyrille Rossant
• Damián Avila
• Matt Henderson
• Matthew Brett
• Matthias Bussonnier
• Michael Droettboom
• Mike McKerns
• Nathan Goldbaum
• Pablo de Oliveira
• Pankaj Pandey
• Pascal Schetelat
• Paul Ivanov
• Paul Moore
• Pere Vilas
• Peter Davis
• Philippe Mallet-Ladeira
• Preston Holmes
• Puneeth Chaganti
• Richard Everson
• Roberto Bonvallet
• Samuel Ainsworth
• Sean Vig
• Shashi Gowda
• Skipper Seabold
• Stephan Rave
• Steve Fox
• Steven Silvester
• stonebig
• Susan Tan
• Sylvain Corlay
• Takeshi Kanmae
• Ted Drain
• Thomas A Caswell
• Thomas Kluyver

2.8. Issues closed in the 2.x development cycle
We closed a total of 1121 issues, 687 pull requests and 434 regular issues; this is the full list (generated with the script `tools/github_stats.py`):

**Pull Requests (687):**

- **PR #5487**: remove weird unicode space in the new copyright header
- **PR #5476**: For 2.0: Fix links in Notebook Help Menu
- **PR #5337**: Examples reorganization
- **PR #5436**: CodeMirror shortcuts in QuickHelp
- **PR #5444**: Fix numeric verification for Int and Float text widgets.
- **PR #5449**: Stretch keyboard shortcut dialog
- **PR #5473**: Minor corrections of git-hooks setup instructions
- **PR #5471**: Add coding magic comment to nbconvert Python template
- **PR #5452**: `print_figure` returns unicode for svg
- **PR #5450**: proposal: remove codename
- **PR #5462**: DOC : fixed minor error in using topological sort
- **PR #5463**: make `spin_thread` tests more forgiving of slow VMs
- **PR #5464**: Fix starting notebook server with file/directory at command line.
- **PR #5453**: remove gitwash
- **PR #5454**: Improve history API docs
- **PR #5431**: update `github_stats` and `gh_api` for 2.0
- **PR #5290**: Add dual mode JS tests
- **PR #5451**: check that a handler is actually registered in `ShortcutManager.handles`
- **PR #5447**: Add `%%%python2` cell magic
- **PR #5439**: Point to the stable SymPy docs, not the dev docs
- **PR #5437**: Install `jquery-ui` images
- PR #5434: fix check for empty cells in rst template
- PR #5432: update links in notebook help menu
- PR #5435: Update whatsnew (notebook tour)
- PR #5433: Document extraction of octave and R magics
- PR #5428: Update COPYING.txt
- PR #5426: Separate get_session_info between HistoryAccessor and HistoryManager
- PR #5419: move prompts from margin to main column on small screens
- PR #5430: Make sure `element` is correct in the context of displayed JS
- PR #5396: prevent saving of partially loaded notebooks
- PR #5429: Fix tooltip pager feature
- PR #5330: Updates to shell reference doc
- PR #5404: Fix broken accordion widget
- PR #5339: Don’t use fork to start the notebook in js tests
- PR #5320: Fix for Tooltip & completer click focus bug.
- PR #5421: Move configuration of Python test controllers into setup()
- PR #5418: fix typo in ssh launcher send_file
- PR #5403: remove alt– shortcut
- PR #5389: better log message in deprecated files/ redirect
- PR #5333: Fix filenbmanager.list_dirs fails for Windows user profile directory
- PR #5390: finish PR #5333
- PR #5326: Some gardening on iptest result reporting
- PR #5375: remove unnecessary onload hack from mathjax macro
- PR #5368: Flexbox classes specificity fixes
- PR #5331: fix raw_input CSS
- PR #5395: urlencode images for rst files
- PR #5049: update quickhelp on adding and removing shortcuts
- PR #5391: Fix Gecko (Netscape) keyboard handling
- PR #5387: Respect 'r' characters in nbconvert.
- PR #5399: Revert PR #5388
- PR #5388: Suppress output even when a comment follows ;. Fixes #4525.
- PR #5394: nbconvert doc update
- PR #5359: do not install less sources

2.8. Issues closed in the 2.x development cycle
• PR #5346: give hint on where to find custom.js
• PR #5357: catch exception in copystat
• PR #5380: Remove DefineShortVerb... line from latex base template
• PR #5376: elide long containers in pretty
• PR #5310: remove raw cell placeholder on focus, closes #5238
• PR #5332: semantic names for indicator icons
• PR #5386: Fix import of socketserver on Python 3
• PR #5360: remove some redundant font-family: monospace
• PR #5379: don’t instantiate Application just for default logger
• PR #5372: Don’t autoclose strings
• PR #5296: unify keyboard shortcut and codemirror interaction
• PR #5349: Make Hub.registration_timeout configurable
• PR #5340: install bootstrap-tour css
• PR #5335: Update docstring for deeppreload module
• PR #5321: Improve assignment regex to match more tuple unpacking syntax
• PR #5325: add NotebookNotary to NotebookApp’s class list
• PR #5313: avoid loading preprocessors twice
• PR #5308: fix HTML capitalization in Highlight2HTML
• PR #5295: OutputArea.append_type functions are not prototype methods
• PR #5318: Fix local import of select_figure_formats
• PR #5300: Fix NameError: name ‘_rl’ is not defined
• PR #5292: focus next cell on shift+enter
• PR #5291: debug occasional error in test_queue_status
• PR #5289: Finishing up #5274 (widget paths fixes)
• PR #5232: Make nbconvert html full output like notebook’s html.
• PR #5288: Correct initial state of kernel status indicator
• PR #5253: display any output from this session in terminal console
• PR #4802: Tour of the notebook UI (was UI elements inline with highlighting)
• PR #5285: Update signature presentation in pinfo classes
• PR #5268: Refactoring Notebook.command_mode
• PR #5226: Don’t run PYTHONSTARTUP file if a file or code is passed
• PR #5283: Remove Widget.closed attribute
• PR #5279: nbconvert: Make sure node is at least version 0.9.12
• PR #5281: fix a typo introduced by a rebased PR
• PR #5280: append Firefox overflow-x fix
• PR #5277: check that PIL can save JPEG to BytesIO
• PR #5044: Store timestamps for modules to autoreload
• PR #5278: Update what’s new doc from pr files
• PR #5276: Fix kernel restart in case connection file is deleted.
• PR #5272: allow highlighting language to be set from notebook metadata
• PR #5158: log refusal to serve hidden directories
• PR #5188: New events system
• PR #5265: Missing class def for TimeoutError
• PR #5267: normalize unicode in notebook API tests
• PR #5076: Refactor keyboard handling
• PR #5241: Add some tests for utils
• PR #5261: Don’t allow edit mode up arrow to continue past index == 0
• PR #5223: use on-load event to trigger resizable images
• PR #5252: make one strftime call at import of jsonutil
• PR #5153: Dashboard sorting
• PR #5169: Allow custom header
• PR #5242: clear _reply_content cache before using it
• PR #5194: require latex titles to be ascii
• PR #5244: try to avoid EADDRINUSE errors on travis
• PR #5245: support extracted output in HTML template
• PR #5209: make input_area css generic to cells
• PR #5246: less %pylab, more cowbell!
• PR #4895: Improvements to %run completions
• PR #5243: Add Javascript to base display priority list.
• PR #5175: Audit .html() calls take #2
• PR #5146: Dual mode bug fixes.
• PR #5207: Children fire event
• PR #5215: Dashboard “Running” Tab
• PR #5240: Remove unused IPython.nbconvert.utils.console module

2.8. Issues closed in the 2.x development cycle
• PR #5239: Fix exclusion of tests directories from coverage reports
• PR #5203: capture some logging/warning output in some tests
• PR #5216: fixup positional arg handling in notebook app
• PR #5229: get _ipython_display_ method safely
• PR #5234: DOC : modified docs is HasTraits.traits and HasTraits.class_traits
• PR #5221: Change widget children List to Tuple.
• PR #5231: don’t forget base_url when updating address bar in rename
• PR #5173: Moved widget files into static/widgets/*
• PR #5222: Unset PYTHONWARNINGS envvar before running subprocess tests.
• PR #5172: Prevent page breaks when printing notebooks via print-view.
• PR #4985: Add automatic Closebrackets function to Codemirror.
• PR #5202: handle nodejs executable on debian
• PR #5112: band-aid for completion
• PR #5187: handle missing output metadata in nbconvert
• PR #5181: use gnureadline on OS X
• PR #5136: set default value from signature defaults in interact
• PR #5132: remove application/pdf->pdf transform in javascript
• PR #5116: reorganize who knows what about paths
• PR #5165: Don’t introspect __call__ for simple callables
• PR #5170: Added msg_throttle sync=True widget traitlet
• PR #5191: Translate markdown link to rst
• PR #5037: FF Fix: alignment and scale of text widget
• PR #5179: remove websocket url
• PR #5110: add InlineBackend.print_figure_kwargs
• PR #5147: Some template URL changes
• PR #5100: remove base_kernel_url
• PR #5163: Simplify implementation of TemporaryWorkingDirectory.
• PR #5166: remove mktemp usage
• PR #5133: don’t use combine option on ucs package
• PR #5089: Remove legacy azure nbmanager
• PR #5159: remove append_json reference
• PR #5095: handle image size metadata in nbconvert html
• PR #5156: fix IPython typo, closes #5155
• PR #5150: fix a link that was broken
• PR #5114: use non-breaking space for button with no description
• PR #4778: add APIs for installing notebook extensions
• PR #5125: Fix the display of functions with keyword-only arguments on Python 3.
• PR #5097: minor notebook logging changes
• PR #5047: only validate package_data when it might be used
• PR #5121: fix remove event in KeyboardManager.register_events
• PR #5119: Removed ‘list’ view from Variable Inspector example
• PR #4925: Notebook manager api fixes
• PR #4996: require print_method to be a bound method
• PR #5108: require specifying the version for gh-pages
• PR #5111: Minor typo in docstring of IPython.parallel DirectView
• PR #5098: mostly debugging changes for IPython.parallel
• PR #5087: trust cells with no output
• PR #5059: Fix incorrect patch logic in widget code
• PR #5075: More flexible box model fixes
• PR #5091: Provide logging messages in ipcluster log when engine or controllers fail to start
• PR #5090: Print a warning when iptest is run from the IPython source directory
• PR #5077: flush replies when entering an eventloop
• PR #5055: Minimal changes to import IPython from IronPython
• PR #5078: Updating JS tests README.md
• PR #5083: don’t create js test directories unless they are being used
• PR #5062: adjust some events in nb_roundtrip
• PR #5043: various unicode / url fixes
• PR #5066: remove (almost) all mentions of pylab from our examples

2.8. Issues closed in the 2.x development cycle
• PR #4977: ensure scp destination directories exist (with mkdir -p)
• PR #5053: Move&rename JS tests
• PR #5067: show traceback in widget handlers
• PR #4920: Adding PDFFormatter and kernel side handling of PDF display data
• PR #5048: Add edit/command mode indicator
• PR #5061: make execute button in menu bar match shift-enter
• PR #5052: Add q to toggle the pager.
• PR #5070: fix flex: auto
• PR #5065: Add example of using annotations in interact
• PR #5063: another pass on Interact example notebooks
• PR #5051: FF Fix: code cell missing hscroll (2)
• PR #4960: Interact/Interactive for widget
• PR #5045: Clear timeout in multi-press keyboard shortcuts.
• PR #5060: Change ‘bind’ to ‘link’
• PR #5039: Expose kernel_info method on inprocess kernel client
• PR #5058: Fix iopubwatcher.py example script.
• PR #5035: FF Fix: code cell missing hscroll
• PR #5040: Polishing some docs
• PR #5001: Add directory navigation to dashboard
• PR #5042: Remove duplicated Channel ABC classes.
• PR #5036: FF Fix: ext link icon same line as link text in help menu
• PR #4975: setup.py changes for 2.0
• PR #4774: emit event on appended element on dom
• PR #5023: Widgets- add ability to pack and unpack arrays on JS side.
• PR #5003: Fix pretty reprs of super() objects
• PR #4974: make paste focus the pasted cell
• PR #5012: Make SelectionWidget.values a dict
• PR #5018: Prevent ‘iptest IPython’ from trying to run.
• PR #5025: citation2latex filter (using HTMLParser)
• PR #5027: pin lessc to 1.4
• PR #4952: Widget test inconsistencies
• PR #5014: Fix command mode & popup view bug
• PR #4842: more subtle kernel indicator
• PR #5017: Add notebook examples link to help menu.
• PR #5015: don’t write cell.trusted to disk
• PR #5007: Update whatsnew doc from PR files
• PR #5010: Fixes for widget alignment in FF
• PR #4901: Add a convenience class to sync traitlet attributes
• PR #5008: updated explanation of ‘pyin’ messages
• PR #5004: Fix widget vslider spacing
• PR #4933: Small Widget inconsistency fixes
• PR #4979: add versioning notes to small message spec changes
• PR #4893: add font-awesome 3.2.1
• PR #4982: Live readout for slider widgets
• PR #4813: make help menu a template
• PR #4939: Embed qtconsole docs (continued)
• PR #4964: remove shift-= merge keyboard shortcut
• PR #4504: Allow input transformers to raise SyntaxError
• PR #4929: Fixing various modal/focus related bugs
• PR #4971: Fixing issues with js tests
• PR #4972: Work around problem in doctest discovery in Python 3.4 with PyQt
• PR #4937: pickle arrays with dtype=object
• PR #4934: ipython profile create respects --ipython-dir
• PR #4954: generate unicode filename
• PR #4845: Add Origin Checking.
• PR #4916: Fine tuning the behavior of the modal UI
• PR #4966: Ignore sys.argv for NotebookNotary in tests
• PR #4967: Fix typo in warning about web socket being closed
• PR #4965: Remove mention of iplogger from setup.py
• PR #4962: Fixed typos in quick-help text
• PR #4953: add utils.wait_for_idle in js tests
• PR #4870: ipython_directive, report except/warn in block and add :okexcept: :okwarning: options to suppress
• PR #4662: Menu cleanup

2.8. Issues closed in the 2.x development cycle
• PR #4824: sign notebooks
• PR #4943: Docs shotgun 4
• PR #4848: avoid import of nearby temporary with %edit
• PR #4950: Two fixes for file upload related bugs
• PR #4927: there shouldn’t be a ‘files/’ prefix in FileLink[s]
• PR #4928: use importlib.machinery when available
• PR #4949: Remove the docscape modules, which are part of numpydoc
• PR #4849: Various unicode fixes (mostly on Windows)
• PR #4932: always point py3compat.input to builtin_mod.input
• PR #4807: Correct handling of ansi colour codes when nbconverting to latex
• PR #4922: Python nbconvert output shouldn’t have output
• PR #4912: Skip some Windows io failures
• PR #4919: flush output before showing tracebacks
• PR #4915: ZMQCompleter inherits from IPCompleter
• PR #4890: better cleanup channel FDs
• PR #4880: set profile name from profile_dir
• PR #4853: fix setting image height/width from metadata
• PR #4786: Reduce spacing of heading cells
• PR #4680: Minimal pandoc version warning
• PR #4908: detect builtin docstrings in oinspect
• PR #4911: Don’t use python -m package on Windows Python 2
• PR #4909: sort dictionary keys before comparison, ordering is not guaranteed
• PR #4374: IPEP 23: Backbone.js Widgets
• PR #4903: use https for all embeds
• PR #4894: Shortcut changes
• PR #4897: More detailed documentation about kernel_cmd
• PR #4891: Squash a few Sphinx warnings from nbconvert.utils.lexers docstrings
• PR #4679: JPG compression for inline pylab
• PR #4708: Fix indent and center
• PR #4789: fix IPython.embed
• PR #4655: prefer marked to pandoc for markdown2html
• PR #4876: don’t show tooltip if object is not found
• PR #4873: use ‘combine’ option to ucs package
• PR #4732: Accents in notebook names and in command-line (nbconvert)
• PR #4867: Update URL for Lawrence Hall of Science webcam image
• PR #4868: Static path fixes
• PR #4858: fix tb_offset when running a file
• PR #4826: some $.html( -> $.text()
• PR #4847: add js kernel_info request
• PR #4832: allow NotImplementedError in formatters
• PR #4803: BUG: fix cython magic support in ipython_directive
• PR #4865: build listed twice in .gitignore. Removing one.
• PR #4851: fix tooltip token regex for single-character names
• PR #4846: Remove some leftover traces of irunner
• PR #4820: fix regex for cleaning old logs with ipcluster
• PR #4844: adjustments to notebook app logging
• PR #4840: Error in Session.send_raw()
• PR #4819: update CodeMirror to 3.21
• PR #4823: Minor fixes for typos/inconsistencies in parallel docs
• PR #4811: document code mirror tab and shift-tab
• PR #4795: merge reveal templates
• PR #4796: update components
• PR #4806: Correct order of packages for unicode in nbconvert to LaTeX
• PR #4800: Qt frontend: Handle ‘aborted’ prompt replies.
• PR #4794: Compatibility fix for Python3 (Issue #4783 )
• PR #4799: minor js test fix
• PR #4788: warn when notebook is started in pylab mode
• PR #4772: Notebook server info files
• PR #4797: be conservative about kernel_info implementation
• PR #4787: non-python kernels run python code with qtconsole
• PR #4565: various display type validations
• PR #4703: Math macro in jinja templates.
• PR #4781: Fix “Source” text for the “Other Syntax” section of the “Typesetting Math” notebook
• PR #4776: Manually document py3compat module.

2.8. Issues closed in the 2.x development cycle
• PR #4533: propagate display metadata to all mimetypes
• PR #4785: Replacing a for-in loop by an index loop on an array
• PR #4780: Updating CSS for UI example.
• PR #3605: Modal UI
• PR #4758: Python 3.4 fixes
• PR #4735: add some HTML error pages
• PR #4775: Update whatsnew doc from PR files
• PR #4760: Make examples and docs more Python 3 aware
• PR #4773: Don’t wait forever for notebook server to launch/die for tests
• PR #4768: Qt console: Fix _prompt_pos accounting on timer flush output.
• PR #4727: Remove Nbconvert template loading magic
• PR #4763: Set numpydoc options to produce fewer Sphinx warnings.
• PR #4770: alway define aliases, even if empty
• PR #4766: add python -m entry points for everything
• PR #4767: remove manpages for irunner, iplogger
• PR #4751: Added --post-serve explanation into the nbconvert docs.
• PR #4762: whitelist alphanumeric characters for cookie_name
• PR #4625: Deprecate %profile magic
• PR #4745: warn on failed formatter calls
• PR #4746: remove redundant cls alias on Windows
• PR #4749: Fix bug in determination of public ips.
• PR #4715: restore use of tornado static_url in templates
• PR #4748: fix race condition in profiledir creation.
• PR #4720: never use ssh multiplexer in tunnels
• PR #4658: Bug fix for #4643: Regex object needs to be reset between calls in toolt...
• PR #4561: Add Formatter.pop(type)
• PR #4712: Docs shotgun 3
• PR #4713: Fix saving kernel history in Python 2
• PR #4744: don’t use lazily-evaluated rc.ids in wait_for_idle
• PR #4740: %env can’t set variables
• PR #4737: check every link when detecting virutalenv
• PR #4738: don’t inject help into user_ns
• PR #4739: skip html nbconvert tests when their dependencies are missing
• PR #4730: Fix stripping continuation prompts when copying from Qt console
• PR #4725: Doc fixes
• PR #4656: Nbconvert HTTP service
• PR #4710: make @interactive decorator friendlier with dill
• PR #4722: allow purging local results as long as they are not outstanding
• PR #4549: Updated IPython console lexers.
• PR #4570: Update IPython directive
• PR #4719: Fix comment typo in prefilter.py
• PR #4575: make sure to encode URL components for API requests
• PR #4718: Fixed typo in displaypub
• PR #4716: Remove input_prefilter hook
• PR #4691: survive failure to bind to localhost in zmq.iostream
• PR #4696: don’t do anything if add_anchor fails
• PR #4711: some typos in the docs
• PR #4700: use if main block in entry points
• PR #4692: setup.py symlink improvements
• PR #4265: JSON configuration file
• PR #4505: Nbconvert latex markdown images2
• PR #4608: transparent background match ... all colors
• PR #4678: allow ipython console to handle text/plain display
• PR #4706: remove irunner, iplogger
• PR #4701: Delete an old dictionary available for selecting the alignment of text.
• PR #4702: Making reveal font-size a relative unit.
• PR #4649: added a quiet option to %cpaste to suppress output
• PR #4690: Option to spew subprocess streams during tests
• PR #4688: Fixed various typos in docstrings.
• PR #4645: CasperJs utility functions.
• PR #4670: Stop bundling the numpydoc Sphinx extension
• PR #4675: common IPython prefix for ModIndex
• PR #4672: Remove unused ‘attic’ module
• PR #4671: Fix docstrings in utils.text

2.8. Issues closed in the 2.x development cycle 65
• PR #4669: add missing help strings to HistoryManager configurables
• PR #4668: Make non-ASCII docstring unicode
• PR #4650: added a note about sharing of nbconvert tempates
• PR #4646: Fixing various output related things:
• PR #4665: check for libedit in readline on OS X
• PR #4606: Make running PYTHONSTARTUP optional
• PR #4654: Fixing left padding of text cells to match that of code cells.
• PR #4306: add raw_mimetype metadata to raw cells
• PR #4576: Tighten up the vertical spacing on cells and make the padding of cells more consistent
• PR #4353: Don’t reset the readline completer after each prompt
• PR #4567: Adding prompt area to non-CodeCells to indent content.
• PR #4446: Use SVG plots in OctaveMagic by default due to lack of Ghostscript on Windows Octave
• PR #4613: remove configurable.created
• PR #4631: Use argument lists for command help tests
• PR #4633: Modifies test_get_long_path_name_win32() to allow for long path names in temp dir
• PR #4642: Allow docs to build without PyQt installed.
• PR #4641: Don’t check for wx in the test suite.
• PR #4622: make QtConsole Lexer configurable
• PR #4594: Fixed #2923 Move Save Away from Cut in toolbar
• PR #4593: don’t interfere with set_next_input contents in qtconsole
• PR #4640: Support matplotlib’s Gtk3 backend in –pylab mode
• PR #4639: Minor import fix to get qtconsole with –pylab=qt working
• PR #4637: Fixed typo in links.txt.
• PR #4634: Fix nbruin in notebooks with non-code cells.
• PR #4632: Restore the ability to run tests from a function.
• PR #4624: Fix crash when $EDITOR is non-ASCII
• PR #4453: Play nice with App Nap
• PR #4541: relax ipconfig matching on Windows
• PR #4552: add pickleutil.use_dill
• PR #4590: Font awesome for IPython slides
• PR #4589: Inherit the width of pre code inside the input code cells.
• PR #4588: Update reveal.js CDN to 2.5.0.
• PR #4569: store cell toolbar preset in notebook metadata
• PR #4609: Fix bytes regex for Python 3.
• PR #4581: Writing unicode to stdout
• PR #4591: Documenting codemirror shortcuts.
• PR #4607: Tutorial doc should link to user config intro
• PR #4601: test that rename fails with 409 if it would clobber
• PR #4599: re-cast int/float subclasses to int/float in json_clean
• PR #4542: new ipython history clear subcommand
• PR #4568: don’t use lazily-evaluated rc.ids in wait_for_idle
• PR #4572: DOC: %profile docstring should reference %prun
• PR #4571: no longer need 3 suffix on travis, tox
• PR #4566: Fixing cell_type in CodeCell constructor.
• PR #4563: Specify encoding for reading notebook file.
• PR #4452: support notebooks in %run
• PR #4546: fix warning condition on notebook startup
• PR #4540: Apidocs3
• PR #4553: Fix Python 3 handling of urllib
• PR #4543: make hiding of initial namespace optional
• PR #4517: send shutdown_request on exit of ipython console
• PR #4528: improvements to bash completion
• PR #4532: Hide dynamically defined metaclass base from Sphinx.
• PR #4515: Spring Cleaning, and Load speedup
• PR #4529: note routing identities needed for input requests
• PR #4514: allow restart in %run -d
• PR #4527: add redirect for 1.0-style ‘files/’ prefix links
• PR #4526: Allow unicode arguments to passwd_check on Python 2
• PR #4403: Global highlight language selection.
• PR #4250: outputarea.js: Wrap inline SVGs inside an iframe
• PR #4521: Read wav files in binary mode
• PR #4444: Css cleaning
• PR #4523: Use username and password for MongoDB on ShiningPanda
• PR #4510: Update whatsnew from PR files
• PR #4441: add setup.py jsversion
• PR #4518: Fix for race condition in url file decoding.
• PR #4497: don’t automatically unpack datetime objects in the message spec
• PR #4506: wait for empty queues as well as load-balanced tasks
• PR #4492: Configuration docs refresh
• PR #4508: Fix some uses of map() in Qt console completion code.
• PR #4498: Daemon StreamCapturer
• PR #4499: Skip clipboard test on unix systems if headless.
• PR #4460: Better clipboard handling, esp. with pywin32
• PR #4496: Pass nbformat object to write call to save .py script
• PR #4466: various pandoc latex fixes
• PR #4473: Setup for Python 2/3
• PR #4459: protect against broken repr in lib.pretty
• PR #4457: Use ~/.ipython as default config directory
• PR #4489: check realpath of env in init_virtualenv
• PR #4490: fix possible race condition in test_await_data
• PR #4476: Fix: Remove space added by display(JavaScript) on page reload
• PR #4398: [Notebook] Deactivate tooltip on tab by default.
• PR #4480: Docs shotgun 2
• PR #4488: fix typo in message spec doc
• PR #4479: yet another JS race condition fix
• PR #4477: Allow incremental builds of the html_noapi docs target
• PR #4470: Various Config object cleanups
• PR #4410: make close-and-halt work on new tabs in Chrome
• PR #4469: Python 3 & getcwd
• PR #4451: fix: allow JS test to run after shutdown test
• PR #4456: Simplify StreamCapturer for subprocess testing
• PR #4464: Correct description for Bytes traitlet type
• PR #4465: Clean up MANIFEST.in
• PR #4461: Correct TypeError message in svg2pdf
• PR #4458: use signalstatus if exit status is undefined
• PR #4438: Single codebase Python 3 support (again)
• PR #4198: Version conversion, support for X to Y even if Y < X (nbformat)
• PR #4415: More tooltips in the Notebook menu
• PR #4450: remove monkey patch for older versions of tornado
• PR #4423: Fix progress bar and scrolling bug.
• PR #4435: raise 404 on not found static file
• PR #4442: fix and add shim for change introduce by #4195
• PR #4436: allow require("nbextensions/extname") to load from IPYTHONDIR/nbextensions
• PR #4437: don’t compute etags in static file handlers
• PR #4427: notebooks should always have one checkpoint
• PR #4425: fix js pythonisme
• PR #4195: IPEP 21: widget messages
• PR #4434: Fix broken link for Dive Into Python.
• PR #4428: bump minimum tornado version to 3.1.0
• PR #4302: Add an Audio display class
• PR #4285: Notebook javascript test suite using CasperJS
• PR #4420: Allow checking for backports via milestone
• PR #4426: set kernel cwd to notebook’s directory
• PR #4389: By default, Magics inherit from Configurable
• PR #4393: Capture output from subprocs during test, and display on failure
• PR #4419: define InlineBackend configurable in its own file
• PR #4303: Multidirectory support for the Notebook
• PR #4371: Restored ipython profile locate dir and fixed typo. (Fixes #3708).
• PR #4414: Specify unicode type properly in rmagic
• PR #4413: don’t instantiate IPython shell as class attr
• PR #4400: Remove 5s wait on inactivity on GUI inputhook loops
• PR #4412: Fix traitlet _notify_trait by-ref issue
• PR #4378: split adds new cell above, rather than below
• PR #4405: Bring display of builtin types and functions in line with Py 2
• PR #4367: clean up of documentation files
• PR #4401: Provide a name of the HistorySavingThread
• PR #4384: fix menubar height measurement

2.8. Issues closed in the 2.x development cycle 69
• PR #4377: fix tooltip cancel
• PR #4293: Factorise code in tooltip for julia monkeypatching
• PR #4292: improve js-completer logic.
• PR #4363: set_next_input: keep only last input when repeatedly called in a single cell
• PR #4382: Use safe_hasattr in dir2
• PR #4379: fix (CTRL-M -) shortcut for splitting cell in FF
• PR #4380: Test and fixes for localinterfaces
• PR #4372: Don’t assume that SyntaxTB is always called with a SyntaxError
• PR #4342: Return value directly from the try block and avoid a variable
• PR #4154: Center LaTeX and figures in markdown
• PR #4311: %load -s to load specific functions or classes
• PR #4350: WinHPC launcher fixes
• PR #4345: Make irunner compatible with upcoming pexpect 3.0 interface
• PR #4276: Support container methods in config
• PR #4359: test_pylabtools also needs to modify matplotlib.rcParamsOrig
• PR #4355: remove hardcoded box-orient
• PR #4333: Add Edit Notebook Metadata to Edit menu
• PR #4349: Script to update What’s New file
• PR #4348: Call PDF viewer after latex compiling (nbconvert)
• PR #4346: getpass() on Windows & Python 2 needs bytes prompt
• PR #4304: use netifaces for faster IPython.utils.localinterfaces
• PR #4305: Add even more ways to populate localinterfaces
• PR #4313: remove strip_math_space
• PR #4325: Some changes to improve readability.
• PR #4281: Adjust tab completion widget if too close to bottom of page.
• PR #4347: Remove pycolor script
• PR #4322: Scroll to the top after change of slides in the IPython slides
• PR #4289: Fix scrolling output (not working post clear_output changes)
• PR #4343: Make parameters for kernel start method more general
• PR #4237: Keywords should shadow magic functions
• PR #4338: adjust default value of level in sync_imports
• PR #4328: Remove unused loop variable.
• PR #4340: fix mathjax download url to new GitHub format
• PR #4336: use simple replacement rather than string formatting in format_kernel_cmd
• PR #4264: catch unicode error listing profiles
• PR #4314: catch EACCES when binding notebook app
• PR #4324: Remove commented addthis toolbar
• PR #4327: Use the with statement to open a file.
• PR #4318: fix initial sys.path
• PR #4315: Explicitly state what version of Pandoc is supported in docs/install
• PR #4316: underscore missing on notebook_p4
• PR #4295: Implement boundary option for load magic (#1093)
• PR #4300: traits defaults are strings not object
• PR #4297: Remove an unreachable return statement.
• PR #4260: Use subprocess for system_raw
• PR #4277: add nbextensions
• PR #4294: don’t require tornado 3 in --post serve
• PR #4270: adjust Scheduler timeout logic
• PR #4278: add -a to easy_install command in libedit warning
• PR #4282: Enable automatic line breaks in MathJax.
• PR #4279: Fixing line-height of list items in tree view.
• PR #4253: fixes #4039.
• PR #4131: Add module’s name argument in %%cython magic
• PR #4269: Add mathletters option and longtable package to latex_base.tplx
• PR #4230: Switch correctly to the user’s default matplotlib backend after inline.
• PR #4271: Hopefully fix ordering of output on ShiningPanda
• PR #4239: more informative error message for bad serialization
• PR #4263: Fix excludes for IPython.testing
• PR #4112: nbconvert: Latex template refactor
• PR #4261: Fixing a formatting error in the custom display example notebook.
• PR #4259: Fix Windows test exclusions
• PR #4229: Clear_output: Animation & widget related changes.
• PR #4151: Refactor alias machinery
• PR #4153: make timeit return an object that contains values

2.8. Issues closed in the 2.x development cycle
• PR #4258: to-backport label is now 1.2
• PR #4242: Allow passing extra arguments to iptest through for nose
• PR #4257: fix unicode argv parsing
• PR #4166: avoid executing code in utils.localinterfaces at import time
• PR #4214: engine ID metadata should be unicode, not bytes
• PR #4232: no highlight if no language specified
• PR #4218: Fix display of SyntaxError when .py file is modified
• PR #4207: add setup.py css command
• PR #4224: clear previous callbacks on execute
• PR #4180: Iptest refactoring
• PR #4105: JS output area misaligned
• PR #4220: Various improvements to docs formatting
• PR #4187: Select adequate highlighter for cell magic languages
• PR #4228: update -dev docs to reflect latest stable version
• PR #4219: Drop bundled argpase
• PR #3851: Adds an explicit newline for pretty-printing.
• PR #3622: Drop fakemodule
• PR #4080: change default behavior of database task storage
• PR #4197: enable cython highlight in notebook
• PR #4225: Updated docstring for core.display.Image
• PR #4175: nbconvert: Jinjaless exporter base
• PR #4208: Added a lightweight “htmlcore” Makefile entry
• PR #4209: Magic doc fixes
• PR #4217: avoid importing numpy at the module level
• PR #4213: fixed dead link in examples/notebooks readme to Part 3
• PR #4183: ESC should be handled by CM if tooltip is not on
• PR #4193: Update for #3549: Append Firefox overflow-x fix
• PR #4205: use TextIOWrapper when communicating with pandoc subprocess
• PR #4204: remove some extraneous print statements from IPython.parallel
• PR #4201: HeadingCells cannot be split or merged
• PR #4048: finish up speaker-notes PR
• PR #4079: trigger Kernel.status_started after websockets open
• PR #4186: moved DummyMod to proper namespace to enable dill pickling
• PR #4190: update version-check message in setup.py and IPython.__init__
• PR #4188: Allow user_ns trait to be None
• PR #4189: always fire LOCAL_IPS.extend(PUBLIC_IPS)
• PR #4174: various issues in markdown and rst templates
• PR #4178: add missing data_javascript
• PR #4168: Py3 failing tests
• PR #4181: nbconvert: Fix, sphinx template not removing new lines from headers
• PR #4043: don’t ‘restore_bytes’ in from_JSON
• PR #4149: reuse more kernels in kernel tests
• PR #4163: Fix for incorrect default encoding on Windows.
• PR #4136: catch javascript errors in any output
• PR #4171: add nbconvert config file when creating profiles
• PR #4172: add ability to check what PRs should be backported in backport_pr
• PR #4167: –fast flag for test suite!
• PR #4125: Basic exercise of ipython [subcommand] -h and help-all
• PR #4085: nbconvert: Fix sphinx preprocessor date format string for Windows
• PR #4159: don’t split .cell and div.cell CSS
• PR #4165: Remove use of parametric tests
• PR #4158: generate choices for --gui configurable from real mapping
• PR #4083: Implement a better check for hidden values for %who etc.
• PR #4147: Reference notebook examples, fixes #4146.
• PR #4065: do not include specific css in embedable one
• PR #4092: nbconvert: Fix for unicode html headers, Windows + Python 2.x
• PR #4074: close Client sockets if connection fails
• PR #4064: Store default codemirror mode in only 1 place
• PR #4104: Add way to install MathJax to a particular profile
• PR #4161: Select name when renaming a notebook
• PR #4160: Add quotes around ".[notebook]" in readme
• PR #4144: help_end transformer shouldn’t pick up ? in multiline string
• PR #4090: Add LaTeX citation handling to nbconvert
• PR #4143: update example custom.js

2.8. Issues closed in the 2.x development cycle
• PR #4142: DOC: unwrap openssl line in public_server doc
• PR #4126: update tox.ini
• PR #4141: add files with a separate add call in backport_pr
• PR #4137: Restore autorestore option for storemagic
• PR #4098: pass profile-dir instead of profile name to Kernel
• PR #4120: support input in Python 2 kernels
• PR #4088: nbconvert: Fix coalescestreams line with incorrect nesting causing strange behavior
• PR #4060: only strip continuation prompts if regular prompts seen first
• PR #4132: Fixed name error bug in function safe_unicode in module py3compat.
• PR #4121: move test_kernel from IPython.zmq to IPython.kernel
• PR #4118: ZMQ heartbeat channel: catch EINTR exceptions and continue.
• PR #4070: New changes should go into pr/ folder
• PR #4054: use unicode for HTML export
• PR #4106: fix a couple of default block values
• PR #4107: update parallel magic tests with capture_output API
• PR #4102: Fix clashes between debugger tests and coverage.py
• PR #4115: Update docs on declaring a magic function
• PR #4101: restore accidentally removed EngineError
• PR #4096: minor docs changes
• PR #4094: Update target branch before backporting PR
• PR #4069: Drop monkeypatch for pre-1.0 nose
• PR #4056: respect pylab_import_all when --pylab specified at the command-line
• PR #4091: Make Qt console banner configurable
• PR #4086: fix missing errno import
• PR #4084: Use msvcrt.getwch() for Windows pager.
• PR #4073: rename post_processors submodule to postprocessors
• PR #4075: Update supported Python versions in tools/test_pr
• PR #4068: minor bug fix, define ‘cell’ in dialog.js.
• PR #4044: rename call methods to transform and postprocess
• PR #3744: capture rich output as well as stdout/err in capture_output
• PR #3969: “use strict” in most (if not all) our javascript
• PR #4030: exclude .git in MANIFEST.in
- PR #4047: Use istype() when checking if canned object is a dict
- PR #4031: don’t close_fds on Windows
- PR #4029: bson.Binary moved
- PR #3883: skip test on unix when x11 not available
- PR #3863: Added working speaker notes for slides.
- PR #4035: Fixed custom jinja2 templates being ignored when setting template_path
- PR #4002: Drop Python 2.6 and 3.2
- PR #4026: small doc fix in nbconvert
- PR #4016: Fix IPython.start_* functions
- PR #4021: Fix parallel.client.View map() on numpy arrays
- PR #4022: DOC: fix links to matplotlib, notebook docs
- PR #4018: Fix warning when running IPython.kernel tests
- PR #4017: Add REPL-like printing of final/return value to %%R cell magic
- PR #4019: Test skipping without unicode paths
- PR #4008: Transform code before %prun/%%prun runs
- PR #4014: Fix typo in ipapp
- PR #3997: DOC: typos + rewording in examples/notebooks/Cell Magics.ipynb
- PR #3914: nbconvert: Transformer tests
- PR #3987: get files list in backport_pr
- PR #3923: nbconvert: Writer tests
- PR #3974: nbconvert: Fix app tests on Window7 w/ Python 3.3
- PR #3937: make tab visible in codemirror and light red background
- PR #3933: nbconvert: Post-processor tests
- PR #3978: fix --existing with non-localhost IP
- PR #3939: minor checkpoint cleanup
- PR #3955: complete on % for magic in notebook
- PR #3981: BF: fix nbconert rst input prompt spacing
- PR #3960: Don’t make sphinx a dependency for importing nbconvert
- PR #3973: logging.Formatter is not new-style in 2.6

Issues (434):
- #5476: For 2.0: Fix links in Notebook Help Menu
- #5337: Examples reorganization

2.8. Issues closed in the 2.x development cycle
• #5436: CodeMirror shortcuts in QuickHelp
• #5444: Fix numeric verification for Int and Float text widgets.
• #5443: Int and Float Widgets don’t allow negative signs
• #5449: Stretch keyboard shortcut dialog
• #5471: Add coding magic comment to nbconvert Python template
• #5470: UTF-8 Issue When Converting Notebook to a Script.
• #5369: FormatterWarning for SVG matplotlib output in notebook
• #5460: Can’t start the notebook server specifying a notebook
• #2918: CodeMirror related issues.
• #5431: update github_stats and gh_api for 2.0
• #4887: Add tests for modal UI
• #5290: Add dual mode JS tests
• #5448: Cmd+/ shortcut doesn’t work in IPython master
• #5447: Add %%python2 cell magic
• #5442: Make a “python2” alias or rename the “python”cell magic.
• #2495: non-ascii characters in the path
• #4554: dictDB: Exception due to str to datetime comparission
• #5006: Comm code is not run in the same context as notebook code
• #5118: Weird interact behavior
• #5401: Empty code cells in nbconvert rst output cause problems
• #5434: fix check for empty cells in rst template
• #4944: Trouble finding ipynb path in Windows 8
• #4605: Change the url of Editor Shorcuts in the notebook menu.
• #5425: Update COPYING.txt
• #5348: BUG: HistoryAccessor.get_session_info(0) - exception
• #5293: Javascript(“element.append()”) looks broken.
• #5363: Disable saving if notebook has stopped loading
• #5189: Tooltip pager mode is broken
• #5330: Updates to shell reference doc
• #5397: Accordion widget broken
• #5106: Flexbox CSS specificity bugs
• #5297: tooltip triggers focus bug
• #5417: scp checking for existence of directories: directory names are incorrect
• #5302: Parallel engine registration fails for slow engines
• #5334: notebook’s split-cell shortcut dangerous / incompatible with Neo layout (for instance)
• #5324: Style of raw_input UI is off in notebook
• #5350: Converting notebooks with spaces in their names to RST gives broken images
• #5049: update quickhelp on adding and removing shortcuts
• #4941: Eliminating display of intermediate stages in progress bars
• #5345: nbconvert to markdown does not use backticks
• #5357: catch exception in copystat
• #5351: Notebook saving fails on smb share
• #4946: TeX produced cannot be converted to PDF
• #5347: pretty print list too slow
• #5238: Raw cell placeholder is not removed when you edit the cell
• #5382: Qtconsole doesn’t run in Python 3
• #5378: Unexpected and new conflict between PyFileConfigLoader and IPythonQtConsoleApp
• #4945: Heading/cells positioning problem and cell output wrapping
• #5084: Consistent approach for HTML/JS output on nbviewer
• #4902: print preview does not work, custom.css not found
• #5336: TypeError in bootstrap-tour.min.js
• #5303: Changed Hub.registration_timeout to be a config input.
• #995: Paste-able mode in terminal
• #5305: Tuple unpacking for shell escape
• #5232: Make nbconvert html full output like notebook’s html.
• #5224: Audit nbconvert HTML output
• #5253: display any output from this session in terminal console
• #5251: ipython console ignoring some stream messages?
• #4802: Tour of the notebook UI (was UI elements inline with highlighting)
• #5103: Moving Constructor definition to the top like a Function definition
• #5264: Test failures on master with Anaconda
• #4833: Serve /usr/share/javascript at/_sysassets/javascript/ in notebook
• #5071: Prevent %pylab from clobbering interactive
• #5282: Exception in widget __del__ methods in Python 3.4.

2.8. Issues closed in the 2.x development cycle
• #5280: append Firefox overflow-x fix
• #5120: append Firefox overflow-x fix, again
• #4127: autoreload shouldn’t rely on .pyc modification times
• #5272: allow highlighting language to be set from notebook metadata
• #5050: Notebook cells truncated with Firefox
• #4839: Error in Session.send_raw()
• #5188: New events system
• #5076: Refactor keyboard handling
• #4886: Refactor and consolidate different keyboard logic in JavaScript code
• #5002: the green cell border moving forever in Chrome, when there are many code cells.
• #5259: Codemirror still active in command mode
• #5219: Output images appear as small thumbnails (Notebook)
• #4829: Not able to connect qtconsole in Windows 8
• #5152: Hide __pycache__ in dashboard directory list
• #5151: Case-insensitive sort for dashboard list
• #4603: Warn when overwriting a notebook with upload
• #4895: Improvements to %run completions
• #3459: Filename completion when run script with %run
• #5225: Add JavaScript to nbconvert HTML display priority
• #5034: Audit the places where we call .html(something)
• #5094: Dancing cells in notebook
• #4999: Notebook focus effects
• #5149: Clicking on a TextBoxWidget in FF completely breaks dual mode.
• #5207: Children fire event
• #5227: display_method of objects with custom __getattr__
• #5236: Cursor keys do not work to leave Markdown cell while it’s being edited
• #5205: Use CTuple traitlet for Widget children
• #5230: notebook rename does not respect url prefix
• #5218: Test failures with Python 3 and enabled warnings
• #5115: Page Breaks for Print Preview Broken by display: flex - Simple CSS Fix
• #5024: Make nbconvert HTML output smart about page breaking
• #4985: Add automatic Closebrackets function to Codemirror.
• #5184: print ‘xa’ crashes the interactive shell
• #5214: Downloading notebook as Python (.py) fails
• #5211: AttributeError: ‘module’ object has no attribute ‘_outputfile’
• #5206: [CSS?] Inconsistencies in nbconvert divs and IPython Notebook divs?
• #5201: node != nodejs within Debian packages
• #5112: band-aid for completion
• #4860: Completer As-You-Type Broken
• #5116: reorganize who knows what about paths
• #4973: Adding security.js with 1st attempt at is_safe
• #5164: test_oinspect.test_calltip_builtin failure with python3.4
• #5127: Widgets: skip intermediate callbacks during throttling
• #5013: Widget alignment differs between FF and Chrome
• #5141: tornado error static file
• #5160: TemporaryWorkingDirectory incompatible with python3.4
• #5140: WIP: %kernels magic
• #4987: Widget lifecycle problems
• #5129: UCS package break latex export on non-ascii
• #4986: Cell horizontal scrollbar is missing in FF but not in Chrome
• #4685: nbconvert ignores image size metadata
• #5155: Notebook logout button does not work (source typo)
• #2678: Ctrl-m keyboard shortcut clash on Chrome OS
• #5113: ButtonWidget without caption wrong height.
• #4778: add APIs for installing notebook extensions
• #5046: python setup.py failed vs git submodule update worked
• #4925: Notebook manager api fixes
• #5073: Cannot align widgets horizontally in the notebook
• #4996: require print_method to be a bound method
• #4990: _repr_html_ exception reporting corner case when using type(foo)
• #5099: Notebook: Changing base_project_url results in failed WebSockets call
• #5096: Client.map is not fault tolerant
• #4997: Inconsistent %matplotlib qt behavior
• #5041: Remove more .html(...) calls.

2.8. Issues closed in the 2.x development cycle
• #5078: Updating JS tests README.md
• #4977: ensure scp destination directories exist (with mkdir -p)
• #3411: ipython parallel: scp failure.
• #5064: Errors during interact display at the terminal, not anywhere in the notebook
• #4921: Add PDF formatter and handling
• #4920: Adding PDFFormatter and kernel side handling of PDF display data
• #5048: Add edit/command mode indicator
• #4889: Add UI element for indicating command/edit modes
• #5052: Add q to toggle the pager.
• #5000: Closing pager with keyboard in modal UI
• #5069: Box model changes broke the Keyboard Shortcuts help modal
• #4960: Interact/Interactive for widget
• #4883: Implement interact/interactive for widgets
• #5038: Fix multiple press keyboard events
• #5054: UnicodeDecodeError: ‘ascii’ codec can’t decode byte 0xc6 in position 1: ordinal not in range(128)
• #5031: Bug during integration of IPython console in Qt application
• #5057: iopubwatcher.py example is broken.
• #4747: Add event for output_area adding an output
• #5001: Add directory navigation to dashboard
• #5016: Help menu external-link icons break layout in FF
• #4885: Modal UI behavior changes
• #5009: notebook signatures don’t work
• #4975: setup.py changes for 2.0
• #4774: emit event on appended element on dom
• #5020: Python Lists translated to javascript objects in widgets
• #5003: Fix pretty reprs of super() objects
• #5012: Make SelectionWidget.values a dict
• #4961: Bug when constructing a selection widget with both values and labels
• #4283: A < in a markdown cell strips cell content when converting to latex
• #4006: iptest IPython broken
• #4251: & escaped to &amp; in tex?
• #5027: pin lessc to 1.4
• #4323: Take 2: citation2latex filter (using HTMLParser)
• #4196: Printing notebook from browser gives 1-page truncated output
• #4842: more subtle kernel indicator
• #4057: No path to notebook examples from Help menu
• #5015: don’t write cell.trusted to disk
• #4617: Changed url link in Help dropdown menu.
• #4976: Container widget layout broken on Firefox
• #4981: Vertical slider layout broken
• #4793: Message spec changes related to `clear_output`
• #4982: Live readout for slider widgets
• #4813: make help menu a template
• #4989: Filename tab completion completely broken
• #1380: Tab should insert 4 spaces in # comment lines
• #2888: spaces vs tabs
• #1193: Allow resizing figures in notebook
• #4504: Allow input transformers to raise SyntaxError
• #4697: Problems with height after toggling header and toolbar...
• #4951: TextWidget to code cell command mode bug.
• #4809: Arbitrary scrolling (jumping) in clicks in modal UI for notebook
• #4971: Fixing issues with js tests
• #4972: Work around problem in doctest discovery in Python 3.4 with PyQt
• #4892: IPython.qt test failure with python3.4
• #4863: BUG: cannot create an OBJECT array from memory buffer
• #4704: Subcommand `profile` ignores –ipython-dir
• #4845: Add Origin Checking.
• #4870: ipython_directive, report except/warn in block and add :okexcept: :okwarning: options to suppress
• #4956: Shift-Enter does not move to next cell
• #4662: Menu cleanup
• #4824: sign notebooks
• #4848: avoid import of nearby temporary with `%edit`
• #4731: %edit files mistakenly import modules in /tmp
• #4950: Two fixes for file upload related bugs
• #4871: Notebook upload fails after Delete
• #4825: File Upload URL set incorrectly
• #3867: display.FileLinks should work in the exported html version of a notebook
• #4948: reveal: ipython css overrides reveal themes
• #4947: reveal: slides that are too big?
• #4051: Test failures with Python 3 and enabled warnings
• #3633: outstanding issues over in ipython/nbconvert repo
• #4087: Sympy printing in the example notebook
• #4627: Document various QtConsole embedding approaches.
• #4849: Various unicode fixes (mostly on Windows)
• #3653: autocompletion in “from package import <tab>”
• #4583: overwrite? prompt gets EOFError in 2 process
• #4807: Correct handling of ansi colour codes when nbconverting to latex
• #4611: Document how to compile .less files in dev docs.
• #4618: “Editor Shortcuts” link is broken in help menu dropdown notebook
• #4522: DeprecationWarning: the sets module is deprecated
• #4368: No symlink from ipython to ipython3 when inside a python3 virtualenv
• #4234: Math without $$ doesn’t show up when converted to slides
• #4194: config.TerminalIPythonApp.nosep does not work
• #1491: prefilter not called for multi-line notebook cells
• #4001: Windows IPython executable /scripts/ipython not working
• #3959: think more carefully about text wrapping in nbconvert
• #4907: Test for traceback depth fails on Windows
• #4906: Test for IPython.embed() fails on Windows
• #4912: Skip some Windows io failures
• #3700: stdout/stderr should be flushed printing exception output...
• #1181: greedy completer bug in terminal console
• #2032: check for a few places we should be using DEFAULT_ENCODING
• #4882: Too many files open when starting and stopping kernel repeatedly
• #4880: set profile name from profile_dir
• #4238: parallel.Client() not using profile that notebook was run with?
• #4853: fix setting image height/width from metadata
• #4786: Reduce spacing of heading cells
• #4680: Minimal pandoc version warning
• #3707: nbconvert: Remove IPython magic commands from –format=”python” output
• #4130: PDF figures as links from png or svg figures
• #3919: Allow –profile to be passed a dir.
• #2136: Handle hard newlines in pretty printer
• #4790: Notebook modal UI: “merge cell below” key binding, `shift+=`, does not work with some keyboard layouts
• #4884: Keyboard shortcut changes
• #1184: slow handling of keyboard input
• #4913: Mathjax, Markdown, tex, env* and italic
• #3972: nbconvert: Template output testing
• #4903: use https for all embeds
• #4874: –debug does not work if you set .kernel_cmd
• #4679: JPG compression for inline pylab
• #4708: Fix indent and center
• #4789: fix IPython.embed
• #4759: Application._load_config_files log parameter default fails
• #3153: docs / file menu: explain how to exit the notebook
• #4791: Did updates to ipython_directive bork support for cython magic snippets?
• #4655: prefer marked to pandoc for markdown2html
• #3441: Fix focus related problems in the notebook
• #3402: Feature Request: Save As (latex, html,..etc) as a menu option in Notebook rather than explicit need to invoke nbconvert
• #3224: Revisit layout of notebook area
• #2746: rerunning a cell with long output (exception) scrolls to much (html notebook)
• #2667: can’t save opened notebook if accidently delete the notebook in the dashboard
• #3026: Reporting errors from _repr_<type>_.methods
• #1844: Notebook does not exist and permalinks

2.8. Issues closed in the 2.x development cycle
• #2450: [closed PR] Prevent jumping of window to input when output is clicked.
• #3166: IPEP 16: Notebook multi directory dashboard and URL mapping
• #3691: Slight misalignment of Notebook menu bar with focus box
• #4875: Empty tooltip with object_found = false still being shown
• #4432: The SSL cert for the MathJax CDN is invalid and URL is not protocol agnostic
• #2633: Help text should leave current cell active
• #3976: DOC: Pandas link on the notebook help menu?
• #4082: /new handler redirect cached by browser
• #4298: Slow ipython –pylab and ipython notebook startup
• #4545: %store magic not working
• #4610: toolbar UI enhancements
• #4782: New modal UI
• #4732: Accents in notebook names and in command-line (nbconvert)
• #4752: link broken in docs/examples
• #4835: running ipython on python files adds an extra traceback frame
• #4792: repr_html exception warning on qtconsole with pandas #4745
• #4834: function tooltip issues
• #4808: Docstrings in Notebook not displayed properly and introspection
• #4846: Remove some leftover traces of irunner
• #4810: ipcluster bug in clean_logs flag
• #4812: update CodeMirror for the notebook
• #671: add migration guide for old IPython config
• #4783: ipython 2dev under windows / (win)python 3.3 experiment
• #4772: Notebook server info files
• #4765: missing build script for highlight.js
• #4787: non-python kernels run python code with qtconsole
• #4703: Math macro in jinja templates.
• #4595: ipython notebook XSS vulnerable
• #4776: Manually document py3compat module.
• #4686: For-in loop on an array in cell.js
• #3605: Modal UI
• #4769: Ipython 2.0 will not startup on py27 on windows
• #4482: reveal.js converter not including CDN by default?
• #4761: ipv6 address triggers cookie exception
• #4580: rename or remove %profile magic
• #4643: Docstring does not open properly
• #4714: Static URLs are not auto-versioned
• #2573: document code mirror keyboard shortcuts
• #4717: hang in parallel.Client when using SSHAgent
• #4544: Clarify the requirement for pyreadline on Windows
• #3451: revisit REST /new handler to avoid systematic crawling.
• #2922: File => Save as ‘.py’ saves magic as code
• #4728: Copy/Paste stripping broken in version > 0.13.x in QTConsole
• #4539: Nbconvert: Latex to PDF conversion fails on notebooks with accented letters
• #4721: purge_results with jobid crashing - looking for insight
• #4620: Notebook with ? in title defies autosave, renaming and deletion.
• #4574: Hash character in notebook name breaks a lot of things
• #4709: input_prefilter hook not called
• #1680: qtconsole should support –no-banner and custom banner
• #4689: IStream IP address configurable
• #4698: Missing “if __name__ == ‘__main__’:” check in /usr/bin/ipython
• #4191: NBConvert: markdown inline and locally referenced files have incorrect file location for latex
• #2865: %%%!? does not display the shell execute docstring
• #1551: Notebook should be saved before printing
• #4612: remove Configurable.created?
• #4629: Lots of tests fail due to space in sys.executable
• #4644: Fixed URLs for notebooks
• #4621: IPython 1.1.0 Qtconsole syntax highlighting highlights python 2 only built-ins when using python 3
• #2923: Move Delete Button Away from Save Button in the HTML notebook toolbar
• #4615: UnicodeDecodeError
• #4431: ipython slow in os x mavericks?
• #4538: DOC: document how to change ipcontroller-engine.json in case controller was started with –ip="*"
• #4551: Serialize methods and closures

2.8. Issues closed in the 2.x development cycle
• #4081: [Nbconvert][reveal] link to font awesome?
• #4602: “ipcluster stop” fails after “ipcluster start –daemonize” using python3.3
• #4578: NBconvert fails with unicode errors when --stdout and file redirection is specified and HTML entities are present
• #4600: Renaming new notebook to an exist name silently deletes the old one
• #4598: Qtconsole docstring pop-up fails on method containing defaulted enum argument
• #951: Remove Tornado monkeypatch
• #4564: Notebook save failure
• #4562: nbconvert: Default encoding problem on OS X
• #1675: add file_to_run=file.ipynb capability to the notebook
• #4516: ipython console doesn’t send a shutdown_request
• #3043: can’t restart pdb session in ipython
• #4524: Fix bug with non ascii passwords in notebook login
• #1866: problems rendering an SVG?
• #4520: unicode error when trying Audio(‘data/Bach Cello Suite #3.wav’)
• #4493: Qtconsole cannot print an ISO8601 date at nanosecond precision
• #4502: intermittent parallel test failure test_purge_everything
• #4495: firefox 25.0: notebooks report “Notebook save failed”, .py script save fails, but .ipynb save succeeds
• #4245: nbconvert latex: code highlighting causes error
• #4486: Test for whether inside virtualenv does not work if directory is symlinked
• #4485: Incorrect info in “Messaging in IPython” documentation.
• #4447: Ipcontroller broken in current HEAD on windows
• #4241: Audio display object
• #4463: Error on empty c.Session.key
• #4454: UnicodeDecodeError when starting Ipython notebook on a directory containing a file with a non-ascii character
• #3801: Autocompletion: Fix issue #3723 – ordering of completions for magic commands and variables with same name
• #3723: Code completion: ‘matplotlib’ and ‘%matplotlib’
• #4396: Always checkpoint al least once?
• #2524: [Notebook] Clear kernel queue
• #2292: Client side tests for the notebook
• #4424: Dealing with images in multidirectory environment
• #4388: Make writing configurable magics easier
• #852: Notebook should be saved before downloading
• #3708: ipython profile locate should also work
• #1349: ? may generate hundreds of cell
• #4381: Using hasattr for trait_names instead of just looking for it directly/using __dir__?
• #4361: Crash Ultratraceback/ session history
• #3044: IPython notebook autocomplete for filename string converts multiple spaces to a single space
• #3346: Up arrow history search shows duplicates in Qtconsole
• #3496: Fix import errors when running tests from the source directory
• #4114: If default profile doesn’t exist, can’t install mathjax to any location
• #4335: TestPylabSwitch.test_qt fails
• #4291: serve like option for nbconvert –to latex
• #1824: Exception before prompting for password during ssh connection
• #4309: Error in nbconvert - closing </code> tag is not inserted in HTML under some circumstances
• #4351: /parallel/apps/launcher.py error
• #3603: Upcoming issues with nbconvert
• #4296: sync_imports() fails in python 3.3
• #4339: local mathjax install doesn’t work
• #4334: NotebookApp.webapp_settings static_url_prefix causes crash
• #4308: Error when use “ipython notebook” in win7 64 with python2.7.3 64.
• #4317: Relative imports broken in the notebook (Windows)
• #3658: Saving Notebook clears “Kernel Busy” status from the page and titlebar
• #4312: Link broken on ipython-doc stable
• #1093: Add boundary options to %load
• #3619: Multi-dir webservice design
• #4299: Nbconvert, default_preprocessors to list of dotted name not list of obj
• #3210: IPython.parallel tests seem to hang on ShiningPanda
• #4280: MathJax Automatic Line Breaking
• #4039: Celltoolbar example issue
• #4247: nbconvert –to latex: error when converting greek letter
• #4273: %%capture not capturing rich objects like plots (IPython 1.1.0)
• #3866: Vertical offsets in LaTeX output for nbconvert
• #3631: xkcd mode for the IPython notebook
• #4243: Test exclusions not working on Windows
• #4256: IPython no longer handles unicode file names
• #3656: Audio displayobject
• #4223: Double output on Ctrl-enter-enter
• #4184: nbconvert: use r pygmentize backend when highlighting “%%R” cells
• #3851: Adds an explicit newline for pretty-printing.
• #3622: Drop fakemodule
• #4122: Nbconvert [windows]: Inconsistent line endings in markdown cells exported to latex
• #3819: nbconvert add extra blank line to code block on Windows.
• #4203: remove spurious print statement from parallel annotated functions
• #4200: Notebook: merging a heading cell and markdown cell cannot be undone
• #3747: ipynb -> ipynb transformer
• #4024: nbconvert markdown issues
• #3903: on Windows, ‘ipython3 nbconvert “C:/blabla/first_trial.ipynb” –to slides’ gives an unexpected result, and ‘–post serve’ fails
• #4095: Catch js error in append html in stream/pyerr
• #1880: Add parallelism to test_pr
• #4085: nbconvert: Fix sphinx preprocessor date format string for Windows
• #4156: Specifying –gui=tk at the command line
• #4146: Having to prepend ‘files/’ to markdown image paths is confusing
• #3818: nbconvert can’t handle Heading with Chinese characters on Japanese Windows OS.
• #4134: multi-line parser fails on ‘”’ in comment, qtconsole and notebook.
• #3998: sample custom.js needs to be updated
• #4078: StoreMagic.autorestore not working in 1.0.0
• #3990: Builtin input doesn’t work over zmq
• #4015: nbconvert fails to convert all the content of a notebook
• #4059: Issues with Ellipsis literal in Python 3
• #2310: “ZMQError: Interrupted system call” from RichIPythonWidget
• #3807: qtconsole ipython 0.13.2 - html/xhtml export fails
• #4103: Wrong default argument of DirectView.clear
• #4100: parallel.client.client references undefined error.EngineError
• #484: Drop nosepatch
• #3350: Added longlist support in ipdb.
• #1591: Keying ‘q’ doesn’t quit the interactive help in Wins7
• #40: The tests in test_process fail under Windows
• #3744: capture rich output as well as stdout/err in capture_output
• #3742: %%capture to grab rich display outputs
• #3863: Added working speaker notes for slides.
• #4013: Iptest fails in dual python installation
• #4005: IPython.start_kernel doesn’t work.
• #4020: IPython parallel map fails on numpy arrays
• #3914: nbconvert: Transformer tests
• #3923: nbconvert: Writer tests
• #3945: nbconvert: commandline tests fail Win7x64 Py3.3
• #3937: make tab visible in codemirror and light red background
• #3935: No feedback for mixed tabs and spaces
• #3933: nbconvert: Post-processor tests
• #3977: unable to complete remote connections for two-process
• #3939: minor checkpoint cleanup
• #3955: complete on % for magic in notebook
• #3954: all magics should be listed when completing on %
• #3980: nbconvert rst output lacks needed blank lines
• #3968: TypeError: super() argument 1 must be type, not classobj (Python 2.6.6)
• #3880: nbconvert: R&D remaining tests
• #2440: IPEP 4: Python 3 Compatibility

2.9 1.0 Series

2.9.1 Release 1.0.0: An Afternoon Hack

IPython 1.0 requires Python 2.6.5 or 3.2.1. It does not support Python 3.0, 3.1, or 2.5.

This is a big release. The principal milestone is the addition of IPython.nbconvert, but there has been a great deal of work improving all parts of IPython as well.
The previous version (0.13) was released on June 30, 2012, and in this development cycle we had:

- ~12 months of work.
- ~700 pull requests merged.
- ~600 issues closed (non-pull requests).
- contributions from ~150 authors.
- ~4000 commits.

The amount of work included in this release is so large that we can only cover here the main highlights; please see our detailed release statistics for links to every issue and pull request closed on GitHub as well as a full list of individual contributors. It includes

**Reorganization**

There have been two major reorganizations in IPython 1.0:

- Added IPython.kernel for all kernel-related code. This means that IPython.zmq has been removed, and much of it is now in IPython.kernel.zmq, some of it being in the top-level IPython.kernel.

- We have removed the frontend subpackage, as it caused unnecessary depth. So what was IPython.frontend.qt is now IPython.qt, and so on. The one difference is that the notebook has been further flattened, so that IPython.frontend.html.notebook is now just IPython.html. There is a shim module, so IPython.frontend is still importable in 1.0, but there will be a warning.

- The IPython sphinx directives are now installed in IPython.sphinx, so they can be imported by other projects.

**Public APIs**

For the first time since 0.10 (sorry, everyone), there is an official public API for starting IPython:

```python
from IPython import start_ipython
start_ipython()
```

This is what packages should use that start their own IPython session, but don’t actually want embedded IPython (most cases). IPython.embed() is used for embedding IPython into the calling namespace, similar to calling Pdb.set_trace(), whereas start_ipython() will start a plain IPython session, loading config and startup files as normal.

We also have added:

```python
from IPython import get_ipython
```

Which is a library function for getting the current IPython instance, and will return None if no IPython instance is running. This is the official way to check whether your code is called from inside an IPython session. If you want to check for IPython without unnecessarily importing IPython, use this function:
```python
def get_ipython():
    """return IPython instance if there is one, None otherwise""
    import sys
    if "IPython" in sys.modules:
        import IPython
        return IPython.get_ipython()
```

Core

- The input transformation framework has been reworked. This fixes some corner cases, and adds more flexibility for projects which use IPython, like SymPy & SAGE. For more details, see Custom input transformation.
- Exception types can now be displayed with a custom traceback, by defining a __render_traceback__() method which returns a list of strings, each containing one line of the traceback.
- A new command, ipython history trim can be used to delete everything but the last 1000 entries in the history database.
- __file__ is defined in both config files at load time, and .ipy files executed with %run.
- %logstart and %logappend are no longer broken.
- Add glob expansion for %run, e.g. %run -g script.py *.txt.
- Expand variables ($foo) in Cell Magic argument line.
- By default, iptest will exclude various slow tests. All tests can be run with iptest --all.
- SQLite history can be disabled in the various cases that it does not behave well.
- %edit works on interactively defined variables.
- editor hooks have been restored from quarantine, enabling TextMate as editor, etc.
- The env variable PYTHONSTARTUP is respected by IPython.
- The %matplotlib magic was added, which is like the old %pylab magic, but it does not import anything to the interactive namespace. It is recommended that users switch to %matplotlib and explicit imports.
- The --matplotlib command line flag was also added. It invokes the new %matplotlib magic and can be used in the same way as the old --pylab flag. You can either use it by itself as a flag (--matplotlib), or you can also pass a backend explicitly (--matplotlib qt or --matplotlib=wx, etc).

Backwards incompatible changes

- Calling InteractiveShell.prefilter() will no longer perform static transformations - the processing of escaped commands such as %magic and !system, and stripping input prompts from code blocks. This functionality was duplicated in IPython.core.inputsplitter, and the

2.9. 1.0 Series
latter version was already what IPython relied on. A new API to transform input will be ready before release.

- Functions from `IPython.lib.inputhook` to control integration with GUI event loops are no longer exposed in the top level of `IPython.lib`. Code calling these should make sure to import them from `IPython.lib.inputhook`.

- For all kernel managers, the `sub_channel` attribute has been renamed to `iopub_channel`.

- Users on Python versions before 2.6.6, 2.7.1 or 3.2 will now need to call `IPython.utils.doctestreload.doctest_reload()` to make doctests run correctly inside IPython. Python releases since those versions are unaffected. For details, see PR #3068 and Python issue 8048.

- The `InteractiveShell.cache_main_mod()` method has been removed, and `new_main_mod()` has a different signature, expecting a filename where earlier versions expected a namespace. See PR #3555 for details.

- The short-lived plugin system has been removed. Extensions are the way to go.

### NbConvert

The major milestone for IPython 1.0 is the addition of `IPython.nbconvert` - tools for converting IPython notebooks to various other formats.

**Warning:** nbconvert is α-level preview code in 1.0

To use nbconvert to convert various file formats:

```bash
ipython nbconvert --to html *.ipynb
```

See `ipython nbconvert --help` for more information. nbconvert depends on pandoc for many of the translations to and from various formats.

**See also:**
nbconvert

### Notebook

Major changes to the IPython Notebook in 1.0:

- The notebook is now autosaved, by default at an interval of two minutes. When you press ‘save’ or Ctrl-S, a checkpoint is made, in a hidden folder. This checkpoint can be restored, so that the autosave model is strictly safer than traditional save. If you change nothing about your save habits, you will always have a checkpoint that you have written, and an autosaved file that is kept up to date.

- The notebook supports `raw_input()` / `input()`, and thus also `%debug`, and many other Python calls that expect user input.

- You can load custom javascript and CSS in the notebook by editing the files `$({ipython locate profile})/static/custom/custom.js, css.`
• Add `%%html`, `%%svg`, `%%javascript`, and `%%latex` cell magics for writing raw output in notebook cells.

• add a redirect handler and anchors on heading cells, so you can link across notebooks, directly to heading cells in other notebooks.

• Images support width and height metadata, and thereby 2x scaling (retina support).

• `_repr_foo_` methods can return a tuple of (data, metadata), where metadata is a dict containing metadata about the displayed object. This is used to set size, etc. for retina graphics. To enable retina matplotlib figures, simply set `InlineBackend.figure_format = 'retina'` for 2x PNG figures, in your `IPython config file` or via the `%config` magic.

• Add `display.FileLink` and `FileLinks` for quickly displaying HTML links to local files.

• Cells have metadata, which can be edited via cell toolbars. This metadata can be used by external code (e.g. reveal.js or exporters), when examining the notebook.

• Fix an issue parsing LaTeX in markdown cells, which required users to type `\\`, instead of `\`

• Notebook templates are rendered with Jinja instead of Tornado.

• `%%file` has been renamed `%writefile` (`%file` is deprecated).

• ANSI (and VT100) color parsing has been improved in both performance and supported values.

• The static files path can be found as `IPython.html.DEFAULT_STATIC_FILES_PATH`, which may be changed by package managers.

• IPython’s CSS is installed in `static/css/style.min.css` (all style, including bootstrap), and `static/css/ipython.min.css`, which only has IPython’s own CSS. The latter file should be useful for embedding IPython notebooks in other pages, blogs, etc.

• The Print View has been removed. Users are encouraged to test `ipython nbconvert` to generate a static view.

### Javascript Components

The javascript components used in the notebook have been updated significantly.

• updates to jQuery (2.0) and jQueryUI (1.10)

• Update CodeMirror to 3.14

• Twitter Bootstrap (2.3) for layout

• Font-Awesome (3.1) for icons

• highlight.js (7.3) for syntax highlighting

• marked (0.2.8) for markdown rendering

• require.js (2.1) for loading javascript

Some relevant changes that are results of this:
• Markdown cells now support GitHub-flavored Markdown (GFM), which includes ```python code blocks and tables.
• Notebook UI behaves better on more screen sizes.
• Various code cell input issues have been fixed.

**Kernel**

The kernel code has been substantially reorganized.

New features in the kernel:

• Kernels support ZeroMQ IPC transport, not just TCP
• The message protocol has added a top-level metadata field, used for information about messages.
• Add a `data_pub` message that functions much like `display_pub`, but publishes raw (usually pickled) data, rather than representations.
• Ensure that `sys.stdout.encoding` is defined in Kernels.
• Stdout from forked subprocesses should be forwarded to frontends (instead of crashing).

**IPEP 13**

The KernelManager has been split into a `KernelManager` and a `KernelClient`. The Manager owns a kernel and starts / signals / restarts it. There is always zero or one KernelManager per Kernel. Clients communicate with Kernels via zmq channels, and there can be zero-to-many Clients connected to a Kernel at any given time.

The KernelManager now automatically restarts the kernel when it dies, rather than requiring user input at the notebook or QtConsole UI (which may or may not exist at restart time).

**In-process kernels**

The Python-language frontends, particularly the Qt console, may now communicate with in-process kernels, in addition to the traditional out-of-process kernels. An in-process kernel permits direct access to the kernel namespace, which is necessary in some applications. It should be understood, however, that the in-process kernel is not robust to bad user input and will block the main (GUI) thread while executing. Developers must decide on a case-by-case basis whether this tradeoff is appropriate for their application.

**Parallel**

IPython.parallel has had some refactoring as well. There are many improvements and fixes, but these are the major changes:

• Connections have been simplified. All ports and the serialization in use are written to the connection file, rather than the initial two-stage system.
• Serialization has been rewritten, fixing many bugs and dramatically improving performance serializing large containers.

• Load-balancing scheduler performance with large numbers of tasks has been dramatically improved.

• There should be fewer (hopefully zero) false-positives for engine failures.

• Increased compatibility with various use cases that produced serialization / argument errors with map, etc.

• The controller can attempt to resume operation if it has crashed, by passing `ipcontroller --restore`.

• Engines can monitor the Hub heartbeat, and shutdown if the Hub disappears for too long.

• add HTCondor support in launchers

QtConsole

Various fixes, including improved performance with lots of text output, and better drag and drop support. The initial window size of the qtconsole is now configurable via `IPythonWidget.width` and `IPythonWidget.height`.

2.10 Issues closed in the 1.0 development cycle

2.10.1 Issues closed in 1.1

GitHub stats for 2013/08/08 - 2013/09/09 (since 1.0)

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 25 authors contributed 337 commits.

• Benjamin Ragan-Kelley
• Bing Xia
• Bradley M. Froehle
• Brian E. Granger
• Damián Avila
• dhirschfeld
• Dražen Lučanin
• gmbecker
• Jake Vanderplas
• Jason Grout
• Jonathan Frederic
We closed a total of 76 issues, 58 pull requests and 18 regular issues; this is the full list (generated with the script tools/github_stats.py):

Pull Requests (58):

- PR #4188: Allow user_ns trait to be None
- PR #4189: always fire LOCAL_IPS.extend(PUBLIC_IPS)
- PR #4174: various issues in markdown and rst templates
- PR #4178: add missing data_javascript
- PR #4181: nbconvert: Fix, sphinx template not removing new lines from headers
- PR #4043: don’t ‘restore_bytes’ in from_JSON
- PR #4163: Fix for incorrect default encoding on Windows.
- PR #4136: catch javascript errors in any output
- PR #4171: add nbconvert config file when creating profiles
- PR #4125: Basic exercise of ipython [subcommand] -h and help-all
- PR #4085: nbconvert: Fix sphinx preprocessor date format string for Windows
- PR #4159: don’t split .cell and div.cell CSS
- PR #4158: generate choices for --gui configurable from real mapping
- PR #4065: do not include specific css in embedable one
- PR #4092: nbconvert: Fix for unicode html headers, Windows + Python 2.x
• PR #4074: close Client sockets if connection fails
• PR #4064: Store default codemirror mode in only 1 place
• PR #4104: Add way to install MathJax to a particular profile
• PR #4144: help_end transformer shouldn’t pick up ? in multiline string
• PR #4143: update example custom.js
• PR #4142: DOC: unwrap openssl line in public_server doc
• PR #4141: add files with a separate add call in backport_pr
• PR #4137: Restore autorestore option for storemagic
• PR #4098: pass profile-dir instead of profile name to Kernel
• PR #4120: support input in Python 2 kernels
• PR #4088: nbconvert: Fix coalescestreams line with incorrect nesting causing strange behavior
• PR #4060: only strip continuation prompts if regular prompts seen first
• PR #4132: Fixed name error bug in function safe_unicode in module py3compat.
• PR #4121: move test_kernel from IPython.zmq to IPython.kernel
• PR #4118: ZMQ heartbeat channel: catch EINTR exceptions and continue.
• PR #4054: use unicode for HTML export
• PR #4106: fix a couple of default block values
• PR #4115: Update docs on declaring a magic function
• PR #4101: restore accidentally removed EngineError
• PR #4096: minor docs changes
• PR #4056: respect pylab_import_all when --pylab specified at the command-line
• PR #4091: Make Qt console banner configurable
• PR #4086: fix missing errno import
• PR #4030: exclude .git in MANIFEST.in
• PR #4047: Use istype() when checking if canned object is a dict
• PR #4031: don’t close_fds on Windows
• PR #4029: bson.Binary moved
• PR #4035: Fixed custom jinja2 templates being ignored when setting template_path
• PR #4026: small doc fix in nbconvert
• PR #4016: Fix IPython.start_* functions
• PR #4021: Fix parallel.client.View map() on numpy arrays
• PR #4022: DOC: fix links to matplotlib, notebook docs

2.10. Issues closed in the 1.0 development cycle
• PR #4018: Fix warning when running IPython.kernel tests
• PR #4019: Test skipping without unicode paths
• PR #4008: Transform code before %prun/%%prun runs
• PR #4014: Fix typo in ipapp
• PR #3987: get files list in backport_pr
• PR #3974: nbconvert: Fix app tests on Window7 w/ Python 3.3
• PR #3978: fix --existing with non-localhost IP
• PR #3939: minor checkpoint cleanup
• PR #3981: BF: fix nbconvert rst input prompt spacing
• PR #3960: Don’t make sphinx a dependency for importing nbconvert
• PR #3973: logging.Formatter is not new-style in 2.6

Issues (18):
• #4024: nbconvert markdown issues
• #4095: Catch js error in append html in stream/pyerr
• #4156: Specifying –gui=tk at the command line
• #3818: nbconvert can’t handle Heading with Chinese characters on Japanese Windows OS.
• #4134: multi-line parser fails on “” in comment, qtconsole and notebook.
• #3998: sample custom.js needs to be updated
• #4078: StoreMagic.autorestore not working in 1.0.0
• #3990: Buitlin input doesn’t work over zmq
• #4015: nbconvert fails to convert all the content of a notebook
• #4059: Issues with Ellipsis literal in Python 3
• #4103: Wrong default argument of DirectView.clear
• #4100: parallel.client.client references undefined error.EngineError
• #4005: IPython.start_kernel doesn’t work.
• #4020: IPython parallel map fails on numpy arrays
• #3945: nbconvert: commandline tests fail Win7x64 Py3.3
• #3977: unable to complete remote connections for two-process
• #3980: nbconvert rst output lacks needed blank lines
• #3968: TypeError: super() argument 1 must be type, not classobj (Python 2.6.6)
2.10.2 Issues closed in 1.0

GitHub stats for 2012/06/30 - 2013/08/08 (since 0.13)

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 155 authors contributed 4258 commits.

- Aaron Meurer
- Adam Davis
- Ahmet Bakan
- Alberto Valverde
- Allen Riddell
- Anders Hovmöller
- Andrea Bedini
- Andrew Spiers
- Andrew Vandever
- Anthony Scopatz
- Anton Akhmerov
- Anton I. Sipos
- Antony Lee
- Aron Ahmadia
- Benedikt Sauer
- Benjamin Jones
- Benjamin Ragan-Kelley
- Benjie Chen
- Boris de Laage
- Brad Reisfeld
- Bradley M. Froehle
- Brian E. Granger
- Cameron Bates
- Cavendish McKay
- chapmanb
- Chris Beaumont
- Chris Laumann
• Christoph Gohlke
• codebraker
• codespaced
• Corran Webster
• DamianHeard
• Damián Avila
• Dan Kilman
• Dan McDougall
• Danny Staple
• David Hirschfeld
• David P. Sanders
• David Warde-Farley
• David Wolever
• David Wyde
• debjan
• Diane Trout
• dkua
• Dominik Dabrowski
• Donald Curtis
• Dražen Lučanin
• drevicko
• Eric O. LEBIGOT
• Erik M. Bray
• Erik Tollerud
• Eugene Van den Bulke
• Evan Patterson
• Fernando Perez
• Francesco Montesano
• Frank Murphy
• Greg Caporaso
• Guy Haskin Fernald
• guziy
• Hans Meine
• Harry Moreno
• henryiii
• Ivan Djokic
• Jack Feser
• Jake Vanderplas
• jakobgager
• James Booth
• Jan Schulz
• Jason Grout
• Jeff Knisley
• Jens Hedegaard Nielsen
• jeremiahbuddha
• Jerry Fowler
• Jessica B. Hamrick
• Jez Ng
• John Zwinck
• Jonathan Frederic
• Jonathan Taylor
• Joon Ro
• Joseph Lansdowne
• Juergen Hasch
• Julian Taylor
• Jussi Sainio
• Jörgen Stenarson
• kevin
• klonuo
• Konrad Hinsen
• Kyle Kelley
• Lars Solberg
• Lessandro Mariano
• Mark Sienkiewicz at STScI
• Martijn Vermaat
• Martin Spacek
• Matthias Bussonnier
• Maxim Grechkin
• Maximilian Albert
• MercuryRising
• Michael Droettboom
• Michael Shuffett
• Michał Górny
• Mikhail Korobov
• mr.Shu
• Nathan Goldbaum
• ocefpaf
• Ohad Ravid
• Olivier Grisel
• Olivier Verdier
• Owen Healy
• Pankaj Pandey
• Paul Ivanov
• Pawel Jasinski
• Pietro Berkes
• Piti Ongmongkolkul
• Puneeth Chaganti
• Rich Wareham
• Richard Everson
• Rick Lupton
• Rob Young
• Robert Kern
• Robert Marchman
• Robert McGibbon
• Rui Pereira
• Rustam Safin
• Ryan May
• s8weber
• Samuel Ainsworth
• Sean Vig
• Siyu Zhang
• Skylar Saveland
• slojo404
• smithj1
• Stefan Karpinski
• Stefan van der Walt
• Steven Silvester
• Takafumi Arakaki
• Takeshi Kanmae
• tcmulcahy
• teegaar
• Thomas Kluyver
• Thomas Robitaille
• Thomas Spura
• Thomas Weißschuh
• Timothy O’Donnell
• Tom Dimiduk
• ugrurthemaster
• urielshaolin
• v923z
• Valentin Haenel
• Victor Zverovich
• 23. Trevor King
• y-p
• Yoav Ram
• Zbigniew Jędrzejewski-Szmek
• Zoltán Vörös
We closed a total of 1484 issues, 793 pull requests and 691 regular issues; this is the full list (generated with the script tools/github_stats.py):

Pull Requests (793):

- PR #3958: doc update
- PR #3965: Fix ansi color code for background yellow
- PR #3964: Fix casing of message.
- PR #3942: Pass on install docs
- PR #3962: exclude IPython.lib.kernel in iptest
- PR #3961: Longpath test fix
- PR #3905: Remove references to 0.11 and 0.12 from config/overview.rst
- PR #3951: nbconvert: fixed latex characters not escaped properly in nbconvert
- PR #3949: log fatal error when PDF conversion fails
- PR #3947: nbconvert: Make writer & post-processor aliases case insensitive.
- PR #3938: Recompile css.
- PR #3948: sphinx and PDF tweaks
- PR #3943: nbconvert: Serve post-processor Windows fix
- PR #3934: nbconvert: fix logic of verbose flag in PDF post processor
- PR #3929: swallow enter event in rename dialog
- PR #3924: nbconvert: Backport fixes
- PR #3925: Replace –pylab flag with –matplotlib in usage
- PR #3910: Added explicit error message for missing configuration arguments.
- PR #3913: grffile to support spaces in notebook names
- PR #3918: added check_for_tornado, closes #3916
- PR #3917: change docs/examples refs to be just examples
- PR #3908: what’s new tweaks
- PR #3896: two column quickhelp dialog, closes #3895
- PR #3911: explicitly load python mode before IPython mode
- PR #3901: don’t force . relative path, fix #3897
- PR #3891: fix #3889
- PR #3892: Fix documentation of Kernel.stop_channels
- PR #3888: posixify paths for Windows latex
- PR #3882: quick fix for #3881
- PR #3877: don’t use shell=True in PDF export
- PR #3878: minor template loading cleanup
- PR #3855: nbconvert: Filter tests
- PR #3879: finish 3870
- PR #3870: Fix for converting notebooks that contain unicode characters.
- PR #3876: Update parallel_winhpc.rst
- PR #3872: removing vim-ipython, since it has it’s own repo
- PR #3871: updating docs
- PR #3873: remove old examples
- PR #3868: update CodeMirror component to 3.15
- PR #3865: Escape filename for pdflatex in nbconvert
- PR #3861: remove old external.js
- PR #3864: add keyboard shortcut to docs
- PR #3834: This PR fixes a few issues with nbconvert tests
- PR #3840: prevent profile_dir from being undefined
- PR #3859: Add “An Afternoon Hack” to docs
- PR #3854: Catch errors filling readline history on startup
- PR #3857: Delete extra auto
- PR #3845: nbconvert: Serve from original build directory
- PR #3846: Add basic logging to nbconvert
- PR #3850: add missing store_history key to Notebook execute_requests
- PR #3844: update payload source
- PR #3830: mention metadata / display_data similarity in pyout spec
- PR #3848: fix incorrect empty-docstring
- PR #3836: Parse markdown correctly when mathjax is disabled
- PR #3849: skip a failing test on windows
- PR #3828: signature_scheme lives in Session
- PR #3831: update nbconvert doc with new CLI
- PR #3822: add output flag to nbconvert
- PR #3780: Added serving the output directory if html-based format are selected.
- PR #3764: Cleanup nbconvert templates
- PR #3829: remove now-duplicate ‘this is dev’ note
• PR #3814: add `ConsoleWidget.execute_on_complete_input` flag
• PR #3826: try rtfd
• PR #3821: add sphinx prolog
• PR #3817: relax timeouts in terminal console and tests
• PR #3825: fix more tests that fail when pandoc is missing
• PR #3824: don’t set target on internal markdown links
• PR #3816: `s/pylab/matplotlib` in docs
• PR #3812: Describe differences between `start_ipython` and `embed`
• PR #3805: Print View has been removed
• PR #3820: Make it clear that 1.0 is not released yet
• PR #3784: nbconvert: Export flavors & PDF writer (ipy dev meeting)
• PR #3800: semantic-versionify version number for non-releases
• PR #3802: Documentation .txt to .rst
• PR #3765: cleanup terminal console iopub handling
• PR #3720: Fix for #3719
• PR #3787: re-raise `KeyboardInterrupt` in raw_input
• PR #3770: Organizing reveal’s templates.
• PR #3751: Use `link(2)` when possible in nbconvert
• PR #3792: skip tests that require pandoc
• PR #3782: add Importing Notebooks example
• PR #3752: nbconvert: Add `cwd` to `sys.path`
• PR #3789: fix raw_input in qtconsole
• PR #3756: document the wire protocol
• PR #3749: convert IPython syntax to Python syntax in nbconvert python template
• PR #3793: Closes #3788
• PR #3794: Change logo link to ipython.org
• PR #3746: Raise a named exception when pandoc is missing
• PR #3781: comply with the message spec in the notebook
• PR #3779: remove bad `if logged_in` preventing new-notebook without login
• PR #3743: remove notebook read-only view
• PR #3732: add delay to autosave in beforeunload
• PR #3761: Added rm_math_space to markdown cells in the basichtml.tpl to be rendered ok by mathjax after the nbconversion.

• PR #3758: nbconvert: Filter names cleanup
• PR #3769: Add configurability to tabcompletion timeout
• PR #3771: Update px pylab test to match new output of pylab
• PR #3741: better message when notebook format is not supported
• PR #3753: document Ctrl-C not working in ipython kernel
• PR #3766: handle empty metadata in pyout messages more gracefully.
• PR #3736: my attempt to fix #3735
• PR #3759: nbconvert: Provide a more useful error for invalid use case.
• PR #3760: nbconvert: Allow notebook filenames without their extensions
• PR #3750: nbconvert: Add cwd to default templates search path.
• PR #3748: Update nbconvert docs
• PR #3734: Nbconvert: Export extracted files into nbname_files subdirectory
• PR #3733: Nicer message when pandoc is missing, closes #3730
• PR #3722: fix two failing test in IPython.lib
• PR #3704: Start what’s new for 1.0
• PR #3705: Complete rewrite of IPython Notebook documentation: docs/source/interactive/htmlnotebook.txt
• PR #3709: Docs cleanup
• PR #3716: raw_input fixes for kernel restarts
• PR #3683: use %matplotlib in example notebooks
• PR #3686: remove quarantine
• PR #3699: svg2pdf unicode fix
• PR #3695: fix SVG2PDF
• PR #3685: fix Pager.detach
• PR #3675: document new dependencies
• PR #3690: Fixing some css minors in full_html and reveal.
• PR #3671: nbconvert tests
• PR #3692: Fix rename notebook - show error with invalid name
• PR #3409: Prevent qtconsole frontend freeze on lots of output.
• PR #3660: refocus active cell on dialog close
• PR #3598: Statelessify mathjaxutils

2.10. Issues closed in the 1.0 development cycle
• PR #3673: enable comment/uncomment selection
• PR #3677: remove special-case in get_home_dir for frozen dists
• PR #3674: add CONTRIBUTING.md
• PR #3670: use Popen command list for ipexec
• PR #3568: pylab import adjustments
• PR #3559: add create.Cell and delete.Cell js events
• PR #3606: push cell magic to the head of the transformer line
• PR #3607: NbConvert: Writers, No YAML, and stuff...
• PR #3665: Pywin32 skips
• PR #3669: set default client_class for QtKernelManager
• PR #3662: add strip_encoding_cookie transformer
• PR #3641: increase patience for slow kernel startup in tests
• PR #3651: remove a bunch of unused default_config_file assignments
• PR #3630: CSS adjustments
• PR #3645: Don’t require HistoryManager to have a shell
• PR #3643: don’t assume tested ipython is on the PATH
• PR #3654: fix single-result AsyncResults
• PR #3601: Markdown in heading cells (take 2)
• PR #3652: Remove old docs/examples
• PR #3621: catch any exception appending output
• PR #3585: don’t blacklist builtin names
• PR #3647: Fix frontend deprecation warnings in several examples
• PR #3649: fixAsyncResult.get_dict for single result
• PR #3648: Fix store magic test
• PR #3650: Fix, config_file_name was ignored
• PR #3640: Gcf.get_active() can return None
• PR #3571: Added shortcuts to split cell, merge cell above and merge cell below.
• PR #3635: Added missing slash to print-pdf call.
• PR #3487: Drop patch for compatibility with pyreadline 1.5
• PR #3338: Allow filename with extension in find_cmd in Windows.
• PR #3628: Fix test for Python 3 on Windows.
• PR #3642: Fix typo in docs
• PR #3627: use DEFAULT_STATIC_FILES_PATH in a test instead of package dir
• PR #3624: fix some unicode in zmqhandlers
• PR #3460: Set calling program to UNKNOWN, when argv not in sys
• PR #3632: Set calling program to UNKNOWN, when argv not in sys (take #2)
• PR #3629: Use new entry point for python -m IPython
• PR #3626: passing cell to showInPager, closes #3625
• PR #3618: expand terminal color support
• PR #3623: raise UsageError for unsupported GUI backends
• PR #3071: Add magic function %drun to run code in debugger
• PR #3608: a nicer error message when using %pylab magic
• PR #3592: add extra_config_file
• PR #3612: updated .mailmap
• PR #3616: Add examples for interactive use of MPI.
• PR #3615: fix regular expression for ANSI escapes
• PR #3586: Corrected a typo in the format string for strftime the sphinx.py transformer of nbconvert
• PR #3611: check for markdown no longer needed, closes #3610
• PR #3555: Simplify caching of modules with %run
• PR #3583: notebook small things
• PR #3594: Fix duplicate completion in notebook
• PR #3600: parallel: Improved logging for errors during BatchSystemLauncher.stop
• PR #3595: Revert “allow markdown in heading cells”
• PR #3538: add IPython.start_ipython
• PR #3562: Allow custom nbconvert template loaders
• PR #3582: pandoc adjustments
• PR #3560: Remove max_msg_size
• PR #3591: Refer to Setuptools instead of Distribute
• PR #3590: IPython.sphinxext needs an __init__.py
• PR #3581: Added the possibility to read a custom.css file for tweaking the final html in full_html and reveal templates.
• PR #3576: Added support for markdown in heading cells when they are nbconverted.
• PR #3575: tweak run -d message to ‘continue execution’
• PR #3569: add PYTHONSTARTUP to startup files
• PR #3567: Trigger a single event on js app initialized
• PR #3565: style.min.css shoudl always exist...
• PR #3531: allow markdown in heading cells
• PR #3577: Simplify codemirror ipython-mode
• PR #3495: Simplified regexp, and suggestions for clearer regexps.
• PR #3578: Use adjustbox to specify figure size in nbconvert -> latex
• PR #3572: Skip import irunner test on Windows.
• PR #3574: correct static path for CM modes autoload
• PR #3558: Add IPython.sphinxext
• PR #3561: mention double-control-C to stop notebook server
• PR #3566: fix event names
• PR #3564: Remove trivial nbconvert example
• PR #3540: allow cython cache dir to be deleted
• PR #3527: cleanup stale, unused exceptions in parallel.error
• PR #3529: ensure raw_input returns str in zmq shell
• PR #3541: respect image size metadata in qtconsole
• PR #3550: Fixing issue preventing the correct read of images by full_html and reveal exporters.
• PR #3557: open markdown links in new tabs
• PR #3556: remove mention of nonexistent _marqv in macro
• PR #3552: set overflow-x: hidden on Firefox only
• PR #3554: Fix missing import os in latex exporter.
• PR #3546: Don’t hardcode latex posix paths in nbconvert
• PR #3551: fix path prefix in nbconvert
• PR #3533: Use a CDN to get reveal.js library.
• PR #3498: When a notebook is written to file, name the metadata name u”
• PR #3548: Change to standard save icon in Notebook toolbar
• PR #3539: Don’t hardcode posix paths in nbconvert
• PR #3508: notebook supports raw_input and %debug now
• PR #3526: ensure ‘default’ is first in cluster profile list
• PR #3525: basic timezone info
• PR #3532: include nbconvert templates in installation
• PR #3515: update CodeMirror component to 3.14
• PR #3513: add ‘No Checkpoints’ to Revert menu
• PR #3536: format positions are required in Python 2.6.x
• PR #3521: Nbconvert fix, silent fail if template doesn’t exist
• PR #3530: update %store magic docstring
• PR #3528: fix local mathjax with custom base_project_url
• PR #3518: Clear up unused imports
• PR #3506: %store -r restores saved aliases and directory history, as well as variables
• PR #3516: make css highlight style configurable
• PR #3523: Exclude frontend shim from docs build
• PR #3514: use bootstrap disabled instead of ui-state-disabled
• PR #3520: Added relative import of RevealExporter to __init__.py inside exporters module
• PR #3507: fix HTML capitalization in nbconvert exporter classes
• PR #3512: fix nbconvert filter validation
• PR #3511: Get Tracer working after ipapi.get replaced with get_ipython
• PR #3510: use window.onbeforeunload= for nav-away warning
• PR #3504: don’t use parent=self in handlers
• PR #3500: Merge nbconvert into IPython
• PR #3478: restore “unsaved changes” warning on unload
• PR #3493: add a dialog when the kernel is auto-restarted
• PR #3488: Add test suite for autoreload extension
• PR #3484: Catch some pathological cases inside oinspect
• PR #3481: Display R errors without Python traceback
• PR #3468: fix %magic output
• PR #3430: add parent to Configurable
• PR #3491: Remove unexpected keyword parameter to remove_kernel
• PR #3485: SymPy has changed its recommended way to initialize printing
• PR #3486: Add test for non-ascii characters in docstrings
• PR #3483: Inputtransformer: Allow classic prompts without space
• PR #3482: Use an absolute path to iptest, because the tests are not always run from $IPYTHONDIR.
• PR #3381: enable 2x (retina) display
• PR #3450: Flatten IPython.frontend
• PR #3477: pass config to subapps

2.10. Issues closed in the 1.0 development cycle 111
• PR #3466: Kernel fails to start when username has non-ascii characters
• PR #3465: Add HTCondor bindings to IPython.parallel
• PR #3463: fix typo, closes #3462
• PR #3456: Notice for users who disable javascript
• PR #3453: fix cell execution in firefox, closes #3447
• PR #3393: [WIP] bootstrapify
• PR #3440: Fix installing mathjax from downloaded file via command line
• PR #3431: Provide means for starting the Qt console maximized and with the menu bar hidden
• PR #3425: base IPClusterApp inherits from BaseIPythonApp
• PR #3433: Update IPythonexternalpath__init__.py
• PR #3298: Some fixes in IPython Sphinx directive
• PR #3428: process escapes in mathjax
• PR #3420: thank -> thanks
• PR #3416: Fix doc: “principle” not “principal”
• PR #3413: more unique filename for test
• PR #3364: Inject requirejs in notebook and start using it.
• PR #3390: Fix %paste with blank lines
• PR #3403: fix creating config objects from dicts
• PR #3401: rollback #3358
• PR #3373: make cookie_secret configurable
• PR #3307: switch default ws_url logic to js side
• PR #3392: Restore anchor link on h2-h6
• PR #3369: Use different threshold for (auto)scroll in output
• PR #3370: normalize unicode notebook filenames
• PR #3372: base default cookie name on request host+port
• PR #3378: disable CodeMirror drag/drop on Safari
• PR #3358: workaround spurious CodeMirror scrollbars
• PR #3371: make setting the notebook dirty flag an event
• PR #3366: remove long-dead zmq frontend.py and completer.py
• PR #3382: cull Session digest history
• PR #3330: Fix get_ipython_dir when $HOME is /
• PR #3319: IPEP 13: user-expressions and user-variables
• PR #3384: comments in tools/gitwash_dumper.py changed (‘” to “”)
• PR #3387: Make submodule checks work under Python 3.
• PR #3357: move anchor-link off of heading text
• PR #3351: start basic tests of ipcluster Launchers
• PR #3377: allow class.__module__ to be None
• PR #3340: skip submodule check in package managers
• PR #3328: decode subprocess output in launchers
• PR #3368: Reenable bracket matching
• PR #3356: Mpr fixes
• PR #3336: Use new input transformation API in %time magic
• PR #3325: Organize the JS and less files by component.
• PR #3342: fix test_find_cmd_python
• PR #3354: catch socket.error in utils.localinterfaces
• PR #3341: fix default cluster count
• PR #3286: don’t use get_ipython from builtins in library code
• PR #3333: notebookapp: add missing whitespace to warnings
• PR #3323: Strip prompts even if the prompt isn’t present on the first line.
• PR #3321: Reorganize the python/server side of the notebook
• PR #3320: define __file__ in config files
• PR #3317: rename %%file to %%writefile
• PR #3304: set unlimited HWM for all relay devices
• PR #3315: Update Sympy_printing extension load
• PR #3310: further clarify Image docstring
• PR #3285: load extensions in builtin trap
• PR #3308: Speed upAsyncResult._wait_for_outputs(0)
• PR #3294: fix callbacks as optional in js kernel.execute
• PR #3276: Fix: “python ABS/PATH/TO/ipython.py” fails
• PR #3301: allow python3 tests without python installed
• PR #3282: allow view.map to work with a few more things
• PR #3284: remove ipython.py entry point
• PR #3281: fix ignored IOPub messages with no parent
• PR #3275: improve submodule messages / git hooks

2.10. Issues closed in the 1.0 development cycle
• PR #3239: Allow “x” icon and esc key to close pager in notebook
• PR #3290: Improved heartbeat controller to engine monitoring for long running tasks
• PR #3142: Better error message when CWD doesn’t exist on startup
• PR #3066: Add support for relative import to %run -m (fixes #2727)
• PR #3269: protect highlight.js against unknown languages
• PR #3267: add missing return
• PR #3101: use marked / highlight.js instead of pagedown and prettify
• PR #3264: use https url for submodule
• PR #3263: fix set_last_checkpoint when no checkpoint
• PR #3258: Fix submodule location in setup.py
• PR #3254: fix a few URLs from previous PR
• PR #3240: remove js components from the repo
• PR #3158: IPEP 15: autosave the notebook
• PR #3252: move images out of _static folder into _images
• PR #3251: Fix for cell magics in Qt console
• PR #3250: Added a simple __html__() method to the HTML class
• PR #3249: remove copy of sphinx inheritance_diagram.py
• PR #3235: Remove the unused print notebook view
• PR #3238: Improve the design of the tab completion UI
• PR #3242: Make changes of Application.log_format effective
• PR #3219: Workaround so only one CTRL-C is required for a new prompt in –gui=qt
• PR #3190: allow formatters to specify metadata
• PR #3231: improve discovery of public IPs
• PR #3233: check prefixes for swallowing kernel args
• PR #3234: Removing old autogrow JS code.
• PR #3232: Update to CodeMirror 3 and start to ship our components
• PR #3229: The HTML output type accidentally got removed from the OutputArea.
• PR #3228: Typo in IPython.Parallel documentation
• PR #3226: Text in rename dialog was way too big - making it <p>.
• PR #3225: Removing old restuctured text handler and web service.
• PR #3222: make BlockingKernelClient the default Client
• PR #3223: add missing mathjax_url to new settings dict
• PR #3089: add stdin to the notebook
• PR #3221: Remove references to HTMLCell (dead code)
• PR #3205: add ignored *args to HasTraits constructor
• PR #3088: cleanup IPython handler settings
• PR #3201: use much faster regexp for ansi coloring
• PR #3220: avoid race condition in profile creation
• PR #3011: IPEP 12: add KernelClient
• PR #3217: informative error when trying to load directories
• PR #3174: Simple class
• PR #2979: CM configurable Take 2
• PR #3215: Updates storemagic extension to allow for specifying variable name to load
• PR #3181: backport If-Modified-Since fix from tornado
• PR #3200: IFrame (VimeoVideo, ScribdDocument, ...)
• PR #3186: Fix small inconsistency in nbconvert: etype -> ename
• PR #3212: Fix issue #2563, “core.profiledir.check_startup_dir() doesn’t work inside py2exe’d installation”
• PR #3211: Fix inheritance_diagram Sphinx extension for Sphinx 1.2
• PR #3208: Update link to extensions index
• PR #3203: Separate InputSplitter for transforming whole cells
• PR #3189: Improve completer
• PR #3194: finish up PR #3116
• PR #3188: Add new keycodes
• PR #2695: Key the root modules cache by sys.path entries.
• PR #3182: clarify %%file docstring
• PR #3163: BUG: Fix the set and frozenset pretty printer to handle the empty case correctly
• PR #3180: better UsageError for cell magic with no body
• PR #3184: Cython cache
• PR #3175: Added missing s
• PR #3173: Little bits of documentation cleanup
• PR #2635: Improve Windows start menu shortcuts (#2)
• PR #3172: Add missing import in IPython parallel magics example
• PR #3170: default application logger shouldn’t propagate

2.10. Issues closed in the 1.0 development cycle
• PR #3159: Autocompletion for zsh
• PR #3105: move DEFAULT_STATIC_FILES_PATH to IPython.html
• PR #3144: minor bower tweaks
• PR #3141: Default color output for ls on OSX
• PR #3137: fix dot syntax error in inheritance diagram
• PR #3072: raise UnsupportedOperation on iostream.fileno()
• PR #3147: Notebook support for a reverse proxy which handles SSL
• PR #3152: make qtconsole size at startup configurable
• PR #3162: adding stream kwarg to current.new_output
• PR #2981: IPEP 10: kernel side filtering of display formats
• PR #3058: add redirect handler for notebooks by name
• PR #3041: support non-modules in @require
• PR #2447: Stateful line transformers
• PR #3108: fix some O(N) and O(N^2) operations in parallel.map
• PR #2791: forward stdout from forked processes
• PR #3157: use Python 3-style for pretty-printed sets
• PR #3148: closes #3045, #3123 for tornado < version 3.0
• PR #3143: minor heading-link tweaks
• PR #3136: Strip useless ANSI escape codes in notebook
• PR #3126: Prevent errors when pressing arrow keys in an empty notebook
• PR #3135: quick dev installation instructions
• PR #2889: Push pandas dataframes to R magic
• PR #3068: Don’t monkeypatch doctest during IPython startup.
• PR #3133: fix argpase version check
• PR #3102: set spellcheck=false in CodeCell inputarea
• PR #3064: add anchors to heading cells
• PR #3097: PyQt 4.10: use self._document = self.document()
• PR #3117: propagate automagic change to shell
• PR #3118: don’t give up on weird os names
• PR #3115: Fix example
• PR #2640: fix quarantine/ipy_editors.py
• PR #3070: Add info make target that was missing in old Sphinx
• PR #3082: A few small patches to image handling
• PR #3078: fix regular expression for detecting links in stdout
• PR #3054: restore default behavior for automatic cluster size
• PR #3073: fix ipython usage text
• PR #3083: fix DisplayMagics.html docstring
• PR #3080: noted sub_channel being renamed to iopub_channel
• PR #3079: actually use IPKernelApp.kernel_class
• PR #3076: Improve notebook.js documentation
• PR #3063: add missing %html magic
• PR #3075: check for SIGUSR1 before using it, closes #3074
• PR #3051: add width:100% to vbox for webkit / FF consistency
• PR #2999: increase registration timeout
• PR #2997: fix DictDB default size limit
• PR #3033: on resume, print server info again
• PR #3062: test double pyximport
• PR #3046: cast kernel cwd to bytes on Python 2 on Windows
• PR #3038: remove xml from notebook magic docstrings
• PR #3032: fix time format to international time format
• PR #3022: Fix test for Windows
• PR #3024: changed instances of ‘outout’ to ‘output’ in alt texts
• PR #3013: py3 workaround for reload in cythonmagic
• PR #2961: time magic: shorten unnecessary output on windows
• PR #2987: fix local files examples in markdown
• PR #2998: fix css in .output_area pre
• PR #3003: add Sinclude /etc/inputrc to suggested ~/.inputrc
• PR #2957: Refactor qt import logic. Fixes #2955
• PR #2994: expanduser on %%-file targets
• PR #2983: fix run-all (that-> this)
• PR #2964: fix count when testing composite error output
• PR #2967: shows entire session history when only startsess is given
• PR #2942: Move CM IPython theme out of codemirror folder
• PR #2929: Cleanup cell insertion
• PR #2933: Minordocupdate
• PR #2968: fix notebook deletion.
• PR #2966: Added assert msg to extract_hist_ranges()
• PR #2959: Add command to trim the history database.
• PR #2681: Don’t enable pylab mode, when matplotlib is not importable
• PR #2901: Fix inputhook_wx on osx
• PR #2871: truncate potentially long CompositeErrors
• PR #2951: use istype on lists/tuples
• PR #2946: fix qtconsole history logic for end-of-line
• PR #2954: fix logic for append_javascript
• PR #2941: fix baseUrl
• PR #2903: Specify toggle value on cell line number
• PR #2911: display order in output area configurable
• PR #2897: Dont rely on BaseProjectUrl data in body tag
• PR #2894: Cm configurable
• PR #2927: next release will be 1.0
• PR #2932: Simplify using notebook static files from external code
• PR #2915: added small config section to notebook docs page
• PR #2924: safe_run_module: Silence SystemExit codes 0 and None.
• PR #2906: Unpatch/Monkey patch CM
• PR #2921: add menu item for undo delete cell
• PR #2917: Don’t add logging handler if one already exists.
• PR #2910: Respect DB_IP and DB_PORT in mongodb tests
• PR #2926: Don’t die if stderr/stdout do not support set_parent() #2925
• PR #2885: get monospace pager back
• PR #2876: fix celltoolbar layout on FF
• PR #2904: Skip remaining IPC test on Windows
• PR #2908: fix last remaining KernelApp reference
• PR #2905: fix a few remaining KernelApp/IPKernelApp changes
• PR #2900: Don’t assume test case for %time will finish in 0 time
• PR #2893: exclude fabfile from tests
• PR #2884: Correct import for kernelmanager on Windows
• PR #2882: Utils cleanup
• PR #2883: Don’t call ast.fix_missing_locations unless the AST could have been modified
• PR #2855: time(it) magic: Implement minutes/hour formatting and “%%time” cell magic
• PR #2874: Empty cell warnings
• PR #2819: tweak history prefix search (up/^p) in qtconsole
• PR #2868: Import performance
• PR #2877: minor css fixes
• PR #2880: update examples docs with kernel move
• PR #2878: Pass host environment on to kernel
• PR #2599: func_kw_complete for builtin and cython with embededsignature=True using docstring
• PR #2792: Add key “unique” to history_request protocol
• PR #2872: fix payload keys
• PR #2869: Fixing styling of toolbar selects on FF.
• PR #2708: Less css
• PR #2854: Move kernel code into IPython.kernel
• PR #2864: Fix %run -t -N<N> TypeError
• PR #2852: future pyzmq compatibility
• PR #2863: whatsnew/version0.9.txt: Fix ‘~/.python’ -> ‘~/.ipython’ typo
• PR #2861: add missing KernelManager to ConsoleApp class list
• PR #2850: Consolidate host IP detection in utils.localinterfaces
• PR #2859: Correct docstring of ipython.py
• PR #2831: avoid string version comparisons in external.qt
• PR #2844: this should address the failure in #2732
• PR #2849: utils/data: Use list comprehension for uniq_stable()
• PR #2839: add jinja to install docs / setup.py
• PR #2841: Miscellaneous docs fixes
• PR #2811: Still more KernelManager cleanup
• PR #2820: add ‘=’ to greedy completer delims
• PR #2818: log user tracebacks in the kernel (INFO-level)
• PR #2828: Clean up notebook Javascript
• PR #2829: avoid comparison error in dictdb hub history
• PR #2830: BUG: Opening parenthesis after non-callable raises ValueError
• PR #2718: try to fallback to pysqlite2.dbapi2 as sqlite3 in core.history
• PR #2816: in %edit, don’t save “last_call” unless last call succeeded
• PR #2817: change ol format order
• PR #2537: Organize example notebooks
• PR #2815: update release/authors
• PR #2808: improve patience for slow Hub in client tests
• PR #2812: remove nonfunctional –la short arg in cython magic
• PR #2810: remove dead util.upgradedir
• PR #1671: __future__ environments
• PR #2804: skip ipc tests on Windows
• PR #2789: Fixing styling issues with CellToolbar.
• PR #2805: fix KeyError creating ZMQStreams in notebook
• PR #2775: General cleanup of kernel manager code.
• PR #2340: Initial Code to reduce parallel.Client caching
• PR #2799: Exit code
• PR #2800: use type(obj) is cls as switch when canning
• PR #2801: Fix a breakpoint bug
• PR #2795: Remove outdated code from extensions.autoreload
• PR #2796: P3K: fix cookie parsing under Python 3.x (+ duplicate import is removed)
• PR #2724: In-process kernel support (take 3)
• PR #2687: [WIP] Metaui slideshow
• PR #2788: Chrome frame awareness
• PR #2649: Add version_request/reply messaging protocol
• PR #2753: add %%px --local for local execution
• PR #2783: Prefilter shouldn’t touch execution_count
• PR #2333: UI For Metadata
• PR #2396: create a ipynb3 json schema and a validator
• PR #2757: check for complete pyside presence before trying to import
• PR #2782: Allow the %run magic with ‘-b’ to specify a file.
• PR #2778: P3K: fix DeprecationWarning under Python 3x
• PR #2776: remove non-functional View.kill method
• PR #2755: can interactively defined classes
• PR #2774: Removing unused code in the notebook MappingKernelManager.
• PR #2773: Fixed minor typo causing AttributeError to be thrown.
• PR #2609: Add ‘unique’ option to history_request messaging protocol
• PR #2769: Allow shutdown when no engines are registered
• PR #2766: Define __file__ when we %edit a real file.
• PR #2476: allow %edit <variable> to work when interactively defined
• PR #2763: Reset readline delimiters after loading rmagic.
• PR #2460: Better handling of __file__ when running scripts.
• PR #2617: Fix for units argument. Adds a res argument.
• PR #2738: Unicode content crashes the pager (console)
• PR #2749: Tell Travis CI to test on Python 3.3 as well
• PR #2744: Don’t show ‘try %paste’ message while using magics
• PR #2728: shift tab for tooltip
• PR #2741: Add note to %cython Black-Scholes example warning of missing erf.
• PR #2743: BUG: Octavemagic inline plots not working on Windows: Fixed
• PR #2740: Following #2737 this error is now a name error
• PR #2737: Rmagic: error message when moving an non-existant variable from python to R
• PR #2723: diverse fixes for project url
• PR #2731: %Rpush: Look for variables in the local scope first.
• PR #2544: Infinite loop when multiple debuggers have been attached.
• PR #2726: Add qthelp docs creation
• PR #2730: added blockquote CSS
• PR #2729: Fix Read the doc build, Again
• PR #2446: [alternate 2267] Offline mathjax
• PR #2716: remove unexisting headings level
• PR #2717: One liner to fix debugger printing stack traces when lines of context are larger than source.
• PR #2713: Doc bugfix: user_ns is not an attribute of Magic objects.
• PR #2690: Fix ‘import ‘... completion for py3 & egg files.
• PR #2691: Document OpenMP in %%%cython magic
• PR #2699: fix jinja2 rendering for password protected notebooks
• PR #2700: Skip notebook testing if jinja2 is not available.
• PR #2692: Add %%%cython magics to generated documentation.

2.10. Issues closed in the 1.0 development cycle 121
• PR #2685: Fix pretty print of types when __module__ is not available.
• PR #2686: Fix tox.ini
• PR #2604: Backslashes are misinterpreted as escape-sequences by the R-interpreter.
• PR #2689: fix error in doc (arg->kwarg) and pep-8
• PR #2683: for downloads, replaced window.open with window.location.assign
• PR #2659: small bugs in js are fixed
• PR #2363: Refactor notebook templates to use Jinja2
• PR #2662: qtconsole: wrap argument list in tooltip to match width of text body
• PR #2328: addition of classes to generate a link or list of links from files local to the IPython HTML notebook
• PR #2668: pylab_not_importable: Catch all exceptions, not just RuntimeErrors.
• PR #2663: Fix issue #2660: parsing of help and version arguments
• PR #2656: Fix irunner tests when $PYTHONSTARTUP is set
• PR #2312: Add bracket matching to code cells in notebook
• PR #2571: Start to document Javascript
• PR #2641: undefined that -> this
• PR #2638: Fix %paste in Python 3 on Mac
• PR #2301: Ast transformers
• PR #2616: Revamp API docs
• PR #2572: Make ‘Paste Above’ the default paste behavior.
• PR #2574: Fix #2244
• PR #2582: Fix displaying history when output cache is disabled.
• PR #2591: Fix for Issue #2584
• PR #2526: Don’t kill paramiko tunnels when receiving ^C
• PR #2559: Add pssource, pfile, pinfo2 commands to ipdb.
• PR #2546: use 4 Pythons to build 4 Windows installers
• PR #2561: Fix display of plain text containing multiple carriage returns before line feed
• PR #2549: Add a simple ‘undo’ for cell deletion.
• PR #2525: Add event to kernel execution/shell reply.
• PR #2554: Avoid stopping in ipdb until we reach the main script.
• PR #2404: Option to limit search result in history magic command
• PR #2294: inputhook_qt4: Use QEventLoop instead of starting up the QCoreApplication
• PR #2233: Refactored Drag and Drop Support in Qt Console
• PR #1747: switch between hsplit and vsplit paging (request for feedback)
• PR #2530: Adding time offsets to the video
• PR #2542: Allow starting IPython as python -m IPython.
• PR #2534: Do not unescape backslashes in Windows (shellglob)
• PR #2517: Improved MathJax, bug fixes
• PR #2511: trigger default remote_profile_dir when profile_dir is set
• PR #2491: color is supported in ironpython
• PR #2462: Track which extensions are loaded
• PR #2464: Locate URLs in text output and convert them to hyperlinks.
• PR #2490: add ZMQInteractiveShell to IPEngineApp class list
• PR #2498: Don’t catch tab press when something selected
• PR #2527: Run All Above and Run All Below
• PR #2513: add GitHub uploads to release script
• PR #2529: Windows aware tests for shellglob
• PR #2478: Fix doctest_run_option_parser for Windows
• PR #2519: clear In[ ] prompt numbers again
• PR #2467: Clickable links
• PR #2500: Add encoding attribute to OutStream class.
• PR #2349: ENH: added StackExchange-style MathJax filtering
• PR #2503: Fix traceback handling of SyntaxErrors without line numbers.
• PR #2492: add missing ‘qtconsole’ extras_require
• PR #2480: Add deprecation warnings for sympyprinting
• PR #2334: Make the ipengine monitor the ipcontroller heartbeat and die if the ipcontroller goes down
• PR #2479: use new _winapi instead of removed _subprocess
• PR #2474: fix bootstrap name conflicts
• PR #2469: Treat __init__.pyc same as __init__.py in module_list
• PR #2165: Add -g option to %run to glob expand arguments
• PR #2468: Tell git to ignore __pycache__ directories.
• PR #2421: Some notebook tweaks.
• PR #2291: Remove old plugin system
• PR #2127: Ability to build toolbar in JS

2.10. Issues closed in the 1.0 development cycle
• PR #2445: changes for ironpython
• PR #2420: Pass ipython_dir to __init__() method of TerminalInteractiveShell’s superclass.
• PR #2432: Revert #1831, the __file__ injection in safe_execfile / safe_execfile_ipy.
• PR #2216: Autochange highlight with cell magics
• PR #1946: Add image message handler in ZMQTerminalInteractiveShell
• PR #2424: skip find_cmd when setting up script magics
• PR #2389: Catch sqlite DatabaseErrors in more places when reading the history database
• PR #2395: Don’t catch ImportError when trying to unpack module functions
• PR #1868: enable IPC transport for kernels
• PR #2437: don’t let log cleanup prevent engine start
• PR #2441: sys.maxsize is the maximum length of a container.
• PR #2442: allow iptest to be interrupted
• PR #2240: fix message built for engine dying during task
• PR #2369: Block until kernel termination after sending a kill signal
• PR #2439: Py3k: Octal (0777 -> 0o777)
• PR #2326: Detachable pager in notebook.
• PR #2377: Fix installation of man pages in Python 3
• PR #2407: add IPython version to message headers
• PR #2408: Fix Issue #2366
• PR #2405: clarify TaskScheduler.hwm doc
• PR #2399: IndentationError display
• PR #2400: Add scroll_to_cell(cell_number) to the notebook
• PR #2401: unmock read-the-docs modules
• PR #2311: always perform requested trait assignments
• PR #2393: New option n to limit history search hits
• PR #2386: Adapt inline backend to changes in matplotlib
• PR #2392: Remove suspicious double quote
• PR #2387: Added -L library search path to cythonmagic cell magic
• PR #2370: qtconsole: Create a prompt newline by inserting a new block (w/o formatting)
• PR #1715: Fix for #1688, traceback-unicode issue
• PR #2378: use Singleton.instance() for embed() instead of manual global
• PR #2373: fix missing imports in core.interactiveshell
• PR #2368: remove notification widget leftover
• PR #2327: Parallel: Support get/set of nested objects in view (e.g. dv[‘a.b’])
• PR #2362: Clean up ProgressBar class in example notebook
• PR #2346: Extra xterm identification in set_term_title
• PR #2352: Notebook: Store the username in a cookie whose name is unique.
• PR #2358: add backport_pr to tools
• PR #2365: fix names of notebooks for download/save
• PR #2364: make clients use ‘location’ properly (fixes #2361)
• PR #2354: Refactor notebook templates to use Jinja2
• PR #2339: add bash completion example
• PR #2345: Remove references to ‘version’ no longer in argparse. Github issue #2343.
• PR #2347: adjust division error message checking to account for Python 3
• PR #2305: RemoteError._render_traceback_ calls self.render_traceback
• PR #2338: Normalize line endings for ipexec_validate, fix for #2315.
• PR #2192: Introduce Notification Area
• PR #2329: Better error messages for common magic commands.
• PR #2337: ENH: added StackExchange-style MathJax filtering
• PR #2331: update css for qtconsole in doc
• PR #2317: adding cluster_id to parallel.Client.__init__
• PR #2130: Add -l option to %R magic to allow passing in of local namespace
• PR #2196: Fix for bad command line argument to latex
• PR #2300: bug fix: was crashing when sqlite3 is not installed
• PR #2184: Expose store_history to execute_request messages.
• PR #2308: Add welcome_message option to enable_pylab
• PR #2302: Fix variable expansion on ‘self’
• PR #2299: Remove code from prefilter that duplicates functionality in inputsplitter
• PR #2295: allow pip install from github repository directly
• PR #2280: fix SSH passwordless check for OpenSSH
• PR #2290: nbmanager
• PR #2288: s/assertEquals/assertEqual (again)
• PR #2287: Removed outdated dev docs.
• PR #2218: Use redirect for new notebooks

2.10. Issues closed in the 1.0 development cycle
• PR #2277: nb: up/down arrow keys move to begin/end of line at top/bottom of cell
• PR #2045: Refactoring notebook managers and adding Azure backed storage.
• PR #2271: use display instead of send_figure in inline backend hooks
• PR #2278: allow disabling SQLite history
• PR #2225: Add “–annotate” option to %%cython magic.
• PR #2246: serialize individual args/kwargs rather than the containers
• PR #2274: CLN: Use name to id mapping of notebooks instead of searching.
• PR #2270: SSHLauncher tweaks
• PR #2269: add missing location when disambiguating controller IP
• PR #2263: Allow docs to build on http://readthedocs.org/
• PR #2256: Adding data publication example notebook.
• PR #2255: better flush iopub with AsyncResults
• PR #2261: Fix: longest_substr([]) -> ‘’
• PR #2260: fix mpr again
• PR #2242: Document globbing in %history -g <pattern>.
• PR #2250: fix html in notebook example
• PR #2245: Fix regression in embed() from pull-request #2096.
• PR #2248: track sha of master in test_pr messages
• PR #2238: Fast tests
• PR #2211: add data publication message
• PR #2236: minor test_pr tweaks
• PR #2231: Improve Image format validation and add html width,height
• PR #2232: Reapply monkeypatch to inspect.findsource()
• PR #2235: remove spurious print statement from setupbase.py
• PR #2222: adjust how canning deals with import strings
• PR #2224: fix css typo
• PR #2223: Custom tracebacks
• PR #2214: use KernelApp.exec_lines/files in IPythonApp
• PR #2199: Wrap JS published by %%javascript in try/catch
• PR #2212: catch errors in markdown javascript
• PR #2190: Update code mirror 2.22 to 2.32
• PR #2200: documentation build broken in bb429da5b
• PR #2194: clean nan/inf in json_clean
• PR #2198: fix mpr for earlier git version
• PR #2175: add FileFindHandler for Notebook static files
• PR #1990: can func_defaults
• PR #2069: start improving serialization in parallel code
• PR #2202: Create a unique & temporary IPYTHONDIR for each testing group.
• PR #2204: Work around lack of os.kill in win32.
• PR #2148: win32 iptest: Use subprocess.Popen() instead of os.system().
• PR #2179: Pylab switch
• PR #2124: Add an API for registering magic aliases.
• PR #2169: ipdb: pdef, pdoc, pinfo magics all broken
• PR #2174: Ensure consistent indentation in %magic.
• PR #1930: add size-limiting to the DictDB backend
• PR #2189: Fix IPython.lib.latextools for Python 3
• PR #2186: removed references to h5py dependence in octave magic documentation
• PR #2183: Include the kernel object in the event object passed to kernel events
• PR #2185: added test for %store, fixed storemagic
• PR #2138: Use breqn.sty in dvipng backend if possible
• PR #2182: handle undefined param in notebooklist
• PR #1831: fix #1814 set __file__ when running .ipy files
• PR #2051: Add a metadata attribute to messages
• PR #1471: simplify IPython.parallel connections and enable Controller Resume
• PR #2181: add %%javascript, %%svg, and %%latex display magics
• PR #2116: different images in 00_notebook-tour
• PR #2092: %prun: Restore stats.stream after running print_stream.
• PR #2159: show message on notebook list if server is unreachable
• PR #2176: fix git mpr
• PR #2152: [qtconsole] Namespace not empty at startup
• PR #2177: remove numpy install from travis/tox scripts
• PR #2090: New keybinding for code cell execution + cell insertion
• PR #2160: Updating the parallel options pricing example
• PR #2168: expand line in cell magics

2.10. Issues closed in the 1.0 development cycle
• PR #2170: Fix tab completion with IPython.embed_kernel().
• PR #2096: embed(): Default to the future compiler flags of the calling frame.
• PR #2163: fix ‘remote_profile_dir’ typo in SSH launchers
• PR #2158: [2to3 compat ] Tuple params in func defs
• PR #2089: Fix unittest DeprecationWarnings
• PR #2142: Refactor test_pr.py
• PR #2140: 2to3: Apply has_key fixer.
• PR #2131: Add option append (-a) to %save
• PR #2117: use explicit url in notebook example
• PR #2133: Tell git that *.py files contain Python code, for use in word-diffs.
• PR #2134: Apply 2to3 next fix.
• PR #2126: ipcluster broken with any batch launcher (PBS/LSF/SGE)
• PR #2104: Windows make file for Sphinx documentation
• PR #2074: Make BG color of inline plot configurable
• PR #2123: BUG: Look up the _repr_pretty_ method on the class within the MRO rather...
• PR #2100: [in progress] python 2 and 3 compatibility without 2to3, second try
• PR #2128: open notebook copy in different tabs
• PR #2073: allows password and prefix for notebook
• PR #1993: Print View
• PR #2086: re-aliad %ed to %edit in qtconsole
• PR #2110: Fixes and improvements to the input splitter
• PR #2101: fix completer deleting newline
• PR #2102: Fix logging on interactive shell.
• PR #2088: Fix (some) Python 3.2 ResourceWarnings
• PR #2064: conform to pep 3110
• PR #2076: Skip notebook ‘static’ dir in test suite.
• PR #2063: Remove umlauts so py3 installations on LANG=C systems succeed.
• PR #2068: record sysinfo in sdist
• PR #2067: update tools/release_windows.py
• PR #2065: Fix parentheses typo
• PR #2062: Remove duplicates and auto-generated files from repo.
• PR #2061: use explicit tuple in exception
• PR #2060: change minus to - or (hy in manpages

Issues (691):

• #3940: Install process documentation overhaul
• #3946: The PDF option for --post should work with lowercase
• #3957: Notebook help page broken in Firefox
• #3894: nbconvert test failure
• #3887: 1.0.0a1 shows blank screen in both firefox and chrome (windows 7)
• #3703: nbconvert: Output options – names and documentataion
• #3931: Tab completion not working during debugging in the notebook
• #3936: Ipcluster plugin is not working with Ipython 1.0dev
• #3941: IPython Notebook kernel crash on Win7x64
• #3926: Ending Notebook renaming dialog with return creates new-line
• #3932: Incorrect empty docstring
• #3928: Passing variables to script from the workspace
• #3774: Notebooks with spaces in their names breaks nbconvert latex graphics
• #3916: tornado needs its own check
• #3915: Link to Parallel examples “found on GitHub” broken in docs
• #3895: Keyboard shortcuts box in notebook doesn’t fit the screen
• #3912: IPython.utils fails automated test for RC1 1.0.0
• #3636: Code cell missing highlight on load
• #3897: under Windows, “ipython3 nbconvert “C:/blabla/first_try.ipynb” –to latex –post PDF” POST processing action fails because of a bad parameter
• #3900: python3 install syntax errors (OS X 10.8.4)
• #3899: nbconvert to latex fails on notebooks with spaces in file name
• #3881: Temporary Working Directory Test Fails
• #2750: A way to freeze code cells in the notebook
• #3893: Resize Local Image Files in Notebook doesn’t work
• #3823: nbconvert on windows: tex and paths
• #3885: under Windows, “ipython3 nbconvert “C:/blabla/first_try.ipynb” –to latex” write “‘” instead of “/” to reference file path in the .tex file
• #3889: test_qt fails due to assertion error ‘qt4’ != ‘qt’
• #3890: double post, disregard this issue
• #3689: nbconvert, remaining tests

2.10. Issues closed in the 1.0 development cycle
• #3874: Up/Down keys don’t work to “Search previous command history” (besides Ctrl-p/Ctrl-n)
• #3853: CodeMirror locks up in the notebook
• #3862: can only connect to an ipcluster started with v1.0.0-dev (master branch) using an older ipython (v0.13.2), but cannot connect using ipython (v1.0.0-dev)
• #3869: custom css not working.
• #2960: Keyboard shortcuts
• #3795: ipcontroller process goes to 100% CPU, ignores connection requests
• #3553: Ipython and pylab crashes in windows and canopy
• #3837: Cannot set custom mathjax url, crash notebook server.
• #3808: “Naming” releases ?
• #2431: TypeError: must be string without null bytes, not str
• #3856: ? at end of comment causes line to execute
• #3731: nbconvert: add logging for the different steps of nbconvert
• #3835: Markdown cells do not render correctly when mathjax is disabled
• #3843: nbconvert to rst: leftover “In[ ]”
• #3799: nbconvert: Ability to specify name of output file
• #3726: Document when IPython.start_ipython() should be used versus IPython.embed()
• #3778: Add no more readonly view in what’s new
• #3754: No Print View in Notebook in 1.0dev
• #3798: IPython 0.12.1 Crashes on autocompleting sqlalchemy.func.row_number properties
• #3811: Opening notebook directly from the command line with multi-directory support installed
• #3775: Annoying behavior when clicking on cell after execution (Ctrl+Enter)
• #3809: Possible to add some bpython features?
• #3810: Printing the contents of an image file messes up shell text
• #3702: nbconvert: Default help message should be that of –help
• #3735: Nbconvert 1.0.0a1 does not take into account the pdf extensions in graphs
• #3719: Bad strftime format, for windows, in nbconvert exporter
• #3786: Zmq errors appearing with Ctrl-C in console/qtconsole
• #3019: disappearing scrollbar on tooltip in Chrome 24 on Ubuntu 12.04
• #3785: ipdb completely broken in Qt console
• #3796: Document the meaning of milestone/issues-tags for users.
• #3788: Do not auto show tooltip if docstring empty.
• #1366: [Web page] No link to front page from documentation
• #3739: nbconvert (to slideshow) misses some of the math in markdown cells
• #3768: increase and make timeout configurable in console completion.
• #3724: ipcluster only running on one cpu
• #1592: better message for unsupported nbformat
• #2049: Can not stop “ipython kernel” on windows
• #3757: Need direct entry point to given notebook
• #3745: ImportError: cannot import name check_linecache_ipython
• #3701: nbconvert: Final output file should be in same directory as input file
• #3738: history -o works but history with -n produces identical results
• #3740: error when attempting to run ‘make’ in docs directory
• #3737: ipython nbconvert crashes with ValueError: Invalid format string.
• #3730: nbconvert: unhelpful error when pandoc isn’t installed
• #3718: markdown cell cursor misaligned in notebook
• #3710: multiple input fields for %debug in the notebook after resetting the kernel
• #3713: PyCharm has problems with IPython working inside PyPy created by virtualenv
• #3712: Code completion: Complete on dictionary keys
• #3680: --pylab and --matplotlib flag
• #3698: nbconvert: Unicode error with minus sign
• #3693: nbconvert does not process SVGs into PDFs
• #3688: nbconvert, figures not extracting with Python 3.x
• #3542: note new dependencies in docs / setup.py
• #2556: [pagedown] do not target_blank anchor link
• #3684: bad message when %pylab fails due import other than matplotlib
• #3682: ipython notebook pylab inline import_all=False
• #3596: MathjaxUtils race condition?
• #1540: Comment/uncomment selection in notebook
• #2702: frozen setup: permission denied for default ipython_dir
• #3672: allow_none on Number-like traits.
• #2411: add CONTRIBUTING.md
• #481: IPython terminal issue with Qt4Agg on XP SP3
• #2664: How to preserve user variables from import clashing?
• #3436: enable_pylab(import_all=False) still imports np
• #2630: lib.pylabtools.figsize : NameError when using Qt4Agg backend and %pylab magic.
• #3154: Notebook: no event triggered when a Cell is created
• #3579: Nbconvert: SVG are not transformed to PDF anymore
• #3604: MathJax rendering problem in %%latex cell
• #3668: AttributeError: ‘BlockingKernelClient’ object has no attribute ‘started_channels’
• #3245: SyntaxError: encoding declaration in Unicode string
• #3639: %pylab inline in IPYTHON notebook throws “RuntimeError: Cannot activate multiple GUI eventloops”
• #3663: frontend deprecation warnings
• #3661: run -m not behaving like python -m
• #3597: re-do PR #3531 - allow markdown in Header cell
• #3053: Markdown in header cells is not rendered
• #3655: IPython finding its way into pasted strings.
• #3620: uncaught errors in HTML output
• #3646: get_dict() error
• #3004: %load_ext rmagic fails when legacy ipy_user_conf.py is installed (in ipython 0.13.1 / OSX 10.8)
• #3638: setp() issue in ipython notebook with figure references
• #3634: nbconvert reveal to pdf conversion ignores styling, prints only a single page.
• #1307: Remove pyreadline workarounds, we now require pyreadline >= 1.7.1
• #3316: find_cmd test failure on Windows
• #3494: input() in notebook doesn’t work in Python 3
• #3427: Deprecate $ as mathjax delimiter
• #3625: Pager does not open from button
• #3149: Miscellaneous small nbconvert feedback
• #3617: 256 color escapes support
• #3609: %pylab inline blows up for single process ipython
• #2934: Publish the Interactive MPI Demo Notebook
• #3614: ansi escapes broken in master (ls –color)
• #3610: If you don’t have markdown, python setup.py install says no pygments
• #3547: %run modules clobber each other
• #3602: import_item fails when one tries to use DottedObjectName instead of a string
• #3563: Duplicate tab completions in the notebook
• #3599: Problems trying to run IPython on python3 without installing...
• #2937: too long completion in notebook
• #3479: Write empty name for the notebooks
• #3505: nbconvert: Failure in specifying user filter
• #1537: think a bit about namespaces
• #3124: Long multiline strings in Notebook
• #3464: run -d message unclear
• #2706: IPython 0.13.1 ignoring PYTHONSTARTUP
• #3587: LaTeX escaping bug in nbconvert when exporting to HTML
• #3213: Long running notebook died with a coredump
• #3580: Running ipython with pypy on windows
• #3573: custom.js not working
• #3544: IPython.lib test failure on Windows
• #3352: Install Sphinx extensions
• #2971: [notebook]user needs to press ctrl-c twice to stop notebook server should be put into terminal window
• #2413: ipython3 qtconsole fails to install: ipython 0.13 has no such extra feature ‘qtconsole’
• #2618: documentation is incorrect for install process
• #2595: mac 10.8 qtconsole export history
• #2586: cannot store aliases
• #2714: ipython qtconsole print unittest messages in console instead his own window.
• #2669: cython magic failing to work with openmp.
• #3256: Vagrant pandas instance of iPython Notebook does not respect additional plotting arguments
• #3010: cython magic fail if cache dir is deleted while in session
• #2044: prune unused names from parallel.error
• #1145: Online help utility broken in QtConsole
• #3439: Markdown links no longer open in new window (with change from pagedown to marked)
• #3476: _margv for macros seems to be missing
• #3499: Add reveal.js library (version 2.4.0) inside IPython
• #2771: Wiki Migration to GitHub
• #2887: ipcontroller purging some engines during connect

2.10. Issues closed in the 1.0 development cycle 133
• #626: Enable Resuming Controller
• #2824: Kernel restarting after message “Kernel XXXX failed to respond to heartbeat”
• #2823: %%cython magic gives ImportError: dlopen(long_file_name.so, 2): image not found
• #2891: In IPython for Python 3, system site-packages comes before user site-packages
• #2928: Add magic “watch” function (example)
• #2931: Problem rendering pandas dataframe in Firefox for Windows
• #2939: [notebook] Figure legend not shown in inline backend if outside the box of the axes
• #2972: [notebook] in Markdown mode, press Enter key at the end of <some http link>, the next line is indented unexpectedly
• #3069: Instructions for installing IPython notebook on Windows
• #3444: Encoding problem: cannot use if user’s name is not ascii?
• #3335: Reenable bracket matching
• #3386: Magic %paste not working in Python 3.3.2. TypeError: Type str doesn’t support the buffer API
• #3543: Exception shutting down kernel from notebook dashboard (0.13.1)
• #3549: Codecell size changes with selection
• #3445: Adding newlines in %%latex cell
• #3237: [notebook] Can’t close a notebook without errors
• #2916: colon invokes auto(un)indent in markdown cells
• #2167: Indent and dedent in htmlnotebook
• #3545: Notebook save button icon not clear
• #3534: nbconvert incompatible with Windows?
• #3489: Update example notebook that raw_input is allowed
• #3396: Notebook checkpoint time is displayed an hour out
• #3261: Empty revert to checkpoint menu if no checkpoint...
• #2984: “print” magic does not work in Python 3
• #3524: Issues with pyzmq and ipython on EPD update
• #2434: %store magic not auto-restoring
• #2720: base_url and static path
• #2234: Update various low resolution graphics for retina displays
• #2842: Remember passwords for pw-protected notebooks
• #3244: qtconsole: ValueError('close_fds is not supported on Windows platforms if you redirect stdin/stdout/stderr'.)
• #2215:AsyncResult.wait(0) can hang waiting for the client to get results?
• #2268: provide mean to retrieve static data path
• #1905: Expose UI for worksheets within each notebook
• #2380: Qt inputhook prevents modal dialog boxes from displaying
• #3185: prettify on double //
• #2821: Test failure: IPython.parallel.tests.test_client.test_resubmit_header
• #2475: [Notebook] Line is deindented when typing eg a colon in markdown mode
• #2470: Do not destroy valid notebooks
• #860: Allow the standalone export of a notebook to HTML
• #2652: notebook with qt backend crashes at save image location popup
• #1587: Improve kernel restarting in the notebook
• #2710: Saving a plot in Mac OS X backend crashes IPython
• #2596: notebook “Last saved:” is misleading on file opening.
• #2671: TypeError :NoneType when executed “ipython qtconsole” in windows console
• #2703: Notebook scrolling breaks after pager is shown
• #2803: KernelManager and KernelClient should be two separate objects
• #2693: TerminalIPythonApp configuration fails without ipython_config.py
• #2531: IPython 0.13.1 python 2 32-bit installer includes 64-bit ipython*.exe launchers in the scripts folder
• #2520: Control-C kills port forwarding
• #2279: Setting __file__ to None breaks Mayavi import
• #2161: When logged into notebook, long titles are incorrectly positioned
• #1292: Notebook, Print view should not be editable...
• #1731: test parallel launchers
• #3227: Improve documentation of ipcontroller and possible BUG
• #2896: IPController very unstable
• #3517: documentation build broken in head
• #3522: UnicodeDecodeError: ‘ascii’ codec can’t decode byte on Pycharm on Windows
• #3448: Please include MathJax fonts with IPython Notebook
• #3519: IPython Parallel map mysteriously turns pandas Series into numpy ndarray
• #3345: IPython embedded shells ask if I want to exit, but I set confirm_exit = False
• #3509: IPython won’t close without asking “Are you sure?” in Firefox

2.10. Issues closed in the 1.0 development cycle
• #3471: Notebook jinja2/markupsafe dependencies in manual
• #3502: Notebook broken in master
• #3302: autoreload does not work in ipython 0.13.x, python 3.3
• #3475: no warning when leaving/closing notebook on master without saved changes
• #3490: No obvious feedback when kernel crashes
• #1912: Move all autoreload tests to their own group
• #2577: sh.py and ipython for python 3.3
• #3467: %magic doesn’t work
• #3501: Editing markdown cells that wrap has off-by-one errors in cursor positioning
• #3492: IPython for Python3
• #3474: unexpected keyword argument to remove_kernel
• #2283: TypeError when using ‘?’ after a string in a %logstart session
• #2787: rmagic and pandas DataFrame
• #2605: Ellipsis literal triggers AttributeError
• #1179: Test unicode source in pinfo
• #2055: drop Python 3.1 support
• #2293: IPEP 2: Input transformations
• #2790: %paste and %cpaste not removing ”...” lines
• #3480: Testing fails because iptest.py cannot be found
• #2580: will not run within PIL build directory
• #2797: RMagic, Dataframe Conversion Problem
• #2838: Empty lines disappear from triple-quoted literals.
• #3050: Broken link on IPython.core.display page
• #3473: Config not passed down to subcommands
• #3462: Setting log_format in config file results in error (and no format changes)
• #3311: Notebook (occasionally) not working on windows (Sophos AV)
• #3461: Cursor positioning off by a character in auto-wrapped lines
• #3454: _repr_html_ error
• #3457: Space in long Paragraph Markdown cell with Chinese or Japanese
• #3447: Run Cell Does not Work
• #1373: Last lines in long cells are hidden
• #1504: Revisit serialization in IPython.parallel
• #1459: Can’t connect to 2 HTTPS notebook servers on the same host
• #678: Input prompt stripping broken with multiline data structures
• #3001: IPython.notebook.dirty flag is not set when a cell has unsaved changes
• #3077: Multiprocessing semantics in parallel.view.map
• #3056: links across notebooks
• #3120: Tornado 3.0
• #3156: update pretty to use Python 3 style for sets
• #3197: Can’t escape multiple dollar signs in a markdown cell
• #3309: Image() signature/doc improvements
• #3415: Bug in IPython/external/path/__init__.py
• #3446: Feature suggestion: Download matplotlib figure to client browser
• #3295: autoexported notebooks: only export explicitly marked cells
• #3442: Notebook: Summary table extracted from markdown headers
• #3438: Zooming notebook in chrome is broken in master
• #1378: Implement autosave in notebook
• #3437: Highlighting matching parentheses
• #3435: module search segfault
• #3424: ipcluster --version
• #3434: 0.13.2 Ipython/genutils.py doesn’t exist
• #3426: Feature request: Save by cell and not by line #: IPython %save magic
• #3412: Non Responsive Kernel: Running a Django development server from an IPython Notebook
• #3408: Save cell toolbar and slide type metadata in notebooks
• #3246: %%paste regression with blank lines
• #3404: Weird error with $variable and grep in command line magic (!command)
• #3405: Key auto-completion in dictionaries?
• #3259: Codemirror linenumber css broken
• #3397: Vertical text misalignment in Markdown cells
• #3391: Revert #3358 once fix integrated into CM
• #3360: Error 500 while saving IPython notebook
• #3375: Frequent Safari/WebKit crashes
• #3365: zmq frontend
• #2654: User_expression issues

2.10. Issues closed in the 1.0 development cycle 137
• #3389: Store history as plain text
• #3388: Ipython parallel: open TCP connection created for each result returned from engine
• #3385: setup.py failure on Python 3
• #3376: Setting __module__ to None breaks pretty printing
• #3374: ipython qtconsole does not display the prompt on OSX
• #3380: simple call to kernel
• #3379: TaskRecord key ‘started’ not set
• #3241: notebook conection time out
• #3334: magic interpreter interpretes non magic commands?
• #3326: python3.3: Type error when launching SGE cluster in IPython notebook
• #3349: pip3 doesn’t run 2to3?
• #3347: Longlist support in ipdb
• #3343: Make pip install / easy_install faster
• #3337: git submodules broke nightly PPA builds
• #3206: Copy/Paste Regression in QtConsole
• #3329: Buggy linewrap in Mac OSX Terminal (Mountain Lion)
• #3327: Qt version check broken
• #3303: parallel tasks never finish under heavy load
• #1381: ‘\’ for equation continuations require an extra “’ in markdown cells
• #3314: Error launching iPython
• #3306: Test failure when running on a Vagrant VM
• #3280: IPython.utils.process.getoutput returns stderr
• #3299: variables named _ or __ exhibit incorrect behavior
• #3196: add an “x” or similar to htmlnotebook pager
• #3293: Several 404 errors for js files Firefox
• #3292: syntax highlighting in chrome on OSX 10.8.3
• #3288: Latest dev version hangs on page load
• #3283: ipython dev retains directory information after directory change
• #3279: custom.css is not overridden in the dev IPython (1.0)
• #2727: %run -m doesn’t support relative imports
• #3268: GFM triple backquote and unknown language
• #3273: Suppressing all plot related outputs
• #3272: Backspace while completing load previous page
• #3260: Js error in savewidget
• #3247: scrollbar in notebook when not needed?
• #3243: notebook: option to view json source from browser
• #3265: 404 errors when running IPython 1.0dev
• #3257: setup.py not finding submodules
• #3253: Incorrect Qt and PySide version comparison
• #3248: Cell magics broken in Qt console
• #3012: Problems with the less based style.min.css
• #2390: Image width/height don’t work in embedded images
• #3236: cannot set TerminalIPythonApp.log_format
• #3214: notebook kernel dies if started with invalid parameter
• #2980: Remove HTMLCell ?
• #3128: qtconsole hangs on importing pylab (using X forwarding)
• #3198: Hitting recursive depth causing all notebook pages to hang
• #3218: race conditions in profile directory creation
• #3177: OverflowError exception in handlers.py
• #2563: core.profiledir.check_startup_dir() doesn’t work inside py2exe’d installation
• #3207: [Feature] folders for ipython notebook dashboard
• #3178: cell magics do not work with empty lines after #2447
• #3204: Default plot() colors unsuitable for red-green colorblind users
• #1789: :\n/*foo
* turns into :\n*(foo) in triple-quoted strings.
• #3202: File cell magic fails with blank lines
• #3199: %%%cython -a stopped working?
• #2688: obsolete imports in import autocompletion
• #3192: Python2, Unhandled exception, __builtin__.True = False
• #3179: script magic error message loop
• #3009: use XDG_CACHE_HOME for cython objects
• #3059: Bugs in 00_notebook_tour example.
• #3104: Integrate a javascript file manager into the notebook front end
• #3176: Particular equation not rendering (notebook)
• #1133: [notebook] readonly and upload files/UI
• #2975: [notebook] python file and cell toolbar
• #3017: SciPy.weave broken in IPython notebook/qtconsole
• #3161: paste macro not reading spaces correctly
• #2835: %paste not working on WinXpSP3/ipython-0.13.1.py2-win32-PROPER.exe/python27
• #2628: Make transformers work for lines following decorators
• #2612: Multiline String containing ":n?foon” confuses interpreter to replace ?foo with get_ipython().magic(u’pin foo’)
• #2539: Request: Enable cell magics inside of .ipy scripts
• #2507: Multiline string does not work (includes ...) with doctest type input in IPython notebook
• #2164: Request: Line breaks in line magic command
• #3106: poor parallel performance with many jobs
• #2438: print inside multiprocessing crashes Ipython kernel
• #3155: Bad md5 hash for package 0.13.2
• #3045: [Notebook] Ipython Kernel does not start if disconnected from internet(/network?)
• #3146: Using celery in python 3.3
• #3145: The notebook viewer is down
• #2385: grep –color not working well with notebook
• #3131: Quickly install from source in a clean virtualenv?
• #3139: Rolling log for ipython
• #3127: notebook with pylab=inline appears to call figure.draw twice
• #3129: Walking up and down the call stack
• #3123: Notebook crashed if unplugged ethernet cable
• #3121: NB should use normalize.css? was #3049
• #3087: Disable spellchecking in notebook
• #3084: ipython pyqt 4.10 incompatibilty, QTextBlockUserData
• #3113: Fails to install under Jython 2.7 beta
• #3110: Render of h4 headers is not correct in notebook (error in renderedhtml.css)
• #3109: BUG: read_csv: dtype={‘id’ : np.str}: Datatype not understood
• #3107: Autocompletion of object attributes in arrays
• #3103: Reset locale setting in qtconsole
• #3090: python3.3 Entry Point not found
• #3081: UnicodeDecodeError when using Image(data=”some.jpeg”)
• #2834: url regexp only finds one link
• #3091: qtconsole breaks doctest.testmod() in Python 3.3
• #3074: SIGUSR1 not available on Windows
• #2996: registration::purging stalled registration high occurrence in small clusters
• #3065: diff-ability of notebooks
• #3067: Crash with pygit2
• #3061: Bug handling Ellipsis
• #3049: NB css inconsistent behavior between ff and webkit
• #3039: unicode errors when opening a new notebook
• #3048: Installning ipython qtConsole should be easier att Windows
• #3042: Profile creation fails on 0.13.2 branch
• #3035: docstring typo/inconsistency: mention of an xml notebook format?
• #3031: HDF5 library segfault (possibly due to mismatching headers?)
• #2991: In notebook importing sympy closes ipython kernel
• #3027: f.__globals__ causes an error in Python 3.3
• #3020: Failing test test_interactiveshell.TestAstTransform on Windows
• #3023: alt text for “click to expand output” has typo in alt text
• #2963: %history to print all input history of a previous session when line range is omitted
• #3014: Ctrl-C finishes notebook immediately
• #3018: IPython installed within virtualenv. WARNING “Please install IPython inside the virtualenv”
• #3014: Ctrl-C finishes notebook immediately
• #3007: cython_pyximport reload broken in python3
• #2955: Incompatible Qt imports when running inprocess_qtconsole
• #3006: [IPython 0.13.1] The check of PyQt version is wrong
• #3005: Renaming a notebook to an existing notebook name overwrites the other file
• #2940: Abort trap in IPython Notebook after installing matplotlib
• #3000: issue #3000
• #2995: ipython_directive.py fails on multiline when prompt number < 100
• #2993: File magic (%%file) does not work with paths beginning with tilde (e.g., ~/anaconda/stuff.txt)
• #2992: Cell-based input for console and qt frontends?
• #2425: Liaise with Spyder devs to integrate newer IPython
• #2986: requesting help in a loop can damage a notebook

2.10. Issues closed in the 1.0 development cycle
• #2978: v1.0-dev build errors on Arch with Python 3.
• #2557: [refactor] Insert_cell_at_index()
• #2969: ipython command does not work in terminal
• #2762: OSX wxPython (osx_cocoa, 64bit) command “%gui wx” blocks the interpreter
• #2956: Silent importing of submodules differs from standard Python3.2 interpreter’s behavior
• #2943: Up arrow key history search gets stuck in QTConsole
• #2953: using ‘nonlocal’ declaration in global scope causes ipython3 crash
• #2952: qtconsole ignores exec_lines
• #2949: ipython crashes due to atexit()
• #2947: From rmagic to an R console
• #2938: docstring pane not showing in notebook
• #2936: Tornado assumes invalid signature for parse_qs on Python 3.1
• #2935: unable to find python after easy_install / pip install
• #2920: Add undo-cell deletion menu
• #2914: BUG:saving a modified .py file after loading a module kills the kernel
• #2925: BUG: kernel dies if user sets sys.stderr or sys.stdout to a file object
• #2909: LaTeX sometimes fails to render in markdown cells with some curly bracket + underscore combinations
• #2898: Skip ipc tests on Windows
• #2902: ActiveState attempt to build ipython 0.12.1 for python 3.2.2 for Mac OS failed
• #2899: Test failure in IPython.core.tests.test_magic.test_time
• #2890: Test failure when fabric not installed
• #2892: IPython tab completion bug for paths
• #1340: Allow input cells to be collapsed
• #2881: ? command in notebook does not show help in Safari
• #2751: %timeit should use minutes to format running time in long running cells
• #2879: When importing a module with a wrong name, ipython crashes
• #2862: %timeit should warn of empty contents
• #2485: History navigation breaks in qtconsole
• #2785: gevent input hook
• #2843: Sliently running code in clipboard (with paste, cpaste and variants)
• #2784: %run -t -N<N> error
• #2732: Test failure with FileLinks class on Windows
• #2860: ipython help notebook -> KeyError: ‘KernelManager’
• #2858: Where is the installed ipython script?
• #2856: Edit code entered from ipython in external editor
• #2722: IPC transport option not taking effect?
• #2473: Better error messages in ipengine/ipcontroller
• #2836: Cannot send builtin module definitions to IP engines
• #2833: Any reason not to use super()?
• #2781: Cannot interrupt infinite loops in the notebook
• #2150: clippath_demo.py in matplotlib example does not work with inline backend
• #2634: Numbered list in notebook markdown cell renders with Roman numerals instead of numbers
• #2230: IPython crashing during startup with “AttributeError: ‘NoneType’ object has no attribute ‘rstrip’”
• #2483: nbviewer bug? with multi-file gists
• #2466: mistyping ed -p breaks ed -p
• #2477: Glob expansion tests fail on Windows
• #2622: doc issue: notebooks that ship with Ipython .13 are written for python 2.x
• #2626: Add “Cell -> Run All Keep Going” for notebooks
• #1223: Show last modification date of each notebook
• #2621: user request: put link to example notebooks in Dashboard
• #2564: grid blanks plots in ipython pylab inline mode (interactive)
• #2532: Django shell (IPython) gives NameError on dict comprehensions
• #2188: ipython crashes on ctrl-c
• #2391: Request: nbformat API to load/save without changing version
• #2355: Restart kernel message even though kernel is perfectly alive
• #2306: Garbled input text after reverse search on Mac OS X
• #2297: ipdb with separate kernel/client pushing stdout to kernel process only
• #2180: Have [kernel busy] overridden only by [kernel idle]
• #1188: Pylab with OSX backend keyboard focus issue and hang
• #2107: test_octavemagic.py[everything] fails
• #1212: Better understand/document browser compatibility
• #1585: Refactor notebook templates to use Jinja2 and make each page a separate directory

2.10. Issues closed in the 1.0 development cycle
• #1443: xticks scaling factor partially obscured with qtconsole and inline plotting
• #1209: can’t make %result work as in doc.
• #1200: IPython 0.12 Windows install fails on Vista
• #1127: Interactive test scripts for Qt/nb issues
• #959: Matplotlib figures hide
• #2071: win32 installer issue on Windows XP
• #2610: ZMQInteractiveShell.colors being ignored
• #2505: Markdown Cell incorrectly highlighting after “<”
• #165: Installer fails to create Start Menu entries on Windows
• #2356: failing traceback in terminal ipython for first exception
• #2145: Have dashboard show when server disconnect
• #2098: Do not crash on kernel shutdown if json file is missing
• #2813: Offline MathJax is broken on 0.14dev
• #2807: Test failure: IPython.parallel.tests.test_client.TestClient.test_purge_everything
• #2486: Readline’s history search in ipython console does not clear properly after cancellation with Ctrl+C
• #2709: Cython -la doesn’t work
• #2767: What is IPython.utils.upgradedir?
• #2210: Placing matplotlib legend outside axis bounds causes inline display to clip it
• #2553: IPython Notebooks not robust against client failures
• #2536: ImageDraw in Ipython notebook not drawing lines
• #2264: Feature request: Versioning messaging protocol
• #2589: Creation of ~300+ MPI-spawned engines causes instability in ipcluster
• #2672: notebook: inline option without pylab
• #2673: Indefinite Articles & Traitlets
• #2705: Notebook crashes Safari with select and drag
• #2721: dreload kills ipython when it hits zmq
• #2806: ipython.parallel doesn’t discover globals under Python 3.3
• #2794: _exit_code behaves differently in terminal vs ZMQ frontends
• #2793: IPython.parallel issue with pushing pandas TimeSeries
• #1085: In process kernel for Qt frontend
• #2760: IndexError: list index out of range with Python 3.2
• #2780: Save and load notebooks from github
• #2772: AttributeError: ‘Client’ object has no attribute ‘kill’
• #2754: Fail to send class definitions from interactive session to engines namespaces
• #2764: TypeError while using ‘cd’
• #2765: name ‘__file__’ is not defined
• #2540: Wrap tooltip if line exceeds threshold?
• #2394: Startup error on ipython qtconsole (version 0.13 and 0.14-dev
• #2440: IPEP 4: Python 3 Compatibility
• #1814: __file__ is not defined when file end with .ipy
• #2759: R magic extension interferes with tab completion
• #2615: Small change needed to rmagic extension.
• #2748: collapse parts of a html notebook
• #1661: %paste still bugs about IndentationError and says to use %paste
• #2742: Octavemagic fails to deliver inline images in IPython (on Windows)
• #2739: wiki.ipython.org contaminated with prescription drug spam
• #2588: Link error while executing code from cython example notebook
• #2550: Rpush magic doesn’t find local variables and doesn’t support comma separated lists of variables
• #2675: Markdown/html blockquote need css.
• #2419: TerminalInteractiveShell.__init__() ignores value of ipython_dir argument
• #1523: Better LaTeX printing in the qtconsole with the sympy profile
• #2719: ipython fails with pkg_resources.DistributionNotFound: ipython==0.13
• #2715: url crashes nbviewer.ipython.org
• #2555: “import” module completion on MacOSX
• #2707: Problem installing the new version of IPython in Windows
• #2696: SymPy magic bug in IPython Notebook
• #2684: pretty print broken for types created with PyType_FromSpec
• #2533: rmagic breaks on Windows
• #2661: Qtconsole tooltip is too wide when the function has many arguments
• #2679: ipython3 qtconsole via Homebrew on Mac OS X 10.8 - pyqt/pyside import error
• #2646: pylab_not_importable
• #2587: cython magic pops 2 CLI windows upon execution on Windows

2.10. Issues closed in the 1.0 development cycle
• #2660: Certain arguments (-h, –help, –version) never passed to scripts run with ipython
• #2665: Missing docs for rmagic and some other extensions
• #2611: Travis wants to drop 3.1 support
• #2658: Incorrect parsing of raw multiline strings
• #2655: Test fails if from __future__ import print_function in .pythonrc.py
• #2651: nonlocal with no existing variable produces too many errors
• #2645: python3 is a pain (minor unicode bug)
• #2637: %paste in Python 3 on Mac doesn’t work
• #2624: Error on launching IPython on Win 7 and Python 2.7.3
• #2608: disk IO activity on cursor press
• #1275: Markdown parses LaTeX math symbols as its formatting syntax in notebook
• #2613: display(Math(...)) doesn’t render tau correctly
• #925: Tab-completion in Qt console needn’t use pager
• #2607: %load_ext sympy.interactive.ipythonprinting dammaging output
• #2593: Toolbar button to open qtconsole from notebook
• #2602: IPython html documentation for downloading
• #2598: ipython notebook –pylab=inline replaces built-in any()
• #2244: small issue: wrong printout
• #2590: add easier way to execute scripts in the current directory
• #2581: %hist does not work when InteractiveShell.cache_size = 0
• #2584: No file COPYING
• #2578: AttributeError: ‘module’ object has no attribute ‘TestCase’
• #2576: One of my notebooks won’t load any more – is there a maximum notebook size?
• #2560: Notebook output is invisible when printing strings with rrn line endings
• #2566: if pyside partially present ipython qtconsole fails to load even if pyqt4 present
• #1308: ipython qtconsole –ssh=server –existing ... hangs
• #1679: List command doesn’t work in ipdb debugger the first time
• #2545: pypi win32 installer creates 64bit executibles
• #2080: Event loop issues with IPython 0.12 and PyQt4 (QDialog.exec_ and more)
• #2541: Allow python -m IPython
• #2508: subplots_adjust() does not work correctly in ipython notebook
• #2289: Incorrect mathjax rendering of certain arrays of equations
• #2487: Selecting and indenting
• #2521: more fine-grained ‘run’ controls, such as ‘run from here’ and ‘run until here’
• #2535: Funny bounding box when plot with text
• #2523: History not working
• #2514: Issue with zooming in qtconsole
• #2220: No sys.stdout.encoding in kernel based IPython
• #2512: ERROR: Internal Python error in the inspect module.
• #2496: Function passwd does not work in QtConsole
• #1453: make engines reconnect/die when controller was restarted
• #2481: ipython notebook – clicking in a code cell’s output moves the screen to the top of the code cell
• #2488: Undesired plot outputs in Notebook inline mode
• #2482: ipython notebook – download may not get the latest notebook
• #2471: _subprocess module removed in Python 3.3
• #2374: Issues with man pages
• #2316: parallel.Client.__init__ should take cluster_id kwarg
• #2457: Can a R library wrapper be created with Rmagic?
• #1575: Fallback frontend for console when connecting pylab=inline-enabled kernel?
• #2097: Do not crash if history db is corrupted
• #2435: ipengines fail if clean_logs enabled
• #2429: Using warnings.warn() results in TypeError
• #2422: Multiprocessing in ipython notebook kernel crash
• #2426: ipython crashes with the following message. I do not what went wrong. Can you help me identify the problem?
• #2423: Docs typo?
• #2257: pip install -e fails
• #2418: rmagic can’t run R’s read.csv on data files with NA data
• #2417: HTML notebook: Backspace sometimes deletes multiple characters
• #2275: notebook: “Down_Arrow” on last line of cell should move to end of line
• #2414: 0.13.1 does not work with current EPD 7.3-2
• #2409: there is a redundant None
• #2410: Use /usr/bin/python3 instead of /usr/bin/python
• #2366: Notebook Dashboard –notebook-dir and fullpath

2.10. Issues closed in the 1.0 development cycle
• #2406: Inability to get docstring in debugger
• #2398: Show line number for IndentationErrors
• #2314: HTML lists seem to interfere with the QtConsole display
• #1688: unicode exception when using %run with failing script
• #1884: IPython.embed changes color on error
• #2381: %time doesn’t work for multiline statements
• #1435: Add size keywords in Image class
• #2372: interactiveshell.py misses urllib and io_open imports
• #2371: iPython not working
• #2367: Tab expansion moves to next cell in notebook
• #2359: nbviewer alters the order of print and display() output
• #2227: print name for IPython Notebooks has become uninformative
• #2361: client doesn’t use connection file’s ‘location’ in disambiguating ‘interface’
• #2357: failing traceback in terminal ipython for first exception
• #2343: Installing in a python 3.3b2 or python 3.3rc1 virtual environment.
• #2315: Failure in test: “Test we’re not loading modules on startup that we shouldn’t.”
• #2351: Multiple Notebook Apps: cookies not port specific, clash with each other
• #2350: running unittest from qtconsole prints output to terminal
• #2303: remote tracebacks broken since 952d0d6 (PR #2223)
• #2330: qtconsole does not hightlight tab-completion suggestion with custom stylesheet
• #2325: Parsing Tex formula fails in Notebook
• #2324: Parsing Tex formula fails
• #1474: Add argument to run -n for custom namespace
• #2318: C-m n/p don’t work in Markdown cells in the notebook
• #2309: time.time() in ipython notebook producing impossible results
• #2307: schedule tasks on newly arrived engines
• #2313: Allow Notebook HTML/JS to send messages to Python code
• #2304: ipengine throws KeyError: url
• #1878: shell access using ! will not fill class or function scope vars
• #2253: %paste does not retrieve clipboard contents under screen/tmux on OS X
• #1510: Add-on (or Monkey-patch) infrastructure for HTML notebook
• #2273: triple quote and %s at beginning of line with %paste
• #2243: Regression in .embed()
• #2266: SSH passwordless check with OpenSSH checks for the wrong thing
• #2217: Change NewNotebook handler to use 30x redirect
• #2276: config option for disabling history store
• #2239: can’t use parallel.Reference in view.map
• #2272: Sympy piecewise messed up rendering
• #2252: %paste throws an exception with empty clipboard
• #2259: git-mpr is currently broken
• #2247: Variable expansion in shell commands should work in substrings
• #2026: Run ‘fast’ tests only
• #2241: read a list of notebooks on server and bring into browser only notebook
• #2237: please put python and text editor in the web only ipython
• #2053: Improvements to the IPython.display.Image object
• #1456: ERROR: Internal Python error in the inspect module.
• #2221: Avoid importing from IPython.parallel in core
• #2213: Can’t trigger startup code in Engines
• #1464: Strange behavior for backspace with lines ending with more than 4 spaces in notebook
• #2187: NaN in object_info_reply JSON causes parse error
• #214: system command requiring administrative privileges
• #2195: Unknown option no-edit in git-mpr
• #2201: Add documentation build to tools/test_pr.py
• #2205: Command-line option for default Notebook output collapsing behavior
• #1927: toggle between inline and floating figures
• #2171: Can’t start StarCluster after upgrading to IPython 0.13
• #2173: oct2py v >= 0.3.1 doesn’t need h5py anymore
• #2099: storemagic needs to use self.shell
• #2166: DirectView map_sync() with Lambdas Using Generators
• #2091: Unable to use print_stats after %prun -r in notebook
• #2132: Add fail-over for pastebin
• #2156: Make it possible to install ipython without nasty gui dependencies
• #2154: Scrolled long output should be off in print view by default
• #2162: Tab completion does not work with IPython.embed_kernel()
• #2157: iPython 0.13 / github-master cannot create logfile from scratch
• #2151: missing newline when a magic is called from the qtconsole menu
• #2139: 00_notebook_tour Image example broken on master
• #2143: Add a %%cython_annotate magic
• #2135: Running IPython from terminal
• #2093: Makefile for building Sphinx documentation on Windows
• #2122: Bug in pretty printing
• #2120: Notebook “Make a Copy...” keeps opening duplicates in the same tab
• #1997: password cannot be used with url prefix
• #2129: help/doc displayed multiple times if requested in loop
• #2121: ipdb does not support input history in qtconsole
• #2114: %logstart doesn’t log
• #2085: %ed magic fails in qtconsole
• #2119: iPython fails to run on MacOS Lion
• #2052: %pylab inline magic does not work on windows
• #2111: Ipython won’t start on W7
• #2112: Strange internal traceback
• #2108: Backslash () at the end of the line behavior different from default Python
• #1425: Ampersands can’t be typed sometimes in notebook cells
• #1513: Add expand/collapse support for long output elements like stdout and tracebacks
• #2087: error when starting ipython
• #2103: Ability to run notebook file from commandline
• #2082: Qt Console output spacing
• #2083: Test failures with Python 3.2 and PYTHONWARNINGS="d"
• #2094: about inline
• #2077: Starting IPython3 on the terminal
• #1760: easy_install ipython fails on py3.2-win32
• #2075: Local Mathjax install causes iptest3 error under python3
• #2057: setup fails for python3 with LANG=C
• #2070: shebang on Windows
• #2054: sys_info missing git hash in sdists
• #2059: duplicate and modified files in documentation
• #2056: except-shadows-builtins osm.py:687
• #2058: hyphen-used-as-minus-sign in manpages

2.11 0.13 Series

2.11.1 Release 0.13

IPython 0.13 contains several major new features, as well as a large amount of bug and regression fixes. The previous version (0.12) was released on December 19 2011, and in this development cycle we had:

• ~6 months of work.
• 373 pull requests merged.
• 742 issues closed (non-pull requests).
• contributions from 62 authors.
• 1760 commits.
• a diff of 114226 lines.

The amount of work included in this release is so large, that we can only cover here the main highlights; please see our detailed release statistics for links to every issue and pull request closed on GitHub as well as a full list of individual contributors.

Major Notebook improvements: new user interface and more

The IPython Notebook, which has proven since its release to be wildly popular, has seen a massive amount of work in this release cycle, leading to a significantly improved user experience as well as many new features.

The first user-visible change is a reorganization of the user interface; the left panel has been removed and was replaced by a real menu system and a toolbar with icons. Both the toolbar and the header above the menu can be collapsed to leave an unobstructed working area:
The notebook handles very long outputs much better than before (this was a serious usability issue when running processes that generated massive amounts of output). Now, in the presence of outputs longer than \sim 100 lines, the notebook will automatically collapse to a scrollable area and the entire left part of this area controls the display: one click in this area will expand the output region completely, and a double-click will hide it completely. This figure shows both the scrolled and hidden modes:
Note: The auto-folding of long outputs is disabled in Firefox due to bugs in its scrolling behavior. See PR #2047 for details.

Uploading notebooks to the dashboard is now easier: in addition to drag and drop (which can be finicky sometimes), you can now click on the upload text and use a regular file dialog box to select notebooks to upload. Furthermore, the notebook dashboard now auto-refreshes its contents and offers buttons to shut down any running kernels (PR #1739):
Cluster management

The notebook dashboard can now also start and stop clusters, thanks to a new tab in the dashboard user interface:

![Dashboard interface showing cluster management](image)

This interface allows, for each profile you have configured, to start and stop a cluster (and optionally override the default number of engines corresponding to that configuration). While this hides all error reporting, once you have a configuration that you know works smoothly, it is a very convenient interface for controlling your parallel resources.

New notebook format

The notebooks saved now use version 3 of our format, which supports heading levels as well as the concept of ‘raw’ text cells that are not rendered as Markdown. These will be useful with converters we are developing, to pass raw markup (say LaTeX). That conversion code is still under heavy development and not quite ready for prime time, but we welcome help on this front so that we can merge it for full production use as soon as possible.

**Note:** v3 notebooks can *not* be read by older versions of IPython, but we provide a simple script that you can use in case you need to export a v3 notebook to share with a v2 user.

JavaScript refactoring

All the client-side JavaScript has been decoupled to ease reuse of parts of the machinery without having to build a full-blown notebook. This will make it much easier to communicate with an IPython kernel from existing web pages and to integrate single cells into other sites, without loading the full notebook document-like UI. PR #1711.

This refactoring also enables the possibility of writing dynamic javascript widgets that are returned from Python code and that present an interactive view to the user, with callbacks in Javascript executing calls to
the Kernel. This will enable many interactive elements to be added by users in notebooks.

An example of this capability has been provided as a proof of concept in examples/widgets that lets you directly communicate with one or more parallel engines, acting as a mini-console for parallel debugging and introspection.

**Improved tooltips**

The object tooltips have gained some new functionality. By pressing tab several times, you can expand them to see more of a docstring, keep them visible as you fill in a function’s parameters, or transfer the information to the pager at the bottom of the screen. For the details, look at the example notebook 01_notebook_introduction.ipynb.

**Other improvements to the Notebook**

These are some other notable small improvements to the notebook, in addition to many bug fixes and minor changes to add polish and robustness throughout:

- The notebook pager (the area at the bottom) is now resizeable by dragging its divider handle, a feature that had been requested many times by just about anyone who had used the notebook system. PR #1705.

- It is now possible to open notebooks directly from the command line; for example: `ipython notebook path/` will automatically set `path/` as the notebook directory, and `ipython notebook path/foo.ipynb` will further start with the `foo.ipynb` notebook opened. PR #1686.

- If a notebook directory is specified with `--notebook-dir` (or with the corresponding configuration flag `NotebookManager.notebook_dir`), all kernels start in this directory.

- Fix codemirror clearing of cells with `Ctrl-Z`; PR #1965.

- Text (markdown) cells now line wrap correctly in the notebook, making them much easier to edit PR #1330.

- PNG and JPEG figures returned from plots can be interactively resized in the notebook, by dragging them from their lower left corner. PR #1832.

- Clear In [] prompt numbers on “Clear All Output”. For more version-control-friendly .ipynb files, we now strip all prompt numbers when doing a “Clear all output”. This reduces the amount of noise in commit-to-commit diffs that would otherwise show the (highly variable) prompt number changes. PR #1621.

- The notebook server now requires two consecutive `Ctrl-C` within 5 seconds (or an interactive confirmation) to terminate operation. This makes it less likely that you will accidentally kill a long-running server by typing `Ctrl-C` in the wrong terminal. PR #1609.

- Using `Ctrl-S` (or `Cmd-S` on a Mac) actually saves the notebook rather than providing the fairly useless browser html save dialog. PR #1334.
Fig. 2.1: The new notebook tooltips.
• Allow accessing local files from the notebook (in urls), by serving any local file as the url `files/<relativepath>`. This makes it possible to, for example, embed local images in a notebook. PR #1211.

Cell magics

We have completely refactored the magic system, finally moving the magic objects to standalone, independent objects instead of being the mixin class we’d had since the beginning of IPython (PR #1732). Now, a separate base class is provided in `IPython.core.magic.Magics` that users can subclass to create their own magics. Decorators are also provided to create magics from simple functions without the need for object orientation. Please see the `Magic command system` docs for further details.

All builtin magics now exist in a few subclasses that group together related functionality, and the new `IPython.core.magics` package has been created to organize this into smaller files.

This cleanup was the last major piece of deep refactoring needed from the original 2001 codebase.

We have also introduced a new type of magic function, prefixed with `%%` instead of `%`, which operates at the whole-cell level. A cell magic receives two arguments: the line it is called on (like a line magic) and the body of the cell below it.

Cell magics are most natural in the notebook, but they also work in the terminal and qt console, with the usual approach of using a blank line to signal cell termination.

For example, to time the execution of several statements:

```python
%%timeit x = 0  # setup
for i in range(100000):
    x += i**2
```

This is particularly useful to integrate code in another language, and cell magics already exist for shell scripts, Cython, R and Octave. Using `%%script /usr/bin/foo`, you can run a cell in any interpreter that accepts code via stdin.

Another handy cell magic makes it easy to write short text files: `%%file ~/save/to/here.txt`.

The following cell magics are now included by default; all those that use special interpreters (Perl, Ruby, bash, etc.) assume you have the requisite interpreter installed:

- `%%!`: run cell body with the underlying OS shell; this is similar to prefixing every line in the cell with `!`.
- `%%bash`: run cell body under bash.
- `%%capture`: capture the output of the code in the cell (and stderr as well). Useful to run codes that produce too much output that you don’t even want scrolled.
- `%%file`: save cell body as a file.
- `%%perl`: run cell body using Perl.
- `%%prun`: run cell body with profiler (cell extension of `%prun`).
- `%%python3`: run cell body using Python 3.
• `%%ruby`: run cell body using Ruby.
• `%%script`: run cell body with the script specified in the first line.
• `%%sh`: run cell body using sh.
• `%%sx`: run cell with system shell and capture process output (cell extension of `%%sx`).
• `%%system`: run cell with system shell (`%%!` is an alias to this).
• `%%timeit`: time the execution of the cell (extension of `%%timeit`).

This is what some of the script-related magics look like in action:

IPython also creates aliases for a few common interpreters, such as bash, ruby, perl, etc.

These are all equivalent to `%%script <name>`

```
In [4]: %ruby
puts "Hello from Ruby #{RUBY_VERSION}"
Hello from Ruby 1.8.7
```

```
In [5]: %bash
echo "hello from $BASH"
hello from /usr/local/bin/bash
```

In addition, we have also a number of extensions that provide specialized magics. These typically require additional software to run and must be manually loaded via `%load_ext <extension name>`, but are extremely useful. The following extensions are provided:

**Cython magics (extension cythonmagic)** This extension provides magics to automatically build and compile Python extension modules using the Cython language. You must install Cython separately, as well as a C compiler, for this to work. The examples directory in the source distribution ships with a full notebook demonstrating these capabilities:
The %cython magic

Probably the most important magic is the %cython magic. This is similar to the %cython_pypximport magic, but doesn't require you to specify a module name. Instead, the %cython magic uses manages everything using temporary files in the ~/.cython/magic directory. All of the symbols in the Cython module are imported automatically by the magic.

Here is a simple example of a Black-Scholes options pricing algorithm written in Cython:

```python
In [6]: %cython
import cython
from libc.math cimport exp, sqrt, pow, log, erf

cdef double std_norm_cdf(double x) nogil:
    return 0.5*(1+erf(x/sqrt(2.0)))

def black_scholes(double s, double k, double t, double v,
                  double rf, double div, double cp):
    """Price an option using the Black-Scholes model.

    s : initial stock price
    k : strike price
    t : expiration time
    v : volatility
    rf : risk-free rate
    div : dividend
    cp : +1/-1 for call/put
    """

    cdef double d1, d2, optprice
    with nogil:
        d1 = (log(s/k)+(rf-div+0.5*pow(v,2))*t)/(v*sqrt(t))
        d2 = d1 - v*sqrt(t)
        optprice = cp*s*exp(-div*t)*std_norm_cdf(cp*d1) - \
                    cp*k*exp(-rf*t)*std_norm_cdf(cp*d2)
    return optprice

In [7]: black_scholes(100.0, 100.0, 1.0, 0.3, 0.03, 0.0, -1)
```

```
Out[7]: 10.327861752731728
```

```
In [8]: %timeit black_scholes(100.0, 100.0, 1.0, 0.3, 0.03, 0.0, -1)
```

```
1000000 loops, best of 3: 821 ns per loop
```

Octave magics (extension octavemagic) This extension provides several magics that support calling code written in the Octave language for numerical computing. You can execute single-lines or whole blocks of Octave code, capture both output and figures inline (just like matplotlib plots), and have variables automatically converted between the two languages. To use this extension, you must have Octave installed as well as the oct2py package. The examples directory in the source distribution ships with a full notebook demonstrating these capabilities:
In [24]: `octave -s 500,500`

# butterworth filter, order 2, cutoff pi/2 radians
b = [0.292893218813452 0.585786437626905 0.292893218813452];
a = [1 0 0.171572875253810];
freqz(b, a, 32);

In [25]: `octave -s 600,200 -f png`

subplot(121);
[x, y] = meshgrid(0:0.1:3);
r = sin(x - 0.5).^2 + cos(y - 0.5).^2;
surf(x, y, r);
subplot(122);
sombrero()
R magics (extension rmagic) This extension provides several magics that support calling code written in the R language for statistical data analysis. You can execute single-lines or whole blocks of R code, capture both output and figures inline (just like matplotlib plots), and have variables automatically converted between the two languages. To use this extension, you must have R installed as well as the rpy2 package that bridges Python and R. The examples directory in the source distribution ships with a full notebook demonstrating these capabilities:
We can redo the above calculations in one ipython cell. We might also want to add some output such as a summary from R or perhaps the standard plotting diagnostics of the lm.

```python
In [16]: %R -i x y -o xycoef
   :  X Ylm = lm(Y~X)
   :  XYcoef = coef(XYlm)
   :  print(summary(XYlm))
   :  par(mfrow=c(2,2))
   :  plot(XYlm)
```

```
Call:
  lm(formula = Y ~ X)

Residuals:
   1     2     3     4     5
    0.3   0.9  -1.0   0.1   0.2

Coefficients:         Estimate  Std. Error t value Pr(>|t|)
  (Intercept)  3.2000      0.6164    5.191    0.0139 *
   X          0.0000      0.2517    0.3576   0.0374 *
---
Signif. codes:  0 ‘****’ 0.001 ‘***’ 0.01 ‘**’ 0.05 ‘*’ 0.1 ‘.’ 1

Residual standard error: 0.7958 on 3 degrees of freedom
Multiple R-squared: 0.81,  Adjusted R-squared: 0.7467
F-statistic: 12.79 on 1 and 3 DF,  p-value: 0.03739
```
Tab completer improvements

Useful tab-completion based on live inspection of objects is one of the most popular features of IPython. To make this process even more user-friendly, the completers of both the Qt console and the Notebook have been reworked.

The Qt console comes with a new ncurses-like tab completer, activated by default, which lets you cycle through the available completions by pressing tab, or select a completion with the arrow keys (PR #1851).

```
In [2]: import numpy.random as random
```

```
In [3]: numpy.random.<tab>                                    
   ...    numpy.random.binomial                                
   ...    numpy.random.lognormal                               
   ...    numpy.random.logseries     standard_normal          
   ...    numpy.random.chisquare      standard_t              
   ...    numpy.random.dirichlet    standard_normal          
   ...    numpy.random.exponential standard gamma             
   ...    numpy.random.f              standard_t              
   ...    numpy.random.mtrand       standard_normal          
   ...    numpy.random.multinomial  triangular                
   ...    numpy.random.mtrand       uniform                   
   ...    numpy.random.random_integers standard_normal       
   ...    numpy.random.random_sample standard_normal          
```

Fig. 2.2: The new improved Qt console’s ncurses-like completer allows to easily navigate thought long list of completions.

In the notebook, completions are now sourced both from object introspection and analysis of surrounding code, so limited completions can be offered for variables defined in the current cell, or while the kernel is busy (PR #1711).

We have implemented a new configurable flag to control tab completion on modules that provide the `__all__` attribute:

```
IPCompleter.limit_to__all__= Boolean
```

This instructs the completer to honor `__all__` for the completion. Specifically, when completing on `object.<tab>`, if True: only those names in `obj.__all__` will be included. When False [default]: the `__all__` attribute is ignored. PR #1529.

Improvements to the Qt console

The Qt console continues to receive improvements and refinements, despite the fact that it is by now a fairly mature and robust component. Lots of small polish has gone into it, here are a few highlights:

- A number of changes were made to the underlying code for easier integration into other projects such as Spyder (PR #2007, PR #2024).
• Improved menus with a new Magic menu that is organized by magic groups (this was made possible by the reorganization of the magic system internals). PR #1782.

• Allow for restarting kernels without clearing the qtconsole, while leaving a visible indication that the kernel has restarted. PR #1681.

• Allow the native display of jpeg images in the qtconsole. PR #1643.

Parallel

The parallel tools have been improved and fine-tuned on multiple fronts. Now, the creation of an IPython.parallel.Client object automatically activates a line and cell magic function px that sends its code to all the engines. Further magics can be easily created with the Client.activate() method, to conveniently execute code on any subset of engines. PR #1893.

The %%px cell magic can also be given an optional targets argument, as well as a --out argument for storing its output.

A new magic has also been added, %pxconfig, that lets you configure various defaults of the parallel magics. As usual, type %pxconfig? for details.

The exception reporting in parallel contexts has been improved to be easier to read. Now, IPython directly reports the remote exceptions without showing any of the internal execution parts:

```
In [1]: from IPython.parallel import Client
c = Client()
```

```
In [2]: %px 1/0
```

```
[0:execute]:                                                                                       
ZeroDivisionError                               Traceback (most recent call                       
last)<ipython-input-1-05c9758a9c21> in <module>()                                               
----> 1 1/0                                       
ZeroDivisionError: integer division or modulo by zero
```

```
[1:execute]:                                                                                       
ZeroDivisionError                               Traceback (most recent call                       
last)<ipython-input-1-05c9758a9c21> in <module>()                                               
----> 1 1/0                                       
ZeroDivisionError: integer division or modulo by zero
```

The parallel tools now default to using NoDB as the storage backend for intermediate results. This means that the default usage case will have a significantly reduced memory footprint, though certain advanced features are not available with this backend.

The parallel magics now display all output, so you can do parallel plotting or other actions with complex display. The px magic has now both line and cell modes, and in cell mode finer control has been added about how to collate output from multiple engines. PR #1768.

There have also been incremental improvements to the SSH launchers:

• add to_send/fetch steps for moving connection files around.
• add SSHProxyEngineSetLauncher, for invoking ipcluster engines on a remote host. This can be used to start a set of engines via PBS/SGE/MPI remotely.

This makes the SSHLauncher usable on machines without shared filesystems.

A number of ‘sugar’ methods/properties were added toAsyncResult that are quite useful (PR #1548) for everyday work:

• ar.wall_time = received - submitted
• ar.serial_time = sum of serial computation time
• ar.elapsed = time since submission (wall_time if done)
• ar.progress = (int) number of sub-tasks that have completed
• len(ar) = # of tasks
• ar.wait_interactive(): prints progress

Added Client.spin_thread() / stop_spin_thread() for running spin in a background thread, to keep zmq queue clear. This can be used to ensure that timing information is as accurate as possible (at the cost of having a background thread active).

Set TaskScheduler.hwm default to 1 instead of 0. 1 has more predictable/intuitive behavior, if often slower, and thus a more logical default. Users whose workloads require maximum throughput and are largely homogeneous in time per task can make the optimization themselves, but now the behavior will be less surprising to new users. PR #1294.

**Kernel/Engine unification**

This is mostly work ‘under the hood’, but it is actually a major achievement for the project that has deep implications in the long term: at last, we have unified the main object that executes as the user’s interactive shell (which we refer to as the IPython kernel) with the objects that run in all the worker nodes of the parallel computing facilities (the IPython engines). Ever since the first implementation of IPython’s parallel code back in 2006, we had wanted to have these two roles be played by the same machinery, but a number of technical reasons had prevented that from being true.

In this release we have now merged them, and this has a number of important consequences:

• It is now possible to connect any of our clients (qtconsole or terminal console) to any individual parallel engine, with the exact behavior of working at a ‘regular’ IPython console/qtconsole. This makes debugging, plotting, etc. in parallel scenarios vastly easier.

• Parallel engines can always execute arbitrary ‘IPython code’, that is, code that has magics, shell extensions, etc. In combination with the %%px magics, it is thus extremely natural for example to send to all engines a block of Cython or R code to be executed via the new Cython and R magics. For example, this snippet would send the R block to all active engines in a cluster:

```latex
%%px
%%R
... R code goes here
```
• It is possible to embed not only an interactive shell with the `IPython.embed()` call as always, but now you can also embed a *kernel* with `IPython.embed_kernel()`. Embedding an IPython kernel in an application is useful when you want to use `IPython.embed()` but don’t have a terminal attached on stdin and stdout.

• The new `IPython.parallel.bind_kernel()` allows you to promote Engines to listening Kernels, and connect QtConsoles to an Engine and debug it directly.

In addition, having a single core object through our entire architecture also makes the project conceptually cleaner, easier to maintain and more robust. This took a lot of work to get in place, but we are thrilled to have this major piece of architecture finally where we’d always wanted it to be.

**Official Public API**

We have begun organizing our API for easier public use, with an eye towards an official IPython 1.0 release which will firmly maintain this API compatible for its entire lifecycle. There is now an `IPython.display` module that aggregates all display routines, and the `traitlets.config` namespace has all public configuration tools. We will continue improving our public API layout so that users only need to import names one level deeper than the main `IPython` package to access all public namespaces.

**IPython notebook file icons**

The directory `docs/resources` in the source distribution contains SVG and PNG versions of our file icons, as well as an `Info.plist.example` file with instructions to install them on Mac OSX. This is a first draft of our icons, and we encourage contributions from users with graphic talent to improve them in the future.

**New top-level locate command**

Add `locate` entry points; these would be useful for quickly locating IPython directories and profiles from other (non-Python) applications. PR #1762.

Examples:

```
$> ipython locate
/Users/me/.ipython

$> ipython locate profile foo
/Users/me/.ipython/profile_foo

$> ipython locate profile
/Users/me/.ipython/profile_default

$> ipython locate profile dne
[ProfileLocate] Profile u'dne' not found.
```
Other new features and improvements

• `%install_ext`: A new magic function to install an IPython extension from a URL. E.g. `%install_ext https://bitbucket.org/birkenfeld/ipython-physics/raw/default/physics.py`.

• The `%loadpy` magic is no longer restricted to Python files, and has been renamed `%load`. The old name remains as an alias.

• New command line arguments will help external programs find IPython folders: `ipython locate` finds the user’s IPython directory, and `ipython locate profile foo` finds the folder for the ‘foo’ profile (if it exists).

• The `IPYTHON_DIR` environment variable, introduced in the Great Reorganization of 0.11 and existing only in versions 0.11-0.13, has been deprecated. As described in PR #1167, the complexity and confusion of migrating to this variable is not worth the aesthetic improvement. Please use the historical `IPYTHONDIR` environment variable instead.

• The default value of `interactivity` passed from `run_cell()` to `run_ast_nodes()` is now configurable.

• New `%alias_magic` function to conveniently create aliases of existing magics, if you prefer to have shorter names for personal use.

• We ship unminified versions of the JavaScript libraries we use, to better comply with Debian’s packaging policies.

• Simplify the information presented by `obj?/obj??` to eliminate a few redundant fields when possible. PR #2038.

• Improved continuous integration for IPython. We now have automated test runs on Shining Panda and Travis-CI, as well as Tox support.

• The vim-ipython functionality (externally developed) has been updated to the latest version.

• The `%save` magic now has a `-f` flag to force overwriting, which makes it much more usable in the notebook where it is not possible to reply to interactive questions from the kernel. PR #1937.

• Use dvipng to format sympy.Matrix, enabling display of matrices in the Qt console with the sympy printing extension. PR #1861.

• Our messaging protocol now has a reasonable test suite, helping ensure that we don’t accidentally deviate from the spec and possibly break third-party applications that may have been using it. We encourage users to contribute more stringent tests to this part of the test suite. PR #1627.

• Use LaTeX to display, on output, various built-in types with the SymPy printing extension. PR #1399.

• Add Gtk3 event loop integration and example. PR #1588.

• `clear_output` improvements, which allow things like progress bars and other simple animations to work well in the notebook (PR #1563):
  
  - `clear_output()` clears the line, even in terminal IPython, the QtConsole and plain Python as well, by printing `r` to streams.
  
  - `clear_output()` avoids the flicker in the notebook by adding a delay, and firing immediately upon the next actual display message.
- `display_javascript` hides its `output_area` element, so using `display` to run a bunch of javascript doesn’t result in ever-growing vertical space.

- Add simple support for running inside a virtualenv. While this doesn’t supplant proper installation (as users should do), it helps ad-hoc calling of IPython from inside a virtualenv. PR #1388.

**Major Bugs fixed**

In this cycle, we have *closed over 740 issues*, but a few major ones merit special mention:

- The `%pastebin` magic has been updated to point to gist.github.com, since unfortunately http://paste.pocoo.org has closed down. We also added a `-d` flag for the user to provide a gist description string. PR #1670.

- Fix `%paste` that would reject certain valid inputs. PR #1258.

- Fix sending and receiving of Numpy structured arrays (those with composite dtypes, often used as recarrays). PR #2034.

- Reconnect when the websocket connection closes unexpectedly. PR #1577.

- Fix truncated representation of objects in the debugger by showing at least 80 characters’ worth of information. PR #1793.

- Fix logger to be Unicode-aware: logging could crash ipython if there was unicode in the input. PR #1792.

- Fix images missing from XML/SVG export in the Qt console. PR #1449.

- Fix deepreload on Python 3. PR #1625, as well as having a much cleaner and more robust implementation of deepreload in general. PR #1457.

**Backwards incompatible changes**

- The exception `IPython.core.error.TryNext` previously accepted arguments and keyword arguments to be passed to the next implementation of the hook. This feature was removed as it made error message propagation difficult and violated the principle of loose coupling.

**2.12 Issues closed in the 0.13 development cycle**

**2.12.1 Issues closed in 0.13**

GitHub stats since IPython 0.12 (2011/12/19 - 2012/06/30)

These lists are automatically generated, and may be incomplete or contain duplicates.

The following 62 authors contributed 1760 commits.

- Aaron Culich
- Aaron Meurer
2.12. Issues closed in the 0.13 development cycle
• Paul
• Paul Ivanov
• Piotr Zolnierczuk
• Piti Ongmongkolkul
• Puneeth Chaganti
• Robert Kern
• Ross Jones
• Roy Hyunjin Han
• Scott Tsai
• Skipper Seabold
• Stefan van der Walt
• Steven Johnson
• Takafumi Arakaki
• Ted Wright
• Thomas Hisch
• Thomas Kluyver
• Thomas Spura
• Thomi Richards
• Tim Couper
• Timo Paulssen
• Toby Gilham
• Tony S Yu
• 23. Trevor King
• Walter Doerwald
• anatoly techtonik
• fawce
• mcelrath
• wilsaj

We closed a total of 1115 issues, 373 pull requests and 742 regular issues; this is the full list (generated with the script `tools/github_stats.py`):

Pull Requests (373):

- PR #1943: add screenshot and link into releasenotes
• PR #1954: update some example notebooks
• PR #2048: move _encode_binary to jsonutil.encode_images
• PR #2050: only add quotes around xunit-file on Windows
• PR #2047: disable auto-scroll on mozilla
• PR #2015: Fixes for %paste with special transformations
• PR #2046: Iptest unicode
• PR #1939: Namespaces
• PR #2042: increase auto-scroll threshold to 100 lines
• PR #2043: move RemoteError import to top-level
• PR #2036: %alias_magic
• PR #1968: Proposal of icons for .ipynb files
• PR #2037: remove ipython-qtconsole gui-script
• PR #2038: add extra clear warning to shell doc
• PR #2029: Ship unminified js
• PR #2007: Add custom_control and custom_page_control variables to override the Qt widgets used by qtconsole
• PR #2034: fix&test push/pull recarrays
• PR #2028: Reduce unhelpful information shown by pinfo
• PR #2030: check wxPython version in inputhook
• PR #2024: Make interactive_usage a bit more rst friendly
• PR #2031: disable ^C^C confirmation on Windows
• PR #2027: match stdin encoding in frontend readline test
• PR #2025: Fix parallel test on WinXP - wait for resource cleanup.
• PR #2016: BUG: test runner fails in Windows if filenames contain spaces.
• PR #2020: Fix home path expansion test in Windows.
• PR #2021: Fix Windows pathname issue in ‘odd encoding’ test.
• PR #2022: don’t check writability in test for get_home_dir when HOME is undefined
• PR #1996: frontend test tweaks
• PR #2014: relax profile regex in notebook
• PR #2012: Mono cursor offset
• PR #2004: Clarify generic message spec vs. Python message API in docs
• PR #2010: notebook: Print a warning (but do not abort) if no webbrowser can be found.

2.12. Issues closed in the 0.13 development cycle
• PR #2002: Refactor %magic into a lsmagic_docs API function.
• PR #1999: %magic help: display line and cell magics in alphabetical order.
• PR #1981: Clean BG processes created by %%script on kernel exit
• PR #1994: Fix RST misformatting.
• PR #1951: minor notebook startup/notebook-dir adjustments
• PR #1974: Allow path completion on notebook.
• PR #1964: allow multiple instances of a Magic
• PR #1991: fix _ofind attr in %page
• PR #1988: check for active frontend in update_restart_checkbox
• PR #1979: Add support for tox (http://tox.testrun.org/) and Travis CI (http://travis-ci.org/)
• PR #1970: dblclick to restore size of images
• PR #1978: Notebook names truncating at the first period
• PR #1825: second attempt at scrolled long output
• PR #1934: Cell/Worksheet metadata
• PR #1746: Confirm restart (configuration option, and checkbox UI)
• PR #1944: [qtconsole] take %,%% prefix into account for completion
• PR #1973: fix another FreeBSD $HOME symlink issue
• PR #1967: Fix psums example description in docs
• PR #1965: fix for #1678, undo no longer clears cells
• PR #1952: avoid duplicate “Websockets closed” dialog on ws close
• PR #1962: Support unicode prompts
• PR #1955: update to latest version of vim-ipython
• PR #1945: Add –proc option to %%script
• PR #1956: move import RemoteError after get_exc_info
• PR #1950: Fix for copy action (Ctrl+C) when there is no pager defined in qtconsole
• PR #1948: Fix help string for InteractiveShell.ast_node_interactivity
• PR #1942: swallow stderr of which in utils.process.find_cmd
• PR #1940: fix completer css on some Chrome versions
• PR #1938: remove remaining references to deprecated XREP/XREQ names
• PR #1925: Fix styling of superscripts and subscripts. Closes #1924.
• PR #1936: increase duration of save messages
• PR #1937: add %save -f
• PR #1935: add version checking to pyreadline import test
• PR #1849: Octave magics
• PR #1759: github, merge PR(s) just by number(s)
• PR #1931: Win py3fixes
• PR #1933: oinspect.find_file: Additional safety if file cannot be found.
• PR #1932: Fix adding functions to CommandChainDispatcher with equal priority on Py 3
• PR #1928: Select NoDB by default
• PR #1923: Add IPython syntax support to the %timeit magic, in line and cell mode
• PR #1926: Make completer recognize escaped quotes in strings.
• PR #1893: Update Parallel Magics and Exception Display
• PR #1921: magic_arguments: dedent but otherwise preserve indentation.
• PR #1919: Use oinspect in CodeMagics._find_edit_target
• PR #1918: don’t warn in iptest if deathrow/quarantine are missing
• PR #1917: Fix for %pdef on Python 3
• PR #1913: Fix for #1428
• PR #1911: temporarily skip autoreload tests
• PR #1909: Fix for #1908, use os.path.normcase for safe filename comparisons
• PR #1907: py3compat fixes for %%script and tests
• PR #1906: ofind finds non-unique cell magics
• PR #1845: Fixes to inspection machinery for magics
• PR #1902: Workaround fix for gh-1632; minimal revert of gh-1424
• PR #1900: Cython libs
• PR #1899: add ScriptMagics to class list for generated config
• PR #1898: minimize manpages
• PR #1897: use glob for bad exclusion warning
• PR #1855: %%script and %%file magics
• PR #1870: add %%%capture for capturing stdout/err
• PR #1861: Use dvipng to format sympy.Matrix
• PR #1867: Fix 1px margin bouncing of selected menu item.
• PR #1889: Reconnect when the websocket connection closes unexpectedly
• PR #1886: Fix a bug in renaming notebook
• PR #1895: Fix error in test suite with ip.system()
- PR #1762: add `locate` entry points
- PR #1883: Fix vertical offset due to bold/italics, and bad browser fonts.
- PR #1875: re-write columnize, with intermediate step.
- PR #1851: new completer for qtconsole.
- PR #1892: Remove suspicious quotes in interactiveshell.py
- PR #1864: Rmagic exceptions
- PR #1829: [notebook] don’t care about leading prct in completion
- PR #1832: Make svg, jpeg and png images resizable in notebook.
- PR #1674: HTML Notebook carriage-return handling, take 2
- PR #1882: Remove importlib dependency which not available in Python 2.6.
- PR #1879: Correct stack depth for variable expansion in !system commands
- PR #1841: [notebook] deduplicate completion results
- PR #1850: Remove args/kwarg handling in TryNext, fix %paste error messages.
- PR #1663: Keep line-endings in ipynb
- PR #1815: Make : invalid in filenames in the Notebook JS code.
- PR #1819: doc: cleanup the parallel psums example a little
- PR #1839: External cleanup
- PR #1782: fix Magic menu in qtconsole, split in groups
- PR #1862: Minor bind_kernel improvements
- PR #1857: Prevent jumping of window to input when output is clicked.
- PR #1856: Fix 1px jumping of cells and menus in Notebook.
- PR #1852: fix chained resubmissions
- PR #1780: Rmagic extension
- PR #1847: add `InlineBackend` to `ConsoleApp` class list
- PR #1836: preserve header for resubmitted tasks
- PR #1828: change default extension to .ipy for `%save -r`
- PR #1800: Reintroduce recall
- PR #1830: lsmagic lists magics in alphabetical order
- PR #1773: Update SymPy profile: SymPy’s latex() can now print set and frozenset
- PR #1761: Edited documentation to use `IPYTHONDIR` in place of `~/.ipython`
- PR #1822: aesthetics pass on `AsyncResult.display_outputs`
- PR #1821: ENTER submits the rename notebook dialog.
• PR #1820: NotebookApp: Make the number of ports to retry user configurable.
• PR #1816: Always use filename as the notebook name.
• PR #1813: Add assert_in method to nose for Python 2.6
• PR #1711: New Tooltip, New Completer and JS Refactor
• PR #1798: a few simple fixes for docs/parallel
• PR #1812: Ensure AsyncResult.display_outputs doesn’t display empty streams
• PR #1811: warn on nonexistent exclusions in iptest
• PR #1810: fix for #1809, failing tests in IPython.zmq
• PR #1808: Reposition alternate upload for firefox [need cross browser/OS/language test]
• PR #1742: Check for custom_exceptions only once
• PR #1807: add missing cython exclusion in iptest
• PR #1805: Fixed a vcvarsall.bat error on win32/Py2.7 when trying to compile with m...
• PR #1739: Dashboard improvement (necessary merge of #1658 and #1676 + fix #1492)
• PR #1770: Cython related magic functions
• PR #1707: Accept –gui=<...> switch in IPython qtconsole.
• PR #1797: Fix comment which breaks Emacs syntax highlighting.
• PR #1795: fix %gui magic
• PR #1793: Raise repr limit for strings to 80 characters (from 30).
• PR #1794: don’t use XDG path on OS X
• PR #1792: Unicode-aware logger
• PR #1791: update zmqshell magics
• PR #1787: DOC: Remove regression from qt-console docs.
• PR #1758: test_pr, fallback on http if git protocol fail, and SSL errors...
• PR #1748: Fix some tests for Python 3.3
• PR #1755: test for pygments before running qt tests
• PR #1771: Make default value of interactivity passed to run_ast_nodes configurable
• PR #1784: restore loadpy to load
• PR #1768: Update parallel magics
• PR #1779: Tidy up error raising in magic decorators.
• PR #1769: Allow cell mode timeit without setup code.
• PR #1716: Fix for fake filenames in verbose traceback
• PR #1763: [qtconsole] fix append_plain_html -> append_html

2.12. Issues closed in the 0.13 development cycle
• PR #1732: Refactoring of the magics system and implementation of cell magics
• PR #1630: Merge divergent Kernel implementations
• PR #1705: [notebook] Make pager resizable, and remember size...
• PR #1606: Share code for %pycat and %loadpy, make %pycat aware of URLs
• PR #1757: Open IPython notebook hyperlinks in a new window using target=_blank
• PR #1754: Fix typo encounters->encounters
• PR #1753: Clear window title when kernel is restarted
• PR #1449: Fix for bug #735: Images missing from XML/SVG export
• PR #1743: Tooltip completer js refactor
• PR #1681: add qt config option to clear_on_kernel_restart
• PR #1733: Tooltip completer js refactor
• PR #1727: terminate kernel after embed_kernel tests
• PR #1737: add HistoryManager to ipapp class list
• PR #1686: ENH: Open a notebook from the command line
• PR #1709: fixes #1708, failing test in arg_split on windows
• PR #1718: Use CRegExp trait for regular expressions.
• PR #1729: Catch failure in repr() for %whos
• PR #1726: use eval for command-line args instead of exec
• PR #1724: fix scatter/gather with targets='all'
• PR #1725: add –no-ff to git pull in test_pr
• PR #1721: Tooltip completer js refactor
• PR #1657: Add wait optional argument to hooks.editor
• PR #1717: Define generic sys.ps{1,2,3}, for use by scripts.
• PR #1691: Finish PR #1446
• PR #1710: update MathJax CDN url for https
• PR #1713: Make autocall regexp’s configurable.
• PR #1703: Allow TryNext to have an error message without it affecting the command chain
• PR #1714: minor adjustments to test_pr
• PR #1704: ensure all needed qt parts can be imported before settling for one
• PR #1706: Mark test_push_numpy_nocopy as a known failure for Python 3
• PR #1698: fix tooltip on token with number
• PR #1245: pythonw py3k fixes for issue #1226

Chapter 2. What’s new in IPython
• PR #1685: Add script to test pull request
• PR #1693: deprecate IPYTHON_DIR in favor of IPYTHONDIR
• PR #1695: Avoid deprecated warnings from ipython-qtc.
• PR #1694: Add quote to notebook to allow it to load
• PR #1689: Fix sys.path missing ‘’ as first entry in ipython kernel.
• PR #1687: import Binary from bson instead of pymongo
• PR #1616: Make IPython.core.display.Image less notebook-centric
• PR #1684: CLN: Remove redundant function definition.
• PR #1670: Point %pastebin to gist
• PR #1669: handle pyout messages in test_message_spec
• PR #1295: add binary-tree engine interconnect example
• PR #1642: Cherry-picked commits from 0.12.1 release
• PR #1659: Handle carriage return characters (“r”) in HTML notebook output.
• PR #1656: ensure kernels are cleaned up in embed_kernel tests
• PR #1664: InteractiveShell.run_code: Update docstring.
• PR #1662: Delay flushing softspace until after cell finishes
• PR #1643: handle jpg/jpeg in the qtconsole
• PR #1652: add patch_pyzmq() for backporting a few changes from newer pyzmq
• PR #1650: DOC: moving files with SSH launchers
• PR #1357: add IPython.embed_kernel()
• PR #1640: Finish up embed_kernel
• PR #1651: Remove bundled Itpl module
• PR #1634: incremental improvements to SSH launchers
• PR #1649: move examples/test_embed into examples/tests/embed
• PR #1633: Fix installing extension from local file on Windows
• PR #1645: Exclude UserDict when deep reloading NumPy.
• PR #1637: Removed a ‘:’ which shouldn’t have been there
• PR #1631: TST: QApplication doesn’t quit early enough with PySide.
• PR #1629: evaluate a few dangling validate_message generators
• PR #1621: clear In[] prompt numbers on “Clear All Output”
• PR #1627: Test the Message Spec
• PR #1624: Fixes for byte-compilation on Python 3

2.12. Issues closed in the 0.13 development cycle
• PR #1615: Add show() method to figure objects.
• PR #1625: Fix deepreload on Python 3
• PR #1620: pyin message now have execution_count
• PR #1457: Update deepreload to use a rewritten knee.py. Fixes dreload(numpy).
• PR #1613: allow map / parallel function for single-engine views
• PR #1609: exit notebook cleanly on SIGINT, SIGTERM
• PR #1607: cleanup sqlitedb temporary db file after tests
• PR #1608: don’t rely on timedelta.total_seconds inAsyncResult
• PR #1599: Fix for %run -d on Python 3
• PR #1602: Fix %env magic on Python 3.
• PR #1603: Remove python3 profile
• PR #1604: Exclude IPython.quarantine from installation
• PR #1600: Specify encoding for io.open in notebook_reformat tests
• PR #1605: Small fixes for Animation and Progress notebook
• PR #1529: __all__ feature, improvement to dir2, and tests for both
• PR #1548: add sugar methods/properties to AsyncResult
• PR #1535: Fix pretty printing dispatch
• PR #1399: Use LaTeX to print various built-in types with the SymPy printing extension
• PR #1597: re-enter kernel.eventloop after catching SIGINT
• PR #1490: rename plaintext cell -> raw cell
• PR #1480: Fix %notebook magic, etc. nbformat unicode tests and fixes
• PR #1588: Gtk3 integration with ipython works.
• PR #1595: Examples syntax (avoid errors installing on Python 3)
• PR #1526: Find encoding for Python files
• PR #1594: Fix writing git commit ID to a file on build with Python 3
• PR #1556: shallow-copy DictDB query results
• PR #1502: small changes in response to pyflakes pass
• PR #1445: Don’t build sphinx docs for sdists
• PR #1538: store git commit hash in utils._sysinfo instead of hidden data file
• PR #1546: attempt to suppress exceptions in channel threads at shutdown
• PR #1559: update tools/github_stats.py to use GitHub API v3
• PR #1563: clear_output improvements
• PR #1560: remove obsolete discussion of Twisted/trial from testing docs
• PR #1569: BUG: qtconsole – non-standard handling of a and b. [Fixes #1561]
• PR #1573: BUG: Ctrl+C crashes wx pylab kernel in qtconsole.
• PR #1568: fix PR #1567
• PR #1567: Fix: openssh_tunnel did not parse port in server
• PR #1565: fixAsyncResult.abort
• PR #1552: use os.getcwd in NotebookManager
• PR #1541: display_pub flushes stdout/err
• PR #1544: make MultiKernelManager.kernel_manager_class configurable
• PR #1517: Fix indentation bug in IPython/lib/pretty.py
• PR #1519: BUG: Include the name of the exception type in its pretty format.
• PR #1489: Fix zero-copy push
• PR #1477: fix dangling buffer in IPython.parallel.util
• PR #1514: DOC: Fix references to IPython.lib.pretty instead of the old location
• PR #1481: BUG: Improve placement of CallTipWidget
• PR #1496: BUG: LBYL when clearing the output history on shutdown.
• PR #1508: fix sorting profiles in clustermanager
• PR #1495: BUG: Fix pretty-printing for overzealous objects
• PR #1472: more general fix for #662
• PR #1483: updated magic_history docstring
• PR #1383: First version of cluster web service.
• PR #1398: fix %tb after SyntaxError
• PR #1440: Fix for failing testsuite when using –with-xml-coverage on windows.
• PR #1419: Add %install_ext magic function.
• PR #1424: Win32 shell interactivity
• PR #1468: Simplify structure of a Job in the TaskScheduler
• PR #1447: 1107 - Tab autocompletion can suggest invalid syntax
• PR #1469: Fix typo in comment (insert space)
• PR #1463: Fix completion when importing modules in the cwd.
• PR #1466: Fix for issue #1437, unfriendly windows qtconsole error handling
• PR #1432: Fix ipython directive
• PR #1465: allow ipython help subcommand syntax

2.12. Issues closed in the 0.13 development cycle
• PR #1416: Conditional import of ctypes in inputhook
• PR #1462: expedite parallel tests
• PR #1410: Add javascript library and css stylesheet loading to JS class.
• PR #1448: Fix for #875 Never build unicode error messages
• PR #1458: use eval to uncan References
• PR #1450: load mathjax from CDN via https
• PR #1451: include heading level in JSON
• PR #1444: Fix pyhton -> python typos
• PR #1414: ignore errors in shell.var_expand
• PR #1430: Fix for tornado check for tornado < 1.1.0
• PR #1413: get_home_dir expands symlinks, adjust test accordingly
• PR #1385: updated and prettified magic doc strings
• PR #1406: Browser selection
• PR #1377: Saving non-ascii history
• PR #1402: fix symlinked /home issue for FreeBSD
• PR #1405: Only monkeypatch xunit when the tests are run using it.
• PR #1395: Xunit & KnownFailure
• PR #1396: Fix for %tb magic.
• PR #1386: Jsd3
• PR #1388: Add simple support for running inside a virtualenv
• PR #1391: Improve Hub/Scheduler when no engines are registered
• PR #1369: load header with engine id when engine dies in TaskScheduler
• PR #1353: Save notebook as script using unicode file handle.
• PR #1352: Add ‘-m mod : run library module as a script’ option.
• PR #1363: Fix some minor color/style config issues in the qtconsole
• PR #1371: Adds a quiet keyword to sync_imports
• PR #1387: Fixing Cell menu to update cell type select box.
• PR #1296: Wx gui example: fixes the broken example for %gui wx.
• PR #1372: ipcontroller cleans up connection files unless reuse=True
• PR #1374: remove calls to meaningless ZMQStream.on_err
• PR #1370: allow draft76 websockets (Safari)
• PR #1368: Ensure handler patterns are str, not unicode
• PR #1361: Notebook bug fix branch
• PR #1364: avoid jsonlib returning Decimal
• PR #1362: Don’t log complete contents of history replies, even in debug
• PR #1347: fix weird magic completion in notebook
• PR #1346: fixups for alternate URL prefix stuff
• PR #1336: crack at making notebook.html use the layout.html template
• PR #1331: RST and heading cells
• PR #1247: fixes a bug causing extra newlines after comments.
• PR #1332: notebook - allow prefixes in URL path.
• PR #1341: Don’t attempt to tokenize binary files for tracebacks
• PR #1334: added key handler for control-s to notebook, seems to work pretty well
• PR #1338: Fix see also in docstrings so API docs build
• PR #1335: Notebook toolbar UI
• PR #1299: made notebook.html extend layout.html
• PR #1318: make Ctrl-D in qtconsole act same as in terminal (ready to merge)
• PR #1328: Coverage
• PR #1206: don’t preserve fixConsole output in json
• PR #1330: Add linewrapping to text cells (new feature in CodeMirror).
• PR #1309: Inoculate clearcmd extension into %reset functionality
• PR #1327: Updatecm2
• PR #1326: Removing Ace edit capability.
• PR #1325: forgotten selected_cell -> get_selected_cell
• PR #1316: Pass subprocess test runners a suitable location for xunit output
• PR #1303: Updatecm
• PR #1312: minor heartbeat tweaks
• PR #1306: Fix %prun input parsing for escaped characters (closes #1302)
• PR #1301: New “Fix for issue #1202” based on current master.
• PR #1289: Make autoreload extension work on Python 3.
• PR #1288: Don’t ask for confirmation when stdin isn’t available
• PR #1294: TaskScheduler.hwm default to 1 instead of 0
• PR #1283: HeartMonitor.period should be an Integer
• PR #1264: Aceify
• PR #1284: a fix for GH 1269
• PR #1213: BUG: Minor typo in history_console_widget.py
• PR #1267: add NoDB for non-recording Hub
• PR #1222: allow Reference as callable in map/apply
• PR #1257: use self.kernel_manager_class in qtconsoleapp
• PR #1253: set auto_create flag for notebook apps
• PR #1262: Heartbeat no longer shares the app’s Context
• PR #1229: Fix display of SyntaxError in Python 3
• PR #1256: Dewijmoize
• PR #1246: Skip tests that require X, when importing pylab results in RuntimeError.
• PR #1211: serve local files in notebook-dir
• PR #1224: edit text cells on double-click instead of single-click
• PR #1187: misc notebook: connection file cleanup, first heartbeat, startup flush
• PR #1207: fix loadpy duplicating newlines
• PR #1129: Unified setup.py
• PR #1199: Reduce IPython.external.*
• PR #1218: Added -q option to %prun for suppression of the output, along with editing thedochelp string.
• PR #1217: Added -q option to %prun for suppression of the output, along with editing the dochelp string
• PR #1175: core.completer: Clean up excessive and unused code.
• PR #1196: docs: looks like a file path might have been accidentally pasted in the middle of a word
• PR #1190: Fix link to Chris Fonnesbeck blog post about 0.11 highlights.

Issues (742):

• #1943: add screenshot and link into releasenotes
• #1570: [notebook] remove ‘left panel’ references from example.
• #1954: update some example notebooks
• #2048: move _encode_binary to jsonutil.encode_images
• #2050: only add quotes around xunit-file on Windows
• #2047: disable auto-scroll on mozilla
• #1258: Magic %paste error
• #2015: Fixes for %paste with special transformations
• #760: Windows: test runner fails if repo path contains spaces
• #2046: Iptest unicode
• #1939: Namespaces
• #2042: increase auto-scroll threshold to 100 lines
• #2043: move RemoteError import to top-level
• #641: In %magic help, remove duplicate aliases
• #2036: %alias_magic
• #1968: Proposal of icons for .ipynb files
• #825: keyboardinterrupt crashes gtk gui when gtk.set_interactive is not available
• #1971: Remove duplicate magics docs
• #2040: Namespaces for cleaner public APIs
• #2039: ipython parallel import exception
• #2035: Getdefaultencoding test error with sympy 0.7.1-git
• #2037: remove ipython-qtconsole gui-script
• #1516: ipython-qtconsole script isn’t installed for Python 2.x
• #1297: “ipython -p sh” is in documentation but doesn’t work
• #2038: add extra clear warning to shell doc
• #1265: please ship unminified js and css sources
• #2029: Ship unminified js
• #1920: Provide an easy way to override the Qt widget used by qtconsole
• #2007: Add custom_control and custom_page_control variables to override the Qt widgets used by qtconsole
• #2009: In %magic help, remove duplicate aliases
• #2033: ipython parallel pushing and pulling recarrays
• #2034: fix&test push/pull recarrays
• #2028: Reduce unhelpful information shown by pinfo
• #1992: Tab completion fails with many spaces in filename
• #1885: handle too old wx
• #2030: check wxPython version in inputhook
• #2024: Make interactive_usage a bit more rst friendly
• #2031: disable ^C^C confirmation on Windows
• #2023: Unicode test failure on OS X
• #2027: match stdin encoding in frontend readline test

2.12. Issues closed in the 0.13 development cycle 183
• #1901: Windows: parallel test fails assert, leaves 14 python processes alive
• #2025: Fix parallel test on WinXP - wait for resource cleanup.
• #1986: Line magic function %R not found. (Rmagic)
• #1712: test failure in ubuntu package daily build
• #1183: 0.12 testsuite failures
• #2016: BUG: test runner fails in Windows if filenames contain spaces.
• #1806: Alternate upload methods in firefox
• #2019: Windows: home directory expansion test fails
• #2020: Fix home path expansion test in Windows.
• #2017: Windows core test error - filename quoting
• #2021: Fix Windows pathname issue in `odd encoding` test.
• #1998: call to nt.assert_true(path._writable_dir(home)) returns false in test_path.py
• #2022: don’t check writability in test for get_home_dir when HOME is undefined
• #1589: Test failures and docs don’t build on Mac OS X Lion
• #1996: frontend test tweaks
• #2011: Notebook server can’t start cluster with hyphen-containing profile name
• #2014: relax profile regex in notebook
• #2013: brew install pyqt
• #2005: Strange output artifacts in footer of notebook
• #2012: Mono cursor offset
• #2004: Clarify generic message spec vs. Python message API in docs
• #2006: Don’t crash when starting notebook server if runnable browser not found
• #2010: notebook: Print a warning (but do not abort) if no webbrowser can be found.
• #2008: pip install virtualenv
• #2003: Wrong case of rmagic in docs
• #2002: Refactor %magic into a Lsmagic_docs API function.
• #2000: kernel.js consistency with generic IPython message format.
• #1999: %magic help: display line and cell magics in alphabetical order.
• #1635: test_prun_quotes fails on Windows
• #1984: Cannot restart Notebook when using %script --bg
• #1981: Clean BG processes created by %script on kernel exit
• #1994: Fix RST misformatting.
• #1949: Introduce Notebook Magics
• #1985: Kernels should start in notebook dir when manually specified
• #1980: Notebook should check that –notebook-dir exists
• #1951: minor notebook startup/notebook-dir adjustments
• #1969: tab completion in notebook for paths not triggered
• #1974: Allow path completion on notebook.
• #1964: allow multiple instances of a Magic
• #1960: %page not working
• #1991: fix _ofind attr in %page
• #1982: Shutdown qtconsole problem?
• #1988: check for active frontend in update_restart_checkbox
• #1979: Add support for tox (http://tox.testrun.org/) and Travis CI (http://travis-ci.org/)
• #1989: Parallel: output of %px and %px${suffix} is inconsistent
• #1966: ValueError: packer could not serialize a simple message
• #1987: Notebook: MathJax offline install not recognized
• #1970: dblclick to restore size of images
• #1983: Notebook does not save heading level
• #1978: Notebook names truncating at the first period
• #1553: Limited size of output cells and provide scroll bars for such output cells
• #1825: second attempt at scrolled long output
• #1915: add cell-level metadata
• #1934: Cell/Worksheet metadata
• #1746: Confirm restart (configuration option, and checkbox UI)
• #1790: Commenting function.
• #1767: Tab completion problems with cell magics
• #1944: [qtconsole] take %,%% prefix into account for completion
• #1973: fix another FreeBSD $HOME symlink issue
• #1972: Fix completion of ‘%tim’ in the Qt console
• #1887: Make it easy to resize jpeg/png images back to original size.
• #1967: Fix psums example description in docs
• #1678: ctrl-z clears cell output in notebook when pressed enough times
• #1965: fix for #1678, undo no longer clears cells

2.12. Issues closed in the 0.13 development cycle
• #1952: avoid duplicate “Websockets closed” dialog on ws close
• #1961: UnicodeDecodeError on directory with unicode chars in prompt
• #1963: styling prompt, {color.Normal} excepts
• #1962: Support unicode prompts
• #1959: %page not working on qtconsole for Windows XP 32-bit
• #1955: update to latest version of vim-ipython
• #1945: Add --proc option to %script
• #1957: fix indentation in kernel.js
• #1956: move import RemoteError after get_exc_info
• #1950: Fix for copy action (Ctrl+C) when there is nopager defined in qtconsole
• #1948: Fix help string for InteractiveShell.ast_node_interactivity
• #1941: script magics cause terminal spam
• #1942: swallow stderr of which in utils.process.find_cmd
• #1833: completer draws slightly too small on Chrome
• #1940: fix completer css on some Chrome versions
• #1938: remove remaining references to deprecated XREP/XREQ names
• #1924: HTML superscripts not shown raised in the notebook
• #1925: Fix styling of superscripts and subscripts. Closes #1924.
• #1461: User notification if notebook saving fails
• #1936: increase duration of save messages
• #1542: %save magic fails in clients without stdin if file already exists
• #1937: add %save -f
• #1572: pyreadline version dependency not correctly checked
• #1935: add version checking to pyreadline import test
• #1849: Octave magics
• #1759: github, merge PR(s) just by number(s)
• #1931: Win py3fixes
• #1646: Meaning of restart parameter in client.shutdown() unclear
• #1933: oinspect.find_file: Additional safety if file cannot be found.
• #1916: %paste doesn’t work on py3
• #1932: Fix adding functions to CommandChainDispatcher with equal priority on Py 3
• #1928: Select NoDB by default
• #1923: Add IPython syntax support to the %timeit magic, in line and cell mode
• #1926: Make completer recognize escaped quotes in strings.
• #1929: Ipython-qtconsole (0.12.1) hangs with Python 2.7.3, Windows 7 64 bit
• #1409: [qtconsole] forward delete bring completion into current line
• #1922: py3k compatibility for setupegg.py
• #1598: document that sync_imports() can’t handle “import foo as bar”
• #1893: Update Parallel Magics and Exception Display
• #1890: Docstrings for magics that use @magic_arguments are rendered wrong
• #1921: magic_arguments: dedent but otherwise preserve indentation.
• #1919: Use oinspect in CodeMagics._find_edit_target
• #1918: don’t warn in iptest if deathrow/quarantine are missing
• #1914: %pdef failing on python3
• #1917: Fix for %pdef on Python 3
• #1428: Failing test that prun does not clobber string escapes
• #1913: Fix for #1428
• #1911: temporarily skip autoreload tests
• #1549: autoreload extension crashes ipython
• #1908: find_file errors on windows
• #1909: Fix for #1908, use os.path.normcase for safe filename comparisons
• #1907: py3compat fixes for %%script and tests
• #1904: %%px? doesn’t work, shows info for %px, general cell magic problem
• #1906: ofind finds non-unique cell magics
• #1894: Win64 binary install fails
• #1799: Source file not found for magics
• #1845: Fixes to inspection machinery for magics
• #1774: Some magics seems broken
• #1586: Clean up tight coupling between Notebook, CodeCell and Kernel Javascript objects
• #1632: Win32 shell interactivity apparently broke qtconsole “cd” magic
• #1902: Workaround fix for gh-1632; minimal revert of gh-1424
• #1900: Cython libs
• #1503: Cursor is offset in notebook in Chrome 17 on Linux
• #1426: Qt console doesn’t handle the --gui flag correctly.

2.12. Issues closed in the 0.13 development cycle
• #1180: Can’t start IPython kernel in Spyder
• #581: test IPython.zmq
• #1593: Name embedded in notebook overrides filename
• #1899: add ScriptMagics to class list for generated config
• #1618: generate or minimize manpages
• #1898: minimize manpages
• #1896: Windows: apparently spurious warning ‘Excluding nonexistent file’ ... test_exampleip
• #1897: use glob for bad exclusion warning
• #1215: updated %quickref to show short-hand for %sc and %sx
• #1855: %%script and %%file magics
• #1863: Ability to silence a cell in the notebook
• #1870: add %capture for capturing stdout/err
• #1861: Use dvipng to format sympy.Matrix
• #1867: Fix 1px margin bouncing of selected menu item.
• #1889: Reconnect when the websocket connection closes unexpectedly
• #1577: If a notebook loses its network connection WebSockets won’t reconnect
• #1886: Fix a bug in renaming notebook
• #1895: Fix error in test suite with ip.system()
• #1762: add locate entry points
• #1883: Fix vertical offset due to bold/italics, and bad browser fonts.
• #1875: re-write columnize, with intermediate step.
• #1860: IPython.utils.columnize sometime wrong...
• #1851: new completer for qtconsole.
• #1892: Remove suspicious quotes in interactiveshell.py
• #1854: Class %hierarchy and graphiz %dot magics
• #1827: Sending tracebacks over ZMQ should protect against unicode failure
• #1864: Rmagic exceptions
• #1829: [notebook] don’t care about leading prct in completion
• #1832: Make svg, jpeg and png images resizable in notebook.
• #1674: HTML Notebook carriage-return handling, take 2
• #1874: cython_magic uses importlib, which doesn’t ship with py2.6
• #1882: Remove importlib dependency which not available in Python 2.6.
• #1878: shell access using ! will not fill class or function scope vars
• #1879: Correct stack depth for variable expansion in !system commands
• #1840: New JS completer should merge completions before display
• #1841: [notebook] deduplicate completion results
• #1736: no good error message on missing tkinter and %paste
• #1741: Display message from TryNext error in magic_paste
• #1850: Remove args/kwarg handling in TryNext, fix %paste error messages.
• #1663: Keep line-endings in ipynb
• #1872: Matplotlib window freezes using interactive plot in qtconsole
• #1869: Improve CodeMagics._find_edit_target
• #1781: Colons in notebook name causes notebook deletion without warning
• #1815: Make : invalid in filenames in the Notebook JS code.
• #1819: doc: cleanup the parallel psums example a little
• #1838: externals cleanup
• #1839: External cleanup
• #1782: fix Magic menu in qtconsole, split in groups
• #1862: Minor bind_kernel improvements
• #1859: kernmagic during console startup
• #1857: Prevent jumping of window to input when output is clicked.
• #1856: Fix 1px jumping of cells and menus in Notebook.
• #1848: task fails with “AssertionError: not enough buffers!” after second resubmit
• #1852: fix chained resubmissions
• #1780: Rmagic extension
• #1853: Fix jumpy notebook behavior
• #1842: task with UnmetDependency error still owned by engine
• #1847: add InlineBackend to ConsoleApp class list
• #1846: Exceptions within multiprocessing crash Ipython notebook kernel
• #1843: Notebook does not exist and permalinks
• #1837: edit magic broken in head
• #1834: resubmitted tasks doesn’t have same session name
• #1836: preserve header for resubmitted tasks
• #1776: fix magic menu in qtconsole

2.12. Issues closed in the 0.13 development cycle
• #1828: change default extension to .ipy for %save -r
• #1800: Reintroduce recall
• #1671: __future__ environments
• #1830: lsmagic lists magics in alphabetical order
• #1835: Use Python import in ipython profile config
• #1773: Update SymPy profile: SymPy’s latex() can now print set and frozenset
• #1761: Edited documentation to use IPYTHONDIR in place of ~/.ipython
• #1772: notebook autocomplete fail when typing number
• #1822: aesthetics pass on AsyncResult.display_outputs
• #1460: Redirect http to https for notebook
• #1287: Refactor the notebook tab completion tooltip
• #1596: In rename dialog, <return> should submit
• #1821: ENTER submits the rename notebook dialog.
• #1750: Let the user disable random port selection
• #1820: NotebookApp: Make the number of ports to retry user configurable.
• #1816: Always use filename as the notebook name.
• #1775: assert in not present on Python 2.6
• #1813: Add assert_in method to nose for Python 2.6
• #1498: Add tooltip keyboard shortcuts
• #1711: New Tooltip, New Completer and JS Refactor
• #1798: a few simple fixes for docs/parallel
• #1818: possible bug with latex / markdown
• #1647: Aborted parallel tasks can’t be resubmitted
• #1817: Change behavior of ipython notebook –port=...
• #1738: IPython.embed_kernel issues
• #1610: Basic bold and italic in HTML output cells
• #1576: Start and stop kernels from the notebook dashboard
• #1515: impossible to shutdown notebook kernels
• #1812: EnsureAsyncResult.display_outputs doesn’t display empty streams
• #1811: warn on nonexistent exclusions in iptest
• #1809: test suite error in IPython.zmq on windows
• #1810: fix for #1809, failing tests in IPython.zmq
• #1808: Reposition alternate upload for firefox [need cross browser/OS/language test]
• #1742: Check for custom_exceptions only once
• #1802: cythonmagic tests should be skipped if Cython not available
• #1062: warning message in IPython.extensions test
• #1807: add missing cython exclusion in iptest
• #1805: Fixed a vcvarsall.bat error on win32/Py2.7 when trying to compile with mingw.
• #1803: MPI parallel %px bug
• #1804: Fixed a vcvarsall.bat error on win32/Py2.7 when trying to compile with mingw.
• #1492: Drag target very small if IPython Dashboard has no notebooks
• #1562: Offer a method other than drag-n-drop to upload notebooks
• #1739: Dashboard improvement (necessary merge of #1658 and #1676 + fix #1492)
• #1770: Cython related magic functions
• #1532: qtconsole does not accept –gui switch
• #1707: Accept –gui=<...> switch in IPython qtconsole.
• #1797: Fix comment which breaks Emacs syntax highlighting.
• #1796: %gui magic broken
• #1795: fix %gui magic
• #1788: extreme truncating of return values
• #1793: Raise repr limit for strings to 80 characters (from 30).
• #1794: don’t use XDG path on OS X
• #1777: ipython crash on wrong encoding
• #1792: Unicode-aware logger
• #1791: update zmqshell magics
• #1787: DOC: Remove regression from qt-console docs.
• #1785: IPython.utils.tests.test_process.SubProcessTestCase
• #1758: test_pr, fallback on http if git protocol fail, and SSL errors...
• #1786: Make notebook save failures more salient
• #1748: Fix some tests for Python 3.3
• #1755: test for pygments before running qt tests
• #1771: Make default value of interactivity passed to run_ast_nodes configurable
• #1783: part of PR #1606 (loadpy -> load) erased by magic refactoring.
• #1784: restore loadpy to load

2.12. Issues closed in the 0.13 development cycle
• #1768: Update parallel magics
• #1778: string exception in IPython/core/magic.py:232
• #1779: Tidy up error raising in magic decorators.
• #1769: Allow cell mode timeit without setup code.
• #1716: Fix for fake filenames in verbose traceback
• #1763: [qtconsole] fix append_plain_html -> append_html
• #1766: Test failure in IPython.parallel
• #1611: IPEPI: Cell magics and general cleanup of the Magic system
• #1732: Refactoring of the magics system and implementation of cell magics
• #1765: test_pr should clearn PYTHONPATH for the subprocesses
• #1630: Merge divergent Kernel implementations
• #1705: [notebook] Make pager resizable, and remember size...
• #1606: Share code for %pycat and %loadpy, make %pycat aware of URLs
• #1720: Adding interactive inline plotting to notebooks with flot
• #1701: [notebook] Open HTML links in a new window by default
• #1757: Open IPython notebook hyperlinks in a new window using target=_blank
• #1735: Open IPython notebook hyperlinks in a new window using target=_blank
• #1754: Fix typo enconters->encounters
• #1753: Clear window title when kernel is restarted
• #735: Images missing from XML/SVG export (for me)
• #1449: Fix for bug #735 : Images missing from XML/SVG export
• #1752: Reconnect Websocket when it closes unexpectedly
• #1751: Reconnect Websocket when it closes unexpectedly
• #1749: Load MathJax.js using HTTPS when IPython notebook server is HTTPS
• #1743: Tooltip completer js refactor
• #1700: A module for sending custom user messages from the kernel.
• #1745: htmlnotebook: Cursor is off
• #1728: ipython crash with matplotlib during picking
• #1681: add qt config option to clear_on_kernel_restart
• #1733: Tooltip completer js refactor
• #1676: Kernel status/shutdown from dashboard
• #1658: Alternate notebook upload methods
• #1727: terminate kernel after embed_kernel tests
• #1737: add HistoryManager to ipapp class list
• #945: Open a notebook from the command line
• #1686: ENH: Open a notebook from the command line
• #1709: fixes #1708, failing test in arg_split on windows
• #1718: Use CRegExp trait for regular expressions.
• #1729: Catch failure in repr() for %whos
• #1726: use eval for command-line args instead of exec
• #1723: scatter/gather fail with targets='all'
• #1724: fix scatter/gather with targets='all'
• #1725: add –no-ff to git pull in test_pr
• #1722: unicode exception when evaluating expression with non-ascii characters
• #1721: Tooltip completer js refactor
• #1657: Add wait optional argument to hooks.editor
• #123: Define sys.ps{1,2}
• #1717: Define generic sys.ps{1,2,3}, for use by scripts.
• #1442: cache-size issue in qtconsole
• #1691: Finish PR #1446
• #1446: Fixing Issue #1442
• #1710: update MathJax CDN url for https
• #81: Autocall fails if first function argument begins with “-“ or “+
• #1713: Make autocall regexp’s configurable.
• #211: paste command not working
• #1703: Allow TryNext to have an error message without it affecting the command chain
• #1714: minor adjustments to test_pr
• #1509: New tooltip for notebook
• #1697: Major refactoring of the Notebook, Kernel and CodeCell JavaScript.
• #788: Progress indicator in the notebook (and perhaps the Qt console)
• #1034: Single process Qt console
• #1557: magic function conflict while using –pylab
• #1476: Pylab figure objects not properly updating
• #1704: ensure all needed qt parts can be imported before settling for one

2.12. Issues closed in the 0.13 development cycle
• #1708: test failure in arg_split on windows
• #1706: Mark test_push_numpy_nocopy as a known failure for Python 3
• #1696: notebook tooltip fail on function with number
• #1698: fix tooltip on token with number
• #1226: Windows GUI only (pythonw) bug for IPython on Python 3.x
• #1245: pythonw py3k fixes for issue #1226
• #1417: Notebook Completer Class
• #1690: [Bogus] Deliberately make a test fail
• #1685: Add script to test pull request
• #1167: Settle on a choice for $IPYTHONDIR
• #1693: deprecate IPYTHON_DIR in favor of IPYTHONDIR
• #1672: ipython-qtconsole.desktop is using a deprecated format
• #1695: Avoid deprecated warnings from ipython-qtconsole.desktop.
• #1694: Add quote to notebook to allow it to load
• #1240: sys.path missing ‘’ as first entry when kernel launched without interface
• #1689: Fix sys.path missing “” as first entry in ipython kernel.
• #1683: Parallel controller failing with Pymongo 2.2
• #1687: import Binary from bson instead of pymongo
• #1616: Make IPython.core.display.Image less notebook-centric
• #1684: CLN: Remove redundant function definition.
• #1655: Add %open magic command to open editor in non-blocking manner
• #1677: middle-click paste broken in notebook
• #1670: Point %pastebin to gist
• #1667: Test failure in test_message_spec
• #1668: Test failure in IPython.zmq.tests.test_message_spec.test_complete ‘pyout’ != ‘status’
• #1669: handle pyout messages in test_message_spec
• #1295: add binary-tree engine interconnect example
• #1642: Cherry-picked commits from 0.12.1 release
• #1659: Handle carriage return characters (“r”) in HTML notebook output.
• #1313: Figure out MathJax 2 support
• #1653: Test failure in IPython.zmq
- #1656: ensure kernels are cleaned up in embed_kernel tests
- #1666: pip install ipython==dev installs version 0.8 from an old svn repo
- #1664: InteractiveShell.run_code: Update docstring.
- #1512: print stuff, should avoid newline
- #1662: Delay flushing softspace until after cell finishes
- #1643: handle jpg/jpeg in the qtconsole
- #966: dreload fails on Windows XP with iPython 0.11 “Unexpected Error”
- #1500: dreload doesn’t seem to exclude numpy
- #1520: kernel crash when showing tooltip (?)
- #1652: add patch_pyzmq() for backporting a few changes from newer pyzmq
- #1650: DOC: moving files with SSH launchers
- #1357: add IPython.embed_kernel()
- #1640: Finish up embed_kernel
- #1651: Remove bundled Itpl module
- #1634: incremental improvements to SSH launchers
- #1649: move examples/test_embed into examples/tests/embed
- #1171: Recognise virtualenvs
- #1479: test_extension failing in Windows
- #1633: Fix installing extension from local file on Windows
- #1644: Update copyright date to 2012
- #1636: Test_depreload breaks pylab irunner tests
- #1645: Exclude UserDict when deep reloading NumPy.
- #1454: make it possible to start engine in ‘disabled’ mode and ‘enable’ later
- #1641: Escape code for the current time in PromptManager
- #1638: ipython console clobbers custom sys.path
- #1637: Removed a ‘:’ which shouldn’t have been there
- #1536: ipython 0.12 embed shell won’t run startup scripts
- #1628: error: QApplication already exists in TestKillRing
- #1631: TST: QApplication doesn’t quit early enough with PySide.
- #1629: evaluate a few dangling validate_message generators
- #1621: clear In[] prompt numbers on “Clear All Output”
- #1627: Test the Message Spec

2.12. Issues closed in the 0.13 development cycle
• #1470: SyntaxError on setup.py install with Python 3
• #1624: Fixes for byte-compilation on Python 3
• #1612: pylab=inline fig.show() non-existent in notebook
• #1615: Add show() method to figure objects.
• #1622: deepreload fails on Python 3
• #1625: Fix deepreload on Python 3
• #1626: Failure in new dreload tests under Python 3.2
• #1623: iPython / matplotlib Memory error with imshow
• #1619: pyin messages should have execution_count
• #1620: pyin message now have execution_count
• #32: dreload produces spurious traceback when numpy is involved
• #1457: Update deepreload to use a rewritten knee.py. Fixes dreload(numpy).
• #1613: allow map / parallel function for single-engine views
• #1609: exit notebook cleanly on SIGINT, SIGTERM
• #1531: Function keyword completion fails if cursor is in the middle of the complete parentheses
• #1607: cleanup sqlitedb temporary db file after tests
• #1608: don’t rely on timedelta.total_seconds inAsyncResult
• #1421: ipython32 %run -d breaks with NameError name ‘execfile’ is not defined
• #1599: Fix for %run -d on Python 3
• #1201: %env magic fails with Python 3.2
• #1602: Fix %env magic on Python 3.
• #1603: Remove python3 profile
• #1604: Exclude IPython.quarantine from installation
• #1601: Security file is not removed after shutdown by Ctrl+C or kill -INT
• #1600: Specify encoding for io.open in notebook_reformat tests
• #1605: Small fixes for Animation and Progress notebook
• #1452: Bug fix for approval
• #13: Improve robustness and debuggability of test suite
• #70: IPython should prioritize __all__ during tab completion
• #1529: __all__ feature, improvement to dir2, and tests for both
• #1475: Custom namespace for %run
• #1564: calling .abort on AsyncMapResult results in traceback
• #1548: add sugar methods/properties to AsyncResult
• #1535: Fix pretty printing dispatch
• #1522: Discussion: some potential Qt console refactoring
• #1399: Use LaTeX to print various built-in types with the SymPy printing extension
• #1597: re-enter kernel.eventloop after catching SIGINT
• #1490: rename plaintext cell -> raw cell
• #1487: %notebook fails in qtconsole
• #1545: trailing newline not preserved in splitline ipynb
• #1480: Fix %notebook magic, etc. nbformat unicode tests and fixes
• #1588: Gtk3 integration with ipython works.
• #1595: Examples syntax (avoid errors installing on Python 3)
• #1526: Find encoding for Python files
• #1594: Fix writing git commit ID to a file on build with Python 3
• #1556: shallow-copy DictDB query results
• #1499: various pyflakes issues
• #1502: small changes in response to pyflakes pass
• #1445: Don’t build sphinx docs for sdists
• #1484: unhide .git_commit_info.ini
• #1538: store git commit hash in utils._sysinfo instead of hidden data file
• #1546: attempt to suppress exceptions in channel threads at shutdown
• #1524: unhide git_commit_info.ini
• #1559: update tools/github_stats.py to use GitHub API v3
• #1563: clear_output improvements
• #1558: Ipython testing documentation still mentions twisted and trial
• #1560: remove obsolete discussion of Twisted/trial from testing docs
• #1561: Qtconsole - nonstandard a and b
• #1569: BUG: qtconsole – non-standard handling of a and b. [Fixes #1561]
• #1574: BUG: Ctrl+C crashes wx pylab kernel in qtconsole
• #1573: BUG: Ctrl+C crashes wx pylab kernel in qtconsole.
• #1590: ‘iPython3 qtconsole’ doesn’t work in Windows 7
• #602: User test the html notebook
• #613: Implement Namespace panel section

2.12. Issues closed in the 0.13 development cycle 197
• #879: How to handle Javascript output in the notebook
• #1255: figure.show() raises an error with the inline backend
• #1467: Document or bundle a git-integrated facility for stripping VCS-unfriendly binary data
• #1237: Kernel status and logout button overlap
• #1319: Running a cell with ctrl+Enter selects text in cell
• #1571: module member autocomplete should respect __all__
• #1566: ipython3 doesn’t run in Win7 with Python 3.2
• #1568: fix PR #1567
• #1567: Fix: openssh_tunnel did not parse port in server
• #1565: fixAsyncResult.abort
• #1550: Crash when starting notebook in a non-ascii path
• #1552: use os.getcwd in NotebookManager
• #1554: wrong behavior of the all function on iterators
• #1541: display_pub flushes stdout/err
• #1539: Asynchronous issue when using clear_display and print x,y,z
• #1544: make MultiKernelManager.kernel_manager_class configurable
• #1494: Untrusted Secure Websocket broken on latest chrome dev
• #1521: only install ipython-qtconsole gui script on Windows
• #1528: Tab completion optionally respects __all__ (+ dir2() cleanup)
• #1527: Making a progress bar work in IPython Notebook
• #1497: __all__ functionality added to dir2(obj)
• #1518: Pretty printing exceptions is broken
• #811: Fixes for ipython unhanded OSErr exception on failure of os.getcwd()
• #1517: Fix indentation bug in IPython/lib/pretty.py
• #1519: BUG: Include the name of the exception type in its pretty format.
• #1525: A hack for auto-complete numpy recarray
• #1489: Fix zero-copy push
• #1401: numpy arrays cannot be used with View.apply() in Python 3
• #1477: fix dangling buffer in IPython.parallel.util
• #1514: DOC: Fix references to IPython.lib.pretty instead of the old location
• #1511: Version comparison error (‘2.1.11’ < ‘2.1.4’ ==> True)
• #1506: “Fixing” the Notebook scroll to help in visually comparing outputs
• #1481: BUG: Improve placement of CallTipWidget
• #1241: When our debugger class is used standalone __oh key errors are thrown
• #676: IPython.embed() from ipython crashes twice on exit
• #1496: BUG: LBYL when clearing the output history on shutdown.
• #1507: python3 notebook: TypeError: unorderable types
• #1508: fix sorting profiles in clustermanager
• #1495: BUG: Fix pretty-printing for overzealous objects
• #1505: SQLite objects created in a thread can only be used in that same thread
• #1482: %history documentation out of date?
• #1501: dreload doesn’t seem to exclude numpy
• #1472: more general fix for #662
• #1486: save state of qtconsole
• #1485: add history search to qtconsole
• #1483: updated magic_history docstring
• #1383: First version of cluster web service.
• #482: test_run.test_tclass fails on Windows
• #1398: fix %tb after SyntaxError
• #1478: key function or lambda in sorted function doesn’t find global variables
• #1415: handle exit/quit/exit()/quit() variants in zmqconsole
• #1440: Fix for failing testsuite when using –with-xml-coverage on windows.
• #1419: Add %install_ext magic function.
• #1424: Win32 shell interactivity
• #1434: Controller should schedule tasks of multiple clients at the same time
• #1268: notebook %reset magic with StdinNotImplementedError
• #1438: from cherrypy import expose fails when running script form parent directory
• #1468: Simplify structure of a Job in the TaskScheduler
• #875: never build unicode error messages
• #1107: Tab autocompletion can suggest invalid syntax
• #1447: 1107 - Tab autocompletion can suggest invalid syntax
• #1469: Fix typo in comment (insert space)
• #1463: Fix completion when importing modules in the cwd.
• #1437: unfriendly error handling with pythonw and ipython-qtconsole

2.12. Issues closed in the 0.13 development cycle
• #1466: Fix for issue #1437, unfriendly windows qtconsole error handling
• #1432: Fix ipython directive
• #1465: allow ipython help subcommand syntax
• #1394: Wishlist: Remove hard dependency on ctypes
• #1416: Conditional import of ctypes in inputhook
• #1462: expedite parallel tests
• #1418: Strict mode in javascript
• #1410: Add javascript library and css stylesheet loading to JS class.
• #1427: #922 again
• #1448: Fix for #875 Never build unicode error messages
• #1458: use eval to uncan References
• #1455: Python3 install fails
• #1450: load mathjax from CDN via https
• #1182: Qtconsole, multiwindow
• #1439: Notebook not storing heading celltype information
• #1451: include heading level in JSON
• #1444: Fix pytont -> python typos
• #1412: Input parsing issue with %prun
• #1414: ignore errors in shell.var_expand
• #1441: (1) Enable IPython.notebook.kernel.execute to publish display_* even it is not called with a code cell and (2) remove empty html element when execute “display_*”
• #1431: Beginner Error: ipython qtconsole
• #1436: “ipython-qtconsole –gui qt” hangs on 64-bit win7
• #1433: websocket connection fails on Chrome
• #1430: Fix for tornado check for tornado < 1.1.0
• #1408: test_get_home_dir_3 failed on Mac OS X
• #1413: get_home_dir expands symlinks, adjust test accordingly
• #1420: fixes #922
• #823: KnownFailure tests appearing as errors
• #1385: updated and prettified magic doc strings
• #1406: Browser selection
• #1411: ipcluster starts 8 engines “successfully” but Client only finds two
• #1375: %history -g -f file encoding issue
• #1377: Saving non-ascii history
• #797: Source introspection needs to be smarter in python 3.2
• #846: Autoreload extension doesn’t work with Python 3.2
• #1360: IPython notebook not starting on winXP
• #1407: Qtconsole segfaults on OSX when displaying some pop-up function tooltips
• #1402: fix symlinked /home issue for FreeBSD
• #1403: pyreadline cyclic dependency with pdb++/pdbpp module
• #1405: Only monkeypatch xunit when the tests are run using it.
• #1404: Feature Request: List/Dictionary tab completion
• #1395: Xunit & KnownFailure
• #1396: Fix for %tb magic.
• #1397: Stay or leave message not working, Safari session lost.
• #1389: pylab=inline inoperant through ssh tunnelling?
• #1386: Jsd3
• #1388: Add simple support for running inside a virtualenv
• #826: Add support for creation of parallel task when no engine is running
• #1391: Improve Hub/Scheduler when no engines are registered
• #1369: load header with engine id when engine dies in TaskScheduler
• #1345: notebook can’t save unicode as script
• #1353: Save notebook as script using unicode file handle.
• #1352: Add ‘-m mod : run library module as a script’ option.
• #1363: Fix some minor color/style config issues in the qtconsole
• #1371: Adds a quiet keyword to sync_imports
• #1390: Blank screen for notebooks on Safari
• #1387: Fixing Cell menu to update cell type select box.
• #645: Standalone WX GUI support is broken
• #1296: Wx gui example: fixes the broken example for %gui wx.
• #1254: typo in notebooklist.js breaks links
• #781: Users should be able to clone a notebook
• #1372: ipcontroller cleans up connection files unless reuse=True
• #1374: remove calls to meaningless ZMQStream.on_err

2.12. Issues closed in the 0.13 development cycle
• #1382: Update RO for Notebook
• #1370: allow draft76 websockets (Safari)
• #1368: Ensure handler patterns are str, not unicode
• #1379: Sage link on website homepage broken
• #1376: FWIW does not work with Chrome 16.0.912.77 Ubuntu 10.10
• #1358: Cannot install ipython on Windows 7 64-bit
• #1367: Ctrl - m t does not toggle output in chrome
• #1359: [sympyprinting] MathJax can’t render root{m}{n}
• #1337: Tab in the notebook after ( should not indent, only give a tooltip
• #1339: Notebook printing broken
• #1344: Ctrl + M + L does not toggle line numbering in htmlnotebook
• #1348: Ctrl + M + M does not switch to markdown cell
• #1361: Notebook bug fix branch
• #1364: avoid jsonlib returning Decimal
• #1362: Don’t log complete contents of history replies, even in debug
• #888: ReST support in notebooks
• #1205: notebook stores HTML escaped text in the file
• #1351: add IPython.embed_kernel()
• #1243: magic commands without % are not completed properly in htmlnotebook
• #1347: fix weird magic completion in notebook
• #1355: notebook.html extends layout.html now
• #1354: min and max in the notebook
• #1346: fixups for alternate URL prefix stuff
• #1336: crack at making notebook.html use the layout.html template
• #1331: RST and heading cells
• #1350: Add ‘-m mod : run library module as a script’ option
• #1247: fixes a bug causing extra newlines after comments.
• #1329: add base_url to notebook configuration options
• #1332: notebook - allow prefixes in URL path.
• #1317: Very slow traceback construction from Cython extension
• #1341: Don’t attempt to tokenize binary files for tracebacks
• #1300: Cell Input collapse
• #1334: added key handler for control-s to notebook, seems to work pretty well
• #1338: Fix see also in docstrings so API docs build
• #1335: Notebook toolbar UI
• #1299: made notebook.html extend layout.html
• #1318: make Ctrl-D in qtconsole act same as in terminal (ready to merge)
• #873: ReST support in notebook frontend
• #1139: Notebook webkit notification
• #1314: Insertcell
• #1328: Coverage
• #1206: don’t preserve fixConsole output in json
• #1330: Add linewrapping to text cells (new feature in CodeMirror).
• #1309: Inoculate clearcmd extension into %reset functionality
• #1327: Updatem2
• #1326: Removing Ace edit capability.
• #1325: forgotten selected_cell -> get_selected_cell
• #1316: Pass subprocess test runners a suitable location for xunit output
• #1315: Collect results from subprocess runners and spit out Xunit XML output.
• #1233: Update CodeMirror to the latest version
• #1234: Refactor how the notebook focuses cells
• #1235: After upgrading CodeMirror check the status of some bugs
• #1236: Review how select is called when notebook cells are inserted
• #1303: Updatem
• #1311: Fixing CM related indentation problems.
• #1304: controller/server load can disrupt heartbeat
• #1312: minor heartbeat tweaks
• #1302: Input parsing with %prun clobbers escapes
• #1306: Fix %prun input parsing for escaped characters (closes #1302)
• #1251: IPython-0.12 can’t import map module on Python 3.1
• #1202: Pyreadline install exclusion for 64 bit windows no longer required, version dependency not correctly specified.
• #1301: New “Fix for issue #1202” based on current master.
• #1242: changed key map name to match changes to python mode

2.12. Issues closed in the 0.13 development cycle
• #1203: Fix for issue #1202
• #1289: Make autoreload extension work on Python 3.
• #1263: Different ‘C-x’ for shortcut, ‘C-m c’ not toCodeCell anymore
• #1259: Replace “from (.|..) import” with absolute imports.
• #1278: took a crack at making notebook.html extend layout.html
• #1210: Add ‘quiet’ option to suppress screen output during %prun calls, edited dochelp
• #1288: Don’t ask for confirmation when stdin isn’t available
• #1290: Cell-level cut & paste overwrites multiple cells
• #1291: Minor, but important fixes to cut/copy/paste.
• #1293: TaskScheduler.hwm default value
• #1294: TaskScheduler.hwm default to 1 instead of 0
• #1281: in Hub: registration_timeout must be an integer, but heartmonitor.period is CFloat
• #1283: HeartMonitor.period should be an Integer
• #1162: Allow merge/split adjacent cells in notebook
• #1264: Aceify
• #1261: Mergesplit
• #1269: Another strange input handling error
• #1284: a fix for GH 1269
• #1232: Dead kernel loop
• #1279: ImportError: cannot import name S1 (from logging)
• #1276: notebook menu item to send a KeyboardInterrupt to the kernel
• #1213: BUG: Minor typo in history_console_widget.py
• #1248: IPython notebook doesn’t work with lastest version of tornado
• #1267: add NoDB for non-recording Hub
• #1222: allow Reference as callable in map/apply
• #1257: use self.kernel_manager_class in qtconsoleapp
• #1220: Open a new notebook while connecting to an existing kernel (opened by qtconsole or terminal or standalone)
• #1253: set auto_create flag for notebook apps
• #1260: heartbeat failure on long gil-holding operation
• #1262: Heartbeat no longer shares the app’s Context
• #1225: SyntaxError display broken in Python 3
• #1229: Fix display of SyntaxError in Python 3
• #1256: Dewijmoize
• #1246: Skip tests that require X, when importing pylab results in RuntimeError.
• #1250: Wijmoize
• #1244: can not imput chinese word “” , exit right now
• #1194: Adding Opera 11 as a compatible browser for ipython notebook
• #1198: Kernel Has Died error in Notebook
• #1211: serve local files in notebook-dir
• #1224: edit text cells on double-click instead of single-click
• #1187: misc notebook: connection file cleanup, first heartbeat, startup flush
• #1207: fix loadpy duplicating newlines
• #1060: Always save the .py file to disk next to the .ipynb
• #1066: execute cell in place should preserve the current insertion-point in the notebook
• #1141: “In” numbers are not invalidated when restarting kernel
• #1231: pip on OSX tries to install files in /System directory.
• #1129: Unified setup.py
• #1199: Reduce IPython.external.*
• #1219: Make all the static files path absolute.
• #1218: Added -q option to %prun for suppression of the output, along with editing the dochelp string.
• #1217: Added -q option to %prun for suppression of the output, along with editing the dochelp string
• #1216: Pdb tab completion does not work in QtConsole
• #1197: Interactive shell trying to: from ... import history
• #1175: core.completer: Clean up excessive and unused code.
• #1208: should dv.sync_import print failed imports ?
• #1186: payloadpage.py not used by qtconsole
• #1204: double newline from %loadpy in python notebook (at least on mac)
• #1192: Invalid JSON data
• #1196: docs: looks like a file path might have been accidentally pasted in the middle of a word
• #1189: Right justify of ‘in’ prompt in variable prompt size configurations
• #1185: ipython console not work proper with stdout...
• #1191: profile/startup files not executed with “notebook”
• #1190: Fix link to Chris Fonnesbeck blog post about 0.11 highlights.

2.12. Issues closed in the 0.13 development cycle
• #1174: Remove %install_default_config and %install_profiles

2.13 0.12 Series

2.13.1 Release 0.12.1

IPython 0.12.1 is a bugfix release of 0.12, pulling only bugfixes and minor cleanup from 0.13, timed for the Ubuntu 12.04 LTS release.

See the list of fixed issues for specific backported issues.

2.13.2 Release 0.12

IPython 0.12 contains several major new features, as well as a large amount of bug and regression fixes. The 0.11 release brought with it a lot of new functionality and major refactorings of the codebase; by and large this has proven to be a success as the number of contributions to the project has increased dramatically, proving that the code is now much more approachable. But in the refactoring inevitably some bugs were introduced, and we have also squashed many of those as well as recovered some functionality that had been temporarily disabled due to the API changes.

The following major new features appear in this version.

An interactive browser-based Notebook with rich media support

A powerful new interface puts IPython in your browser. You can start it with the command ipython notebook:

This new interface maintains all the features of IPython you are used to, as it is a new client that communicates with the same IPython kernels used by the terminal and Qt console. But the web notebook provides for a different workflow where you can integrate, along with code execution, also text, mathematical expressions, graphics, video, and virtually any content that a modern browser is capable of displaying.

You can save your work sessions as documents that retain all these elements and which can be version controlled, emailed to colleagues or saved as HTML or PDF files for printing or publishing statically on the web. The internal storage format is a JSON file that can be easily manipulated for manual exporting to other formats.

This Notebook is a major milestone for IPython, as for years we have tried to build this kind of system. We were inspired originally by the excellent implementation in Mathematica, we made a number of attempts using older technologies in earlier Summer of Code projects in 2005 (both students and Robert Kern developed early prototypes), and in recent years we have seen the excellent implementation offered by the Sage system. But we continued to work on something that would be consistent with the rest of IPython’s design, and it is clear now that the effort was worth it: based on the ZeroMQ communications architecture introduced in version 0.11, the notebook can now retain 100% of the features of the real IPython. But it can also provide the rich media support and high quality Javascript libraries that were not available in browsers even one or two years ago (such as high-quality mathematical rendering or built-in video).
Fig. 2.3: The new IPython notebook showing text, mathematical expressions in LaTeX, code, results and embedded figures created with Matplotlib.

The notebook has too many useful and important features to describe in these release notes; our documentation now contains a directory called examples/notebooks with several notebooks that illustrate various aspects of the system. You should start by reading those named 00_notebook_tour.ipynb and 01_notebook_introduction.ipynb first, and then can proceed to read the others in any order you want.

To start the notebook server, go to a directory containing the notebooks you want to open (or where you want to create new ones) and type:

```
ipython notebook
```

You can see all the relevant options with:

```
ipython notebook --help
ipython notebook --help-all  # even more
```

and just like the Qt console, you can start the notebook server with pylab support by using:

```
ipython notebook --pylab
```

for floating matplotlib windows or:

```
ipython notebook --pylab inline
```

for plotting support with automatically inlined figures. Note that it is now possible also to activate pylab support at runtime via %pylab, so you do not need to make this decision when starting the server.

See the Notebook docs for technical details.
**Two-process terminal console**

Based on the same architecture as the notebook and the Qt console, we also have now a terminal-based console that can connect to an external IPython kernel (the same kernels used by the Qt console or the notebook, in fact). While this client behaves almost identically to the usual IPython terminal application, this capability can be very useful to attach an interactive console to an existing kernel that was started externally. It lets you use the interactive %debug facilities in a notebook, for example (the web browser can’t interact directly with the debugger) or debug a third-party code where you may have embedded an IPython kernel.

This is also something that we have wanted for a long time, and which is a culmination (as a team effort) of the work started last year during the 2010 Google Summer of Code project.

**Tabbed QtConsole**

The QtConsole now supports starting multiple kernels in tabs, and has a menubar, so it looks and behaves more like a real application. Keyboard enthusiasts can disable the menubar with ctrl-shift-M (PR #887).

**Full Python 3 compatibility**

IPython can now be installed from a single codebase on Python 2 and Python 3. The installation process for Python 3 automatically runs 2to3. The same ‘default’ profile is now used for Python 2 and 3 (the previous version had a separate ‘python3’ profile).

**Standalone Kernel**

The ipython kernel subcommand has been added, to allow starting a standalone kernel, that can be used with various frontends. You can then later connect a Qt console or a terminal console to this kernel by typing e.g.:

```
ipython qtconsole --existing
```

if it’s the only one running, or by passing explicitly the connection parameters (printed by the kernel at startup).

**PyPy support**

The terminal interface to IPython now runs under PyPy. We will continue to monitor PyPy’s progress, and hopefully before long at least we’ll be able to also run the notebook. The Qt console may take longer, as Qt is a very complex set of bindings to a huge C++ library, and that is currently the area where PyPy still lags most behind. But for everyday interactive use at the terminal, with this release and PyPy 1.7, things seem to work quite well from our admittedly limited testing.
Fig. 2.4: The improved Qt console for IPython, now with tabs to control multiple kernels and full menu support.
Other important new features

- **SSH Tunnels:** In 0.11, the IPython.parallel Client could tunnel its connections to the Controller via ssh. Now, the QtConsole supports ssh tunneling, as do parallel engines.

- **relaxed command-line parsing:** 0.11 was released with overly-strict command-line parsing, preventing the ability to specify arguments with spaces, e.g. `ipython --pylab qt` or `ipython -c "print 'hi'"`. This has been fixed, by using argparse. The new parsing is a strict superset of 0.11, so any commands in 0.11 should still work in 0.12.

- **HistoryAccessor:** The `HistoryManager` class for interacting with your IPython SQLite history database has been split, adding a parent `HistoryAccessor` class, so that users can write code to access and search their IPython history without being in an IPython session (PR #824).

- **kernel %gui and %pylab:** The `%gui` and `%pylab` magics have been restored to the IPython kernel (e.g. in the qtconsole or notebook). This allows activation of pylab-mode, or eventloop integration after starting the kernel, which was unavailable in 0.11. Unlike in the terminal, this can be set only once, and cannot be changed.

- **%config:** A new `%config` magic has been added, giving easy access to the IPython configuration system at runtime (PR #923).

- **Multiline History:** Multiline readline history has been restored to the Terminal frontend by default (PR #838).

- **%store:** The `%store` magic from earlier versions has been updated and re-enabled (`storemagic`; PR #1029). To autorestore stored variables on startup, specify `c.StoreMagic.autorestore = True` in `ipython_config.py`.

**Major Bugs fixed**

In this cycle, we have *closed over 500 issues*, but a few major ones merit special mention:

- Simple configuration errors should no longer crash IPython. In 0.11, errors in config files, as well as invalid trait values, could crash IPython. Now, such errors are reported, and help is displayed.

- Certain SyntaxErrors no longer crash IPython (e.g. just typing keywords, such as `return`, `break`, etc.). See #704.

- IPython path utils, such as `get_ipython_dir()` now check for write permissions, so IPython should function on systems where the default path resolution might point to a read-only location, such as `HOMESHARE` on Windows (#669).

- `raw_input()` now works in the kernel when multiple frontends are in use. The request will be sent to the frontend that made the request, and an exception is raised if that frontend does not support stdin requests (e.g. the notebook) (#673).

- `zmq` version detection no longer uses simple lexicographical comparison to check minimum version, which prevents 0.11 from working with pyzmq-2.1.10 (PR #758).

- A bug in PySide < 1.0.7 caused crashes on OSX when tooltips were shown (#711). These tooltips are now disabled on old PySide (PR #963).
- IPython no longer crashes when started on recent versions of Python 3 in Windows (#737).

- Instances of classes defined interactively can now be pickled (#29; PR #648). Note that pickling saves a reference to the class definition, so unpickling the instances will only work where the class has been defined.

**Backwards incompatible changes**

- IPython connection information is no longer specified via ip/port directly, rather via json connection files. These files are stored in the security directory, and enable us to turn on HMAC message authentication by default, significantly improving the security of kernels. Various utility functions have been added to IPython.lib.kernel, for easier connecting to existing kernels.

- KernelManager now has one ip, and several port traits, rather than several ip/port pair _addr traits. This better matches the rest of the code, where the ip cannot not be set separately for each channel.

- Custom prompts are now configured using a new class, PromptManager, which has traits for in_template, in2_template (the ...: continuation prompt), out_template and rewrite_template. This uses Python’s string formatting system, so you can use {time} and {cwd}, although we have preserved the abbreviations from previous versions, e.g. \# (prompt number) and \w (working directory). For the list of available fields, refer to the source of IPython/core/prompts.py.

- The class inheritance of the Launchers in IPython.parallel.apps.launcher used by ipcluster has changed, so that trait names are more consistent across batch systems. This may require a few renames in your config files, if you customized the command-line args for launching controllers and engines. The configurable names have also been changed to be clearer that they point to class names, and can now be specified by name only, rather than requiring the full import path of each class, e.g.:

  ```python
  IPClusterEngines.engine_launcher = 'IPython.parallel.apps.launcher.MPIExecEngineSetLauncher'
  IPClusterStart.controller_launcher = 'IPython.parallel.apps.launcher.SSHControllerLauncher'
  ``

  would now be specified as:

  ```python
  IPClusterEngines.engine_launcher_class = 'MPI'
  IPClusterStart.controller_launcher_class = 'SSH'
  ```

  The full path will still work, and is necessary for using custom launchers not in IPython’s launcher module.

  Further, MPIExec launcher names are now prefixed with just MPI, to better match other batch launchers, and be generally more intuitive. The MPIExec names are deprecated, but continue to work.

- For embedding a shell, note that the parameters user_global_ns and global_ns have been deprecated in favour of user_module and module respectively. The new parameters expect a module-like object, rather than a namespace dict. The old parameters remain for backwards compatibility, although user_global_ns is now ignored. The user_ns parameter works the same way as before, and calling embed() with no arguments still works as before.
Development summary and credits

The previous version (IPython 0.11) was released on July 31 2011, so this release cycle was roughly 4 1/2 months long, we closed a total of 515 issues, 257 pull requests and 258 regular issues (a detailed list is available).

Many users and developers contributed code, features, bug reports and ideas to this release. Please do not hesitate in contacting us if we’ve failed to acknowledge your contribution here. In particular, for this release we have had commits from the following 45 contributors, a mix of new and regular names (in alphabetical order by first name):

- Alcides <alcides-at-do-not-span-me.com>
- Ben Edwards <bedwards-at-cs.unm.edu>
- Benjamin Ragan-Kelley <benjaminrk-at-gmail.com>
- Benjamin Thyreau <benjamin.thyreau-at-gmail.com>
- Bernardo B. Marques <bernardo.fire-at-gmail.com>
- Bernard Paulus <bprecyclebin-at-gmail.com>
- Bradley M. Froehle <brad.froehle-at-gmail.com>
- Brian E. Granger <ellisonbg-at-gmail.com>
- Christian Boos <cboos-at-bct-technology.com>
- Daniel Velkov <danielv-at-mylife.com>
- Erik Tollerud <erik.tollerud-at-gmail.com>
- Evan Patterson <epatters-at-enthought.com>
- Felix Werner <Felix.Werner-at-kit.edu>
- Fernando Perez <Fernando.Perez-at-berkeley.edu>
- Gabriel <g2p.code-at-gmail.com>
- Grahame Bowland <grahame-at-angrygoats.net>
- Hannes Schulz <schulz-at-ais.uni-bonn.de>
- Jens Hedegaard Nielsen <jenshnielsen-at-gmail.com>
- Jonathan March <jmarch-at-enthought.com>
- Jörgen Stenarson <jorgen.stenarson-at-bostream.nu>
- Julian Taylor <jtaylor.debian-at-googlemail.com>
- Kefu Chai <tchaikov-at-gmail.com>
- macgyver <neil.rabinowitz-at-merton.ox.ac.uk>
- Matt Cottingham <matt.cottingham-at-gmail.com>
- Matthew Brett <matthew.brett-at-gmail.com>
• Matthias BUSSONNIER <bussonniermatthias-at-gmail.com>
• Michael Droettboom <mdboom-at-gmail.com>
• Nicolas Rougier <Nicolas.Rougier-at-inria.fr>
• Olivier Verdier <olivier.verdier-at-gmail.com>
• Omar Andres Zapata Mesa <andresete.chaos-at-gmail.com>
• Pablo Winant <pablo.winant-at-gmail.com>
• Paul Ivanov <pivanov314-at-gmail.com>
• Pauli Virtanen <pav-at-iki.fi>
• Pete Aykroyd <aykroyd-at-gmail.com>
• Prabhu Ramachandran <prabhu-at-enthought.com>
• Puneeth Chaganti <punchagan-at-gmail.com>
• Robert Kern <robert.kern-at-gmail.com>
• Satrajit Ghosh <satra-at-mit.edu>
• Stefan van der Walt <stefan-at-sun.ac.za>
• Szabolcs Horvát <szhorvat-at-gmail.com>
• Thomas Kluyver <takowl-at-gmail.com>
• Thomas Spura <thomas.spura-at-gmail.com>
• Timo Paulssen <timonator-at-perpetuum-immobile.de>
• Valentin Haenel <valentin.haenel-at-gmx.de>
• Yaroslav Halchenko <debian-at-onerussian.com>

Note: This list was generated with the output of `git log rel-0.11..HEAD --format=* %aN <%aE>' | sed 's/@/\-at\-/' | sed 's/<//g' | sort -u after some cleanup. If you should be on this list, please add yourself.

## 2.14 Issues closed in the 0.12 development cycle

### 2.14.1 Issues closed in 0.12.1

GitHub stats for bugfix release 0.12.1 (12/28/2011-04/16/2012), backporting pull requests from 0.13.

We closed a total of 71 issues: 44 pull requests and 27 issues; this is the full list (generated with the script `tools/github_stats.py`).

This list is automatically generated, and may be incomplete:

Pull Requests (44):
• PR #1175: core.completer: Clean up excessive and unused code.
• PR #1187: misc notebook: connection file cleanup, first heartbeat, startup flush
• PR #1190: Fix link to Chris Fonnesbeck blog post about 0.11 highlights.
• PR #1196: docs: looks like a file path might have been accidentally pasted in the middle of a word
• PR #1206: don’t preserve fixConsole output in json
• PR #1207: fix loadpy duplicating newlines
• PR #1213: BUG: Minor typo in history_console_widget.py
• PR #1218: Added -q option to %prun for suppression of the output, along with editing the dochelp string.
• PR #1222: allow Reference as callable in map/apply
• PR #1229: Fix display of SyntaxError in Python 3
• PR #1246: Skip tests that require X, when importing pylab results in RuntimeError.
• PR #1253: set auto_create flag for notebook apps
• PR #1257: use self.kernel_manager_class in qtconsoleapp
• PR #1262: Heartbeat no longer shares the app’s Context
• PR #1283: HeartMonitor.period should be an Integer
• PR #1284: a fix for GH 1269
• PR #1289: Make autoreload extension work on Python 3.
• PR #1306: Fix %prun input parsing for escaped characters (closes #1302)
• PR #1312: minor heartbeat tweaks
• PR #1318: make Ctrl-D in qtconsole act same as in terminal (ready to merge)
• PR #1341: Don’t attempt to tokenize binary files for tracebacks
• PR #1353: Save notebook as script using unicode file handle.
• PR #1363: Fix some minor color/style config issues in the qtconsole
• PR #1364: avoid jsonlib returning Decimal
• PR #1369: load header with engine id when engine dies in TaskScheduler
• PR #1370: allow draft76 websockets (Safari)
• PR #1374: remove calls to meaningless ZMQStream.on_err
• PR #1377: Saving non-ascii history
• PR #1396: Fix for %tb magic.
• PR #1402: fix symlinked /home issue for FreeBSD
• PR #1413: get_home_dir expands symlinks, adjust test accordingly
• PR #1414: ignore errors in shell.var_expand
• PR #1430: Fix for tornado check for tornado < 1.1.0
• PR #1445: Don’t build sphinx docs for sdists
• PR #1463: Fix completion when importing modules in the cwd.
• PR #1477: fix dangling buffer in IPython.parallel.util
• PR #1495: BUG: Fix pretty-printing for overzealous objects
• PR #1496: BUG: LBYL when clearing the output history on shutdown.
• PR #1514: DOC: Fix references to IPython.lib.pretty instead of the old location
• PR #1517: Fix indentation bug in IPython/lib/pretty.py
• PR #1538: store git commit hash in utils._sysinfo instead of hidden data file
• PR #1599: Fix for %run -d in Python 3
• PR #1602: Fix %env for Python 3
• PR #1607: cleanup sqlitedb temporary db file after tests

Issues (27):

• #676: IPython.embed() from ipython crashes twice on exit
• #846: Autoreload extension doesn’t work with Python 3.2
• #1187: misc notebook: connection file cleanup, first heartbeat, startup flush
• #1191: profile/startup files not executed with “notebook”
• #1197: Interactive shell trying to: from ... import history
• #1198: Kernel Has Died error in Notebook
• #1201: %env magic fails with Python 3.2
• #1204: double newline from %loadpy in python notebook (at least on mac)
• #1208: should dv.sync_import print failed imports ?
• #1225: SyntaxError display broken in Python 3
• #1232: Dead kernel loop
• #1241: When our debugger class is used standalone _oh key errors are thrown
• #1254: typo in notebooklist.js breaks links
• #1260: heartbeat failure on long gil-holding operation
• #1268: notebook %reset magic fails with StdinNotImplementedError
• #1269: Another strange input handling error
• #1281: in Hub: registration_timeout must be an integer, but heartmonitor.period is CFloat
• #1302: Input parsing with %prun clobbers escapes

2.14. Issues closed in the 0.12 development cycle
• #1304: controller/server load can disrupt heartbeat
• #1317: Very slow traceback construction from Cython extension
• #1345: notebook can’t save unicode as script
• #1375: %history -g -f file encoding issue
• #1401: numpy arrays cannot be used with View.apply() in Python 3
• #1408: test_get_home_dir_3 failed on Mac OS X
• #1412: Input parsing issue with %prun
• #1421: ipython32 %run -d breaks with NameError name ‘execfile’ is not defined
• #1484: unhide .git_commit_info.ini

2.14.2 Issues closed in 0.12

In this cycle, from August 1 to December 28 2011, we closed a total of 515 issues, 257 pull requests and 258 regular issues; this is the full list (generated with the script tools/github_stats.py).

Pull requests (257):

• 1174: Remove %install_default_config and %install_profiles
• 1178: Correct string type casting in pinfo.
• 1096: Show class init and call tooltips in notebook
• 1176: Modifications to profile list
• 1173: don’t load gui/pylab in console frontend
• 1168: Add –script flag as shorthand for notebook save_script option.
• 1165: encode image_tag as utf8 in [x]html export
• 1161: Allow %loadpy to load remote URLs that don’t end in .py
• 1158: Add coding header when notebook exported to .py file.
• 1160: don’t ignore ctrl-C during %gui qt
• 1159: Add encoding header to Python files downloaded from notebooks.
• 1155: minor post-execute fixes (#1154)
• 1153: Pager tearing bug
• 1152: Add support for displaying matplotlib axes directly.
• 1079: Login/out button cleanups
• 1151: allow access to user_ns in prompt_manager
• 1120: updated vim-ipython (pending)
• 1150: BUG: Scrolling pager in vsplit on Mac OSX tears.
• 1149: #1148 (win32 arg_split)
• 1147: Put qtconsole foreground when launching
• 1146: allow saving notebook.py next to notebook.ipynb
• 1128: fix pylab StartMenu item
• 1140: Namespaces for embedding
• 1132: [notebook] read-only: disable name field
• 1125: notebook : update logo
• 1135: allow customized template and static file paths for the notebook web app
• 1122: BUG: Issue #755 qt IPythonWidget.execute_file fails if filename contains...
• 1137: rename MPIExecLaunchers to MPLaunchers
• 1130: optionally ignore shlex’s ValueError in arg_split
• 1116: Shlex unicode
• 1073: Storemagic plugin
• 1143: Add post_install script to create start menu entries in Python 3
• 1138: Fix tests to work when ~/.config/ipython contains a symlink.
• 1121: Don’t transform function calls on IPyAutocall objects
• 1118: protect CRLF from carriage-return action
• 1105: Fix for prompts containing newlines.
• 1126: Totally remove pager when read only (notebook)
• 1091: qtconsole : allow copy with shortcut in pager
• 1114: fix magics history in two-process ipython console
• 1113: Fixing #1112 removing failing asserts for test_carriage_return and test_beep
• 1089: Support carriage return (‘r’) and beep (‘b’) characters in the qtconsole
• 1108: Completer usability 2 (rebased of pr #1082)
• 864: Two-process terminal frontend (ipython core branch)
• 1082: usability and cross browser compat for completer
• 1053: minor improvements to text placement in qtconsole
• 1106: Fix display of errors in compiled code on Python 3
• 1077: allow the notebook to run without MathJax
• 1072: If object has a getdoc() method, override its normal docstring.
• 1059: Switch to simple __IPYTHON__ global
• 1070: Execution count after SyntaxError
• 1098: notebook: config section UI
• 1101: workaround spawnb missing from pexpect.__all__
• 1097: typo, should fix #1095
• 1099: qtconsole export xhtml/utf8
• 1083: Prompts
• 1081: Fix wildcard search for updated namespaces
• 1084: write busy in notebook window title...
• 1078: PromptManager fixes
• 1064: Win32 shlex
• 1069: As you type completer, fix on Firefox
• 1039: Base of an as you type completer.
• 1065: Qtconsole fix racecondition
• 507: Prompt manager
• 1056: Warning in code. qtconsole ssh -X
• 1036: Clean up javascript based on js2-mode feedback.
• 1052: Pylab fix
• 648: Usermod
• 969: Pexpect-u
• 1007: Fix paste/cpaste bug and refactor/cleanup that code a lot.
• 506: make ENTER on a previous input field replace current input buffer
• 1040: json/jsonapi cleanup
• 1042: fix firefox (windows) break line on empty prompt number
• 1015: emacs freezes when tab is hit in ipython with latest python-mode
• 1023: flush stdout/stderr at the end of kernel init
• 956: Generate “All magics...” menu live
• 1038: Notebook: don’t change cell when selecting code using shift+up/down.
• 987: Add Tooltip to notebook.
• 1028: Cleaner minimum version comparison
• 998: defer to stdlib for path.get_home_dir()
• 1033: update copyright to 2011/20xx-2011
• 1032: Intercept <esc> avoid closing websocket on Firefox
• 1030: use pyzmq tools where appropriate
1029: Restore pspersistence, including %store magic, as an extension.
1025: Dollar escape
999: Fix issue #880 - more useful message to user when %paste fails
938: changes to get ipython.el to work with the latest python-mode.el
1012: Add logout button.
1020: Dollar formatter for ! shell calls
1019: Use repr() to make quoted strings
1008: don’t use crash_handler by default
1003: Drop consecutive duplicates when refilling readline history
997: don’t unregister interrupted post-exec functions
996: add Integer traitlet
1016: Fix password hashing for Python 3
1014: escape minus signs in manpages
1013: [NumPyExampleDocstring] link was pointing to raw file
1011: Add hashed password support.
1005: Quick fix for os.system requiring str parameter
994: Allow latex formulas in HTML output
955: Websocket Adjustments
979: use system_raw in terminal, even on Windows
989: fix arguments for commands in _process_posix
991: Show traceback, continuing to start kernel if pylab init fails
981: Split likely multiline text when writing JSON notebooks
957: allow change of png DPI in inline backend
968: add wantDirectory to ipdoctest, so that directories will be checked for e
984: Do not expose variables defined at startup to %who etc.
985: Fixes for parallel code on Python 3
963: disable calltips in PySide < 1.0.7 to prevent segfault
976: Getting started on what’s new
929: Multiline history
964: Default profile
961: Disable the pager for the test suite
953: Physics extension

2.14. Issues closed in the 0.12 development cycle
• 950: Add directory for startup files
• 940: allow setting HistoryManager.hist_file with config
• 948: Monkeypatch Tornado 2.1.1 so it works with Google Chrome 16.
• 916: Run p (https://github.com/ipython/ipython/pull/901)
• 923: %config magic
• 920: unordered iteration of AsyncMapResults (+ a couple fixes)
• 941: Follow-up to 387dcd6a, _rl.__doc__ is None with pyreadline
• 931: read-only notebook mode
• 921: Show invalid config message on TraitErrors during init
• 815: Fix #481 using custom qt4 input hook
• 936: Start webbrowser in a thread. Prevents lockup with Chrome.
• 937: add dirty trick for readline import on OSX
• 913: Py3 tests2
• 933: Cancel in qt console closeevent should trigger event.ignore()
• 930: read-only notebook mode
• 910: Make import checks more explicit in %whos
• 926: reincarnate -V cmdline option
• 928: BUG: Set context for font size change shortcuts in ConsoleWidget
• 901: - There is a bug when running the profiler in the magic command (prun) with python3
• 912: Add magic for cls on windows. Fix for #181.
• 905: enable %gui/%pylab magics in the Kernel
• 909: Allow IPython to run without sqlite3
• 887: Qtconsole menu
• 895: notebook download implies save
• 896: Execfile
• 899: Brian’s Notebook work
• 892: don’t close figures every cycle with inline matplotlib backend
• 893: Adding clear_output to kernel and HTML notebook
• 789: Adding clear_output to kernel and HTML notebook.
• 898: Don’t pass unicode sys.argv with %run or ipython script.py
• 897: Add tooltips to the notebook via ‘title’ attr.
• 877: partial fix for issue #678
• 838: reenable multiline history for terminals
• 872: The constructor of Client() checks for AssertionError in validate_url to open a file instead of connection to a URL if it fails.
• 884: Notebook usability fixes
• 883: User notification if notebook saving fails
• 889: Add drop_by_id method to shell, to remove variables added by extensions.
• 891: Ability to open the notebook in a browser when it starts
• 813: Create menu bar for qtconsole
• 876: protect IPython from bad custom exception handlers
• 856: Backgroundjobs
• 868: Warn user if MathJax can’t be fetched from notebook closes #744
• 878: store_history=False default for run_cell
• 824: History access
• 850: Update codemirror to 2.15 and make the code internally more version-agnostic
• 861: Fix for issue #56
• 819: Adding -m option to %run, similar to -m for python interpreter.
• 855: promote aliases and flags, to ensure they have priority over config files
• 862: BUG: Completion widget position and pager focus.
• 847: Allow connection to kernels by files
• 708: Two-process terminal frontend
• 857: make sdist flags work again (e.g. –manifest-only)
• 835: Add Tab key to list of keys that scroll down the paging widget.
• 859: Fix for issue #800
• 848: Python3 setup.py install failiure
• 845: Tests on Python 3
• 802: DOC: extensions: add documentation for the bundled extensions
• 830: contiguous stdout/stderr in notebook
• 761: Windows: test runner fails if repo path (e.g. home dir) contains spaces
• 801: Py3 notebook
• 809: use CFRunLoop directly in ipython kernel --pylab osx
• 841: updated old scipy.org links, other minor doc fixes
• 837: remove all trailing spaces

2.14. Issues closed in the 0.12 development cycle
• 834: Issue https://github.com/ipython/ipython/issues/832 resolution
• 746: ENH: extensions: port autoreload to current API
• 828: fixed permissions (sub-modules should not be executable) + added shebang for run_ipy_in_profiler.py
• 798: pexpect & Python 3
• 804: Magic ‘range’ crash if greater than len(input_hist)
• 821: update tornado dependency to 2.1
• 807: Facilitate ssh tunnel sharing by announcing ports
• 795: Add cluster-id for multiple cluster instances per profile
• 742: Glut
• 668: Greedy completer
• 776: Reworking qtconsole shortcut, add fullscreen
• 790: TST: add future unicode_literals test (#786)
• 775: redirect_in/redirect_out should be constrained to windows only
• 793: Don’t use readline in the ZMQShell
• 743: Pyglet
• 774: basic/initial .mailmap for nice shortlog summaries
• 770: #769 (reopened)
• 784: Parse user code to AST using compiler flags.
• 783: always use StringIO, never cStringIO
• 782: flush stdout/stderr on displayhook call
• 622: Make pylab import all configurable
• 745: Don’t assume history requests succeed in qtconsole
• 725: don’t assume cursor.selectedText() is a string
• 778: don’t override execfile on Python 2
• 663: Python 3 compatibility work
• 762: qtconsole ipython widget’s execute_file fails if filename contains spaces or quotes
• 763: Set context for shortcuts in ConsoleWidget
• 722: PyPy compatibility
• 757: ipython.el is broken in 0.11
• 764: fix “--colors=<color>” option in py-python-command-args.
• 758: use ROUTER/DEALER socket names instead of XREP/XREQ
• 736: enh: added authentication ability for webapp
• 748: Check for tornado before running frontend.html tests.
• 754: restore msg_id/msg_type aliases in top level of msg dict
• 769: Don’t treat bytes objects as json-safe
• 753: DOC: msg[’msg_type’] removed
• 766: fix “–colors=<color>” option in py-python-command-args.
• 765: fix “–colors=<color>” option in py-python-command-args.
• 741: Run PyOs_InputHook in pager to keep plot windows interactive.
• 664: Remove ipythonrc references from documentation
• 750: Tiny doc fixes
• 433: ZMQ terminal frontend
• 734: Allow %magic argument filenames with spaces to be specified with quotes under win32
• 731: respect encoding of display data from urls
• 730: doc improvements for running notebook via secure protocol
• 729: use null char to start markdown cell placeholder
• 727: Minor fixes to the Htmlnotebook
• 726: use bundled argparse if system argparse is < 1.1
• 705: Htmlnotebook
• 723: Add ‘import time’ to IPython/parallel/apps/launcher.py as time.sleep is called without time being imported
• 714: Install mathjax for offline use
• 718: Underline keyboard shortcut characters on appropriate buttons
• 717: Add source highlighting to markdown snippets
• 716: update EvalFormatter to allow arbitrary expressions
• 712: Reset execution counter after cache is cleared
• 713: Align colons in html notebook help dialog
• 709: Allow usage of ‘.’ in notebook names
• 706: Implement static publishing of HTML notebook
• 674: use argparse to parse aliases & flags
• 679: HistoryManager.get_session_info()
• 696: Fix columnize bug, where tab completion with very long filenames would crash Qt console
• 686: add ssh tunnel support to qtconsole

2.14. Issues closed in the 0.12 development cycle
• 685: Add SSH tunneling to engines
• 384: Allow pickling objects defined interactively.
• 647: My fix rpmlint
• 587: don’t special case for py3k+numpy
• 703: make config-loading debug messages more explicit
• 699: make calltips configurable in qtconsole
• 666: parallel tests & extra readline escapes
• 683: BF - allow nose with-doctest setting in environment
• 689: Protect ipkernel from bad messages
• 702: Prevent ipython.py launcher from being imported.
• 701: Prevent ipython.py from being imported by accident
• 670: check for writable dirs, not just existence, in utils.path
• 579: Sessionwork
• 687: add ipython kernel for starting just a kernel
• 627: Qt Console history search
• 646: Generate package list automatically in find_packages
• 660: i658
• 659: don’t crash on bad config files

Regular issues (258):
• 1177: UnicodeDecodeError in py3compat from “xlrd??”
• 1094: Tooltip doesn’t show constructor docstrings
• 1170: double pylab greeting with c.InteractiveShellApp.pylab = “tk” in zmqconsole
• 1166: E-mail cpaste broken
• 1164: IPython qtconsole (0.12) can’t export to html with external png
• 1103: %loadpy should cut out encoding declaration
• 1156: Notebooks downloaded as Python files require a header stating the encoding
• 1157: Ctrl-C not working when GUI/pylab integration is active
• 1154: We should be less aggressive in de-registering post-execution functions
• 1134: “select-all, kill” leaves qtconsole in unusable state
• 1148: A lot of testerrors
• 803: Make doctests work with Python 3
• 1119: Start menu shortcuts not created in Python 3
• 1136: The embedding machinery ignores user_ns
• 607: Use the new IPython logo/font in the notebook header
• 755: qtconsole ipython widget’s execute_file fails if filename contains spaces or quotes
• 1115: shlex_split should return unicode
• 1109: timeit with string ending in space gives “ValueError: No closing quotation”
• 1142: Install problems
• 700: Some SVG images render incorrectly in htmlnotebook
• 1117: quit() doesn’t work in terminal
• 1111: ls broken after merge of #1089
• 1104: Prompt spacing weird
• 1124: Seg Fault 11 when calling PySide using “run” command
• 1088: QtConsole : can’t copy from pager
• 568: Test error and failure in IPython.core on windows
• 1112: testfailure in IPython.frontend on windows
• 1102: magic in IPythonDemo fails when not located at top of demo file
• 629: r and b in qtconsole don’t behave as expected
• 1080: Notebook: tab completion should close on “(
• 973: Qt Console close dialog and on-top Qt Console
• 1087: QtConsole xhtml/Svg export broken ?
• 1067: Parallel test suite hangs on Python 3
• 1018: Local mathjax breaks install
• 993: raw_input redirection to foreign kernels is extremely brittle
• 1100: ipython3 traceback unicode issue from extensions
• 1071: Large html-notebooks hang on load on a slow machine
• 89: %pdoc np.ma.compress shows docstring twice
• 22: Include improvements from anythingipython.el
• 633: Execution count & SyntaxError
• 1095: Uncaught TypeError: Object has no method ‘remove_and_cancell_tooltip’
• 1075: We’re ignoring prompt customizations
• 1086: Can’t open qtconsole from outside source tree
• 1076: namespace changes broke foo.*bar*? syntax
• 1074: pprinting old-style class objects fails (TypeError: ‘tuple’ object is not callable)
- 1063: IPython.utils test error due to missing unicodedata module
- 592: Bug in argument parsing for %run
- 378: Windows path escape issues
- 1068: Notebook tab completion broken in Firefox
- 75: No tab completion after “/”
- 103: customizable cpaste
- 324: Remove code in IPython.testing that is not being used
- 131: Global variables not seen by cprofile.run()
- 851: IPython shell swallows exceptions in certain circumstances
- 882: ipython freezes at start if IPYTHONDIR is on an NFS mount
- 1057: Blocker: Qt console broken after “all magics” menu became dynamic
- 1027: ipython does not like white space at end of file
- 1058: New bug: Notebook asks for confirmation to leave even saved pages.
- 1061: rep (magic recall) under pypy
- 1047: Document the notebook format
- 102: Properties accessed twice for classes defined interactively
- 16: %store raises exception when storing compiled regex
- 67: tab expansion should only take one directory level at the time
- 62: Global variables undefined in interactive use of embedded ipython shell
- 57: debugging with ipython does not work well outside ipython
- 38: Line entry edge case error
- 980: Update parallel docs for new parallel architecture
- 1017: Add small example about ipcluster/ssh startup
- 1041: Proxy Issues
- 967: KernelManagers don’t use zmq eventloop properly
- 1055: “All Magics” display on Ubuntu
- 1054: ipython explodes on syntax error
- 1051: ipython3 set_next_input() failure
- 693: “run -i” no longer works after %reset in terminal
- 29: cPickle works in standard interpreter, but not in IPython
- 1050: ipython3 broken by commit 8bb887c8c2e447bf7
- 1048: Update docs on notebook password
• 1046: Series of questions/issues?
• 1045: crash when exiting - previously launched embedded sub-shell
• 1043: pylab doesn’t work in qtconsole
• 1044: run -p doesn’t work in python 3
• 1010: emacs freezes when ipython-complete is called
• 82: Update devel docs with discussion about good changelogs
• 116: Update release management scripts and release.revision for git
• 1022: Pylab banner shows up with first cell to execute
• 787: Keyboard selection of multiple lines in the notebook behaves inconsistently
• 1037: notepad + jsonlib: TypeError: Only whitespace may be used for indentation.
• 970: Default home not writable, %HOME% does not help (windows)
• 747: HOMESHARE not a good choice for “writable homedir” on Windows
• 810: cleanup utils.path.get_home_dir
• 2: Fix the copyright statement in source code files to be accurate
• 1031: <esc> on Firefox crash websocket
• 684: %Store eliminated in configuration and magic commands in 0.11
• 1026: BUG: wrong default parameter in ask_yes_no
• 880: Better error message if %paste fails
• 1024: autopx magic broken
• 822: Unicode bug in Itpl when expanding shell variables in syscalls with !
• 1009: Windows: regression in cd magic handling of paths
• 833: Crash python with matplotlib and unequal length arrays
• 695: Crash handler initialization is too aggressive
• 1000: Remove duplicates when refilling readline history
• 992: Interrupting certain matplotlib operations leaves the inline backend ‘wedged’
• 942: number traits should cast if value doesn’t change
• 1006: ls crashes when run on a UNC path or with non-ascii args
• 944: Decide the default image format for inline figures: SVG or PNG?
• 842: Python 3 on Windows (pyreadline) - expected an object with the buffer interface
• 1002: ImportError due to incorrect version checking
• 1001: Ipython “source” command?
• 954: IPython embed doesn’t respect namespaces

2.14. Issues closed in the 0.12 development cycle
• 681: pdb freezes inside qtconsole
• 698: crash report “TypeError: can only concatenate list (not “unicode”) to list”
• 978: ipython 0.11 buffers external command output till the cmd is done
• 952: Need user-facing warning in the browser if websocket connection fails
• 988: Error using idsave
• 990: ipython notebook - kernel dies if matplotlib is not installed
• 752: Matplotlib figures showed only once in notebook
• 54: Exception hook should be optional for embedding IPython in GUIs
• 918: IPython.frontend tests fail without tornado
• 986: Views created with c.direct_view() fail
• 697: Filter out from %who names loaded at initialization time
• 932: IPython 0.11 quickref card has superfluous “%recall and”
• 982: png files with executable permissions
• 914: Simpler system for running code after InteractiveShell is initialised
• 911: ipython crashes on startup if readline is missing
• 971: bookmarks created in 0.11 are corrupt in 0.12
• 974: object feature tab-completion crash
• 939: ZMQShell always uses default profile
• 946: Multi-tab Close action should offer option to leave all kernels alone
• 949: Test suite must not require any manual interaction
• 643: enable gui eventloop integration in ipkernel
• 965: ipython is crashed without launch.(python3.2)
• 958: Can’t use os X clipboard on with qtconsole
• 962: Don’t require tornado in the tests
• 960: crash on syntax error on Windows XP
• 934: The latest ipython branch doesn’t work in Chrome
• 870: zmq version detection
• 943: HISTIGNORE for IPython
• 947: qtconsole segfaults at startup
• 903: Expose a magic to control config of the inline pylab backend
• 908: bad user config shouldn’t crash IPython
• 935: Typing break causes IPython to crash.
- 869: Tab completion of ~/ shows no output post 0.10.x
- 904: whos under pypy1.6
- 773: check_security_dir() and check_pid_dir() fail on network filesystem
- 915: OS X Lion Terminal.app line wrap problem
- 886: Notebook kernel crash when specifying –notebook-dir on commandline
- 636: debugger.py: pydb broken
- 808: Ctrl+C during %reset confirm message crash Qtconsole
- 927: Using return outside a function crashes ipython
- 919: Pop-up segfault when moving cursor out of qtconsole window
- 181: cls command does not work on windows
- 917: documentation typos
- 818: %run does not work with non-ascii characters in path
- 907: Errors in custom completer functions can crash IPython
- 867: doc: notebook password authentication howto
- 211: paste command not working
- 900: Tab key should insert 4 spaces in qt console
- 513: [Qt console] cannot insert new lines into console functions using tab
- 906: qtconsoleapp ‘parse_command_line’ doesn’t like –existing anymore
- 638: Qt console –pylab=inline and getfigs(), etc.
- 710: unwanted unicode passed to args
- 436: Users should see tooltips for all buttons in the notebook UI
- 207: ipython crashes if atexit handler raises exception
- 692: use of Tracer() when debugging works but gives error messages
- 690: debugger does not print error message by default in 0.11
- 571: history of multiline entries
- 749: IPython.parallel test failure under Windows 7 and XP
- 890: ipclusterapp.py - helep
- 885: ws-hostname alias not recognized by notebook
- 881: Missing manual.pdf?
- 744: cannot create notebook in offline mode if mathjax not installed
- 865: Make tracebacks from %paste show the code
- 535: exception unicode handling in %run is faulty in qtconsole

2.14. Issues closed in the 0.12 development cycle
• 817: iPython crashed
• 799: %edit magic not working on windows xp in qtconsole
• 732: QTConsole wrongly promotes the index of the input line on which user presses Enter
• 662: ipython test failures on Mac OS X Lion
• 650: Handle bad config files better
• 829: We should not insert new lines after all print statements in the notebook
• 874: ipython-qtconsole: pyzmq Version Comparison
• 640: matplotlib macosx windows don’t respond in qtconsole
• 624: ipython intermittently segfaults when figure is closed (Mac OS X)
• 871: Notebook crashes if a profile is used
• 56: Have %cpaste accept also Ctrl-D as a termination marker
• 849: Command line options to not override profile options
• 806: Provide single-port connection to kernels
• 691: [ wishlist] Automatically find existing kernel
• 688: local security vulnerability: all ports visible to any local user.
• 866: DistributionNotFound on running ipython 0.11 on Windows XP x86
• 673: raw_input appears to be round-robin for qtconsole
• 863: Graceful degradation when home directory not writable
• 800: Timing scripts with run -t -N <N> fails on report output
• 858: Typing ‘continue’ makes ipython0.11 crash
• 840: all processes run on one CPU core
• 843: “import braces” crashes ipython
• 836: Strange Output after IPython Install
• 839: Qtconsole segfaults when mouse exits window with active tooltip
• 827: Add support for checking several limits before running task on engine
• 826: Add support for creation of parallel task when no engine is running
• 832: Improve error message for %logstop
• 831: %logstart in read-only directory forbid any further command
• 814: ipython does not start – DistributionNotFound
• 794: Allow >1 controller per profile
• 820: Tab Completion feature
• 812: Qt console crashes on Ubuntu 11.10
• 816: Import error using Python 2.7 and dateutil2.0 No module named _thread
• 756: qtconsole Windows fails to print error message for ‘%run nonexistent_file’
• 651: Completion doesn’t work on element of a list
• 617: [qtconsole] %hist doesn’t show anything in qtconsole
• 786: from __future__ import unicode_literals does not work
• 779: Using irunner from virtual evn uses systemwide ipython
• 768: codepage handling of output from scripts and shellcommands are not handled properly by qt-console
• 785: Don’t strip leading whitespace in repr() in notebook
• 737: in pickleshare.py line52 should be “if not os.path.isdir(self.root):”?
• 738: in ipthon_win_post_install.py line 38
• 777: print(..., sep=...) raises SyntaxError
• 728: ipcontroller crash with MPI
• 780: qtconsole Out value prints before the print statements that precede it
• 632: IPython Crash Report (0.10.2)
• 253: Unable to install ipython on windows
• 80: Split IPClusterApp into multiple Application subclasses for each subcommand
• 34: non-blocking pendingResult partial results
• 739: Tests fail if tornado not installed
• 719: Better support Pypy
• 667: qtconsole problem with default pylab profile
• 661: ipythonrc referenced in magic command in 0.11
• 665: Source introspection with ?? is broken
• 724: crash - ipython qtconsole, %quickref
• 655: ipython with qtconsole crashes
• 593: HTML Notebook Prompt can be deleted . . .
• 563: use argparse instead of kvloader for flags&aliases
• 751: Tornado version greater than 2.0 needed for firefox 6
• 720: Crash report when importing easter egg
• 740: Ctrl-Enter clears line in notebook
• 772: ipengine fails on Windows with “XXX lineno: 355, opcode: 0”
• 771: Add python 3 tag to setup.py

2.14. Issues closed in the 0.12 development cycle
• 767: non-ascii in __doc__ string crashes qtconsole kernel when showing tooltip
• 733: In Windows, %run fails to strip quotes from filename
• 721: no completion in emacs by ipython(ipython.el)
• 669: Do not accept an ipython_dir that’s not writeable
• 711: segfault on mac os x
• 500: “RuntimeError: Cannot change input buffer during execution” in console_widget.py
• 707: Copy and paste keyboard shortcuts do not work in Qt Console on OS X
• 478: PyZMQ’s use of memoryviews breaks reconstruction of numpy arrays
• 694: Turning off callout tips in qtconsole
• 704: return kills IPython
• 442: Users should have intelligent autoindenting in the notebook
• 615: Wireframe and implement a project dashboard page
• 614: Wireframe and implement a notebook dashboard page
• 606: Users should be able to use the notebook to import/export a notebook to .py or .rst
• 604: A user should be able to leave a kernel running in the notebook and reconnect
• 298: Users should be able to save a notebook and then later reload it
• 649: ipython qtconsole (v0.11): setting “c.IPythonWidget.in_prompt = ‘>>> ’ crashes
• 672: What happened to Exit?
• 658: Put the InteractiveShellApp section first in the auto-generated config files
• 656: [suggestion] dependency checking for pyqt for Windows installer
• 654: broken documentation link on download page
• 653: Test failures in IPython.parallel

2.15 0.11 Series

2.15.1 Release 0.11

IPython 0.11 is a major overhaul of IPython, two years in the making. Most of the code base has been rewritten or at least reorganized, breaking backward compatibility with several APIs in previous versions. It is the first major release in two years, and probably the most significant change to IPython since its inception. We plan to have a relatively quick succession of releases, as people discover new bugs and regressions. Once we iron out any significant bugs in this process and settle down the new APIs, this series will become IPython 1.0. We encourage feedback now on the core APIs, which we hope to maintain stable during the 1.0 series.

Since the internal APIs have changed so much, projects using IPython as a library (as opposed to end-users of the application) are the most likely to encounter regressions or changes that break their existing use
patterns. We will make every effort to provide updated versions of the APIs to facilitate the transition, and we encourage you to contact us on the development mailing list with questions and feedback.

Chris Fonnesbeck recently wrote an excellent post that highlights some of our major new features, with examples and screenshots. We encourage you to read it as it provides an illustrated, high-level overview complementing the detailed feature breakdown in this document.

A quick summary of the major changes (see below for details):

- **Standalone Qt console**: a new rich console has been added to IPython, started with `ipython qtconsole`. In this application we have tried to retain the feel of a terminal for fast and efficient workflows, while adding many features that a line-oriented terminal simply can not support, such as inline figures, full multiline editing with syntax highlighting, graphical tooltips for function calls and much more. This development was sponsored by Enthought Inc.. See below for details.

- **High-level parallel computing with ZeroMQ**: Using the same architecture that our Qt console is based on, we have completely rewritten our high-level parallel computing machinery that in prior versions used the Twisted networking framework. While this change will require users to update their codes, the improvements in performance, memory control and internal consistency across our codebase convinced us it was a price worth paying. We have tried to explain how to best proceed with this update, and will be happy to answer questions that may arise. A full tutorial describing these features was presented at SciPy’11, more details below.

- **New model for GUI/plotting support in the terminal**: Now instead of the various `-Xthread` flags we had before, GUI support is provided without the use of any threads, by directly integrating GUI event loops with Python’s `PyOS_InputHook` API. A new command-line flag `--gui` controls GUI support, and it can also be enabled after IPython startup via the new `%gui` magic. This requires some changes if you want to execute GUI-using scripts inside IPython, see the GUI support section for more details.

- **A two-process architecture**: The Qt console is the first use of a new model that splits IPython between a kernel process where code is executed and a client that handles user interaction. We plan on also providing terminal and web-browser based clients using this infrastructure in future releases. This model allows multiple clients to interact with an IPython process through a well-documented messaging protocol using the ZeroMQ networking library.

- **Refactoring**: the entire codebase has been refactored, in order to make it more modular and easier to contribute to. IPython has traditionally been a hard project to participate because the old codebase was very monolithic. We hope this (ongoing) restructuring will make it easier for new developers to join us.

- **Vim integration**: Vim can be configured to seamlessly control an IPython kernel, see the files in `docs/examples/vim` for the full details. This work was done by Paul Ivanov, who prepared a nice video demonstration of the features it provides.

- **Integration into Microsoft Visual Studio**: Thanks to the work of the Microsoft Python Tools for Visual Studio team, this version of IPython has been integrated into Microsoft Visual Studio’s Python tools open source plug-in. Details below

- **Improved unicode support**: We closed many bugs related to unicode input.

- **Python 3**: IPython now runs on Python 3.x. See Python 3 support for details.
• **New profile model.** Profiles are now directories that contain all relevant information for that session, and thus better isolate IPython use-cases.

• **SQLite storage for history.** All history is now stored in a SQLite database, providing support for multiple simultaneous sessions that won’t clobber each other as well as the ability to perform queries on all stored data.

• **New configuration system.** All parts of IPython are now configured via a mechanism inspired by the Enthought Traits library. Any configurable element can have its attributes set either via files that now use real Python syntax or from the command-line.

• **Pasting of code with prompts.** IPython now intelligently strips out input prompts, be they plain Python ones (```>>>``` and `...`) or IPython ones (`In [N]:` and `...:`). More details [here](#).

**Authors and support**

Over 60 separate authors have contributed to this release, see [below](#) for a full list. In particular, we want to highlight the extremely active participation of two new core team members: Evan Patterson implemented the Qt console, and Thomas Kluvyer started with our Python 3 port and by now has made major contributions to just about every area of IPython.

We are also grateful for the support we have received during this development cycle from several institutions:

• **Enthought Inc** funded the development of our new Qt console, an effort that required developing major pieces of underlying infrastructure, which now power not only the Qt console but also our new parallel machinery. We’d like to thank Eric Jones and Travis Oliphant for their support, as well as Ilan Schnell for his tireless work integrating and testing IPython in the Enthought Python Distribution.

• Nipy/NIH: funding via the NiPy project (NIH grant 5R01MH081909-02) helped us jumpstart the development of this series by restructuring the entire codebase two years ago in a way that would make modular development and testing more approachable. Without this initial groundwork, all the new features we have added would have been impossible to develop.

• Sage/NSF: funding via the grant Sage: Unifying Mathematical Software for Scientists, Engineers, and Mathematicians (NSF grant DMS-1015114) supported a meeting in spring 2011 of several of the core IPython developers where major progress was made integrating the last key pieces leading to this release.

• Microsoft’s team working on Python Tools for Visual Studio developed the integraton of IPython into the Python plugin for Visual Studio 2010.

• Google Summer of Code: in 2010, we had two students developing prototypes of the new machinery that is now maturing in this release: [Omar Zapata](#) and [Gerardo Gutiérrez](#).

**Development summary: moving to Git and Github**

In April 2010, after one breakage too many with bzr, we decided to move our entire development process to Git and Github.com. This has proven to be one of the best decisions in the project’s history, as the combination of git and github have made us far, far more productive than we could be with our previous tools. We first converted our bzr repo to a git one without losing history, and a few weeks later ported all open Launchpad bugs to github issues with their comments mostly intact (modulo some formatting changes).
This ensured a smooth transition where no development history or submitted bugs were lost. Feel free to use our little Launchpad to Github issues porting script if you need to make a similar transition.

These simple statistics show how much work has been done on the new release, by comparing the current code to the last point it had in common with the 0.10 series. A huge diff and ~2200 commits make up this cycle:

```
  git diff $(git merge-base 0.10.2 HEAD)  | wc -l
  288019

  git log $(git merge-base 0.10.2 HEAD)..<HEAD --oneline | wc -l
  2200
```

Since our move to github, 511 issues were closed, 226 of which were pull requests and 285 regular issues (a full list with links is available for those interested in the details). Github’s pull requests are a fantastic mechanism for reviewing code and building a shared ownership of the project, and we are making enthusiastic use of it.

**Note:** This undercounts the number of issues closed in this development cycle, since we only moved to github for issue tracking in May 2010, but we have no way of collecting statistics on the number of issues closed in the old Launchpad bug tracker prior to that.

---

**Qt Console**

IPython now ships with a Qt application that feels very much like a terminal, but is in fact a rich GUI that runs an IPython client but supports inline figures, saving sessions to PDF and HTML, multiline editing with syntax highlighting, graphical calltips and much more:

We hope that many projects will embed this widget, which we’ve kept deliberately very lightweight, into their own environments. In the future we may also offer a slightly more featureful application (with menus and other GUI elements), but we remain committed to always shipping this easy to embed widget.

See the Jupyter Qt Console site for a detailed description of the console’s features and use.

---

**High-level parallel computing with ZeroMQ**

We have completely rewritten the Twisted-based code for high-level parallel computing to work atop our new ZeroMQ architecture. While we realize this will break compatibility for a number of users, we hope to make the transition as easy as possible with our docs, and we are convinced the change is worth it. ZeroMQ provides us with much tighter control over memory, higher performance, and its communications are impervious to the Python Global Interpreter Lock because they take place in a system-level C++ thread. The impact of the GIL in our previous code was something we could simply not work around, given that Twisted is itself a Python library. So while Twisted is a very capable framework, we think ZeroMQ fits our needs much better and we hope you will find the change to be a significant improvement in the long run.

Our manual contains a full description of how to use IPython for parallel computing, and the tutorial presented by Min Ragan-Kelley at the SciPy 2011 conference provides a hands-on complement to the reference docs.
Fig. 2.5: The Qt console for IPython, using inline matplotlib plots.

In [1]: run recarr_simple.py

Seismic stations in the Himalaya

In [2]: from scipy import special as sp
   ...: x = linspace(0, 20, 100)
   ...: for n in range(4):
   ...:     y = sp.jn(n, x)
   ...:     plot(x, y, label=r'$J_{%d}(x)$' % n)
   ...:     axhline(0, color='green', label='_nolegend_')
   ...:     grid()
   ...:     legend()
   ...:     xlabel('$x$')
   ...:     title(r'\textit{Bessel functions $J_{%d}(x)$}')

Out[2]: <matplotlib.text.Text object at 0x7fcdc1795d0>
Refactoring

As of this release, a significant portion of IPython has been refactored. This refactoring is founded on a number of new abstractions. The main new classes that implement these abstractions are:

- `traitlets.HasTraits`
- `traitlets.config.configurable.Configurable`
- `traitlets.config.application.Application`
- `traitlets.config.loader.ConfigLoader`
- `traitlets.config.loader.Config`

We are still in the process of writing developer focused documentation about these classes, but for now our configuration documentation contains a high level overview of the concepts that these classes express.

The biggest user-visible change is likely the move to using the config system to determine the command-line arguments for IPython applications. The benefit of this is that all configurable values in IPython are exposed on the command-line, but the syntax for specifying values has changed. The gist is that assigning values is pure Python assignment. Simple flags exist for commonly used options, these are always prefixed with `-`. The IPython command-line help has the details of all the options (via `ipython --help`), but a simple example should clarify things; the `pylab` flag can be used to start in pylab mode with the qt4 backend:

```
ipython --pylab=qt
```

which is equivalent to using the fully qualified form:

```
ipython --TerminalIPythonApp.pylab=qt
```

The long-form options can be listed via `ipython --help-all`.

ZeroMQ architecture

There is a new GUI framework for IPython, based on a client-server model in which multiple clients can communicate with one IPython kernel, using the ZeroMQ messaging framework. There is already a Qt console client, which can be started by calling `ipython qtconsole`. The protocol is documented.

The parallel computing framework has also been rewritten using ZMQ. The protocol is described here, and the code is in the new `IPython.parallel` module.

Python 3 support

A Python 3 version of IPython has been prepared. For the time being, this is maintained separately and updated from the main codebase. Its code can be found here. The parallel computing components are not perfect on Python3, but most functionality appears to be working. As this work is evolving quickly, the best place to find updated information about it is our Python 3 wiki page.
Unicode

Entering non-ascii characters in unicode literals (u"€") now works properly on all platforms. However, entering these in byte/string literals ("€") will not work as expected on Windows (or any platform where the terminal encoding is not UTF-8, as it typically is for Linux & Mac OS X). You can use escape sequences ("\xe9\xe8") to get bytes above 128, or use unicode literals and encode them. This is a limitation of Python 2 which we cannot easily work around.

Integration with Microsoft Visual Studio

IPython can be used as the interactive shell in the Python plugin for Microsoft Visual Studio, as seen here:

![IPython console embedded in Microsoft Visual Studio.](image)

Fig. 2.6: IPython console embedded in Microsoft Visual Studio.

The Microsoft team developing this currently has a release candidate out using IPython 0.11. We will continue to collaborate with them to ensure that as they approach their final release date, the integration with IPython remains smooth. We’d like to thank Dino Viehland and Shahrokh Mortazavi for the work they have done towards this feature, as well as Wenming Ye for his support of our WinHPC capabilities.

Additional new features

- Added Bytes traitlet, removing Str. All ‘string’ traitlets should either be Unicode if a real string, or Bytes if a C-string. This removes ambiguity and helps the Python 3 transition.
- New magic %loadpy loads a python file from disk or web URL into the current input buffer.
• New magic `%pastebin` for sharing code via the ‘Lodge it’ pastebin.
• New magic `%precision` for controlling float and numpy pretty printing.
• IPython applications initiate logging, so any object can gain access to a the logger of the currently running Application with:

```python
from traitlets.config.application import Application
logger = Application.instance().log
```

• You can now get help on an object halfway through typing a command. For instance, typing `a = zip?` shows the details of `zip()`. It also leaves the command at the next prompt so you can carry on with it.

• The input history is now written to an SQLite database. The API for retrieving items from the history has also been redesigned.

• The IPython.extensions.pretty extension has been moved out of quarantine and fully updated to the new extension API.

• New magics for loading/unloading/reloading extensions have been added: `%load_ext`, `%unload_ext` and `%reload_ext`.

• The configuration system and configuration files are brand new. See the configuration system documentation for more details.

• The `InteractiveShell` class is now a Configurable subclass and has traits that determine the defaults and runtime environment. The `__init__` method has also been refactored so this class can be instantiated and run without the old `ipmaker` module.

• The methods of `InteractiveShell` have been organized into sections to make it easier to turn more sections of functionality into components.

• The embedded shell has been refactored into a truly standalone subclass of `InteractiveShell` called `InteractiveShellEmbed`. All embedding logic has been taken out of the base class and put into the embedded subclass.

• Added methods of `InteractiveShell` to help it cleanup after itself. The `cleanup()` method controls this. We couldn’t do this in `__del__()` because we have cycles in our object graph that prevent it from being called.

• Created a new module `IPython.utils.importstring` for resolving strings like `foo.bar.Bar` to the actual class.

• Completely refactored the `IPython.core.prefilter` module into Configurable subclasses. Added a new layer into the prefilter system, called “transformations” that all new prefilter logic should use (rather than the older “checker/handler” approach).

• Aliases are now components (`IPython.core.alias`).

• New top level `embed()` function that can be called to embed IPython at any place in user’s code. On the first call it will create an `InteractiveShellEmbed` instance and call it. In later calls, it just calls the previously created `InteractiveShellEmbed`.

• Created a configuration system (`traitlets.config.configurable`) that is based on `traitlets`. Configurables are arranged into a runtime containment tree (not inheritance) that i)
automatically propagates configuration information and ii) allows singletons to discover each other in a loosely coupled manner. In the future all parts of IPython will be subclasses of Configurable. All IPython developers should become familiar with the config system.

- Created a new Config for holding configuration information. This is a dict like class with a few extras: i) it supports attribute style access, ii) it has a merge function that merges two Config instances recursively and iii) it will automatically create sub-Config instances for attributes that start with an uppercase character.

- Created new configuration loaders in traitlets.config.loader. These loaders provide a unified loading interface for all configuration information including command line arguments and configuration files. We have two default implementations based on argparse and plain python files. These are used to implement the new configuration system.

- Created a top-level Application class in IPython.core.application that is designed to encapsulate the starting of any basic Python program. An application loads and merges all the configuration objects, constructs the main application, configures and initiates logging, and creates and configures any Configurable instances and then starts the application running. An extended BaseIPythonApplication class adds logic for handling the IPython directory as well as profiles, and all IPython entry points extend it.

- The Type and Instance traitlets now handle classes given as strings, like foo.bar.Bar. This is needed for forward declarations. But, this was implemented in a careful way so that string to class resolution is done at a single point, when the parent HasTraits is instantiated.

- IPython.util.ipstruct has been refactored to be a subclass of dict. It also now has full docstrings and doctests.

- Created a Traits like implementation in traitlets. This is a pure Python, lightweight version of a library that is similar to Enthought’s Traits project, but has no dependencies on Enthought’s code. We are using this for validation, defaults and notification in our new component system. Although it is not 100% API compatible with Enthought’s Traits, we plan on moving in this direction so that eventually our implementation could be replaced by a (yet to exist) pure Python version of Enthought Traits.

- Added a new module IPython.lib.inputhook to manage the integration with GUI event loops using PyOS_InputHook. See the docstrings in this module or the main IPython docs for details.

- For users, GUI event loop integration is now handled through the new %gui magic command. Type %gui? at an IPython prompt for documentation.

- For developers IPython.lib.inputhook provides a simple interface for managing the event loops in their interactive GUI applications. Examples can be found in our examples/lib directory.

**Backwards incompatible changes**

- The Twisted-based IPython.kernel has been removed, and completely rewritten as IPython.parallel, using ZeroMQ.

- Profiles are now directories. Instead of a profile being a single config file, profiles are now self-contained directories. By default, profiles get their own IPython history, log files, and everything. To create a new profile, do ipython profile create <name>.
• All IPython applications have been rewritten to use `KeyValueConfigLoader`. This means that command-line options have changed. Now, all configurable values are accessible from the command-line with the same syntax as in a configuration file.

• The command line options `-wthread`, `-qthread` and `-gthread` have been removed. Use `--gui=wx`, `--gui=qt`, `--gui=gtk` instead.

• The extension loading functions have been renamed to `load_ipython_extension()` and `unload_ipython_extension()`.

• `InteractiveShell` no longer takes an embedded argument. Instead just use the `InteractiveShellEmbed` class.

• `__IPYTHON__` is no longer injected into `__builtin__`.

• `Struct.__init__()` no longer takes None as its first argument. It must be a `dict` or `Struct`.

• `ipmagic()` has been renamed `()`.

• The functions `ipmagic()` and `ipalias()` have been removed from `__builtins__`.

• The references to the global `InteractiveShell` instance `_ip` and `__IP`) have been removed from the user’s namespace. They are replaced by a new function called `get_ipython()` that returns the current `InteractiveShell` instance. This function is injected into the user’s namespace and is now the main way of accessing the running IPython.

• Old style configuration files `ipythonrc` and `ipy_user_conf.py` are no longer supported. Users should migrate there configuration files to the new format described here and here.

• The old IPython extension API that relied on `ipapi()` has been completely removed. The new extension API is described here.

• Support for `qt3` has been dropped. Users who need this should use previous versions of IPython.

• Removed `shellglobals` as it was obsolete.

• Removed all the threaded shells in `IPython.core.shell`. These are no longer needed because of the new capabilities in `IPython.lib.inputhook`.

• New top-level sub-packages have been created: `IPython.core`, `IPython.lib`, `IPython.utils`, `IPython.deathrow`, `IPython.quarantine`. All existing top-level modules have been moved to appropriate sub-packages. All internal import statements have been updated and tests have been added. The build system (setup.py and friends) have been updated. See The IPython API for details of these new sub-packages.

• `IPython.ipapi` has been moved to `IPython.core.ipapi`. `IPython.Shell` and `IPython.iplib` have been split and removed as part of the refactor.

• Extensions has been moved to extensions and all existing extensions have been moved to either `IPython.quarantine` or `IPython.deathrow`. `IPython.quarantine` contains modules that we plan on keeping but that need to be updated. `IPython.deathrow` contains modules that are either dead or that should be maintained as third party libraries.

• Previous IPython GUIs in `IPython.frontend` and `IPython.gui` are likely broken, and have been removed to `IPython.deathrow` because of the refactoring in the core. With proper updates, these should still work.
Known Regressions

We do our best to improve IPython, but there are some known regressions in 0.11 relative to 0.10.2. First of all, there are features that have yet to be ported to the new APIs, and in order to ensure that all of the installed code runs for our users, we have moved them to two separate directories in the source distribution, quarantine and deathrow. Finally, we have some other miscellaneous regressions that we hope to fix as soon as possible. We now describe all of these in more detail.

Quarantine

These are tools and extensions that we consider relatively easy to update to the new classes and APIs, but that we simply haven’t had time for. Any user who is interested in one of these is encouraged to help us by porting it and submitting a pull request on our development site.

Currently, the quarantine directory contains:

<table>
<thead>
<tr>
<th>clearcmd.py</th>
<th>ipy_fsops.py</th>
<th>ipy_signals.py</th>
</tr>
</thead>
<tbody>
<tr>
<td>envpersist.py</td>
<td>ipy_gnuglobal.py</td>
<td>ipy_synchronize_with.py</td>
</tr>
<tr>
<td>ext_rescapture.py</td>
<td>ipy_greedycompleter.py</td>
<td>ipy_system_conf.py</td>
</tr>
<tr>
<td>InterpreterExec.py</td>
<td>ipy_jot.py</td>
<td>ipy_winpdb.py</td>
</tr>
<tr>
<td>ipy_app_completers.py</td>
<td>ipy_profile_doctest.py</td>
<td>ipy_workdir.py</td>
</tr>
<tr>
<td>ipy_autoreload.py</td>
<td>ipy_pydb.py</td>
<td>jobctrl.py</td>
</tr>
<tr>
<td>ipy_completers.py</td>
<td>ipy_rehashdir.py</td>
<td>ledit.py</td>
</tr>
<tr>
<td>ipy_editors.py</td>
<td>ipy_renderer.py</td>
<td>papersistence.py</td>
</tr>
<tr>
<td>ipy_exportdb.py</td>
<td>ipy_server.py</td>
<td>win32clip.py</td>
</tr>
<tr>
<td>ipy_extutil.py</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Deathrow

These packages may be harder to update or make most sense as third-party libraries. Some of them are completely obsolete and have been already replaced by better functionality (we simply haven’t had the time to carefully weed them out so they are kept here for now). Others simply require fixes to code that the current core team may not be familiar with. If a tool you were used to is included here, we encourage you to contact the dev list and we can discuss whether it makes sense to keep it in IPython (if it can be maintained).

Currently, the deathrow directory contains:

<table>
<thead>
<tr>
<th>astyle.py</th>
<th>ipy_defaults.py</th>
<th>ipy_vimserver.py</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtutils.py</td>
<td>ipy_kitcfg.py</td>
<td>numeric_formats.py</td>
</tr>
<tr>
<td>Gnuplot2.py</td>
<td>ipy_legacy.py</td>
<td>numutils.py</td>
</tr>
<tr>
<td>GnuplotInteractive.py</td>
<td>ipy_p4.py</td>
<td>outputtrap.py</td>
</tr>
<tr>
<td>GnuplotRuntime.py</td>
<td>ipy_profile_none.py</td>
<td>PhysicalQInput.py</td>
</tr>
<tr>
<td>ibrowse.py</td>
<td>ipy_profile_numpy.py</td>
<td>PhysicalQInteractive.py</td>
</tr>
<tr>
<td>igrid.py</td>
<td>ipy_profile_scipy.py</td>
<td>quitter.py*</td>
</tr>
<tr>
<td>ipipe.py</td>
<td>ipy_profile_sh.py</td>
<td>scitedirector.py</td>
</tr>
<tr>
<td>iplib.py</td>
<td>ipy_profile_zope.py</td>
<td>Shell.py</td>
</tr>
<tr>
<td>ipy_constants.py</td>
<td>ipy_traits_completer.py</td>
<td>twshell.py</td>
</tr>
</tbody>
</table>
Other regressions

• The machinery that adds functionality to the ‘sh’ profile for using IPython as your system shell has not been updated to use the new APIs. As a result, only the aesthetic (prompt) changes are still implemented. We intend to fix this by 0.12. Tracked as issue 547.

• The installation of scripts on Windows was broken without setuptools, so we now depend on setuptools on Windows. We hope to fix setuptools-less installation, and then remove the setuptools dependency. Issue 539.

• The directory history `_dh` is not saved between sessions. Issue 634.

Removed Features

As part of the updating of IPython, we have removed a few features for the purposes of cleaning up the codebase and interfaces. These removals are permanent, but for any item listed below, equivalent functionality is available.

• The magics Exit and Quit have been dropped as ways to exit IPython. Instead, the lowercase forms of both work either as a bare name (`exit`) or a function call (`exit()`). You can assign these to other names using `exec_lines` in the config file.

Credits

Many users and developers contributed code, features, bug reports and ideas to this release. Please do not hesitate in contacting us if we’ve failed to acknowledge your contribution here. In particular, for this release we have contribution from the following people, a mix of new and regular names (in alphabetical order by first name):

• Aenugu Sai Kiran Reddy <saikrm08-at-gmail.com>
• andy wilson <wilson.andrew.j+github-at-gmail.com>
• Antonio Cuni <antocuni>
• Barry Wark <barrywark-at-gmail.com>
• Beetoju Anuradha <anu.beethoju-at-gmail.com>
• Benjamin Ragan-Kelley <minrk-at-Mercury.local>
• Brad Reisfeld
• Brian E. Granger <ellisonbg-at-gmail.com>
• Christoph Gohlke <cgohlke-at-uci.edu>
• Cody Precord
• dan.milstein
• Darren Dale <dsdale24-at-gmail.com>
• Dav Clark <davclark-at-berkeley.edu>
Note: This list was generated with the output of `git log dev-0.11 HEAD --format='* %aN
<aE>' | sed 's/@//g' | sed 's/>//g' | sort -u` after some cleanup. If you should be on this list, please add yourself.

### 2.16 Issues closed in the 0.11 development cycle

In this cycle, we closed a total of 511 issues, 226 pull requests and 285 regular issues; this is the full list (generated with the script `tools/github_stats.py`). We should note that a few of these were made on the 0.10.x series, but we have no automatic way of filtering the issues by branch, so this reflects all of our development over the last two years, including work already released in 0.10.2:

Pull requests (226):

- 620: Release notes and updates to GUI support docs for 0.11
- 642: fix typo in docs/examples/vim/README.rst
• 631: two-way vim-ipython integration
• 637: print is a function, this allows to properly exit ipython
• 635: support html representations in the notebook frontend
• 639: Updating the credits file
• 628: import pexpect from IPython.external in irunner
• 596: Irun
• 598: Fix templates for CrashHandler
• 590: Desktop
• 600: Fix bug with non-ascii reprs inside pretty-printed lists.
• 618: I617
• 599: Gui Qt example and docs
• 619: manpage update
• 582: Updating sympy profile to match the exec_lines of isympy.
• 578: Check to see if correct source for decorated functions can be displayed
• 589: issue 588
• 591: simulate shell expansion on %run arguments, at least tilde expansion
• 576: Show message about %paste magic on an IndentationError
• 574: Getcwd
• 565: don’t move old config files, keep nagging the user
• 575: Added more docstrings to IPython.zmq.session.
• 567: fix trailing whitespace from resetting indentation
• 564: Command line args in docs
• 560: reorder qt support in kernel
• 561: command-line suggestions
• 556: qt_for_kernel: use matplotlib rcParams to decide between PyQt4 and PySide
• 557: Update usage.py to newapp
• 555: Rm default old config
• 552: update parallel code for py3k
• 504: Updating string formatting
• 551: Make pylab import all configurable
• 496: Qt editing keybindings
• 550: Support v2 PyQt4 APIs and PySide in kernel’s GUI support
• 546: doc update
• 548: Fix sympy profile to work with sympy 0.7.
• 542: issue 440
• 533: Remove unused configobj and validate libraries from externals.
• 538: fix various tests on Windows
• 540: support –pylab flag with deprecation warning
• 537: Docs update
• 536: setup.py install depends on setuptools on Windows
• 480: Get help mid-command
• 462: Str and Bytes traitlets
• 534: Handle unicode properly in IPython.zmq.iostream
• 527: ZMQ displayhook
• 526: Handle asynchronous output in Qt console
• 528: Do not import deprecated functions from external decorators library.
• 454: New BaseIPythonApplication
• 532: Zmq unicode
• 531: Fix Parallel test
• 525: fallback on lsof if otool not found in libedit detection
• 517: Merge IPython.parallel.streamsession into IPython.zmq.session
• 521: use dict.get(key) instead of dict[key] for safety from KeyErrors
• 492: add QtConsoleApp using newapplication
• 485: terminal IPython with newapp
• 486: Use newapp in parallel code
• 511: Add a new line before displaying multiline strings in the Qt console.
• 509: i508
• 501: ignore EINTR in channel loops
• 495: Better selection of Qt bindings when QT_API is not specified
• 498: Check for .pyd as extension for binary files.
• 494: QtConsole zoom adjustments
• 490: fix UnicodeEncodeError writing SVG string to .svg file, fixes #489
• 491: add QtConsoleApp using newapplication
• 479: embed() doesn’t load default config

2.16. Issues closed in the 0.11 development cycle
• 483: Links launchpad -> github
• 419: %xdel magic
• 477: Add n to lines in the log
• 459: use os.system for shell.system in Terminal frontend
• 475: i473
• 471: Add test decorator onlyif_unicode_paths.
• 474: Fix support for raw GTK and WX matplotlib backends.
• 472: Kernel event loop is robust against random SIGINT.
• 460: Share code for magic_edit
• 469: Add exit code when running all tests with iptest.
• 464: Add home directory expansion to IPYTHON_DIR environment variables.
• 455: Bugfix with logger
• 448: Separate out skip_doctest decorator
• 453: Draft of new main BaseIPythonApplication.
• 452: Use list/tuple/dict/set subclass’s overridden __repr__ instead of the pretty
• 398: allow toggle of svg/png inline figure format
• 381: Support inline PNGs of matplotlib plots
• 413: Retries and Resubmit (#411 and #412)
• 370: Fixes to the display system
• 449: Fix issue 447 - inspecting old-style classes.
• 423: Allow type checking on elements of List,Tuple,Set traits
• 400: Config5
• 421: Generalise mechanism to put text at the next prompt in the Qt console.
• 443: pinfo code duplication
• 429: add check_pid, and handle stale PID info in ipcluster.
• 431: Fix error message in test_irunner
• 427: handle different SyntaxError messages in test_irunner
• 424: Irunner test failure
• 430: Small parallel doc typo
• 422: Make ipython-qtconsole a GUI script
• 420: Permit kernel std* to be redirected
• 408: History request
• 388: Add Emacs-style kill ring to Qt console
• 414: Warn on old config files
• 415: Prevent prefilter from crashing IPython
• 418: Minor configuration doc fixes
• 407: Update What’s new documentation
• 410: Install notebook frontend
• 406: install IPython.zmq.gui
• 393: ipdir unicode
• 397: utils.io.Term.cin/out/err -> utils.io.stdin/out/err
• 389: DB fixes and Scheduler HWM
• 374: Various Windows-related fixes to IPython.parallel
• 362: fallback on defaultencoding if filesystemencoding is None
• 382: Shell’s reset method clears namespace from last %run command.
• 385: Update iptest exclusions (fix #375)
• 383: Catch errors in querying readline which occur with pyreadline.
• 373: Remove runlines etc.
• 364: Single output
• 372: Multiline input push
• 363: Issue 125
• 361: don’t rely on setuptools for readline dependency check
• 349: Fix %autopx magic
• 355: History save thread
• 356: Usability improvements to history in Qt console
• 357: Exit autocall
• 353: Rewrite quit()/exit()/Quit()/Exit() calls as magic
• 354: Cell tweaks
• 345: Attempt to address (partly) issue ipython/#342 by rewriting quit(), exit(), etc.
• 352: #342: Try to recover as intelligently as possible if user calls magic().
• 346: Dedent prefix bugfix + tests: #142
• 348: %reset doesn’t reset prompt number.
• 347: Make ip.reset() work the same in interactive or non-interactive code.
• 343: make readline a dependency on OSX

2.16. Issues closed in the 0.11 development cycle
• 344: restore auto debug behavior
• 339: fix for issue 337: incorrect/phantom tooltips for magics
• 254: newparallel branch (add zmq.parallel submodule)
• 334: Hard reset
• 316: Unicode win process
• 332: AST splitter
• 325: Removetwisted
• 330: Magic pastebin
• 331: Tweak config loader for PyPy compatibility.
• 319: Rewrite code to restore readline history after an action
• 329: Do not store file contents in history when running a .ipy file.
• 179: Html notebook
• 323: Add missing external.pexpect to packages
• 295: Magic local scope
• 315: Unicode magic args
• 310: allow Unicode Command-Line options
• 313: Readline shortcuts
• 311: Qtconsole exit
• 312: History memory
• 294: Issue 290
• 292: Issue 31
• 252: Unicode issues
• 235: Fix history magic command’s bugs wrt to full history and add -O option to display full history
• 236: History minus p flag
• 261: Adapt magic commands to new history system.
• 282: SQLite history
• 191: Unbundle external libraries
• 199: Magic arguments
• 204: Emacs completion bugfix
• 293: Issue 133
• 249: Writing unicode characters to a log file. (IPython 0.10.2.git)
• 283: Support for 256-color escape sequences in Qt console
• 281: Refactored and improved Qt console’s HTML export facility
• 237: Fix185 (take two)
• 251: Issue 129
• 278: add basic XDG_CONFIG_HOME support
• 275: inline pylab cuts off labels on log plots
• 280: Add %precision magic
• 259: Pyside support
• 193: Make ipython cProfile-able
• 272: Magic examples
• 219: Doc magic pycat
• 221: Doc magic alias
• 230: Doc magic edit
• 224: Doc magic cpaste
• 229: Doc magic pdef
• 273: Docs build
• 228: Doc magic who
• 233: Doc magic cd
• 226: Doc magic pwd
• 218: Doc magic history
• 231: Doc magic reset
• 225: Doc magic save
• 222: Doc magic timeit
• 223: Doc magic colors
• 203: Small typos in zmq/blockingkernelmanager.py
• 227: Doc magic logon
• 232: Doc magic profile
• 264: Kernel logging
• 220: Doc magic edit
• 268: PyZMQ > 2.0.10
• 267: GitHub Pages (again)
• 266: OSX-specific fixes to the Qt console
• 255: Gitwash typo
• 265: Fix string input2
• 260: Kernel crash with empty history
• 243: New display system
• 242: Fix terminal exit
• 250: always use Session.send
• 239: Makefile command & script for GitHub Pages
• 244: My exit
• 234: Timed history save
• 217: Doc magic lsmagic
• 215: History fix
• 195: Formatters
• 192: Ready colorize bug
• 198: Windows workdir
• 174: Whitespace cleanup
• 188: Version info: update our version management system to use git.
• 158: Ready for merge
• 187: Resolved Print shortcut collision with ctrl-P emacs binding
• 183: cleanup of exit/quit commands for qt console
• 184: Logo added to sphinx docs
• 180: Cleanup old code
• 171: Expose Pygments styles as options
• 170: HTML Fixes
• 172: Fix del method exit test
• 164: Qt frontend shutdown behavior fixes and enhancements
• 167: Added HTML export
• 163: Execution refactor
• 159: Ipy3 preparation
• 155: Ready startup fix
• 152: 0.10.1 sge
• 151: mk_object_info -> object_info
• 149: Simple bug-fix

Regular issues (285):

• 630: new.py in pwd prevents ipython from starting
• 623: Execute DirectView commands while running LoadBalancedView tasks
• 437: Users should have autocompletion in the notebook
• 583: update manpages
• 594: irunner command line options defer to file extensions
• 603: Users should see colored text in tracebacks and the pager
• 597: UnicodeDecodeError: ‘ascii’ codec can’t decode byte 0xc2
• 608: Organize and layout buttons in the notebook panel sections
• 609: Implement controls in the Kernel panel section
• 611: Add kernel status widget back to notebook
• 610: Implement controls in the Cell section panel
• 612: Implement Help panel section
• 621: [qtconsole] on windows xp, cannot PageUp more than once
• 616: Store exit status of last command
• 605: Users should be able to open different notebooks in the cwd
• 302: Users should see a consistent behavior in the Out prompt in the html notebook
• 435: Notebook should not import anything by default
• 595: qtconsole command issue
• 588: ipython-qtconsole uses 100% CPU
• 586: ? + plot() Command B0rks QTConsole Strangely
• 585: %pdoc throws Errors for classes without __init__ or docstring
• 584: %pdoc throws TypeError
• 580: Client instantiation AssertionError
• 569: UnicodeDecodeError during startup
• 572: Indented command hits error
• 573: -wthread breaks indented top-level statements
• 570: “–pylab inline” vs. “–pylab=inline”
• 566: Can’t use exec_file in config file
• 562: update docs to reflect ‘–args=values’
• 558: triple quote and %s at beginning of line
• 554: Update 0.11 docs to explain Qt console and how to do a clean install
• 553: embed() fails if config files not installed
• 8: Ensure %gui qt works with new Mayavi and pylab
• 269: Provide compatibility api for IPython.Shell().start().mainloop()
• 66: Update the main What's New document to reflect work on 0.11
• 549: Don’t check for ‘linux2’ value in sys.platform
• 505: Qt windows created within imported functions won’t show()
• 545: qtconsole ignores exec_lines
• 371: segfault in qtconsole when kernel quits
• 377: Failure: error (nothing to repeat)
• 544: Ipython qtconsole pylab config issue.
• 543: RuntimeError in completer
• 440: %run filename autocompletion “The kernel heartbeat has been inactive ... ” error
• 541: log_level is broken in the ipython Application
• 369: windows source install doesn’t create scripts correctly
• 351: Make sure that the Windows installer handles the top-level IPython scripts.
• 512: Two displayhooks in zmq
• 340: Make sure that the Windows HPC scheduler support is working for 0.11
• 98: Should be able to get help on an object mid-command
• 529: unicode problem in qtconsole for windows
• 476: Separate input area in Qt Console
• 175: Qt console needs configuration support
• 156: Key history lost when debugging program crash
• 470: decorator: uses deprecated features
• 30: readline in OS X does not have correct key bindings
• 503: merge IPython.parallel.streamsession and IPython.zmq.session
• 456: pathname in document punctuated by dots not slashes
• 451: Allow switching the default image format for inline mpl backend
• 79: Implement more robust handling of config stages in Application
• 522: Encoding problems
• 524: otool should not be unconditionally called on osx
• 523: Get profile and config file inheritance working
- 519: qtconsole –pure: “TypeError: string indices must be integers, not str”
- 516: qtconsole –pure: “KeyError: ‘ismagic’”
- 520: qtconsole –pure: “TypeError: string indices must be integers, not str”
- 450: resubmitted tasks sometimes stuck as pending
- 518: JSON serialization problems with ObjectId type (MongoDB)
- 178: Channels should be named for their function, not their socket type
- 515: [ipcluster] termination on os x
- 510: qtconsole: indentation problem printing numpy arrays
- 499: “ZMQError: Interrupted system call” when saving inline figure
- 426: %edit magic fails in qtconsole
- 497: Don’t show info from .pyd files
- 493: QFont::setPointSize: Point size <= 0 (0), must be greater than 0
- 489: UnicodeEncodeError in qt.svg.save_svg
- 458: embed() doesn’t load default config
- 488: Using IPython with RubyPython leads to problems with IPython.parallel.client.client.Client.__init()
- 401: Race condition when running lbview.apply() fast multiple times in loop
- 168: Scrub Launchpad links from code, docs
- 141: garbage collection problem (revisited)
- 59: test_magic.test_obj_del fails on win32
- 457: Backgrounded Tasks not Allowed? (but easy to slip by . . .)
- 297: Shouldn’t use pexpect for subprocesses in in-process terminal frontend
- 110: magic to return exit status
- 473: OSX readline detection fails in the debugger
- 466: tests fail without unicode filename support
- 468: iptest script has 0 exit code even when tests fail
- 465: client.db_query() behaves different with SQLite and MongoDB
- 467: magic_install_default_config test fails when there is no .ipython directory
- 463: IPYTHON_DIR (and IPYTHONDIR) don’t expand tilde to ‘~’ directory
- 446: Test machinery is imported at normal runtime
- 438: Users should be able to use Up/Down for cell navigation
• 439: Users should be able to copy notebook input and output
• 291: Rename special display methods and put them lower in priority than display functions
• 447: Instantiating classes without __init__ function causes kernel to crash
• 444: Ctrl + t in WxIPython Causes Unexpected Behavior
• 445: qt and console Based Startup Errors
• 428: ipcluster doesn’t handle stale pid info well
• 434: 10.0.2 seg fault with rpy2
• 441: Allow running a block of code in a file
• 432: Silent request fails
• 409: Test failure in IPython.lib
• 402: History section of messaging spec is incorrect
• 88: Error when inputting UTF8 CJK characters
• 366: Ctrl-K should kill line and store it, so that Ctrl-y can yank it back
• 425: typo in %gui magic help
• 304: Persistent warnings if old configuration files exist
• 216: crash of ipython when alias is used with %s and echo
• 412: add support to automatic retry of tasks
• 411: add support to continue tasks
• 417: IPython should display things unsorted if it can’t sort them
• 416: wrong encode when printing unicode string
• 376: Failing InputsplitterTest
• 405: TraitError in traitlets.py(332) on any input
• 392: UnicodeEncodeError on start
• 137: sys.getfilesystemencoding return value not checked
• 300: Users should be able to manage kernels and kernel sessions from the notebook UI
• 301: Users should have access to working Kernel, Tabs, Edit, Help menus in the notebook
• 396: cursor move triggers a lot of IO access
• 379: Minor doc nit: –paging argument
• 399: Add task queue limit in engine when load-balancing
• 78: StringTask won’t take unicode code strings
• 391: MongoDB.add_record() does not work in 0.11dev
• 365: newparallel on Windows
- 386: FAIL: test that pushed functions have access to globals
- 387: Interactively defined functions can’t access user namespace
- 118: Snow Leopard ipy_vimserver POLL error
- 394: System escape interpreted in multi-line string
- 26: find_job_cmd is too hasty to fail on Windows
- 368: Installation instructions in dev docs are completely wrong
- 380: qtconsole pager RST - HTML not happening consistently
- 367: Qt console doesn’t support ibus input method
- 375: Missing libraries cause ImportError in tests
- 71: temp file errors in iptest IPython.core
- 350: Decide how to handle displayhook being triggered multiple times
- 360: Remove runlines method
- 125: Exec lines in config should not contribute to line numbering or history
- 20: Robust readline support on OS X’s built-in Python
- 147: On Windows, %page is being too restrictive to split line by rn only
- 326: Update docs and examples for parallel stuff to reflect movement away from Twisted
- 341: Fix Parallel Magics for newparallel
- 338: Usability improvements to Qt console
- 142: unexpected auto-indenting when variables names that start with ‘pass’
- 296: Automatic PDB via %pdb doesn’t work
- 337: exit( and quit( in Qt console produces phantom signature/docstring popup, even though quit() or exit() raises NameError
- 318: %debug broken in master: invokes missing save_history() method
- 307: lines ending with semicolon should not go to cache
- 104: have ipengine run start-up scripts before registering with the controller
- 33: The skip_doctest decorator is failing to work on Shell.MatplotlibShellBase.magic_run
- 336: Missing figure development/figs/iopubfade.png for docs
- 49: %clear should also delete _NN references and Out[NN] ones
- 335: using setuptools installs every script twice
- 306: multiline strings at end of input cause noop
- 327: PyPy compatibility
- 328: %run script.ipy raises “ERROR! Session/line number was not unique in database.”

2.16. Issues closed in the 0.11 development cycle
• 7: Update the changes doc to reflect the kernel config work
• 303: Users should be able to scroll a notebook w/o moving the menu/buttons
• 322: Embedding an interactive IPython shell
• 321: %debug broken in master
• 287: Crash when using %macros in sqlite-history branch
• 55: Can’t edit files whose names begin with numbers
• 284: In variable no longer works in 0.11
• 92: Using multiprocessing module crashes parallel IPython
• 262: Fail to recover history after force-kill.
• 320: Tab completing re.search objects crashes IPython
• 317: IPython.kernel: parallel map issues
• 197: ipython-qtconsole unicode problem in magic ls
• 305: more readline shortcuts in qtconsole
• 314: Multi-line, multi-block cells can’t be executed.
• 308: Test suite should set sqlite history to work in :memory:
• 202: Matplotlib native ‘MacOSX’ backend broken in ‘-pylab’ mode
• 196: IPython can’t deal with unicode file name.
• 25: unicode bug - encoding input
• 290: try/except/else clauses can’t be typed, code input stops too early.
• 43: Implement SSH support in ipcluster
• 6: Update the Sphinx docs for the new ipcluster
• 9: Getting “DeadReferenceError: Calling Stale Broker” after ipcontroller restart
• 132: Ipython prevent south from working
• 27: generics.complete_object broken
• 60: Improve absolute import management for iptest.py
• 31: Issues in magic_whos code
• 52: Document testing process better
• 44: Merge history from multiple sessions
• 182: ipython q4thread in version 10.1 not starting properly
• 143: Ipython.gui.wx.ipython_view.IPShellWidget: ignores user*_ns arguments
• 127: %edit does not work on filenames consisted of pure numbers
• 126: Can’t transfer command line argument to script
• 28: Offer finer control for initialization of input streams
• 58: ipython change char ‘0xe9’ to 4 spaces
• 68: Problems with Control-C stopping ipcluster on Windows/Python2.6
• 24: ipcluster does not start all the engines
• 240: Incorrect method displayed in %psource
• 120: inspect.getsource fails for functions defined on command line
• 212: IPython ignores exceptions in the first evaluation of class attrs
• 108: ipython disables python logger
• 100: Overzealous introspection
• 18: %cpaste freeze sync frontend
• 200: Unicode error when starting ipython in a folder with non-ascii path
• 130: Deadlock when importing a module that creates an IPython client
• 134: multiline block scrolling
• 46: Input to %timeit is not preparsed
• 285: ipcluster local -n 4 fails
• 205: In the Qt console, Tab should insert 4 spaces when not completing
• 145: Bug on MSW systems: idle can not be set as default IPython editor. Fix Suggested.
• 77: ipython oops in cygwin
• 121: If plot windows are closed via window controls, no more plotting is possible.
• 111: Iterator version of TaskClient.map() that returns results as they become available
• 109: WinHPCLauncher is a hard dependency that causes errors in the test suite
• 86: Make IPython work with multiprocessing
• 15: Implement SGE support in ipcluster
• 3: Implement PBS support in ipcluster
• 53: Internal Python error in the inspect module
• 74: Manager() [from multiprocessing module] hangs ipythonx but not ipython
• 51: Out not working with ipythonx
• 201: use session.send throughout zmq code
• 115: multiline specials not defined in 0.11 branch
• 93: when looping, cursor appears at leftmost point in newline
• 133: whitespace after Source introspection
• 50: Ctrl-C with -gthread on Windows, causes uncaught IOError
• 65: Do not use .message attributes in exceptions, deprecated in 2.6
• 76: syntax error when raise is inside except process
• 107: bdist_rpm causes traceback looking for a non-existant file
• 113: initial magic ? (question mark) fails before wildcard
• 128: Pdb instance has no attribute ‘curframe’
• 139: running with -pylab pollutes namespace
• 140: malloc error during tab completion of numpy array member functions starting with ‘c’
• 153: ipy_vimserver traceback on Windows
• 154: using ipython in Slicer3 show how os.environ[‘HOME’] is not defined
• 185: show() blocks in pylab mode with ipython 0.10.1
• 189: Crash on tab completion
• 274: bashism in sshx.sh
• 276: Calling sip.setapi does not work if app has already imported from PyQt4
• 277: matplotlib.image imgshow from 10.1 segfault
• 288: Incorrect docstring in zmq/kernelmanager.py
• 286: Fix IPython.Shell compatibility layer
• 99: blank lines in history
• 129: psearch: TypeError: expected string or buffer
• 190: Add option to format float point output
• 246: Application not conforms XDG Base Directory Specification
• 48: IPython should follow the XDG Base Directory spec for configuration
• 176: Make client-side history persistence readline-independent
• 279: Backtraces when using ipdb do not respect -colour LightBG setting
• 119: Broken type filter in magic_who_ls
• 271: Intermittent problem with print output in Qt console.
• 270: Small typo in IPython developer’s guide
• 166: Add keyboard accelerators to Qt close dialog
• 173: asymmetrical ctrl-A/ctrl-E behavior in multiline
• 45: Autosave history for robustness
• 162: make command history persist in ipythonqt
• 161: make ipythonqt exit without dialog when exit() is called
• 263: [ipython + numpy] Some test errors
• 256: reset docstring ipython 0.10
• 258: allow caching to avoid matplotlib object references
• 248: Can’t open and read files after upgrade from 0.10 to 0.10.0
• 247: ipython + Stackless
• 245: Magic save and macro missing newlines, line ranges don’t match prompt numbers.
• 241: “exit” hangs on terminal version of IPython
• 213: ipython -pylab no longer plots interactively on 0.10.1
• 4: wx frontend don’t display well commands output
• 5: ls command not supported in ipython wx frontend
• 1: Document winhpcjob.py and launcher.py
• 83: Usage of testing.util.DeferredTestCase should be replace with twisted.trial.unittest.TestCase
• 117: Redesign how Component instances are tracked and queried
• 47: IPython.kernel.client cannot be imported inside an engine
• 105: Refactor the task dependencies system
• 210: 0.10.1 doc mistake - New IPython Sphinx directive error
• 209: can’t activate IPython parallel magics
• 206: Buggy linewrap in Mac OSX Terminal
• 194: !sudo <command> displays password in plain text
• 186: %edit issue under OS X 10.5 - IPython 0.10.1
• 11: Create a daily build PPA for ipython
• 144: logo missing from sphinx docs
• 181: cls command does not work on windows
• 169: Kernel can only be bound to localhost
• 36: tab completion does not escape ()
• 177: Report tracebacks of interactively entered input
• 148: dictionary having multiple keys having frozenset fails to print on IPython
• 160: magic_gui throws TypeError when gui magic is used
• 150: History entries ending with parentheses corrupt command line on OS X 10.6.4
• 146: -ipythondir - using an alternative .ipython dir for rc type stuff
• 114: Interactive strings get mangled with “_ip.magic”
• 135: crash on invalid print
• 69: Usage of “mycluster” profile in docs and examples

2.16. Issues closed in the 0.11 development cycle
• 37: Fix colors in output of ResultList on Windows

2.17 0.10 series

2.17.1 Release 0.10.2

IPython 0.10.2 was released April 9, 2011. This is a minor bugfix release that preserves backward compatibility. At this point, all IPython development resources are focused on the 0.11 series that includes a complete architectural restructuring of the project as well as many new capabilities, so this is likely to be the last release of the 0.10.x series. We have tried to fix all major bugs in this series so that it remains a viable platform for those not ready yet to transition to the 0.11 and newer codebase (since that will require some porting effort, as a number of APIs have changed).

Thus, we are not opening a 0.10.3 active development branch yet, but if the user community requires new patches and is willing to maintain/release such a branch, we’ll be happy to host it on the IPython github repositories.

Highlights of this release:

• The main one is the closing of github ticket #185, a major regression we had in 0.10.1 where pylab mode with GTK (or gthread) was not working correctly, hence plots were blocking with GTK. Since this is the default matplotlib backend on Unix systems, this was a major annoyance for many users. Many thanks to Paul Ivanov for helping resolve this issue.

• Fix IOError bug on Windows when used with -gthread.

• Work robustly if $HOME is missing from environment.

• Better POSIX support in ssh scripts (remove bash-specific idioms).

• Improved support for non-ascii characters in log files.

• Work correctly in environments where GTK can be imported but not started (such as a linux text console without X11).

For this release we merged 24 commits, contributed by the following people (please let us know if we omitted your name and we’ll gladly fix this in the notes for the future):

• Fernando Perez
• MinRK
• Paul Ivanov
• Pieter Cristiaan de Groot
• TvrtkoM

2.17.2 Release 0.10.1

IPython 0.10.1 was released October 11, 2010, over a year after version 0.10. This is mostly a bugfix release, since after version 0.10 was released, the development team’s energy has been focused on the 0.11 series.
We have nonetheless tried to backport what fixes we could into 0.10.1, as it remains the stable series that many users have in production systems they rely on.

Since the 0.11 series changes many APIs in backwards-incompatible ways, we are willing to continue maintaining the 0.10.x series. We don’t really have time to actively write new code for 0.10.x, but we are happy to accept patches and pull requests on the IPython github site. If sufficient contributions are made that improve 0.10.1, we will roll them into future releases. For this purpose, we will have a branch called 0.10.2 on github, on which you can base your contributions.

For this release, we applied approximately 60 commits totaling a diff of over 7000 lines:

```
(0.10.1)amirbar[dist]> git diff --oneline rel-0.10.. | wc -l
7296
```

Highlights of this release:

- The only significant new feature is that IPython’s parallel computing machinery now supports natively the Sun Grid Engine and LSF schedulers. This work was a joint contribution from Justin Riley, Satra Ghosh and Matthieu Brucher, who put a lot of work into it. We also improved traceback handling in remote tasks, as well as providing better control for remote task IDs.

- New IPython Sphinx directive contributed by John Hunter. You can use this directive to mark blocks in reStructedText documents as containing IPython syntax (including figures) and they will be executed during the build:

  ```
  In [2]: plt.figure() # ensure a fresh figure
  @savefig psimple.png width=4
  In [3]: plt.plot([1,2,3])
  Out[3]: [<matplotlib.lines.Line2D object at 0x9b74d8c>]
  ```

- Various fixes to the standalone ipython-wx application.

- We now ship internally the excellent argparse library, graciously licensed under BSD terms by Steven Bethard. Now (2010) that argparse has become part of Python 2.7 this will be less of an issue, but Steven’s relicensing allowed us to start updating IPython to using argparse well before Python 2.7. Many thanks!

- Robustness improvements so that IPython doesn’t crash if the readline library is absent (though obviously a lot of functionality that requires readline will not be available).

- Improvements to tab completion in Emacs with Python 2.6.

- Logging now supports timestamps (see %logstart? for full details).

- A long-standing and quite annoying bug where parentheses would be added to print statements, under Python 2.5 and 2.6, was finally fixed.

- Improved handling of libreadline on Apple OSX.

- Fix reload method of IPython demos, which was broken.

- Fixes for the ipipe/ibrowse system on OSX.

- Fixes for Zope profile.

- Fix %timeit reporting when the time is longer than 1000s.
• Avoid lockups with ? or ?? in SunOS, due to a bug in termios.
• The usual assortment of miscellaneous bug fixes and small improvements.

The following people contributed to this release (please let us know if we omitted your name and we’ll gladly fix this in the notes for the future):

• Beni Cherniavsky
• Boyd Waters.
• David Warde-Farley
• Fernando Perez
• Gökhan Sever
• John Hunter
• Justin Riley
• Kiorky
• Laurent Dufrechou
• Mark E. Smith
• Matthieu Brucher
• Satrajit Ghosh
• Sebastian Busch
• Václav Šmilauer

2.17.3 Release 0.10

This release brings months of slow but steady development, and will be the last before a major restructuring and cleanup of IPython’s internals that is already under way. For this reason, we hope that 0.10 will be a stable and robust release so that while users adapt to some of the API changes that will come with the refactoring that will become IPython 0.11, they can safely use 0.10 in all existing projects with minimal changes (if any).

IPython 0.10 is now a medium-sized project, with roughly (as reported by David Wheeler’s sloccount utility) 40750 lines of Python code, and a diff between 0.9.1 and this release that contains almost 28000 lines of code and documentation. Our documentation, in PDF format, is a 495-page long PDF document (also available in HTML format, both generated from the same sources).

Many users and developers contributed code, features, bug reports and ideas to this release. Please do not hesitate in contacting us if we’ve failed to acknowledge your contribution here. In particular, for this release we have contribution from the following people, a mix of new and regular names (in alphabetical order by first name):

• Alexander Clausen: fix #341726.
• Brian Granger: lots of work everywhere (features, bug fixes, etc).
• Daniel Ashbrook: bug report on MemoryError during compilation, now fixed.
• Darren Dale: improvements to documentation build system, feedback, design ideas.
• Fernando Perez: various places.
• Gaël Varoquaux: core code, ipythonx GUI, design discussions, etc. Lots...
• John Hunter: suggestions, bug fixes, feedback.
• Jorgen Stenarson: work on many fronts, tests, fixes, win32 support, etc.
• Laurent Dufréchou: many improvements to ipython-wx standalone app.
• Łukasz Pankowski: prefilter, %edit, demo improvements.
• Matt Foster: TextMate support in %edit.
• Nathaniel Smith: fix #237073.
• Pauli Virtanen: fixes and improvements to extensions, documentation.
• Prabhu Ramachandran: improvements to %timeit.
• Robert Kern: several extensions.
• Sameer D’Costa: help on critical bug #269966.
• Stephan Peijnik: feedback on Debian compliance and many man pages.
• Steven Bethard: we are now shipping his argparse module.
• Tom Fetherston: many improvements to IPython.demo module.
• Ville Vainio: lots of work everywhere (features, bug fixes, etc).
• Vishal Vasta: ssh support in ipcluster.
• Walter Doerwald: work on the IPython.ipipe system.

Below we give an overview of new features, bug fixes and backwards-incompatible changes. For a detailed account of every change made, feel free to view the project log withbzr log.

New features

• New %paste magic automatically extracts current contents of clipboard and pastes it directly, while correctly handling code that is indented or prepended with >>> or ... python prompt markers. A very useful new feature contributed by Robert Kern.
• IPython ‘demos’, created with the IPython.demo module, can now be created from files on disk or strings in memory. Other fixes and improvements to the demo system, by Tom Fetherston.
• Added find_cmd() function to IPython.platutils module, to find commands in a cross-platform manner.
• Many improvements and fixes to Gaël Varoquaux’s ipythonx, a WX-based lightweight IPython instance that can be easily embedded in other WX applications. These improvements have made it possible to now have an embedded IPython in Mayavi and other tools.
• MultiengineClient objects now have a benchmark() method.

• The manual now includes a full set of auto-generated API documents from the code sources, using Sphinx and some of our own support code. We are now using the Numpy Documentation Standard for all docstrings, and we have tried to update as many existing ones as possible to this format.

• The new IPython.Extensions.ipy.pretty extension by Robert Kern provides configurable pretty-printing.

• Many improvements to the ipython-wx standalone WX-based IPython application by Laurent Dufréchou. It can optionally run in a thread, and this can be toggled at runtime (allowing the loading of Matplotlib in a running session without ill effects).

• IPython includes a copy of Steven Bethard’s argparse in the IPython.external package, so we can use it internally and it is also available to any IPython user. By installing it in this manner, we ensure zero conflicts with any system-wide installation you may already have while minimizing external dependencies for new users. In IPython 0.10, We ship argparse version 1.0.

• An improved and much more robust test suite, that runs groups of tests in separate subprocesses using either Nose or Twisted’s trial runner to ensure proper management of Twisted-using code. The test suite degrades gracefully if optional dependencies are not available, so that the ipptest command can be run with only Nose installed and nothing else. We also have more and cleaner test decorators to better select tests depending on runtime conditions, do setup/teardown, etc.

• The new ipcluster now has a fully working ssh mode that should work on Linux, Unix and OS X. Thanks to Vishal Vatsa for implementing this!

• The wonderful TextMate editor can now be used with %edit on OS X. Thanks to Matt Foster for this patch.

• The documentation regarding parallel uses of IPython, including MPI and PBS, has been significantly updated and improved.

• The developer guidelines in the documentation have been updated to explain our workflow using bzr and Launchpad.

• Fully refactored ipcluster command line program for starting IPython clusters. This new version is a complete rewrite and 1) is fully cross platform (we now use Twisted’s process management), 2) has much improved performance, 3) uses subcommands for different types of clusters, 4) uses argparse for parsing command line options, 5) has better support for starting clusters using mpirun, 6) has experimental support for starting engines using PBS. It can also reuse FURL files, by appropriately passing options to its subcommands. However, this new version of ipcluster should be considered a technology preview. We plan on changing the API in significant ways before it is final.

• Full description of the security model added to the docs.

• cd completer: show bookmarks if no other completions are available.

• sh profile: easy way to give ‘title’ to prompt: assign to variable ‘_prompt_title’. It looks like this:

```
[~]|1> _prompt_title = 'sudo!'  
sudo!|~]|2>
```

• %edit: If you do ‘%edit pasted_block’, pasted_block variable gets updated with new data (so repeated editing makes sense)
Bug fixes

- Fix #368719, removed top-level debian/ directory to make the job of Debian packagers easier.
- Fix #291143 by including man pages contributed by Stephan Peijnik from the Debian project.
- Fix #358202, effectively a race condition, by properly synchronizing file creation at cluster startup time.
- `%timeit` now handles correctly functions that take a long time to execute even the first time, by not repeating them.
- Fix #239054, releasing of references after exiting.
- Fix #341726, thanks to Alexander Clausen.
- Fix #269966. This long-standing and very difficult bug (which is actually a problem in Python itself) meant long-running sessions would inevitably grow in memory size, often with catastrophic consequences if users had large objects in their scripts. Now, using `%run` repeatedly should not cause any memory leaks. Special thanks to John Hunter and Sameer D’Costa for their help with this bug.
- Fix #295371, bug in `%history`.
- Improved support for py2exe.
- Fix #270856: IPython hangs with PyGTK
- Fix #270998: A magic with no docstring breaks the ‘%magic magic’
- fix #271684: -c startup commands screw up raw vs. native history
- Numerous bugs on Windows with the new ipcluster have been fixed.
- The ipengine and ipcontroller scripts now handle missing furl files more gracefully by giving better error messages.
- `%rehashx`: Aliases no longer contain dots. python3.0 binary will create alias python30. Fixes: #259716 “commands with dots in them don’t work”
- `%cpaste`: `%cpaste -r` repeats the last pasted block. The block is assigned to pasted_block even if code raises exception.
- Bug #274067 ‘The code in get_home_dir is broken for py2exe’ was fixed.
- Many other small bug fixes not listed here by number (see the bzr log for more info).

Backwards incompatible changes

- ipykit and related files were unmaintained and have been removed.
- The IPython.genutils.doctest_reload() does not actually call reload(doctest) anymore, as this was causing many problems with the test suite. It still resets doctest.master to None.
• While we have not deliberately broken Python 2.4 compatibility, only minor testing was done with Python 2.4, while 2.5 and 2.6 were fully tested. But if you encounter problems with 2.4, please do report them as bugs.

• The `ipcluster` now requires a mode argument; for example to start a cluster on the local machine with 4 engines, you must now type:

```
$ ipcluster local -n 4
```

• The controller now has a `-r` flag that needs to be used if you want to reuse existing furl files. Otherwise they are deleted (the default).

• Remove `ipy_leo.py`. You can use `easy_install ipython-extension` to get it. (done to decouple it from ipython release cycle)

## 2.18 0.9 series

### 2.18.1 Release 0.9.1

This release was quickly made to restore compatibility with Python 2.4, which version 0.9 accidentally broke. No new features were introduced, other than some additional testing support for internal use.

### 2.18.2 Release 0.9

#### New features

• All furl files and security certificates are now put in a read-only directory named `~/.ipython/security`.

• A single function `get_ipython_dir()`, in `IPython.genutils` that determines the user’s IPython directory in a robust manner.

• Laurent’s WX application has been given a top-level script called `ipython-wx`, and it has received numerous fixes. We expect this code to be architecturally better integrated with Gael’s WX ‘ipython widget’ over the next few releases.

• The Editor synchronization work by Vivian De Smedt has been merged in. This code adds a number of new editor hooks to synchronize with editors under Windows.

• A new, still experimental but highly functional, WX shell by Gael Varoquaux. This work was sponsored by Enthought, and while it’s still very new, it is based on a more cleanly organized architecture of the various IPython components. We will continue to develop this over the next few releases as a model for GUI components that use IPython.

• Another GUI frontend, Cocoa based (Cocoa is the OSX native GUI framework), authored by Barry Wark. Currently the WX and the Cocoa ones have slightly different internal organizations, but the whole team is working on finding what the right abstraction points are for a unified codebase.

• As part of the frontend work, Barry Wark also implemented an experimental event notification system that various ipython components can use. In the next release the implications and use patterns of this system regarding the various GUI options will be worked out.
• IPython finally has a full test system, that can test docstrings with IPython-specific functionality. There are still a few pieces missing for it to be widely accessible to all users (so they can run the test suite at any time and report problems), but it now works for the developers. We are working hard on continuing to improve it, as this was probably IPython’s major Achilles heel (the lack of proper test coverage made it effectively impossible to do large-scale refactoring). The full test suite can now be run using the `iptest` command line program.

• The notion of a task has been completely reworked. An `ITask` interface has been created. This interface defines the methods that tasks need to implement. These methods are now responsible for things like submitting tasks and processing results. There are two basic task types: `IPython.kernel.task.StringTask` (this is the old `Task` object, but renamed) and the new `IPython.kernel.task.MapTask`, which is based on a function.

• A new interface, `IPython.kernel.mapper.IMapper` has been defined to standardize the idea of a `map` method. This interface has a single `map` method that has the same syntax as the built-in `map`. We have also defined a mapper factory interface that creates objects that implement `IPython.kernel.mapper.IMapper` for different controllers. Both the multiengine and task controller now have mapping capabilities.

• The parallel function capabilities have been reworks. The major changes are that i) there is now an `@parallel` magic that creates parallel functions, ii) the syntax for multiple variable follows that of `map`, iii) both the multiengine and task controller now have a parallel function implementation.

• All of the parallel computing capabilities from `ipython1-dev` have been merged into IPython proper. This resulted in the following new subpackages: `IPython.kernel`, `IPython.kernel.core`, `traitlets.config`, `IPython.tools` and `IPython.testing`.

• As part of merging in the `ipython1-dev` stuff, the `setup.py` script and friends have been completely refactored. Now we are checking for dependencies using the approach that matplotlib uses.

• The documentation has been completely reorganized to accept the documentation from `ipython1-dev`.

• We have switched to using Foolscap for all of our network protocols in `IPython.kernel`. This gives us secure connections that are both encrypted and authenticated.

• We have a brand new `COPYING.txt` files that describes the IPython license and copyright. The biggest change is that we are putting “The IPython Development Team” as the copyright holder. We give more details about exactly what this means in this file. All developer should read this and use the new banner in all IPython source code files.

• `sh profile: ./foo` runs foo as system command, no need to do `!/foo` anymore

• String lists now support `sort(field, nums = True)` method (to easily sort system command output). Try it with `a = !ls -l ; a.sort(1, nums=True)`.

• ‘%cpaste foo’ now assigns the pasted block as string list, instead of string

• The `ipcluster` script now run by default with no security. This is done because the main usage of the script is for starting things on localhost. Eventually when ipcluster is able to start things on other hosts, we will put security back.
• ‘cd –foo’ searches directory history for string foo, and jumps to that dir. Last part of dir name is checked first. If no matches for that are found, look at the whole path.

Bug fixes

• The Windows installer has been fixed. Now all IPython scripts have .bat versions created. Also, the Start Menu shortcuts have been updated.

• The colors escapes in the multiengine client are now turned off on win32 as they don’t print correctly.

• The IPython.kernel.script.ipengine script was exec’ing mpi_import_statement incorrectly, which was leading the engine to crash when mpi was enabled.

• A few subpackages had missing __init__.py files.

• The documentation is only created if Sphinx is found. Previously, the setup.py script would fail if it was missing.

• Greedy cd completion has been disabled again (it was enabled in 0.8.4) as it caused problems on certain platforms.

Backwards incompatible changes

• The clusterfile options of the ipcluster command has been removed as it was not working and will be replaced soon by something much more robust.

• The IPython.kernel configuration now properly find the user’s IPython directory.

• In ipapi, the make_user_ns() function has been replaced with make_user_namespaces(), to support dict subclasses in namespace creation.

• IPython.kernel.client.Task has been renamed IPython.kernel.client.StringTask to make way for new task types.

• The keyword argument style has been renamed dist in scatter, gather and map.

• Renamed the values that the rename dist keyword argument can have from ‘basic’ to ‘b’.

• IPython has a larger set of dependencies if you want all of its capabilities. See the setup.py script for details.

• The constructors for IPython.kernel.client.MultiEngineClient and IPython.kernel.client.TaskClient no longer take the (ip,port) tuple. Instead they take the filename of a file that contains the FURL for that client. If the FURL file is in your IPYTHONDIR, it will be found automatically and the constructor can be left empty.

• The asynchronous clients in IPython.kernel.asyncclient are now created using the factory functions get_multiengine_client() and get_task_client(). These return a Deferred to the actual client.

• The command line options to ipcontroller and ipengine have changed to reflect the new Foolscap network protocol and the FURL files. Please see the help for these scripts for details.
• The configuration files for the kernel have changed because of the Foolscap stuff. If you were using custom config files before, you should delete them and regenerate new ones.

Changes merged in from IPython1

New features

• Much improved setup.py and setupegg.py scripts. Because Twisted and zope.interface are now easy installable, we can declare them as dependencies in our setupegg.py script.

• IPython is now compatible with Twisted 2.5.0 and 8.x.

• Added a new example of how to use ipython1.kernel.asynclient.

• Initial draft of a process daemon in ipython1.daemon. This has not been merged into IPython and is still in ipython1-dev.

• The TaskController now has methods for getting the queue status.

• The TaskResult objects not have information about how long the task took to run.

• We are attaching additional attributes to exceptions (_ipython_*) that we use to carry additional info around.

• New top-level module asyncclient that has asynchronous versions (that return deferreds) of the client classes. This is designed to users who want to run their own Twisted reactor.

• All the clients in client are now based on Twisted. This is done by running the Twisted reactor in a separate thread and using the blockingCallFromThread() function that is in recent versions of Twisted.

• Functions can now be pushed/pulled to/from engines using MultiEngineClient.push_function() and MultiEngineClient.pull_function().

• Gather/scatter are now implemented in the client to reduce the work load of the controller and improve performance.

• Complete rewrite of the IPython documentation. All of the documentation from the IPython website has been moved into docs/source as restructured text documents. PDF and HTML documentation are being generated using Sphinx.

• New developer oriented documentation: development guidelines and roadmap.

• Traditional ChangeLog has been changed to a more useful changes.txt file that is organized by release and is meant to provide something more relevant for users.

Bug fixes

• Created a proper MANIFEST.in file to create source distributions.

• Fixed a bug in the MultiEngine interface. Previously, multi-engine actions were being collected with a DeferredList with fireononeerrback=1. This meant that methods were returning before all engines had given their results. This was causing extremely odd bugs in certain cases. To
fix this problem, we have 1) set `fireoneerrorback=0` to make sure all results (or exceptions) are in before returning and 2) introduced a `CompositeError` exception that wraps all of the engine exceptions. This is a huge change as it means that users will have to catch `CompositeError` rather than the actual exception.

### Backwards incompatible changes

- All names have been renamed to conform to the lowercase_with_underscore convention. This will require users to change references to all names like `queueStatus` to `queue_status`.

- Previously, methods like `MultiEngineClient.push()` and `MultiEngineClient.push()` used `*args` and `**kwargs`. This was becoming a problem as we weren’t able to introduce new keyword arguments into the API. Now these methods simple take a dict or sequence. This has also allowed us to get rid of the `*All` methods like `pushAll()` and `pullAll()`. These things are now handled with the `targets` keyword argument that defaults to ‘all’.

- The `MultiEngineClient.magicTargets` has been renamed to `MultiEngineClient.targets`.

- All methods in the MultiEngine interface now accept the optional keyword argument `block`.

- Renamed `RemoteController` to `MultiEngineClient` and `TaskController` to `TaskClient`.

- Renamed the top-level module from `api` to `client`.

- Most methods in the multiengine interface now raise a `CompositeError` exception that wraps the user’s exceptions, rather than just raising the raw user’s exception.

- Changed the `setupNS` and `resultNames` in the `Task` class to `push` and `pull`.

### 2.19 0.8 series

#### 2.19.1 Release 0.8.4

This was a quick release to fix an unfortunate bug that slipped into the 0.8.3 release. The `--twisted` option was disabled, as it turned out to be broken across several platforms.

#### 2.19.2 Release 0.8.3

- `pydb` is now disabled by default (due to `%run -d` problems). You can enable it by passing `-pydb` command line argument to IPython. Note that setting it in config file won’t work.

#### 2.19.3 Release 0.8.2

- `%pushd/%popd` behave differently; now “pushd /foo” pushes CURRENT directory and jumps to /foo. The current behaviour is closer to the documented behaviour, and should not trip anyone.
2.19.4 Older releases

Changes in earlier releases of IPython are described in the older file ChangeLog. Please refer to this document for details.
Installation

IPython requires Python 2.7 or 3.3.

See also:

Installing Jupyter The Notebook, nbconvert, and many other former pieces of IPython are now part of Project Jupyter.

3.1 Quickstart

If you have pip, the quickest way to get up and running with IPython is:

$ pip install ipython

To use IPython with notebooks or the Qt console, you should also install jupyter.

To run IPython’s test suite, use the iptest command:

$ iptest

3.2 Overview

This document describes in detail the steps required to install IPython. For a few quick ways to get started with package managers or full Python distributions, see the install page of the IPython website.

Please let us know if you have problems installing IPython or any of its dependencies.

IPython and most dependencies can be installed via pip. In many scenarios, this is the simplest method of installing Python packages. More information about pip can be found on its PyPI page.

More general information about installing Python packages can be found in Python’s documentation.
3.3 Installing IPython itself

Given a properly built Python, the basic interactive IPython shell will work with no external dependencies. However, some Python distributions (particularly on Windows and OS X), don’t come with a working readline module. The IPython shell will work without readline, but will lack many features that users depend on, such as tab completion and command line editing. If you install IPython with pip, then the appropriate readline for your platform will be installed. See below for details of how to make sure you have a working readline.

3.3.1 Installation using pip

If you have pip, the easiest way of getting IPython is:

```
$ pip install ipython
```

That’s it.

3.3.2 Installation from source

If you don’t want to use pip, or don’t have it installed, grab the latest stable tarball of IPython from PyPI. Then do the following:

```
$ tar -xzf ipython.tar.gz
$ cd ipython
$ python setup.py install
```

If you are installing to a location (like /usr/local) that requires higher permissions, you may need to run the last command with sudo.

3.3.3 Installing the development version

It is also possible to install the development version of IPython from our Git source code repository. To do this you will need to have Git installed on your system. Then do:

```
$ git clone --recursive https://github.com/ipython/ipython.git
$ cd ipython
$ python setup.py install
```

Some users want to be able to follow the development branch as it changes. If you have pip, you can replace the last step by:

```
$ pip install -e .
```

This creates links in the right places and installs the command line script to the appropriate places.

Then, if you want to update your IPython at any time, do:

```
$ git pull
```
3.4 Dependencies

IPython relies on a number of other Python packages. Installing using a package manager like pip or conda will ensure the necessary packages are installed. If you install manually, it’s up to you to make sure dependencies are installed. They’re not listed here, because they may change from release to release, so a static list will inevitably get out of date.

It also has one key non-Python dependency which you may need to install separately.

3.4.1 readline

IPython’s terminal interface relies on readline to provide features like tab completion and history navigation. If you only want to use IPython as a kernel for Jupyter notebooks and other frontends, you don’t need readline.

On Windows, to get full console functionality, PyReadline is required. PyReadline is a separate, Windows only implementation of readline that uses native Windows calls through ctypes. The easiest way of installing PyReadline is by using the binary installer available here.

On OS X, if you are using the built-in Python shipped by Apple, you will be missing a proper readline implementation as Apple ships instead a library called libedit that provides only some of readline’s functionality. While you may find libedit sufficient, we have occasional reports of bugs with it and several developers who use OS X as their main environment consider libedit unacceptable for productive, regular use with IPython.

Therefore, IPython on OS X depends on the gnureadline module. We will not consider completion/history problems to be bugs for IPython if you are using libedit.

To get a working readline module on OS X, do (with pip installed):

```
$ pip install gnureadline
```

**Note:** Other Python distributions on OS X (such as Anaconda, fink, MacPorts) already have proper readline so you likely don’t have to do this step.

When IPython is installed with pip, the correct readline should be installed if you specify the terminal optional dependencies:

```
$ pip install "ipython[terminal]"
```

On Linux, readline is normally installed by default. If not, install it from your system package manager. If you are compiling your own Python, make sure you install the readline development headers first.

3.5 Installing the IPython kernel

See also:

**Installing Jupyter** The IPython kernel is the Python execution backend for Jupyter.
The Jupyter Notebook and other frontends automatically ensure that the IPython kernel is available. However, if you want to use a kernel with a different version of Python, or in a virtualenv or conda environment, you’ll need to install that manually.

### 3.5.1 Kernels for Python 2 and 3

If you’re running Jupyter on Python 3, you can set up a Python 2 kernel like this:

```bash
python2 -m pip install ipykernel
python2 -m ipykernel install --user
```

Or using conda, create a Python 2 environment:

```bash
conda create -n ipykernel_py2 python=2 ipykernel
source activate ipykernel_py2 # On Windows, remove the word 'source'
python -m ipykernel install --user
```

If you’re running Jupyter on Python 2 and want to set up a Python 3 kernel, follow the same steps, replacing 2 with 3.

The last command installs a kernel spec file for the current python installation. Kernel spec files are JSON files, which can be viewed and changed with a normal text editor.

### 3.5.2 Kernels for different environments

If you want to have multiple IPython kernels for different virtualenvs or conda environments, you will need to specify unique names for the kernelspecs.

For example, using conda environments:

```bash
source activate myenv
python -m ipykernel install --user --name myenv --display-name "Python (myenv)"
source activate other-env
python -m ipykernel install --user --name other-env --display-name "Python (other-env)"
```

The --name value is used by Jupyter internally. These commands will overwrite any existing kernel with the same name. --display-name is what you see in the notebook menus.
4.1 Introducing IPython

You don’t need to know anything beyond Python to start using IPython – just type commands as you would at the standard Python prompt. But IPython can do much more than the standard prompt. Some key features are described here. For more information, check the tips page, or look at examples in the IPython cookbook.

If you’ve never used Python before, you might want to look at the official tutorial or an alternative, Dive into Python.

4.1.1 The four most helpful commands

The four most helpful commands, as well as their brief description, is shown to you in a banner, every time you start IPython:

<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Introduction and overview of IPython’s features.</td>
</tr>
<tr>
<td>%quickref</td>
<td>Quick reference.</td>
</tr>
<tr>
<td>help</td>
<td>Python’s own help system.</td>
</tr>
<tr>
<td>object?</td>
<td>Details about ‘object’, use ‘object??’ for extra details.</td>
</tr>
</tbody>
</table>

4.1.2 Tab completion

Tab completion, especially for attributes, is a convenient way to explore the structure of any object you’re dealing with. Simply type `object_name.<TAB>` to view the object’s attributes (see the readline section for more). Besides Python objects and keywords, tab completion also works on file and directory names.

4.1.3 Exploring your objects

Typing `object_name?` will print all sorts of details about any object, including docstrings, function definition lines (for call arguments) and constructor details for classes. To get specific information on an object, you can use the magic commands `%pdoc`, `%pdef`, `%psource` and `%pfile`
4.1.4 Magic functions

IPython has a set of predefined ‘magic functions’ that you can call with a command line style syntax. There are two kinds of magics, line-oriented and cell-oriented. **Line magics** are prefixed with the % character and work much like OS command-line calls: they get as an argument the rest of the line, where arguments are passed without parentheses or quotes. **Cell magics** are prefixed with a double %, and they are functions that get as an argument not only the rest of the line, but also the lines below it in a separate argument.

The following examples show how to call the built-in `%%timeit` magic, both in line and cell mode:

```
In [1]: %timeit range(1000)
100000 loops, best of 3: 7.76 us per loop

In [2]: %%timeit x = range(10000)
    ...: max(x)
    ...:
1000 loops, best of 3: 223 us per loop
```

The built-in magics include:

- Functions that work with code: `%run`, `%edit`, `%save`, `%macro`, `%recall`, etc.
- Functions which affect the shell: `%colors`, `%xmode`, `%autoindent`, `%automagic`, etc.
- Other functions such as `%reset`, `%timeit`, `%writefile`, `%load`, or `%paste`.

You can always call them using the % prefix, and if you’re calling a line magic on a line by itself, you can omit even that:

```
run thescript.py
```

You can toggle this behavior by running the `%automagic` magic. Cell magics must always have the %% prefix.

A more detailed explanation of the magic system can be obtained by calling `%magic`, and for more details on any magic function, call `%somemagic?` to read its docstring. To see all the available magic functions, call `%lsmagic`.

**See also:**

- Built-in magic commands
- Cell magics example notebook

### Running and Editing

The `%run` magic command allows you to run any python script and load all of its data directly into the interactive namespace. Since the file is re-read from disk each time, changes you make to it are reflected immediately (unlike imported modules, which have to be specifically reloaded). IPython also includes `dreload`, a recursive reload function.

`%run` has special flags for timing the execution of your scripts (-t), or for running them under the control of either Python’s pdb debugger (-d) or profiler (-p).
The `%edit` command gives a reasonable approximation of multiline editing, by invoking your favorite editor on the spot. IPython will execute the code you type in there as if it were typed interactively.

**Debugging**

After an exception occurs, you can call `%debug` to jump into the Python debugger (pdb) and examine the problem. Alternatively, if you call `%pdb`, IPython will automatically start the debugger on any uncaught exception. You can print variables, see code, execute statements and even walk up and down the call stack to track down the true source of the problem. This can be an efficient way to develop and debug code, in many cases eliminating the need for print statements or external debugging tools.

You can also step through a program from the beginning by calling `%run -d theprogram.py`.

**4.1.5 History**

IPython stores both the commands you enter, and the results it produces. You can easily go through previous commands with the up- and down-arrow keys, or access your history in more sophisticated ways.

Input and output history are kept in variables called `In` and `Out`, keyed by the prompt numbers, e.g. `In[4]`. The last three objects in output history are also kept in variables named `_`, `__` and `___`.

You can use the `%history` magic function to examine past input and output. Input history from previous sessions is saved in a database, and IPython can be configured to save output history.

Several other magic functions can use your input history, including `%edit`, `%rerun`, `%recall`, `%macro`, `%save` and `%pastebin`. You can use a standard format to refer to lines:

```
%pastebin 3 18-20 ~1/1-5
```

This will take line 3 and lines 18 to 20 from the current session, and lines 1-5 from the previous session.

**4.1.6 System shell commands**

To run any command at the system shell, simply prefix it with `!`, e.g.:

```
!ping www.bbc.co.uk
```

You can capture the output into a Python list, e.g.: `files = !ls`. To pass the values of Python variables or expressions to system commands, prefix them with `$`: `!grep -rF $pattern ipython/*`. See our shell section for more details.

**Define your own system aliases**

It’s convenient to have aliases to the system commands you use most often. This allows you to work seamlessly from inside IPython with the same commands you are used to in your system shell. IPython comes with some pre-defined aliases and a complete system for changing directories, both via a stack (see `%pushd`, `%popd` and `%dhist`) and via direct `%cd`. The latter keeps a history of visited directories and allows you to go to any previously visited one.
4.1.7 Configuration

Much of IPython can be tweaked through configuration. To get started, use the command `ipython profile create` to produce the default config files. These will be placed in `~/.ipython/profile_default`, and contain comments explaining what the various options do.

Profiles allow you to use IPython for different tasks, keeping separate config files and history for each one. More details in the profiles section.

Startup Files

If you want some code to be run at the beginning of every IPython session, the easiest way is to add Python (.py) or IPython (.ipy) scripts to your `profile_default/startup/` directory. Files here will be executed as soon as the IPython shell is constructed, before any other code or scripts you have specified. The files will be run in order of their names, so you can control the ordering with prefixes, like `10-myimports.py`.

4.2 Built-in magic commands

4.2.1 Line magics

%alias

Define an alias for a system command.

`%alias alias_name cmd` defines `alias_name` as an alias for `cmd`

Then, typing `alias_name params` will execute the system command `cmd params` (from your underlying operating system).

Aliases have lower precedence than magic functions and Python normal variables, so if `foo` is both a Python variable and an alias, the alias can not be executed until `del foo` removes the Python variable.

You can use the %l specifier in an alias definition to represent the whole line when the alias is called. For example:

```
In [2]: alias bracket echo "Input in brackets: <%l>"
In [3]: bracket hello world
Input in brackets: <hello world>
```

You can also define aliases with parameters using %s specifiers (one per parameter):

```
In [1]: alias parts echo first %s second %s
In [2]: %parts A B
first A second B
In [3]: %parts A
Incorrect number of arguments: 2 expected.
parts is an alias to: 'echo first %s second %s'
```

Note that %l and %s are mutually exclusive. You can only use one or the other in your aliases.
Aliases expand Python variables just like system calls using ! or !! do: all expressions prefixed with ‘$’ get expanded. For details of the semantic rules, see PEP-215: http://www.python.org/peps/pep-0215.html. This is the library used by IPython for variable expansion. If you want to access a true shell variable, an extra $ is necessary to prevent its expansion by IPython:

```
In [6]: alias show echo
In [7]: PATH='A Python string'
In [8]: show $PATH
A Python string
In [9]: show $$PATH
/usr/local/lf9560/bin:/usr/local/intel/compiler70/ia32/bin:...
```

You can use the alias facility to access all of $PATH. See the %rehashx function, which automatically creates aliases for the contents of your $PATH.

If called with no parameters, %alias prints the current alias table.

```
%alias_magic
```

%alias_magic [-l] [-c] name target

Create an alias for an existing line or cell magic.

Examples

```
In [1]: %alias_magic t timeit
Created `%t` as an alias for `%timeit`.
Created `%%t` as an alias for `%%timeit`.

In [2]: %t -n1 pass
1 loops, best of 3: 954 ns per loop

In [3]: %t -n1
   ...: pass
   ...:
1 loops, best of 3: 954 ns per loop

In [4]: %alias_magic --cell whereami pwd
UsageError: Cell magic function `%%pwd` not found.
In [5]: %alias_magic --line whereami pwd
Created `%whereami` as an alias for `%%pwd`.

In [6]: %whereami
Out[6]: u'/home/testuser'
```

positional arguments: name Name of the magic to be created. target Name of the existing line or cell magic.

optional arguments:

- `-l`, `--line` Create a line magic alias.
- `-c`, `--cell` Create a cell magic alias.
%autocall

Make functions callable without having to type parentheses.

Usage:

%autocall [mode]

The mode can be one of: 0->Off, 1->Smart, 2->Full. If not given, the value is toggled on and off (remembering the previous state).

In more detail, these values mean:

0 -> fully disabled
1 -> active, but do not apply if there are no arguments on the line.

In this mode, you get:

In [1]: callable
Out[1]: <built-in function callable>

In [2]: callable 'hello'
------> callable('hello')
Out[2]: False

2 -> Active always. Even if no arguments are present, the callable object is called:

In [2]: float
------> float()
Out[2]: 0.0

Note that even with autocall off, you can still use '/' at the start of a line to treat the first argument on the command line as a function and add parentheses to it:

In [8]: /str 43
------> str(43)
Out[8]: '43'

# all-random (note for auto-testing)

%automagic

Make magic functions callable without having to type the initial %.

Without arguments it toggles on/off (when off, you must call it as %automagic, of course). With arguments it sets the value, and you can use any of (case insensitive):

•on, 1, True: to activate
•off, 0, False: to deactivate.

Note that magic functions have lowest priority, so if there’s a variable whose name collides with that of a magic fn, automagic won’t work for that function (you get the variable instead). However, if you delete the variable (del var), the previously shadowed magic function becomes visible to automagic again.

%bookmark

Manage IPython’s bookmark system.
%bookmark <name> - set bookmark to current dir %bookmark <name> <dir> - set bookmark to <dir> %bookmark -l - list all bookmarks %bookmark -d <name> - remove bookmark %bookmark -r - remove all bookmarks

You can later on access a bookmarked folder with:

```
%cd -b <name>
```

or simply ‘%cd <name>’ if there is no directory called <name> AND there is such a bookmark defined.

Your bookmarks persist through IPython sessions, but they are associated with each profile.

**%cd**

Change the current working directory.

This command automatically maintains an internal list of directories you visit during your IPython session, in the variable _dh. The command %dhist shows this history nicely formatted. You can also do ‘cd -<tab>’ to see directory history conveniently.

**Usage:**

- `cd 'dir'`: changes to directory ‘dir’.
- `cd -`: changes to the last visited directory.
- `cd -<n>`: changes to the n-th directory in the directory history.
- `cd –foo`: change to directory that matches ‘foo’ in history
- `cd -b <bookmark_name>`: jump to a bookmark set by `%bookmark`

*(note: cd <bookmark_name> is enough if there is no directory <bookmark_name>, but a bookmark with the name exists.)*

**Options:**

- `-q`: quiet. Do not print the working directory after the cd command is executed. By default IPython’s cd command does print this directory, since the default prompts do not display path information.

Note that `!cd` doesn’t work for this purpose because the shell where `!command` runs is immediately discarded after executing ‘command’.

**Examples**

```
In [10]: cd parent/child
/home/tsuser/parent/child
```

**%colors**

Switch color scheme for prompts, info system and exception handlers.

Currently implemented schemes: NoColor, Linux, LightBG.

Color scheme names are not case-sensitive.

**Examples**

To get a plain black and white terminal:
%colors nocolor

%config
  configure IPython

  %config Class.[trait=value]

  This magic exposes most of the IPython config system. Any Configurable class should be able to be
  configured with the simple line:

  %config Class.trait=value

  Where value will be resolved in the user’s namespace, if it is an expression or variable name.

Examples

To see what classes are available for config, pass no arguments:

In [1]: %config
Available objects for config:
   TerminalInteractiveShell
   HistoryManager
   PrefilterManager
   AliasManager
   IPCompleter
   PromptManager
   DisplayFormatter

To view what is configurable on a given class, just pass the class name:

In [2]: %config IPCompleter
IPCompleter options

IPCompleter.omit__names=<Enum>
  Current: 2
  Choices: (0, 1, 2)
  Instruct the completer to omit private method names
  Specifically, when completing on `object.<tab>`.
  When 2 [default]: all names that start with '_' will be excluded.
  When 1: all 'magic' names (`__foo__`) will be excluded.
  When 0: nothing will be excluded.
IPCompleter.merge_completions=<CBool>
  Current: True
  Whether to merge completion results into a single list
  If False, only the completion results from the first non-empty
  completer will be returned.
IPCompleter.limit_to__all__=<CBool>
  Current: False
  Instruct the completer to use __all__ for the completion
  Specifically, when completing on `object.<tab>`.
  When True: only those names in obj.__all__ will be included.
  When False [default]: the __all__ attribute is ignored
IPCompleter.greedy=<CBool>
  Current: False
  Activate greedy completion
This will enable completion on elements of lists, results of function calls, etc., but can be unsafe because the code is actually evaluated on TAB.

but the real use is in setting values:

```
In [3]: %config IPCompleter.greedy = True
```

and these values are read from the user_ns if they are variables:

```
In [4]: feeling_greedy=False
In [5]: %config IPCompleter.greedy = feeling_greedy
```

 `%debug`

`%debug [--breakpoint FILE:LINE] [statement [statement ...]]`

Activate the interactive debugger.

This magic command support two ways of activating debugger. One is to activate debugger before executing code. This way, you can set a break point, to step through the code from the point. You can use this mode by giving statements to execute and optionally a breakpoint.

The other one is to activate debugger in post-mortem mode. You can activate this mode simply running `%debug` without any argument. If an exception has just occurred, this lets you inspect its stack frames interactively. Note that this will always work only on the last traceback that occurred, so you must call this quickly after an exception that you wish to inspect has fired, because if another one occurs, it clobbers the previous one.

If you want IPython to automatically do this on every exception, see the `%pdb` magic for more details.

**positional arguments:**

  - `statement` Code to run in debugger. You can omit this in cell magic mode.

**optional arguments:**

  - `--breakpoint <FILE:LINE>, -b <FILE:LINE>` Set break point at LINE in FILE.

 `%dhist`

Print your history of visited directories.

```
%dhist -> print full history
%dhist n -> print last n entries only
%dhist n1 n2 -> print entries between n1 and n2 (n2 not included)
```

This history is automatically maintained by the `%cd` command, and always available as the global list variable `_dh_. You can use `%cd -<n>` to go to directory number `<n>`.

Note that most of time, you should view directory history by entering cd -<TAB>.

 `%dirs`

Return the current directory stack.
%doctest_mode

Toggle doctest mode on and off.

This mode is intended to make IPython behave as much as possible like a plain Python shell, from the perspective of how its prompts, exceptions and output look. This makes it easy to copy and paste parts of a session into doctests. It does so by:

- Changing the prompts to the classic >>> ones.
- Changing the exception reporting mode to ‘Plain’.
- Disabling pretty-printing of output.

Note that IPython also supports the pasting of code snippets that have leading ‘>>>’ and ‘...’ prompts in them. This means that you can paste doctests from files or docstrings (even if they have leading whitespace), and the code will execute correctly. You can then use ‘%history -t’ to see the translated history; this will give you the input after removal of all the leading prompts and whitespace, which can be pasted back into an editor.

With these features, you can switch into this mode easily whenever you need to do testing and changes to doctests, without having to leave your existing IPython session.

%edit

Bring up an editor and execute the resulting code.

Usage:  %edit [options] [args]

%edit runs IPython’s editor hook. The default version of this hook is set to call the editor specified by your $EDITOR environment variable. If this isn’t found, it will default to vi under Linux/Unix and to notepad under Windows. See the end of this docstring for how to change the editor hook.

You can also set the value of this editor via the TerminalInteractiveShell.editor option in your configuration file. This is useful if you wish to use a different editor from your typical default with IPython (and for Windows users who typically don’t set environment variables).

This command allows you to conveniently edit multi-line code right in your IPython session.

If called without arguments, %edit opens up an empty editor with a temporary file and will execute the contents of this file when you close it (don’t forget to save it!).

Options:

- -n <number>: open the editor at a specified line number. By default, the IPython editor hook uses the unix syntax ‘editor +N filename’, but you can configure this by providing your own modified hook if your favorite editor supports line-number specifications with a different syntax.

- -p: this will call the editor with the same data as the previous time it was used, regardless of how long ago (in your current session) it was.

- -r: use ‘raw’ input. This option only applies to input taken from the user’s history. By default, the ‘processed’ history is used, so that magics are loaded in their transformed version to valid Python. If this option is given, the raw input as typed as the command line is used instead. When you exit the editor, it will be executed by IPython’s own processor.

- -x: do not execute the edited code immediately upon exit. This is mainly useful if you are editing programs which need to be called with command line arguments, which you can then do using %run.
Arguments:

If arguments are given, the following possibilities exist:

- If the argument is a filename, IPython will load that into the editor. It will execute its contents with execfile() when you exit, loading any code in the file into your interactive namespace.

- The arguments are ranges of input history, e.g. “7 ~1/4-6”. The syntax is the same as in the %history magic.

- If the argument is a string variable, its contents are loaded into the editor. You can thus edit any string which contains python code (including the result of previous edits).

- If the argument is the name of an object (other than a string), IPython will try to locate the file where it was defined and open the editor at the point where it is defined. You can use %edit function to load an editor exactly at the point where ‘function’ is defined, edit it and have the file be executed automatically.

- If the object is a macro (see %macro for details), this opens up your specified editor with a temporary file containing the macro’s data. Upon exit, the macro is reloaded with the contents of the file.

Note: opening at an exact line is only supported under Unix, and some editors (like kedit and gedit up to Gnome 2.8) do not understand the ‘+NUMBER’ parameter necessary for this feature. Good editors like (X)Emacs, vi, jed, pico and joe all do.

After executing your code, %edit will return as output the code you typed in the editor (except when it was an existing file). This way you can reload the code in further invocations of %edit as a variable, via _<NUMBER> or Out[<NUMBER>], where <NUMBER> is the prompt number of the output.

Note that %edit is also available through the alias %ed.

This is an example of creating a simple function inside the editor and then modifying it. First, start up the editor:

```
In [1]: edit
Editing... done. Executing edited code...
Out[1]: 'def foo():
 print "foo() was defined in an editing session"
'
```

We can then call the function foo():

```
In [2]: foo()
foo() was defined in an editing session
```

Now we edit foo. IPython automatically loads the editor with the (temporary) file where foo() was previously defined:

```
In [3]: edit foo
Editing... done. Executing edited code...
```

And if we call foo() again we get the modified version:

```
In [4]: foo()
foo() has now been changed!
```
Here is an example of how to edit a code snippet successive times. First we call the editor:

In [5]: edit
   Editing... done. Executing edited code...
   hello
Out[5]: "print 'hello'\n"

Now we call it again with the previous output (stored in _):

In [6]: edit _
   Editing... done. Executing edited code...
   hello world
Out[6]: "print 'hello world'\n"

Now we call it with the output #8 (stored in _8, also as Out[8]):

In [7]: edit _8
   Editing... done. Executing edited code...
   hello again
Out[7]: "print 'hello again'\n"

Changing the default editor hook:

If you wish to write your own editor hook, you can put it in a configuration file which you load at startup time. The default hook is defined in the IPython.core.hooks module, and you can use that as a starting example for further modifications. That file also has general instructions on how to set a new hook for use once you’ve defined it.

%env

Get, set, or list environment variables.

Usage:

   %env: lists all environment variables/values
   %env var: get value for var
   %env var=val: set value for var
   %env var=$val: set value for var, using python expansion if possible

%gui

Enable or disable IPython GUI event loop integration.

%gui [GUINAME]

This magic replaces IPython’s threaded shells that were activated using the (pylab/wthread/etc.) command line flags. GUI toolkits can now be enabled at runtime and keyboard interrupts should work without any problems. The following toolkits are supported: wxPython, PyQt4, PyGTK, Tk and Cocoa (OSX):

%gui wx       # enable wxPython event loop integration
%gui qt4|qt   # enable PyQt4 event loop integration
%gui qt5      # enable PyQt5 event loop integration
%gui gtk      # enable PyGTK event loop integration
%gui gtk3     # enable Gtk3 event loop integration
%gui tk       # enable Tk event loop integration
%gui osx      # enable Cocoa event loop integration
%gui

WARNING: after any of these has been called you can simply create an application object, but DO NOT start the event loop yourself, as we have already handled that.

%history

%history [-n] [-o] [-p] [-t] [-f FILENAME] [-g [PATTERN [PATTERN ...]]]
[-l [LIMIT]] [-u]
[range [range ...]]

Print input history (_i<n> variables), with most recent last.

By default, input history is printed without line numbers so it can be directly pasted into an editor. Use -n to show them.

By default, all input history from the current session is displayed. Ranges of history can be indicated using the syntax:

4 Line 4, current session
4–6 Lines 4-6, current session
243/1–5 Lines 1-5, session 243
~2/7 Line 7, session 2 before current
~8/1–~6/5 From the first line of 8 sessions ago, to the fifth line of 6 sessions ago.

Multiple ranges can be entered, separated by spaces

The same syntax is used by %macro, %save, %edit, %rerun

Examples

In [6]: %history -n 4–6
4:a = 12
5:print a**2
6:%history -n 4–6

positional arguments: range
optional arguments:
- n
  print line numbers for each input. This feature is only available if numbered prompts are in use.
- o
  also print outputs for each input.
- p
  print classic ‘>>>’ python prompts before each input. This is useful for making documentation, and in conjunction with -o, for producing doctest-ready output.
- t
  print the ‘translated’ history, as IPython understands it. IPython filters your input and converts it all into valid Python
source before executing it (things like magics or aliases are turned into function calls, for example). With this option, you’ll see the native history instead of the user-entered version: ‘%cd /’ will be seen as ‘get_ipython().magic(‘%cd /’))’ instead of ‘%cd /’.

-\f FILENAME FILENAME: instead of printing the output to the screen, redirect it to the given file. The file is always overwritten, though when it can, IPython asks for confirmation first. In particular, running the command ‘history -f FILENAME’ from the IPython Notebook interface will replace FILENAME even if it already exists without confirmation.

-\g {PATTERN {PATTERN ...]}\> treat the arg as a glob pattern to search for in (full) history. This includes the saved history (almost all commands ever written). The pattern may contain ‘?’ to match one unknown character and ‘*’ to match any number of unknown characters. Use ‘%hist -g’ to show full saved history (may be very long).

-\l {LIMIT}\ get the last n lines from all sessions. Specify n as a single arg, or the default is the last 10 lines.

-\u\ when searching history using -g, show only unique history.

%install_default_config
%install_default_config has been deprecate and will be removed in IPython 5.0.

%install_ext
Download and install an extension from a URL, e.g.:

%install_ext https://bitbucket.org/birkenfeld/ipython-physics/raw/d1310a2a815d/physics.py

The URL should point to an importable Python module - either a .py file or a .zip file.
Parameters:

-\n filename [Specify a name for the file, rather than taking it from] the URL.

%install_profiles
%install_profiles has been deprecated.

%kill_bgscripts
Kill all BG processes started by %%%script and its family.

%load
Load code into the current frontend.

Usage: %load [options] source

where source can be a filename, URL, input history range, macro, or element in the user namespace

Options:
-r <lines>: Specify lines or ranges of lines to load from the source. Ranges could be specified as x-y (x..y) or in python-style x:y (x..(y-1)). Both limits x and y can be left blank (meaning the beginning and end of the file, respectively).

-s <symbols>: Specify function or classes to load from python source.

-y : Don’t ask confirmation for loading source above 200 000 characters.

-n : Include the user’s namespace when searching for source code.

This magic command can either take a local filename, a URL, an history range (see %history) or a macro as argument, it will prompt for confirmation before loading source with more than 200 000 characters, unless -y flag is passed or if the frontend does not support raw_input:

```%load myscript.py
%load 7-27
%load myMacro
%load http://www.example.com/myscript.py
%load -r 5-10 myscript.py
%load -r 10-20,30,40: foo.py
%load -s MyClass,wonder_function myscript.py
%load -n MyClass
%load -n my_module.wonder_function
```

%load_ext
Load an IPython extension by its module name.

%loadpy
Alias of %load

%loadpy has gained some flexibility and dropped the requirement of a .py extension. So it has been renamed simply into %load. You can look at %load’s docstring for more info.

%logoff
Temporarily stop logging.

You must have previously started logging.

%logon
Restart logging.

This function is for restarting logging which you’ve temporarily stopped with %logoff. For starting logging for the first time, you must use the %logstart function, which allows you to specify an optional log filename.

%logstart
Start logging anywhere in a session.

%logstart [-ol-rl-t] [log_name [log_mode]]

If no name is given, it defaults to a file named ‘ipython_log.py’ in your current directory, in ‘rotate’ mode (see below).

‘%logstart name’ saves to file ‘name’ in ‘backup’ mode. It saves your history up to that point and then continues logging.

4.2. Built-in magic commands 293
%logstart takes a second optional parameter: logging mode. This can be one of (note that the modes are given unquoted):

**append** Keep logging at the end of any existing file.

**backup** Rename any existing file to name~ and start name.

**global** Append to a single logfile in your home directory.

**over** Overwrite any existing log.

**rotate** Create rotating logs: name.1~, name.2~, etc.

Options:

- **-o** log also IPython’s output. In this mode, all commands which generate an Out[N]N prompt are recorded to the logfile, right after their corresponding input line. The output lines are always prepended with a ‘#Out’ marker, so that the log remains valid Python code.

Since this marker is always the same, filtering only the output from a log is very easy, using for example a simple awk call:

```
awk -F'#\[Out\]#' '{if($2) {print $2}}' ipython_log.py
```

- **-r** log ‘raw’ input. Normally, IPython’s logs contain the processed input, so that user lines are logged in their final form, converted into valid Python. For example, %Exit is logged as _ip.magic(“Exit”). If the -r flag is given, all input is logged exactly as typed, with no transformations applied.

- **-t** put timestamps before each input line logged (these are put in comments).

%logstate

Print the status of the logging system.

%logstop

Fully stop logging and close log file.

In order to start logging again, a new %logstart call needs to be made, possibly (though not necessarily) with a new filename, mode and other options.

%lsmagic

List currently available magic functions.

%macro

Define a macro for future re-execution. It accepts ranges of history, filenames or string objects.

**Usage:** %macro [options] name n1-n2 n3-n4 ... n5 .. n6 ...

Options:
-r: use ‘raw’ input. By default, the ‘processed’ history is used, so that magics are loaded in their transformed version to valid Python. If this option is given, the raw input as typed at the command line is used instead.

-q: quiet macro definition. By default, a tag line is printed to indicate the macro has been created, and then the contents of the macro are printed. If this option is given, then no printout is produced once the macro is created.

This will define a global variable called name which is a string made of joining the slices and lines you specify (n1,n2,... numbers above) from your input history into a single string. This variable acts like an automatic function which re-executes those lines as if you had typed them. You just type ‘name’ at the prompt and the code executes.

The syntax for indicating input ranges is described in %history.

Note: as a ‘hidden’ feature, you can also use traditional python slice notation, where N:M means numbers N through M-1.

For example, if your history contains (print using %hist -n):

```
44: x=1
45: y=3
46: z=x+y
47: print x
48: a=5
49: print 'x',x,'y',y
```

you can create a macro with lines 44 through 47 (included) and line 49 called my_macro with:

```
In [55]: %macro my_macro 44-47 49
```

Now, typing my_macro (without quotes) will re-execute all this code in one pass.

You don’t need to give the line-numbers in order, and any given line number can appear multiple times. You can assemble macros with any lines from your input history in any order.

The macro is a simple object which holds its value in an attribute, but IPython’s display system checks for macros and executes them as code instead of printing them when you type their name.

You can view a macro’s contents by explicitly printing it with:

```
print macro_name
```

%magic
Print information about the magic function system.
Supported formats: -latex, -brief, -rest

%matplotlib

```
%matplotlib [-l] [gui]
```

Set up matplotlib to work interactively.
This function lets you activate matplotlib interactive support at any point during an IPython session. It does not import anything into the interactive namespace.

If you are using the inline matplotlib backend in the IPython Notebook you can set which figure formats are enabled using the following:

```python
In [1]: from IPython.display import set_matplotlib_formats
In [2]: set_matplotlib_formats('pdf', 'svg')
```

The default for inline figures sets `bbox_inches` to 'tight'. This can cause discrepancies between the displayed image and the identical image created using `savefig`. This behavior can be disabled using the `%config` magic:

```python
In [3]: %config InlineBackend.print_figure_kwargs = ('bbox_inches':None)
```

In addition, see the docstring of `IPython.display.set_matplotlib_formats` and `IPython.display.set_matplotlib_close` for more information on changing additional behaviors of the inline backend.

Examples

To enable the inline backend for usage with the IPython Notebook:

```
In [1]: %matplotlib inline
```

In this case, where the matplotlib default is TkAgg:

```
In [2]: %matplotlib
Using matplotlib backend: TkAgg
```

But you can explicitly request a different GUI backend:

```
In [3]: %matplotlib qt
```

You can list the available backends using the `-l`--list option:

```
In [4]: %matplotlib --list
Available matplotlib backends: ['osx', 'qt4', 'qt5', 'gtk3', 'notebook', 'wx', 'qt', 'gtk', 'tk', 'inline']
```

**positional arguments:**

- `gui` Name of the matplotlib backend to use (`'gtk'`, `'gtk3'`, `'inline'`, `'nbagg'`, `'notebook'`, `'osx'`, `'qt'`, `'qt4'`, `'qt5'`, `'tk'`, `'wx'`). If given, the corresponding matplotlib backend is used, otherwise it will be matplotlib's default (which you can set in your matplotlib config file).

**optional arguments:**

- `-l`, `--list` Show available matplotlib backends

%notebook
%notebook [-e] filename

Export and convert IPython notebooks.

This function can export the current IPython history to a notebook file. For example, to export the history to “foo.ipynb” do “%notebook -e foo.ipynb”. To export the history to “foo.py” do “%notebook -e foo.py”.

**Positional arguments:** filename Notebook name or filename

**Optional arguments:**

- **-e, --export** Export IPython history as a notebook. The filename argument is used to specify the notebook name and format. For example a filename of notebook.ipynb will result in a notebook name of “notebook” and a format of “json”. Likewise using a “.py” file extension will write the notebook as a Python script.

%page

Pretty print the object and display it through a pager.

```
%page [options] OBJECT
```

If no object is given, use _ (last output).

Options:

- **-r**: page str(object), don’t pretty-print it.

%pastebin

Upload code to Github’s Gist paste bin, returning the URL.

**Usage:** %pastebin [-d “Custom description”] 1-7

The argument can be an input history range, a filename, or the name of a string or macro.

Options:

- **-d**: Pass a custom description for the gist. The default will say “Pasted from IPython”.

%pdb

Control the automatic calling of the pdb interactive debugger.

Call as ‘%pdb on’, ‘%pdb 1’, ‘%pdb off’ or ‘%pdb 0’. If called without argument it works as a toggle.

When an exception is triggered, IPython can optionally call the interactive pdb debugger after the traceback printout. %pdb toggles this feature on and off.

The initial state of this feature is set in your configuration file (the option is InteractiveShell.pdb).

If you want to just activate the debugger AFTER an exception has fired, without having to type ‘%pdb on’ and rerunning your code, you can use the %debug magic.

%pdef

Print the call signature for any callable object.

If the object is a class, print the constructor information.
Examples

In [3]: %pdef urllib.urlopen
urllib.urlopen(url, data=None, proxies=None)

%doc
Print the docstring for an object.

If the given object is a class, it will print both the class and the constructor docstrings.

%file
Print (or run through pager) the file where an object is defined.

The file opens at the line where the object definition begins. IPython will honor the environment variable PAGER if set, and otherwise will do its best to print the file in a convenient form.

If the given argument is not an object currently defined, IPython will try to interpret it as a filename (automatically adding a .py extension if needed). You can thus use %pfile as a syntax highlighting code viewer.

%info
Provide detailed information about an object.

‘%info object’ is just a synonym for object? or ?object.

%info2
Provide extra detailed information about an object.

‘%info2 object’ is just a synonym for object?? or ??object.

%popd
Change to directory popped off the top of the stack.

%pprint
Toggle pretty printing on/off.

%precision
Set floating point precision for pretty printing.

Can set either integer precision or a format string.

If numpy has been imported and precision is an int, numpy display precision will also be set, via numpy.set_printoptions.

If no argument is given, defaults will be restored.

Examples

In [1]: from math import pi

In [2]: %precision 3
Out[2]: u'%.3f'

In [3]: pi
Out[3]: 3.142

In [4]: %precision %i
Out[4]: u'\%i'

In [5]: pi
Out[5]: 3

In [6]: \%precision \%e
Out[6]: u'\%e'

In [7]: pi**10
Out[7]: 9.364805e+04

In [8]: \%precision
Out[8]: u'\%r'

In [9]: pi**10
Out[9]: 93648.047476082982

%profile
Print your currently active IPython profile.

See Also
prun [run code using the Python profiler] (prun())

%prun
Run a statement through the python code profiler.

Usage, in line mode:  \%prun [options] statement

Usage, in cell mode:  \%%prun [options] [statement] code... code...

In cell mode, the additional code lines are appended to the (possibly empty) statement in the first line. Cell mode allows you to easily profile multiline blocks without having to put them in a separate function.

The given statement (which doesn’t require quote marks) is run via the python profiler in a manner similar to the profile.run() function. Namespaces are internally managed to work correctly; profile.run cannot be used in IPython because it makes certain assumptions about namespaces which do not hold under IPython.

Options:

- \-l <limit> you can place restrictions on what or how much of the profile gets printed. The limit value can be:
  • A string: only information for function names containing this string is printed.
  • An integer: only these many lines are printed.
  • A float (between 0 and 1): this fraction of the report is printed (for example, use a limit of 0.4 to see the topmost 40% only).

You can combine several limits with repeated use of the option. For example, \-l __init__ \-l 5 will print only the topmost 5 lines of information about class constructors.
-r

return the pstats.Stats object generated by the profiling. This object has all the information about the profile in it, and you can later use it for further analysis or in other functions.

-s <key>

sort profile by given key. You can provide more than one key by using the option several times: `-s key1 -s key2 -s key3...`. The default sorting key is ‘time’.

The following is copied verbatim from the profile documentation referenced below:

When more than one key is provided, additional keys are used as secondary criteria when the there is equality in all keys selected before them.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

<table>
<thead>
<tr>
<th>Valid Arg</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>“calls”</td>
<td>call count</td>
</tr>
<tr>
<td>“cumulative”</td>
<td>cumulative time</td>
</tr>
<tr>
<td>“file”</td>
<td>file name</td>
</tr>
<tr>
<td>“module”</td>
<td>file name</td>
</tr>
<tr>
<td>“pcalls”</td>
<td>primitive call count</td>
</tr>
<tr>
<td>“line”</td>
<td>line number</td>
</tr>
<tr>
<td>“name”</td>
<td>function name</td>
</tr>
<tr>
<td>“nfl”</td>
<td>name/file/line</td>
</tr>
<tr>
<td>“stdname”</td>
<td>standard name</td>
</tr>
<tr>
<td>“time”</td>
<td>internal time</td>
</tr>
</tbody>
</table>

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (i.e., alphabetical). The subtle distinction between “nfl” and “stdname” is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order “20” “3” and “40”. In contrast, “nfl” does a numeric compare of the line numbers. In fact, sort_stats(“nfl”) is the same as sort_stats(“name”, “file”, “line”).

-T <filename>

save profile results as shown on screen to a text file. The profile is still shown on screen.

-D <filename>

save (via dump_stats) profile statistics to given filename. This data is in a format understood by the pstats module, and is generated by a call to the dump_stats() method of profile objects. The profile is still shown on screen.

-q

suppress output to the pager. Best used with -T and/or -D above.

If you want to run complete programs under the profiler’s control, use `%run -p [prof_opts]`
filename.py [args to program] where prof_opts contains profiler specific options as described here.

You can read the complete documentation for the profile module with:

```python
In [1]: import profile; profile.help()
```

### %psearch

Search for object in namespaces by wildcard.

```bash
%psearch [options] PATTERN [OBJECT TYPE]
```

Note: ? can be used as a synonym for %psearch, at the beginning or at the end: both a*? and ?a* are equivalent to "%psearch a*". Still, the rest of the command line must be unchanged (options come first), so for example the following forms are equivalent:

```bash
%psearch -i a* function -i a* function? ?-i a* function
```

Arguments:

- **PATTERN**

  where PATTERN is a string containing * as a wildcard similar to its use in a shell. The pattern is matched in all namespaces on the search path. By default objects starting with a single _ are not matched, many IPython generated objects have a single underscore. The default is case insensitive matching. Matching is also done on the attributes of objects and not only on the objects in a module.

- **[OBJECT TYPE]**

  Is the name of a python type from the types module. The name is given in lowercase without the ending type, ex. StringType is written string. By adding a type here only objects matching the given type are matched. Using all here makes the pattern match all types (this is the default).

Options:

- **-a**: makes the pattern match even objects whose names start with a single underscore. These names are normally omitted from the search.

- **-i/-c**: make the pattern case insensitive/sensitive. If neither of these options are given, the default is read from your configuration file, with the option InteractiveShell.wildcards_case_sensitive. If this option is not specified in your configuration file, IPython’s internal default is to do a case sensitive search.

- **-e/-s NAMESPACE**: exclude/search a given namespace. The pattern you specify can be searched in any of the following namespaces: ‘builtin’, ‘user’, ‘user_global’, ‘internal’, ‘alias’, where ‘builtin’ and ‘user’ are the search defaults. Note that you should not use quotes when specifying namespaces.

  ‘Builtin’ contains the python module builtin, ‘user’ contains all user data, ‘alias’ only contain the shell aliases and no python objects, ‘internal’ contains objects used by IPython. The ‘user_global’ namespace is only used by embedded IPython instances, and it contains module-level globals. You can add namespaces to the search with -s or exclude them with -e (these options can be given more than once).
Examples

%psearch a* -> objects beginning with an a
%psearch -e builtin a* -> objects NOT in the builtin space starting in a
%psearch a* function -> all functions beginning with an a
%psearch re.e* -> objects beginning with an e in module re
%psearch r*.e* -> objects that start with e in modules starting in r
%psearch r*.* string -> all strings in modules beginning with r

Case sensitive search:

%psearch -c a* list all objects beginning with lower case a

Show objects beginning with a single _:

%psearch -a _* list objects beginning with a single underscore

%psource
Print (or run through pager) the source code for an object.

%pushd
Place the current dir on stack and change directory.
Usage: %pushd [‘dirname’]

%pwd
Return the current working directory path.

Examples

In [9]: pwd
Out[9]: '/home/tsuser/sprint/ipython'

%pycat
Show a syntax-highlighted file through a pager.

This magic is similar to the cat utility, but it will assume the file to be Python source and will show it
with syntax highlighting.

This magic command can either take a local filename, an url, an history range (see %history) or a
macro as argument

%pycat myscript.py
%pycat 7-27
%pycat myMacro
%pycat http://www.example.com/myscript.py

%pylab

%pylab [--no-import-all] [gui]

Load numpy and matplotlib to work interactively.

This function lets you activate pylab (matplotlib, numpy and interactive support) at any point during
an IPython session.
%%pylab makes the following imports:

```
import numpy
import matplotlib
from matplotlib import pylab, mlab, pyplot
np = numpy
plt = pyplot

from IPython.display import display
from IPython.core.pylabtools import figsize, getfigs

from pylab import *
from numpy import *
```

If you pass `--no-import-all`, the last two `*` imports will be excluded.

See the `%matplotlib` magic for more details about activating matplotlib without affecting the interactive namespace.

**positional arguments:**

```

gui Name of the matplotlib backend to use (‘gtk’, ‘gtk3’, ‘inline’, ‘nbagg’, ‘notebook’, ‘osx’, ‘qt’, ‘qt4’, ‘qt5’, ‘tk’, ‘wx’). If given, the corresponding matplotlib backend is used, otherwise it will be matplotlib’s default (which you can set in your matplotlib config file).
```

**optional arguments:**

```

--no-import-all Prevent IPython from performing `import *` into the interactive namespace. You can govern the default behavior of this flag with the InteractiveShellApp.pylab_import_all configurable.
```

**%quickref**

Show a quick reference sheet

**%recall**

Repeat a command, or get command to input line for editing.

%recall and %rep are equivalent.

```
•%recall (no arguments):

Place a string version of last computation result (stored in the special ‘_' variable) to the next input prompt. Allows you to create elaborate command lines without using copy-paste:
```

```
In[1]: l = ['hei', 'vaan']
In[2]: ''.join(l)
Out[2]: heivaan
In[3]: %recall
```

%recall 45

Place history line 45 on the next input prompt. Use %hist to find out the number.

%recall 1-4

---

### 4.2. Built-in magic commands

---

303
Combine the specified lines into one cell, and place it on the next input prompt. See %history for the slice syntax.

%recall foo+bar

If foo+bar can be evaluated in the user namespace, the result is placed at the next input prompt. Otherwise, the history is searched for lines which contain that substring, and the most recent one is placed at the next input prompt.

%rehashx

Update the alias table with all executable files in $PATH.

rehashx explicitly checks that every entry in $PATH is a file with execute access (os.X_OK).

Under Windows, it checks executability as a match against a ‘|’-separated string of extensions, stored in the IPython config variable win_exec_ext. This defaults to ‘.exe|com|bat’.

This function also resets the root module cache of module completer, used on slow filesystems.

%reload_ext

Reload an IPython extension by its module name.

%rerun

Re-run previous input

By default, you can specify ranges of input history to be repeated (as with %history). With no arguments, it will repeat the last line.

Options:

- -l <n> : Repeat the last n lines of input, not including the current command.
- -g foo : Repeat the most recent line which contains foo

%reset

Resets the namespace by removing all names defined by the user, if called without arguments, or by removing some types of objects, such as everything currently in IPython’s In[] and Out[] containers (see the parameters for details).

Parameters

- -f : force reset without asking for confirmation.
- -s ['Soft’ reset: Only clears your namespace, leaving history intact.] References to objects may be kept. By default (without this option), we do a ‘hard’ reset, giving you a new session and removing all references to objects from the current session.

in : reset input history

out : reset output history

dhist : reset directory history

array : reset only variables that are NumPy arrays

See Also

reset_selective : invoked as %reset_selective
Examples

In [6]: a = 1
In [7]: a
Out[7]: 1
In [8]: 'a' in _ip.user_ns
Out[8]: True
In [9]: %reset -f
In [1]: 'a' in _ip.user_ns
Out[1]: False
In [2]: %reset -f in
Flushing input history
In [3]: %reset -f dhist in
Flushing directory history
Flushing input history

Notes

Calling this magic from clients that do not implement standard input, such as the ipython notebook interface, will reset the namespace without confirmation.

%reset_selective

Resets the namespace by removing names defined by the user.

Input/Output history are left around in case you need them.

%reset_selective [-f] regex

No action is taken if regex is not included

Options

-f: force reset without asking for confirmation.

See Also

reset: invoked as %reset

Examples

We first fully reset the namespace so your output looks identical to this example for pedagogical reasons; in practice you do not need a full reset:

In [1]: %reset -f

Now, with a clean namespace we can make a few variables and use %reset_selective to only delete names that match our regexp:

In [2]: a=1; b=2; c=3; b1m=4; b2m=5; b3m=6; b4m=7; b2s=8
In [3]: who_ls
Out[3]: ['a', 'b', 'b1m', 'b2m', 'b2s', 'b3m', 'b4m', 'c']
In 

In [4]: %reset_selective -f b[2-3]m
In [5]: who_ls
Out[5]: ['a', 'b', 'blm', 'b2s', 'b4m', 'c']
In [6]: %reset_selective -f d
In [7]: who_ls
Out[7]: ['a', 'b', 'blm', 'b2s', 'b4m', 'c']
In [8]: %reset_selective -f c
In [9]: who_ls
Out[9]: ['a', 'b', 'blm', 'b2s', 'b4m']
In [10]: %reset_selective -f b
In [11]: who_ls
Out[11]: ['a']

Notes
Calling this magic from clients that do not implement standard input, such as the ipython notebook interface, will reset the namespace without confirmation.

%run
Run the named file inside IPython as a program.
Usage:
%run [-n -i -e -G]

Parameters after the filename are passed as command-line arguments to the program (put in sys.argv). Then, control returns to IPython’s prompt.

This is similar to running at a system prompt python file args, but with the advantage of giving you IPython’s tracebacks, and of loading all variables into your interactive namespace for further use (unless -p is used, see below).

The file is executed in a namespace initially consisting only of __name__==’__main__’ and sys.argv constructed as indicated. It thus sees its environment as if it were being run as a stand-alone program (except for sharing global objects such as previously imported modules). But after execution, the IPython interactive namespace gets updated with all variables defined in the program (except for __name__ and sys.argv). This allows for very convenient loading of code for interactive work, while giving each program a ‘clean sheet’ to run in.

Arguments are expanded using shell-like glob match. Patterns ‘*’, ‘?’, ‘[seq]’ and ‘[!seq]’ can be used. Additionally, tilde ‘~’ will be expanded into user’s home directory. Unlike real shells, quotation does not suppress expansions. Use two back slashes (e.g. FLAGS) to suppress expansions. To completely disable these expansions, you can use -G flag.

Options:
-n  __name__ is NOT set to `__main__`, but to the running file's name without extension (as python does under import). This allows running scripts and reloading the definitions in them without calling code protected by an if __name__ == "__main__" clause.

-i  run the file in IPython's namespace instead of an empty one. This is useful if you are experimenting with code written in a text editor which depends on variables defined interactively.

-e  ignore sys.exit() calls or SystemExit exceptions in the script being run. This is particularly useful if IPython is being used to run unitests, which always exit with a sys.exit() call. In such cases you are interested in the output of the test results, not in seeing a traceback of the unittest module.

-t  print timing information at the end of the run. IPython will give you an estimated CPU time consumption for your script, which under Unix uses the resource module to avoid the wraparound problems of time.clock(). Under Unix, an estimate of time spent on system tasks is also given (for Windows platforms this is reported as 0.0).

If -t is given, an additional -N<N> option can be given, where <N> must be an integer indicating how many times you want the script to run. The final timing report will include total and per run results.

For example (testing the script uniq_stable.py):

In [1]: run -t uniq_stable

IPython CPU timings (estimated):
   User :  0.19597 s.
   System:  0.0 s.

In [2]: run -t -N5 uniq_stable

IPython CPU timings (estimated):
Total runs performed: 5
Times :   Total  Per run
User :  0.910862 s,  0.1821724 s.
System:  0.0 s,  0.0 s.

-d  run your program under the control of pdb, the Python debugger. This allows you to execute your program step by step, watch variables, etc. Internally, what IPython does is similar to calling:

```
pdb.run('execfile("YOURFILENAME")')
```

with a breakpoint set on line 1 of your file. You can change the line number for this automatic breakpoint to be <N> by using the -bN option (where N must be an integer). For example:

```
%run -d -b40 myscript
```

will set the first breakpoint at line 40 in myscript.py. Note that the
first breakpoint must be set on a line which actually does something
(not a comment or docstring) for it to stop execution.

Or you can specify a breakpoint in a different file:

```
%run -d -b myotherfile.py:20 myscript
```

When the pdb debugger starts, you will see a (Pdb) prompt. You
must first enter ‘c’ (without quotes) to start execution up to the first
breakpoint.

Entering ‘help’ gives information about the use of the debugger. You
can easily see pdb’s full documentation with “import pdb;pdb.help()”
at a prompt.

-p

run program under the control of the Python profiler module (which
prints a detailed report of execution times, function calls, etc).

You can pass other options after -p which affect the behavior of the
profiler itself. See the docs for %prun for details.

In this mode, the program’s variables do NOT propagate back to the
IPython interactive namespace (because they remain in the names-
pace where the profiler executes them).

Internally this triggers a call to %prun, see its documentation for de-
tails on the options available specifically for profiling.

There is one special usage for which the text above doesn’t apply: if the filename ends with .ipy[nb],
the file is run as ipython script, just as if the commands were written on IPython prompt.

-m

specify module name to load instead of script path. Similar to the -m
option for the python interpreter. Use this option last if you want to
combine with other %run options. Unlike the python interpreter only
source modules are allowed no .pyc or .pyo files. For example:

```
%run -m example
```

will run the example module.

-G
disable shell-like glob expansion of arguments.

%save
Save a set of lines or a macro to a given filename.

Usage:  %save [options] filename n1-n2 n3-n4 ... n5 .. n6 ...

Options:

- f: force overwrite. If file exists, %save will prompt for overwrite unless -f is given.
- a: append to the file instead of overwriting it.
This function uses the same syntax as \%history for input ranges, then saves the lines to the filename you specify.

It adds a `.py` extension to the file if you don’t do so yourself, and it asks for confirmation before overwriting existing files.

If `-r` option is used, the default extension is `.ipy`.

\%sc

Shell capture - run shell command and capture output (DEPRECATED use `!`).

DEPRECATED. Suboptimal, retained for backwards compatibility.

You should use the form `var = !command` instead. Example:

```
"\%sc -l myfiles = ls ~" should now be written as
"myfiles = !ls ~"
```

myfiles.s, myfiles.l and myfiles.n still apply as documented below.

\%sc [options] varname=command

IPython will run the given command using commands.getoutput(), and will then update the user’s interactive namespace with a variable called varname, containing the value of the call. Your command can contain shell wildcards, pipes, etc.

The `=` sign in the syntax is mandatory, and the variable name you supply must follow Python’s standard conventions for valid names.

(A special format without variable name exists for internal use)

Options:

- `-l`: list output. Split the output on newlines into a list before assigning it to the given variable. By default the output is stored as a single string.
- `-v`: verbose. Print the contents of the variable.

In most cases you should not need to split as a list, because the returned value is a special type of string which can automatically provide its contents either as a list (split on newlines) or as a space-separated string. These are convenient, respectively, either for sequential processing or to be passed to a shell command.

For example:

```
# Capture into variable a
In [1]: sc a=ls *py

# a is a string with embedded newlines
In [2]: a
Out[2]: 'setup.py\nwin32_manual_post_install.py'

# which can be seen as a list:
In [3]: a.l
Out[3]: ['setup.py', 'win32_manual_post_install.py']

# or as a whitespace-separated string:
```
In [4]: a.s
Out[4]: 'setup.py win32_manual_post_install.py'

# a.s is useful to pass as a single command line:
In [5]: !wc -l $a.s
   146 setup.py
   130 win32_manual_post_install.py
   276 total

# while the list form is useful to loop over:
In [6]: for f in a.l:
   ...: !wc -l $f
   ...
   146 setup.py
   130 win32_manual_post_install.py

Similarly, the lists returned by the -l option are also special, in the sense that you can equally invoke
the .s attribute on them to automatically get a whitespace-separated string from their contents:
In [7]: sc -l b=ls *py
In [8]: b
Out[8]: ['setup.py', 'win32_manual_post_install.py']
In [9]: b.s
Out[9]: 'setup.py win32_manual_post_install.py'

In summary, both the lists and strings used for output capture have the following special attributes:

| .l (or .list) | value as list. |
| .n (or .nlstr) | value as newline-separated string. |
| .s (or .spstr) | value as space-separated string. |

%set_env
Set environment variables. Assumptions are that either “val” is a name in the user namespace, or val
is something that evaluates to a string.

Usage: %set_env var val: set value for var %set_env var=val: set value for var %set_env var=$val:
   set value for var, using python expansion if possible

%sx
Shell execute - run shell command and capture output (!! is short-hand).

%sx command
IPython will run the given command using commands.getoutput(), and return the result formatted as
a list (split on ‘n’). Since the output is _returned_, it will be stored in ipython’s regular output cache
Out[N] and in the ‘_N’ automatic variables.

Notes:
1) If an input line begins with ‘!!’, then %sx is automatically invoked. That is, while:
!ls

causes ipython to simply issue system('ls'), typing:

!!ls

is a shorthand equivalent to:

%sx ls

2) %sx differs from %sc in that %sx automatically splits into a list, like `%sc -l`. The reason for this is to make it as easy as possible to process line-oriented shell output via further python commands. %sc is meant to provide much finer control, but requires more typing.

3) Just like %sc -l, this is a list with special attributes:

.. l (or .list) : value as list.
.. n (or .nlstr): value as newline-separated string.
.. s (or .spstr): value as whitespace-separated string.

This is very useful when trying to use such lists as arguments to system commands.

**%system**

Shell execute - run shell command and capture output (!! is short-hand).

%sx command

IPython will run the given command using commands.getoutput(), and return the result formatted as a list (split on ‘n’). Since the output is _returned_, it will be stored in ipython’s regular output cache Out[N] and in the ‘_N’ automatic variables.

Notes:

1) If an input line begins with ‘!!’, then %sx is automatically invoked. That is, while:

!!ls

causes ipython to simply issue system('ls'), typing:

!!ls

is a shorthand equivalent to:

%sx ls

2) %sx differs from %sc in that %sx automatically splits into a list, like ‘%sc -l’. The reason for this is to make it as easy as possible to process line-oriented shell output via further python commands. %sc is meant to provide much finer control, but requires more typing.

3) Just like %sc -l, this is a list with special attributes:

.. l (or .list) : value as list.
.. n (or .nlstr): value as newline-separated string.
.. s (or .spstr): value as whitespace-separated string.

This is very useful when trying to use such lists as arguments to system commands.
%tb
Print the last traceback with the currently active exception mode.
See %xmode for changing exception reporting modes.

%time
Time execution of a Python statement or expression.
The CPU and wall clock times are printed, and the value of the expression (if any) is returned. Note
that under Win32, system time is always reported as 0, since it can not be measured.
This function can be used both as a line and cell magic:

• In line mode you can time a single-line statement (though multiple ones can be chained with
  using semicolons).

• In cell mode, you can time the cell body (a directly following statement raises an error).

This function provides very basic timing functionality. Use the timeit magic for more control over the
measurement.

Examples
In [1]: %time 2**128
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00
Out[1]: 34028236692093846346337431768211456L

In [2]: n = 1000000
In [3]: %time sum(range(n))
CPU times: user 1.20 s, sys: 0.05 s, total: 1.25 s
Wall time: 1.37
Out[3]: 499999500000L

In [4]: %time print 'hello world'
  hello world
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00

Note that the time needed by Python to compile the given expression
will be reported if it is more than 0.1s. In this example, the
actual exponentiation is done by Python at compilation time, so while
the expression can take a noticeable amount of time to compute, that
time is purely due to the compilation:

In [5]: %time 3**9999;
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00 s

In [6]: %time 3**999999;
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00 s
Compiler : 0.78 s
%timeit

Time execution of a Python statement or expression

**Usage, in line mode:** %timeit [-n<N> -r<R> [-t|c] -q -p<P> -o] statement

**or in cell mode:** %timeit [-n<N> -r<R> [-t|c] -q -p<P> -o] setup_code code code...

Time execution of a Python statement or expression using the timeit module. This function can be used both as a line and cell magic:

- In line mode you can time a single-line statement (though multiple ones can be chained with using semicolons).
- In cell mode, the statement in the first line is used as setup code (executed but not timed) and the body of the cell is timed. The cell body has access to any variables created in the setup code.

Options:
- `-n<N>`: execute the given statement `<N>` times in a loop. If this value is not given, a fitting value is chosen.
- `-r<R>`: repeat the loop iteration `<R>` times and take the best result. Default: 3
- `-t`: use time.time to measure the time, which is the default on Unix. This function measures wall time.
- `-c`: use time.clock to measure the time, which is the default on Windows and measures wall time. On Unix, resource.getrusage is used instead and returns the CPU user time.
- `-p<P>`: use a precision of `<P>` digits to display the timing result. Default: 3
- `-q`: Quiet, do not print result.
- `-o`: return a TimeitResult that can be stored in a variable to inspect the result in more details.

**Examples**

```plaintext
In [1]: %timeit pass
10000000 loops, best of 3: 53.3 ns per loop

In [2]: u = None

In [3]: %timeit u is None
10000000 loops, best of 3: 184 ns per loop

In [4]: %timeit -r 4 u == None
10000000 loops, best of 4: 242 ns per loop

In [5]: import time

In [6]: %timeit -n1 time.sleep(2)
1 loop, best of 3: 2 s per loop
```

The times reported by %timeit will be slightly higher than those reported by the timeit.py script when variables are accessed. This is due to the fact that %timeit executes the statement in the namespace of the shell, compared with timeit.py, which uses a single setup statement to import function or create variables. Generally, the bias does not matter as long as results from timeit.py are not mixed with those from %timeit.
%unalias
Remove an alias

%unload_ext
Unload an IPython extension by its module name.

Not all extensions can be unloaded, only those which define a `unload_ipython_extension` function.

%who
Print all interactive variables, with some minimal formatting.

If any arguments are given, only variables whose type matches one of these are printed. For example:

```
%who function str
```

will only list functions and strings, excluding all other types of variables. To find the proper type names, simply use `type(var)` at a command line to see how Python prints type names. For example:

```
In [1]: type('hello')
Out[1]: <type 'str'>
```

indicates that the type name for strings is `str`.

%who always excludes executed names loaded through your configuration file and things which are internal to IPython.

This is deliberate, as typically you may load many modules and the purpose of `%who` is to show you only what you’ve manually defined.

Examples

Define two variables and list them with `who`:

```
In [1]: alpha = 123
In [2]: beta = 'test'
In [3]: %who
   alpha  beta
In [4]: %who int
   alpha
In [5]: %who str
   beta
```

%who_ls
Return a sorted list of all interactive variables.

If arguments are given, only variables of types matching these arguments are returned.

Examples

Define two variables and list them with `who_ls`:
In [1]: alpha = 123
In [2]: beta = 'test'

In [3]: %who_ls
Out[3]: ['alpha', 'beta']

In [4]: %who_ls int
Out[4]: ['alpha']

In [5]: %who_ls str
Out[5]: ['beta']

%whos
Like %who, but gives some extra information about each variable.
The same type filtering of %who can be applied here.
For all variables, the type is printed. Additionally it prints:
• For {}, [], (): their length.
• For numpy arrays, a summary with shape, number of elements, typecode and size in memory.
• Everything else: a string representation, snipping their middle if too long.

Examples
Define two variables and list them with whos:

In [1]: alpha = 123
In [2]: beta = 'test'
In [3]: %whos
Variable | Type | Data/Info
---------|------|-----------
alpha    | int  | 123       
beta     | str  | test      

%xdel
Delete a variable, trying to clear it from anywhere that IPython’s machinery has references to it. By
default, this uses the identity of the named object in the user namespace to remove references held
under other names. The object is also removed from the output history.

Options  -n: Delete the specified name from all namespaces, without checking their identity.

%xmode
Switch modes for the exception handlers.
Valid modes: Plain, Context and Verbose.
If called without arguments, acts as a toggle.
4.2.2 Cell magics

```bash
%%bash
%%bash script magic
Run cells with bash in a subprocess.
This is a shortcut for %script bash
```

```capture
%capture [--no-stderr] [--no-stdout] [--no-display] [output]
```

run the cell, capturing stdout, stderr, and IPython’s rich display() calls.

**positional arguments:**

- `output` The name of the variable in which to store output. This is a `utils.io.CapturedIO` object with `stdout/err` attributes for the text of the captured output. `CapturedOutput` also has a `show()` method for displaying the output, and `__call__` as well, so you can use that to quickly display the output. If unspecified, captured output is discarded.

**optional arguments:**

- `--no-stderr` Don’t capture stderr.
- `--no-stdout` Don’t capture stdout.
- `--no-display` Don’t capture IPython’s rich display.

```html
%%html
Render the cell as a block of HTML
```

```javascript
%%javascript
Run the cell block of Javascript code
```

```latex
%%latex
Render the cell as a block of latex
The subset of latex which is support depends on the implementation in the client. In the Jupyter Notebook, this magic only renders the subset of latex defined by MathJax [here](https://docs.mathjax.org/en/v2.5-latest/tex.html).
```

```perl
%%perl
%%perl script magic
Run cells with perl in a subprocess.
This is a shortcut for %script perl
```

```pypy
%%pypy
%%pypy script magic
Run cells with pypy in a subprocess.
This is a shortcut for %script pypy
```
%%python
%%python script magic
Run cells with python in a subprocess.
This is a shortcut for %%script python

%%python2
%%python2 script magic
Run cells with python2 in a subprocess.
This is a shortcut for %%script python2

%%python3
%%python3 script magic
Run cells with python3 in a subprocess.
This is a shortcut for %%script python3

%%ruby
%%ruby script magic
Run cells with ruby in a subprocess.
This is a shortcut for %%script ruby

%%script
%shebang [--proc PROC] [--bg] [--err ERR] [--out OUT]

Run a cell via a shell command
The %%script line is like the #! line of script, specifying a program (bash, perl, ruby, etc.) with which to run.
The rest of the cell is run by that program.

Examples

In [1]: %%script bash
   ...: for i in 1 2 3; do
   ...:   echo $i
   ...: done
1
2
3

optional arguments:
--proc PROC The variable in which to store Popen instance. This is used only when --bg option is given.
--bg Whether to run the script in the background. If given, the only way to see the output of the command is with --out/err.
--err ERR  The variable in which to store stderr from the script. If the script is backgrounded, this will be the stderr pipe, instead of the stderr text itself.

--out OUT  The variable in which to store stdout from the script. If the script is backgrounded, this will be the stdout pipe, instead of the stdout text itself.

%%sh

%%sh script magic
Run cells with sh in a subprocess.
This is a shortcut for %%script sh

%%svg
Render the cell as an SVG literal

%%writefile

%writefile [-a] filename

Write the contents of the cell to a file.
The file will be overwritten unless the -a (--append) flag is specified.

positional arguments:  filename file to write

optional arguments:

-a, --append  Append contents of the cell to an existing file. The file will be created if it does not exist.

4.3 Plotting

One major feature of the IPython kernel is the ability to display plots that are the output of running code cells. The IPython kernel is designed to work seamlessly with the matplotlib plotting library to provide this functionality.

To set this up, before any plotting is performed you must execute the %matplotlib magic command. This performs the necessary behind-the-scenes setup for IPython to work correctly hand in hand with matplotlib; it does not, however, actually execute any Python import commands, that is, no names are added to the namespace.

If the %matplotlib magic is called without an argument, the output of a plotting command is displayed using the default matplotlib backend in a separate window. Alternatively, the backend can be explicitly requested using, for example:

%matplotlib gtk

A particularly interesting backend, provided by IPython, is the inline backend. This is available only for the Jupyter Notebook and the Jupyter QtConsole. It can be invoked as follows:
With this backend, the output of plotting commands is displayed inline within the notebook, directly below the code cell that produced it. The resulting plots will then also be stored in the notebook document.

See also:

Plotting with Matplotlib example notebook

4.4 IPython reference

4.4.1 Command-line usage

You start IPython with the command:

$ ipython [options] files

If invoked with no options, it executes all the files listed in sequence and drops you into the interpreter while still acknowledging any options you may have set in your ipython_config.py. This behavior is different from standard Python, which when called as python -i will only execute one file and ignore your configuration setup.

Please note that some of the configuration options are not available at the command line, simply because they are not practical here. Look into your configuration files for details on those. There are separate configuration files for each profile, and the files look like ipython_config.py or ipython_config_frontendname.py. Profile directories look like profile_profilename and are typically installed in the IPYTHONDIR directory, which defaults to $HOME/.ipython. For Windows users, HOME resolves to C:\Users\YourUserName in most instances.

Command-line Options

To see the options IPython accepts, use ipython --help (and you probably should run the output through a pager such as ipython --help | less for more convenient reading). This shows all the options that have a single-word alias to control them, but IPython lets you configure all of its objects from the command-line by passing the full class name and a corresponding value; type ipython --help-all to see this full list. For example:

ipython --matplotlib qt

is equivalent to:

ipython --TerminalIPythonApp.matplotlib='qt'

Note that in the second form, you must use the equal sign, as the expression is evaluated as an actual Python assignment. While in the above example the short form is more convenient, only the most common options have a short form, while any configurable variable in IPython can be set at the command-line by using the long form. This long form is the same syntax used in the configuration files, if you want to set these options permanently.
4.4.2 Interactive use

IPython is meant to work as a drop-in replacement for the standard interactive interpreter. As such, any code which is valid python should execute normally under IPython (cases where this is not true should be reported as bugs). It does, however, offer many features which are not available at a standard python prompt. What follows is a list of these.

Caution for Windows users

Windows, unfortunately, uses the ‘\’ character as a path separator. This is a terrible choice, because ‘\’ also represents the escape character in most modern programming languages, including Python. For this reason, using ‘/’ character is recommended if you have problems with ‘\’. However, in Windows commands ‘/’ flags options, so you can not use it for the root directory. This means that paths beginning at the root must be typed in a contrived manner like: %copy \opt/foo/bar.txt \tmp

Magic command system

IPython will treat any line whose first character is a % as a special call to a ‘magic’ function. These allow you to control the behavior of IPython itself, plus a lot of system-type features. They are all prefixed with a % character, but parameters are given without parentheses or quotes.

Lines that begin with %% signal a cell magic: they take as arguments not only the rest of the current line, but all lines below them as well, in the current execution block. Cell magics can in fact make arbitrary modifications to the input they receive, which need not even be valid Python code at all. They receive the whole block as a single string.

As a line magic example, the %cd magic works just like the OS command of the same name:

```
In [8]: %cd
/home/fperez
```

The following uses the builtin $timeit in cell mode:

```
In [10]: %%timeit x = range(10000)
    ...: min(x)
    ...: max(x)
    ....
1000 loops, best of 3: 438 us per loop
```

In this case, x = range(10000) is called as the line argument, and the block with min(x) and max(x) is called as the cell body. The $timeit magic receives both.

If you have ‘automagic’ enabled (as it is by default), you don’t need to type in the single % explicitly for line magics; IPython will scan its internal list of magic functions and call one if it exists. With automagic on you can then just type cd mydir to go to directory ‘mydir’:

```
In [9]: cd mydir
/home/fperez/mydir
```

Cell magics always require an explicit %% prefix, automagic calling only works for line magics.
The automagic system has the lowest possible precedence in name searches, so you can freely use variables with the same names as magic commands. If a magic command is ‘shadowed’ by a variable, you will need the explicit % prefix to use it:

In [1]: cd ipython  # %cd is called by automagic
/home/fperez/ipython

In [2]: cd=1  # now cd is just a variable

In [3]: cd ..  # and doesn't work as a function anymore
File "<ipython-input-3-9fedb3aff56c>", line 1
  cd ..
^  
SyntaxError: invalid syntax

In [4]: %cd ..  # but %cd always works
/home/fperez

In [5]: del cd  # if you remove the cd variable, automagic works again

In [6]: cd ipython
/home/fperez/ipython

Line magics, if they return a value, can be assigned to a variable using the syntax l = %sx ls (which in this particular case returns the result of ls as a python list). See below for more information.

Type %magic for more information, including a list of all available magic functions at any time and their docstrings. You can also type %magic_function_name? (see below for information on the ‘?’ system) to get information about any particular magic function you are interested in.

The API documentation for the IPython.core.magic module contains the full docstrings of all currently available magic commands.

See also:

Built-in magic commands A list of the line and cell magics available in IPython by default

Defining custom magics How to define and register additional magic functions

Access to the standard Python help

Simply type help() to access Python’s standard help system. You can also type help(object) for information about a given object, or help(‘keyword’) for information on a keyword. You may need to configure your PYTHONDOCS environment variable for this feature to work correctly.

Dynamic object information

Typing ?word or word? prints detailed information about an object. If certain strings in the object are too long (e.g. function signatures) they get snipped in the center for brevity. This system gives access variable types and values, docstrings, function prototypes and other useful information.
If the information will not fit in the terminal, it is displayed in a pager (less if available, otherwise a basic internal pager).

Typing `??word` or `word??` gives access to the full information, including the source code where possible. Long strings are not snipped.

The following magic functions are particularly useful for gathering information about your working environment:

- `%pdoc <object>`: Print (or run through a pager if too long) the docstring for an object. If the given object is a class, it will print both the class and the constructor docstrings.
- `%pdef <object>`: Print the call signature for any callable object. If the object is a class, print the constructor information.
- `%psource <object>`: Print (or run through a pager if too long) the source code for an object.
- `%pfile <object>`: Show the entire source file where an object was defined via a pager, opening it at the line where the object definition begins.
- `%who/%whos`: These functions give information about identifiers you have defined interactively (not things you loaded or defined in your configuration files). `%who` just prints a list of identifiers and `%whos` prints a table with some basic details about each identifier.

The dynamic object information functions (??, `%pdoc`, `%pfile`, `%pdef`, `%psource) work on object attributes, as well as directly on variables. For example, after doing `import os`, you can use `os.path.abspath??`.

### Readline-based features

These features require the GNU readline library, so they won’t work if your Python installation lacks readline support. We will first describe the default behavior IPython uses, and then how to change it to suit your preferences.

#### Command line completion

At any time, hitting TAB will complete any available python commands or variable names, and show you a list of the possible completions if there’s no unambiguous one. It will also complete filenames in the current directory if no python names match what you’ve typed so far.

#### Search command history

IPython provides two ways for searching through previous input and thus reduce the need for repetitive typing:

1. Start typing, and then use the up and down arrow keys (or Ctrl-p and Ctrl-n) to search through only the history items that match what you’ve typed so far.
2. Hit Ctrl-r: to open a search prompt. Begin typing and the system searches your history for lines that contain what you’ve typed so far, completing as much as it can.
IPython will save your input history when it leaves and reload it next time you restart it. By default, the history file is named `.ipython/profile_name/history.sqlite`.

**Autoindent**

IPython can recognize lines ending in ‘:’ and indent the next line, while also un-indenting automatically after ‘raise’ or ‘return’.

This feature uses the readline library, so it will honor your `~/.inputrc` configuration (or whatever file your `INPUTRC` environment variable points to). Adding the following lines to your `.inputrc` file can make indenting/unindenting more convenient (M-i indents, M-u unindents):

```bash
#if you don't already have a ~/.inputrc file, you need this include:
#include /etc/inputrc

$if Python
 "\M-i": " "
 "\M-u": "\d\d\d\d"
$endif

Note that there are 4 spaces between the quote marks after “M-i” above.

**Warning:** Setting the above indents will cause problems with unicode text entry in the terminal.

**Warning:** Autoindent is ON by default, but it can cause problems with the pasting of multi-line indented code (the pasted code gets re-indented on each line). A magic function `%autoindent` allows you to toggle it on/off at runtime. You can also disable it permanently on in your `ipython_config.py` file (set `TerminalInteractiveShell.autoindent=False`). If you want to paste multiple lines in the terminal, it is recommended that you use `%paste`.

**Customizing readline behavior**

All these features are based on the GNU readline library, which has an extremely customizable interface. Normally, readline is configured via a `.inputrc` file. IPython respects this, and you can also customise readline by setting the following configuration options:

- `InteractiveShell.readline_parse_and_bind`: this holds a list of strings to be executed via a `readline.parse_and_bind()` command. The syntax for valid commands of this kind can be found by reading the documentation for the GNU readline library, as these commands are of the kind which readline accepts in its configuration file.

- `InteractiveShell.readline_remove_delims`: a string of characters to be removed from the default word-delimiters list used by readline, so that completions may be performed on strings which contain them. Do not change the default value unless you know what you’re doing.

You will find the default values in your configuration file.
Session logging and restoring

You can log all input from a session either by starting IPython with the command line switch
`--logfile=foo.py` (see here) or by activating the logging at any moment with the magic function
`%logstart`.

Log files can later be reloaded by running them as scripts and IPython will attempt to ‘replay’ the log by
executing all the lines in it, thus restoring the state of a previous session. This feature is not quite perfect,
but can still be useful in many cases.

The log files can also be used as a way to have a permanent record of any code you wrote while experiment-
ing. Log files are regular text files which you can later open in your favorite text editor to extract code or to
‘clean them up’ before using them to replay a session.

The `%logstart` function for activating logging in mid-session is used as follows:

```
%logstart [log_name [log_mode]]
```

If no name is given, it defaults to a file named ‘ipython_log.py’ in your current working directory, in ‘rotate’
mode (see below).

‘%logstart name’ saves to file ‘name’ in ‘backup’ mode. It saves your history up to that point and then
continues logging.

`%logstart` takes a second optional parameter: logging mode. This can be one of (note that the modes are
given unquoted):

- [over:] overwrite existing log_name.
- [backup:] rename (if exists) to log_name~ and start log_name.
- [append:] well, that says it.
- [rotate:] create rotating logs log_name.1~, log_name.2~, etc.

The `%logoff` and `%logon` functions allow you to temporarily stop and resume logging to a file which had
previously been started with `%logstart`. They will fail (with an explanation) if you try to use them before
logging has been started.

System shell access

Any input line beginning with a ! character is passed verbatim (minus the !, of course) to the underlying
operating system. For example, typing `!ls` will run ‘ls’ in the current directory.

Manual capture of command output and magic output

You can assign the result of a system command to a Python variable with the syntax `myfiles = !ls`. Similarly,
the result of a magic (as long as it returns a value) can be assigned to a variable. For example, the
syntax `myfiles = %sx ls` is equivalent to the above system command example (the `%sx` magic runs
a shell command and captures the output). Each of these gets machine readable output from stdout (e.g.
without colours), and splits on newlines. To explicitly get this sort of output without assigning to a variable,
use two exclamation marks (!!!ls) or the $$sx$$ magic command without an assignment. (However, !! commands cannot be assigned to a variable.)

The captured list in this example has some convenience features. `myfiles.n` or `myfiles.s` returns a string delimited by newlines or spaces, respectively. `myfiles.p` produces path objects from the list items. See String lists for details.

IPython also allows you to expand the value of python variables when making system calls. Wrap variables or expressions in `{braces}`:

```
In [1]: pyvar = 'Hello world'
In [2]: !echo "A python variable: {pyvar}"  
A python variable: Hello world
In [3]: import math
In [4]: x = 8
In [5]: !echo {math.factorial(x)}
40320
```

For simple cases, you can alternatively prepend $ to a variable name:

```
In [6]: !echo $sys.argv  
[/home/fperez/usr/bin/ipython]
In [7]: !echo "A system variable: $$HOME"  
A system variable: /home/fperez
```

Note that $$ is used to represent a literal $.

### System command aliases

The `%alias` magic function allows you to define magic functions which are in fact system shell commands. These aliases can have parameters.

```
%alias alias_name cmd defines ‘alias_name’ as an alias for ‘cmd’
```

Then, typing `alias_name params` will execute the system command ‘cmd params’ (from your underlying operating system).

You can also define aliases with parameters using %s specifiers (one per parameter). The following example defines the parts function as an alias to the command ‘echo first %s second %s’ where each %s will be replaced by a positional parameter to the call to %parts:

```
In [1]: %alias parts echo first %s second %s
In [2]: parts A B
first A second B
In [3]: parts A
ERROR: Alias <parts> requires 2 arguments, 1 given.
```

If called with no parameters, `%alias` prints the table of currently defined aliases.

The `%rehashx` magic allows you to load your entire $PATH as ipython aliases. See its docstring for further details.
Recursive reload

The `IPython.lib.deepreload` module allows you to recursively reload a module: changes made to any of its dependencies will be reloaded without having to exit. To start using it, do:

```python
from IPython.lib.deepreload import reload as dreload
```

Verbose and colored exception traceback printouts

IPython provides the option to see very detailed exception tracebacks, which can be especially useful when debugging large programs. You can run any Python file with the `%run` function to benefit from these detailed tracebacks. Furthermore, both normal and verbose tracebacks can be colored (if your terminal supports it) which makes them much easier to parse visually.

See the magic `%xmode` and `%colors` functions for details.

These features are basically a terminal version of Ka-Ping Yee’s cgitb module, now part of the standard Python library.

Input caching system

IPython offers numbered prompts (In/Out) with input and output caching (also referred to as ‘input history’). All input is saved and can be retrieved as variables (besides the usual arrow key recall), in addition to the `%rep` magic command that brings a history entry up for editing on the next command line.

The following variables always exist:

- `_i, _ii, _iii`: store previous, next previous and next-next previous inputs.
- `In, _ih`: a list of all inputs; `_ih[n]` is the input from line n. If you overwrite `In` with a variable of your own, you can remake the assignment to the internal list with a simple `In=_ih`.

Additionally, global variables named `_i<n>` are dynamically created (`<n>` being the prompt counter), so `_i<n> == _ih[n] == In[n]`.

For example, what you typed at prompt 14 is available as `_i14, _ih[14]` and `In[14]`.

This allows you to easily cut and paste multi line interactive prompts by printing them out: they print like a clean string, without prompt characters. You can also manipulate them like regular variables (they are strings), modify or exec them.

You can also re-execute multiple lines of input easily by using the magic `%rerun` or `%macro` functions. The macro system also allows you to re-execute previous lines which include magic function calls (which require special processing). Type `%macro?` for more details on the macro system.

A history function `%history` allows you to see any part of your input history by printing a range of the `_i` variables.

You can also search (‘grep’) through your history by typing `%hist -g somestring`. This is handy for searching for URLs, IP addresses, etc. You can bring history entries listed by `%hist -g` up for editing with the `%recall` command, or run them immediately with `%rerun`. 
Output caching system

For output that is returned from actions, a system similar to the input cache exists but using _ instead of _i. Only actions that produce a result (NOT assignments, for example) are cached. If you are familiar with Mathematica, IPython’s _ variables behave exactly like Mathematica’s % variables.

The following variables always exist:

- [ ] (a single underscore): stores previous output, like Python’s default interpreter.
- [__] (two underscores): next previous.
- [___] (three underscores): next-next previous.

Additionally, global variables named _<n> are dynamically created (<n> being the prompt counter), such that the result of output <n> is always available as _<n> (don’t use the angle brackets, just the number, e.g. _21).

These variables are also stored in a global dictionary (not a list, since it only has entries for lines which returned a result) available under the names _oh and Out (similar to _ih and In). So the output from line 12 can be obtained as _12, out[12] or _oh[12]. If you accidentally overwrite the Out variable you can recover it by typing Out=_oh at the prompt.

This system obviously can potentially put heavy memory demands on your system, since it prevents Python’s garbage collector from removing any previously computed results. You can control how many results are kept in memory with the configuration option InteractiveShell.cache_size. If you set it to 0, output caching is disabled. You can also use the %reset and %xdel magics to clear large items from memory.

Directory history

Your history of visited directories is kept in the global list _dh, and the magic %cd command can be used to go to any entry in that list. The %dhist command allows you to view this history. Do cd -<TAB> to conveniently view the directory history.

Automatic parentheses and quotes

These features were adapted from Nathan Gray’s LazyPython. They are meant to allow less typing for common situations.

Callable objects (i.e. functions, methods, etc) can be invoked like this (notice the commas between the arguments):

```
In [1]: callable_ob arg1, arg2, arg3
------> callable_ob(arg1, arg2, arg3)
```

Note: This feature is disabled by default. To enable it, use the %autocall magic command. The commands below with special prefixes will always work, however.

You can force automatic parentheses by using ‘/’ as the first character of a line. For example:
In [2]: /globals # becomes 'globals()'

Note that the ‘/’ MUST be the first character on the line! This won’t work:

In [3]: print /globals # syntax error

In most cases the automatic algorithm should work, so you should rarely need to explicitly invoke /. One notable exception is if you are trying to call a function with a list of tuples as arguments (the parenthesis will confuse IPython):

In [4]: zip (1,2,3),(4,5,6) # won’t work

but this will work:

In [5]: /zip (1,2,3),(4,5,6)
-----> zip ((1, 2, 3), (4, 5, 6))
Out[5]: [(1, 4), (2, 5), (3, 6)]

IPython tells you that it has altered your command line by displaying the new command line preceded by --->.

You can force automatic quoting of a function’s arguments by using , or ; as the first character of a line. For example:

In [1]: ,my_function /home/me # becomes my_function("/home/me")

If you use ‘;’ the whole argument is quoted as a single string, while ‘,’ splits on whitespace:

In [2]: ,my_function a b c # becomes my_function("a","b","c")
In [3]: ;my_function a b c # becomes my_function("a b c")

Note that the ‘,’ or ‘;’ MUST be the first character on the line! This won’t work:

In [4]: x = ,my_function /home/me # syntax error

4.4.3 IPython as your default Python environment

Python honors the environment variable PYTHONSTARTUP and will execute at startup the file referenced by this variable. If you put the following code at the end of that file, then IPython will be your working environment anytime you start Python:

```python
import os, IPython
os.environ['PYTHONSTARTUP'] = '' # Prevent running this again
IPython.start_ipython()
raise SystemExit
```

The raise SystemExit is needed to exit Python when it finishes, otherwise you’ll be back at the normal Python >>> prompt.

This is probably useful to developers who manage multiple Python versions and don’t want to have correspondingly multiple IPython versions. Note that in this mode, there is no way to pass IPython any command-line options, as those are trapped first by Python itself.
4.4.4 Embedding IPython

You can start a regular IPython session with

```python
import IPython
IPython.start_ipython(argv=[])  
```

at any point in your program. This will load IPython configuration, startup files, and everything, just as if it were a normal IPython session.

It is also possible to embed an IPython shell in a namespace in your Python code. This allows you to evaluate dynamically the state of your code, operate with your variables, analyze them, etc. Note however that any changes you make to values while in the shell do not propagate back to the running code, so it is safe to modify your values because you won’t break your code in bizarre ways by doing so.

**Note:** At present, embedding IPython cannot be done from inside IPython. Run the code samples below outside IPython.

This feature allows you to easily have a fully functional python environment for doing object introspection anywhere in your code with a simple function call. In some cases a simple print statement is enough, but if you need to do more detailed analysis of a code fragment this feature can be very valuable.

It can also be useful in scientific computing situations where it is common to need to do some automatic, computationally intensive part and then stop to look at data, plots, etc. Opening an IPython instance will give you full access to your data and functions, and you can resume program execution once you are done with the interactive part (perhaps to stop again later, as many times as needed).

The following code snippet is the bare minimum you need to include in your Python programs for this to work (detailed examples follow later):

```python
from IPython import embed
embed()  # this call anywhere in your program will start IPython
```

You can also embed an IPython *kernel*, for use with qtconsole, etc. via IPython.embed_kernel(). This should function work the same way, but you can connect an external frontend (ipython qtconsole or ipython console), rather than interacting with it in the terminal.

You can run embedded instances even in code which is itself being run at the IPython interactive prompt with ‘%run <filename>’. Since it’s easy to get lost as to where you are (in your top-level IPython or in your embedded one), it’s a good idea in such cases to set the in/out prompts to something different for the embedded instances. The code examples below illustrate this.

You can also have multiple IPython instances in your program and open them separately, for example with different options for data presentation. If you close and open the same instance multiple times, its prompt counters simply continue from each execution to the next.

Please look at the docstrings in the `embed` module for more details on the use of this system.

The following sample file illustrating how to use the embedding functionality is provided in the examples directory as `embed_class_long.py`. It should be fairly self-explanatory:
An example of how to embed an IPython shell into a running program.

Please see the documentation in the IPython.Shell module for more details.

The accompanying file embed_class_short.py has quick code fragments for embedding which you can cut and paste in your code once you understand how things work.

The code in this file is deliberately extra-verbose, meant for learning.

```python
from __future__ import print_function

# The basics to get you going:

# IPython injects get_ipython into builtins, so you can know if you have nested copies running.

# Try running this code both at the command line and from inside IPython (with %run example-embed.py)
from traitlets.config.loader import Config
try:
    get_ipython
except NameError:
    nested = 0
    cfg = Config()
    prompt_config = cfg.PromptManager
    prompt_config.in_template = '\nIn \#>: '  # Added \n to add new line
    prompt_config.in2_template = '  \D.: '  # Added \ to add new line
    prompt_config.out_template = 'Out\#>: '
else:
    print("Running nested copies of IPython.")
    print("The prompts for the nested copy have been modified")
    cfg = Config()
    nested = 1

# First import the embeddable shell class
from IPython.terminal.embed import InteractiveShellEmbed

# Now create an instance of the embeddable shell. The first argument is a string with options exactly as you would type them if you were starting IPython at the system command line. Any parameters you want to define for configuration can thus be specified here.
ipshell = InteractiveShellEmbed(config=cfg,
    banner1 = 'Dropping into IPython',
    exit_msg = 'Leaving Interpreter, back to program."
)

# Make a second instance, you can have as many as you want.
cfg2 = cfg.copy()
prompt_config = cfg2.PromptManager
prompt_config.in_template = 'In\#>: '
if not nested:
    prompt_config.in_template = 'In2\#>: '
    prompt_config.in2_template = '  \D.: '
```

Chapter 4. Using IPython for interactive work
prompt_config.out_template = 'Out<\#>: '
ipshell2 = InteractiveShellEmbed(config=cfg,  
    banner1 = 'Second IPython instance.')

print('
Hello. This is printed from the main controller program.
')

# You can then call ipshell() anywhere you need it (with an optional 
# message):
ipshell('***Called from top level. '  
    'Hit Ctrl-D to exit interpreter and continue program.
'  
    'Note that if you use %kill_embedded, you can fully deactivate
'  
    'This embedded instance so it will never turn on again')

print('
Back in caller program, moving along...
')

# Some more details:

# InteractiveShellEmbed instances don't print the standard system banner and 
# messages. The IPython banner (which actually may contain initialization 
# messages) is available as get_ipython().banner in case you want it.

# InteractiveShellEmbed instances print the following information everytime they 
# start:

# - A global startup banner.

# - A call-specific header string, which you can use to indicate where in the 
# execution flow the shell is starting.

# They also print an exit message every time they exit.

# Both the startup banner and the exit message default to None, and can be set 
# either at the instance constructor or at any other time with the 
# by setting the banner and exit_msg attributes.

# The shell instance can be also put in 'dummy' mode globally or on a per-call 
# basis. This gives you fine control for debugging without having to change 
# code all over the place.

# The code below illustrates all this.

# This is how the global banner and exit_msg can be reset at any point
ipshell.banner = 'Entering interpreter - New Banner'
ipshell.exit_msg = 'Leaving interpreter - New exit_msg'

def foo(m):
    s = 'spam'
ipshell('***In foo(). Try %whos, or print s or m:')
    print('foo says m = ', m)

def bar(n):

4.4. IPython reference
s = 'eggs'
ipshell('***In bar(). Try %whos, or print s or n:ʽ)
print('bar says n = ', n)

# Some calls to the above functions which will trigger IPython:
print('Main program calling foo("eggs")\n')
foo('eggs')

# The shell can be put in 'dummy' mode where calls to it silently return. This
# allows you, for example, to globally turn off debugging for a program with a
# single call.
ipshell.dummy_mode = True
print('\nTrying to call IPython which is now "dummy":ʽ)
ipshell()
print('Nothing happened...')
# The global 'dummy' mode can still be overridden for a single call
print('\nOverriding dummy mode manually:ʽ)
ipshell(dummy=False)

# Reactivate the IPython shell
ipshell.dummy_mode = False

print('You can even have multiple embedded instances:ʽ)
ipshell2()

print('Main program calling bar("spam")\n')
bar('spam')

print('Main program finished. Bye!')

Once you understand how the system functions, you can use the following code fragments in your programs
which are ready for cut and paste:

"""Quick code snippets for embedding IPython into other programs.

See embed_class_long.py for full details, this file has the bare minimum code for
cut and paste use once you understand how to use the system."""

#--------------------------------------------
# This code loads IPython but modifies a few things if it detects it's running
# embedded in another IPython session (helps avoid confusion)

try:
    get_ipython
except NameError:
    banner=exit_msg=''
else:
    banner = '*** Nested interpreter ***'
    exit_msg = '*** Back in main IPython ***'

# First import the embed function
from IPython.terminal.embed import InteractiveShellEmbed
# Now create the IPython shell instance. Put ipshell() anywhere in your code
# where you want it to open.
ipshell = InteractiveShellEmbed(banner1=banner, exit_msg=exit_msg)

# This code will load an embeddable IPython shell always with no changes for # nested embeddedings.

from IPython import embed
# Now embed() will open IPython anywhere in the code.

# This code loads an embeddable shell only if NOT running inside # IPython. Inside IPython, the embeddable shell variable ipshell is just a # dummy function.

try:
    get_ipython
except NameError:
    from IPython.terminal.embed import InteractiveShellEmbed
ipshell = InteractiveShellEmbed()
# Now ipshell() will open IPython anywhere in the code
else:
    # Define a dummy ipshell() so the same code doesn't crash inside an # interactive IPython
def ipshell(): pass


4.4.5 Using the Python debugger (pdb)

Running entire programs via pdb

pdb, the Python debugger, is a powerful interactive debugger which allows you to step through code, set breakpoints, watch variables, etc. IPython makes it very easy to start any script under the control of pdb, regardless of whether you have wrapped it into a ‘main()’ function or not. For this, simply type %run -d myscript at an IPython prompt. See the %run command’s documentation for more details, including how to control where pdb will stop execution first.

For more information on the use of the pdb debugger, see Debugger Commands in the Python documentation.

IPython extends the debugger with a few useful additions, like coloring of tracebacks. The debugger will adopt the color scheme selected for IPython.

The where command has also been extended to take as argument the number of context line to show. This allows to a many line of context on shallow stack trace:

And less context on shallower Stack Trace:
Post-mortem debugging

Going into a debugger when an exception occurs can be extremely useful in order to find the origin of subtle bugs, because pdb opens up at the point in your code which triggered the exception, and while your program is at this point ‘dead’, all the data is still available and you can walk up and down the stack frame and understand the origin of the problem.

You can use the `%debug` magic after an exception has occurred to start post-mortem debugging. IPython can also call debugger every time your code triggers an uncaught exception. This feature can be toggled with the `%pdb` magic command, or you can start IPython with the `--pdb` option.

For a post-mortem debugger in your programs outside IPython, put the following lines toward the top of your ‘main’ routine:

```python
import sys
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose', color_scheme='Linux', call_pdb=1)
```

The mode keyword can be either ‘Verbose’ or ‘Plain’, giving either very detailed or normal tracebacks respectively. The color_scheme keyword can be one of ‘NoColor’, ‘Linux’ (default) or ‘LightBG’. These are the same options which can be set in IPython with `--colors` and `--xmode`.

This will give any of your programs detailed, colored tracebacks with automatic invocation of pdb.

4.4.6 Pasting of code starting with Python or IPython prompts

IPython is smart enough to filter out input prompts, be they plain Python ones (```>>>``` and `...`) or IPython ones (`In [N]:` and `...:`). You can therefore copy and paste from existing interactive sessions without worry.

The following is a ‘screenshot’ of how things work, copying an example from the standard Python tutorial:

```python
In [1]: >>> # Fibonacci series:
In [2]: ... # the sum of two elements defines the next
In [3]: ... a, b = 0, 1
In [4]: >>> while b < 10:
   ...:     ... print(b)
   ...:     ... a, b = b, a+b
   ...:
1
1
2
3
5
8
```

And pasting from IPython sessions works equally well:
4.4.7 GUI event loop support

New in version 0.11: The %gui magic and IPython.lib.inpthook.

IPython has excellent support for working interactively with Graphical User Interface (GUI) toolkits, such as wxPython, PyQt4/PySide, PyGTK and Tk. This is implemented using Python’s builtin PyOSInputHook hook. This implementation is extremely robust compared to our previous thread-based version. The advantages of this are:

- GUIs can be enabled and disabled dynamically at runtime.
- The active GUI can be switched dynamically at runtime.
- In some cases, multiple GUIs can run simultaneously with no problems.
- There is a developer API in IPython.lib.inpthook for customizing all of these things.

For users, enabling GUI event loop integration is simple. You simple use the %gui magic as follows:

%gui [GUINAME]

With no arguments, %gui removes all GUI support. Valid GUINAME arguments are wx, qt, gtk and tk. Thus, to use wxPython interactively and create a running wx.App object, do:

%gui wx

You can also start IPython with an event loop set up using the --gui flag:

$ ipython --gui=qt

For information on IPython’s matplotlib integration (and the matplotlib mode) see this section.

For developers that want to use IPython’s GUI event loop integration in the form of a library, these capabilities are exposed in library form in the IPython.lib.inpthook and IPython.lib.guisupport modules. Interested developers should see the module docstrings for more information, but there are a few points that should be mentioned here.

First, the PyOSInputHook approach only works in command line settings where readline is activated. The integration with various eventloops is handled somewhat differently (and more simply) when using the standalone kernel, as in the qtconsole and notebook.

Second, when using the PyOSInputHook approach, a GUI application should not start its event loop. Instead all of this is handled by the PyOSInputHook. This means that applications that are meant to be used both in IPython and as standalone apps need to have special code to detects how the application is being run. We highly recommend using IPython’s support for this. Since the details vary slightly between toolkits,
we point you to the various examples in our source directory `examples/Embedding` that demonstrate these capabilities.

Third, unlike previous versions of IPython, we no longer “hijack” (replace them with no-ops) the event loops. This is done to allow applications that actually need to run the real event loops to do so. This is often needed to process pending events at critical points.

Finally, we also have a number of examples in our source directory `examples/Embedding` that demonstrate these capabilities.

**PyQt and PySide**

When you use `--gui=qt` or `--matplotlib=qt`, IPython can work with either PyQt4 or PySide. There are three options for configuration here, because PyQt4 has two APIs for QString and QVariant: v1, which is the default on Python 2, and the more natural v2, which is the only API supported by PySide. v2 is also the default for PyQt4 on Python 3. IPython’s code for the QtConsole uses v2, but you can still use any interface in your code, since the Qt frontend is in a different process.

The default will be to import PyQt4 without configuration of the APIs, thus matching what most applications would expect. It will fall back to PySide if PyQt4 is unavailable.

If specified, IPython will respect the environment variable `QT_API` used by ETS. ETS 4.0 also works with both PyQt4 and PySide, but it requires PyQt4 to use its v2 API. So if `QT_API=pyside` PySide will be used, and if `QT_API=pyqt` then PyQt4 will be used with the v2 API for QString and QVariant, so ETS codes like MayaVi will also work with IPython.

If you launch IPython in matplotlib mode with `ipython --matplotlib=qt`, then IPython will ask matplotlib which Qt library to use (only if `QT_API` is not set), via the ‘backend.qt4’ rcParam. If matplotlib is version 1.0.1 or older, then IPython will always use PyQt4 without setting the v2 APIs, since neither v2 PyQt nor PySide work.

**Warning:** Note that this means for ETS 4 to work with PyQt4, `QT_API must be set to work with IPython’s qt integration, because otherwise PyQt4 will be loaded in an incompatible mode.

It also means that you must not have `QT_API` set if you want to use `--gui=qt` with code that requires PyQt4 API v1.

### 4.4.8 Plotting with matplotlib

`matplotlib` provides high quality 2D and 3D plotting for Python. `matplotlib` can produce plots on screen using a variety of GUI toolkits, including Tk, PyGTK, PyQt4 and wxPython. It also provides a number of commands useful for scientific computing, all with a syntax compatible with that of the popular Matlab program.

To start IPython with matplotlib support, use the `--matplotlib` switch. If IPython is already running, you can run the `%matplotlib` magic. If no arguments are given, IPython will automatically detect your choice of matplotlib backend. You can also request a specific backend with `%matplotlib backend`, where `backend` must be one of: ‘tk’, ‘qt’, ‘wx’, ‘gtk’, ‘osx’. In the web notebook and Qt console, ‘inline’ is also a valid backend value, which produces static figures inline inside the application window instead of matplotlib’s interactive figures that live in separate windows.
IPython Documentation, Release 4.2.1

4.4.9 Interactive demos with IPython
IPython ships with a basic system for running scripts interactively in sections, useful when presenting code
to audiences. A few tags embedded in comments (so that the script remains valid Python code) divide a
file into separate blocks, and the demo can be run one block at a time, with IPython printing (with syntax
highlighting) the block before executing it, and returning to the interactive prompt after each block. The
interactive namespace is updated after each block is run with the contents of the demo’s namespace.
This allows you to show a piece of code, run it and then execute interactively commands based on the
variables just created. Once you want to continue, you simply execute the next block of the demo. The
following listing shows the markup necessary for dividing a script into sections for execution as a demo:
# -*- coding: utf-8 -*"""A simple interactive demo to illustrate the use of IPython's Demo class.
Any python script can be run as a demo, but that does little more than showing
it on-screen, syntax-highlighted in one shot. If you add a little simple
markup, you can stop at specified intervals and return to the ipython prompt,
resuming execution later.
This is a unicode test, åäö
"""
from __future__ import print_function
print('Hello, welcome to an interactive IPython demo.')
print('Executing this block should require confirmation before proceeding,')
print('unless auto_all has been set to true in the demo object')
# The mark below defines a block boundary, which is a point where IPython will
# stop execution and return to the interactive prompt.
# <demo> --- stop --x = 1
y = 2
# <demo> --- stop --# the mark below makes this block as silent
# <demo> silent
print('This is a silent block, which gets executed but not printed.')
# <demo> --- stop --# <demo> auto
print('This is an automatic block.')
print('It is executed without asking for confirmation, but printed.')
z = x+y
print('z=',x)
# <demo> --- stop --# This is just another normal block.
print('z is now:', z)

4.4. IPython reference

337


print('bye!')

In order to run a file as a demo, you must first make a Demo object out of it. If the file is named myscript.py, the following code will make a demo:

```python
from IPython.lib.demo import Demo
mydemo = Demo('myscript.py')
```

This creates the mydemo object, whose blocks you run one at a time by simply calling the object with no arguments. Then call it to run each step of the demo:

```python
mydemo()
```

Demo objects can be restarted, you can move forward or back skipping blocks, re-execute the last block, etc. See the IPython.lib.demo module and the Demo class for details.

Limitations: These demos are limited to fairly simple uses. In particular, you cannot break up sections within indented code (loops, if statements, function definitions, etc.) Supporting something like this would basically require tracking the internal execution state of the Python interpreter, so only top-level divisions are allowed. If you want to be able to open an IPython instance at an arbitrary point in a program, you can use IPython’s embedding facilities.

### 4.5 IPython as a system shell

#### 4.5.1 Overview

It is possible to adapt IPython for system shell usage. In the past, IPython shipped a special ‘sh’ profile for this purpose, but it had been quarantined since 0.11 release, and in 1.0 it was removed altogether. Nevertheless, much of this section relies on machinery which does not require a custom profile.

You can set up your own ‘sh’ profile to be different from the default profile such that:

- Prompt shows the current directory (see Prompt customization)
- Make system commands directly available (in alias table) by running the %rehashx magic. If you install new programs along your PATH, you might want to run %rehashx to update the alias table
- turn %autocall to full mode

#### 4.5.2 Environment variables

Rather than manipulating os.environ directly, you may like to use the magic %env command. With no arguments, this displays all environment variables and values. To get the value of a specific variable, use %env var. To set the value of a specific variable, use %env foo=bar, %env foo=bar. By default values are considered to be strings so quoting them is unnecessary. However, Python variables are expanded as usual in the magic command, so %env foo=$bar means “set the environment variable foo to the value of the Python variable bar”.

338 Chapter 4. Using IPython for interactive work
4.5.3 Aliases

Once you run \%rehashx, all of your $PATH has been loaded as IPython aliases, so you should be able to type any normal system command and have it executed. See \%alias? and \%unalias? for details on the alias facilities. See also \%rehashx? for details on the mechanism used to load $PATH.

4.5.4 Directory management

Since each command passed by IPython to the underlying system is executed in a subshell which exits immediately, you can NOT use !cd to navigate the filesystem.

IPython provides its own builtin \%cd magic command to move in the filesystem (the % is not required with automagic on). It also maintains a list of visited directories (use %dhist to see it) and allows direct switching to any of them. Type cd? for more details.

\%pushd, \%popd and \%dirs are provided for directory stack handling.

4.5.5 Prompt customization

Here are some prompt configurations you can try out interactively by using the \%config magic:

```
%config PromptManager.in_template = r'{color.LightGreen}\u@\h{color.LightBlue}{color.LightCyan}\Y1{color.LightBlue}{color.Green}|#> '
%config PromptManager.in2_template = r'{color.Green}|{color.LightGreen}\D{color.Green}> '
%config PromptManager.out_template = r'<\#> '
```

You can change the prompt configuration to your liking permanently by editing ipython_config.py:

```
c.PromptManager.in_template = r'{color.LightGreen}\u@\h{color.LightBlue}{color.LightCyan}\Y1{color.LightBlue}{color.Green}|#> '
c.PromptManager.in2_template = r'{color.Green}|{color.LightGreen}\D{color.Green}> '
c.PromptManager.out_template = r'<\#> '
```

Read more about the configuration system for details on how to find ipython_config.py.

4.5.6 String lists

String lists (IPython.utils.text.SList) are handy way to process output from system commands. They are produced by var = !cmd syntax.

First, we acquire the output of ‘ls -l’:

```
[Q:doc/examples]|2> lines = !ls -l ==
['total 23',
'-rw-rw-r- 1 ville None 1163 Sep 30 2006 example-demo.py',
'-rw-rw-r- 1 ville None 1927 Sep 30 2006 example-embed-short.py',
'-rwxrwxrwx 1 ville None 4606 Sep 1 17:15 example-embed.py',
'-rwxrwxrwx 1 ville None 1017 Sep 30 2006 example-gnuplot.py',
'-rwxrwxrwx 1 ville None 339 Jun 11 18:01 extension.py',
'-rwxrwxrwx 1 ville None 113 Dec 20 2006 seteditor.py',
'-rwxrwxrwx 1 ville None 245 Dec 12 2006 seteditor.pyc']
```
Now, let’s take a look at the contents of ‘lines’ (the first number is the list element number):

```
[Q:doc/examples]|3> lines
   <3> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total 23
1: -rw-rw-rw- 1 ville None 1163 Sep 30 2006 example-demo.py
2: -rw-rw-rw- 1 ville None 1927 Sep 30 2006 example-embed-short.py
3: -rwxrwxrwxx 1 ville None 4606 Sep 1 17:15 example-embed.py
4: -rwxrwxrwxx 1 ville None 1017 Sep 30 2006 example-gnuplot.py
5: -rwxrwxrwxx 1 ville None 339 Jun 11 18:01 extension.py
6: -rwxrwxrwxx 1 ville None 113 Dec 20 2006 seteditor.py
7: -rwxrwxrwxx 1 ville None 245 Dec 12 2006 seteditor.pyc
```

Now, let’s filter out the ‘embed’ lines:

```
[Q:doc/examples]|4> l2 = lines.grep('embed', prune=1)
[Q:doc/examples]|5> l2
   <5> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total 23
1: -rw-rw-rw- 1 ville None 1163 Sep 30 2006 example-demo.py
2: -rwxrwxrwxx 1 ville None 4606 Sep 1 17:15 example-embed.py
3: -rwxrwxrwxx 1 ville None 1017 Sep 30 2006 example-gnuplot.py
4: -rwxrwxrwxx 1 ville None 339 Jun 11 18:01 extension.py
5: -rwxrwxrwxx 1 ville None 113 Dec 20 2006 seteditor.py
6: -rwxrwxrwxx 1 ville None 245 Dec 12 2006 seteditor.pyc
```

Now, we want strings having just file names and permissions:

```
[Q:doc/examples]|6> l2.fields(8,0)
   <6> SList (.p, .n, .l, .s, .grep(), .fields() available). Value:

0: total
1: example-demo.py -rw-rw-rw-
2: example-gnuplot.py -rwxrwxrwxx
3: extension.py -rwxrwxrwxx
4: seteditor.py -rwxrwxrwxx
5: seteditor.pyc -rwxrwxrwxx
```

Note how the line with ‘total’ does not raise IndexError.

If you want to split these (yielding lists), call fields() without arguments:

```
[Q:doc/examples]|7> _.fields()
   <7> [[total],
        [example-demo.py', '-rw-rw-rw-'],
        [example-gnuplot.py', '-rwxrwxrwxx'],
        [extension.py', '-rwxrwxrwxx'],
        [seteditor.py', '-rwxrwxrwxx'],
        [seteditor.pyc', '-rwxrwxrwxx']]
```

If you want to pass these separated with spaces to a command (typical for lists if files), use the .s property:
SLists are inherited from normal Python lists, so every list method is available:

```
[Q:doc/examples]|21> lines.append('hey')
```

### Real world example: remove all files outside version control

First, capture output of “hg status”:

```
[Q:/ipython]|28> out = !hg status
==
['M IPython\extensions\ipy_kitcfg.py',
'M IPython\extensions\ipy_rehashdir.py',
'...
'? build\lib\IPython\Debugger.py',
'? build\lib\IPython\extensions\InterpreterExec.py',
'? build\lib\IPython\extensions\InterpreterPasteInput.py',
'...
```

(lines starting with ? are not under version control).

```
[Q:/ipython]|35> junk = out.grep(r'^\?').fields(1)
[Q:/ipython]|36> junk
   <36> SList (.p, .n, .l, .s, .grep(), .fields() availab
```

Now we can just remove these files by doing `rm $junk.s`.

#### The .s, .n, .p properties

The .s property returns one string where lines are separated by single space (for convenient passing to system commands). The .n property return one string where the lines are separated by a newline (i.e. the original output of the function). If the items in string list are file names, .p can be used to get a list of “path” objects for convenient file manipulation.

### 4.6 IPython Tips & Tricks

The IPython cookbook details more things you can do with IPython.
4.6.1 Embed IPython in your programs

A few lines of code are enough to load a complete IPython inside your own programs, giving you the ability to work with your data interactively after automatic processing has been completed. See the embedding section.

4.6.2 Run doctests

Run your doctests from within IPython for development and debugging. The special %doctest_mode command toggles a mode where the prompt, output and exceptions display matches as closely as possible that of the default Python interpreter. In addition, this mode allows you to directly paste in code that contains leading ‘>>>’ prompts, even if they have extra leading whitespace (as is common in doctest files). This combined with the %history -t call to see your translated history allows for an easy doctest workflow, where you can go from doctest to interactive execution to pasting into valid Python code as needed.

4.6.3 Use IPython to present interactive demos

Use the IPython.lib.demo.Demo class to load any Python script as an interactive demo. With a minimal amount of simple markup, you can control the execution of the script, stopping as needed. See here for more.

4.6.4 Suppress output

Put a ‘;’ at the end of a line to suppress the printing of output. This is useful when doing calculations which generate long output you are not interested in seeing. It also keeps the object out of the output cache, so if you’re working with large temporary objects, they’ll be released from memory sooner.

4.6.5 Lightweight ‘version control’

When you call %edit with no arguments, IPython opens an empty editor with a temporary file, and it returns the contents of your editing session as a string variable. Thanks to IPython’s output caching mechanism, this is automatically stored:

```
In [1]: %edit

IPython will make a temporary file named: /tmp/ipython_edit_yR-HCN.py

Editing... done. Executing edited code...

hello - this is a temporary file

Out[1]: "print('hello - this is a temporary file')
"
```

Now, if you call %edit -p, IPython tries to open an editor with the same data as the last time you used %edit. So if you haven’t used %edit in the meantime, this same contents will reopen; however, it will be done in a new file. This means that if you make changes and you later want to find an old version, you
can always retrieve it by using its output number, via ‘%edit _NN’, where NN is the number of the output prompt.

Continuing with the example above, this should illustrate this idea:

In [2]: edit -p

IPython will make a temporary file named: /tmp/ipython_edit_nA09Qk.py

Editing... done. Executing edited code...

hello - now I made some changes

Out[2]: "print('hello - now I made some changes')
"

In [3]: edit _1

IPython will make a temporary file named: /tmp/ipython_edit_gy6-zD.py

Editing... done. Executing edited code...

hello - this is a temporary file

IPython version control at work :)

Out[3]: "print('hello - this is a temporary file')
print('IPython version control at work :')"

This section was written after a contribution by Alexander Belchenko on the IPython user list.

See also:

A Qt Console for Jupyter The Jupyter Notebook
5.1 Configuring IPython

5.1.1 Introduction to IPython configuration

Setting configurable options

Many of IPython’s classes have configurable attributes (see IPython options for the list). These can be configured in several ways.

Python config files

To create the blank config files, run:

```bash
ipython profile create [profilename]
```

If you leave out the profile name, the files will be created for the default profile (see Profiles). These will typically be located in ~/.ipython/profile_default/, and will be named ipython_config.py, ipython_notebook_config.py, etc. The settings in ipython_config.py apply to all IPython commands.

The files typically start by getting the root config object:

```python
c = get_config()
```

You can then configure class attributes like this:

```python
c.InteractiveShell.automagic = False
```

Be careful with spelling—incorrect names will simply be ignored, with no error.

To add to a collection which may have already been defined elsewhere, you can use methods like those found on lists, dicts and sets: append, extend, prepend() (like extend, but at the front), add and update (which works both for dicts and sets):

```python
c.InteractiveShellApp.extensions.append('Cython')
```
New in version 2.0: list, dict and set methods for config values

Example config file

```python
# sample ipython_config.py

c = get_config()

c.TerminalIPythonApp.display_banner = True
c.InteractiveShellApp.log_level = 20
c.InteractiveShellApp.extensions = ['myextension']
c.InteractiveShellApp.exec_lines = ['import numpy', 'import scipy']
c.InteractiveShellApp.exec_files = ['mycode.py', 'fancy.ipy']
c.InteractiveShell.autoindent = True
c.InteractiveShell.colors = 'LightBG'
c.InteractiveShell.confirm_exit = False
c.InteractiveShell.editor = 'nano'
c.InteractiveShell.xmode = 'Context'

c.PromptManager.in_template = 'In [\#]: '
c.PromptManager.in2_template = '.\D.: '
c.PromptManager.out_template = 'Out[\#]: '
c.PromptManager.justify = True

c.PrefilterManager.multi_line_specials = True

c.AliasManager.user_aliases = [('la', 'ls -al')]
```

Command line arguments

Every configurable value can be set from the command line, using this syntax:

```bash
ipython --ClassName.attribute=value
```

Many frequently used options have short aliases and flags, such as `--matplotlib` (to integrate with a matplotlib GUI event loop) or `--pdb` (automatic post-mortem debugging of exceptions).

To see all of these abbreviated options, run:

```bash
ipython --help
ipython notebook --help
# etc.
```

Options specified at the command line, in either format, override options set in a configuration file.
The config magic

You can also modify config from inside IPython, using a magic command:

```
%config IPCompleter.greedy = True
```

At present, this only affects the current session - changes you make to config are not saved anywhere. Also, some options are only read when IPython starts, so they can’t be changed like this.

Profiles

IPython can use multiple profiles, with separate configuration and history. By default, if you don’t specify a profile, IPython always runs in the default profile. To use a new profile:

```
ipython profile create foo  # create the profile foo
ipython --profile=foo       # start IPython using the new profile
```

Profiles are typically stored in *The IPython directory*, but you can also keep a profile in the current working directory, for example to distribute it with a project. To find a profile directory on the filesystem:

```
ipython locate profile foo
```

The IPython directory

IPython stores its files—config, command history and extensions—in the directory `~/.ipython/` by default.

**IPYTHONDIR**

If set, this environment variable should be the path to a directory, which IPython will use for user data. IPython will create it if it does not exist.

```
--ipython-dir=<path>
```

This command line option can also be used to override the default IPython directory.

To see where IPython is looking for the IPython directory, use the command `ipython locate`, or the Python function `IPython.paths.get_ipython_dir()`.

5.1.2 IPython options

Any of the options listed here can be set in config files, at the command line, or from inside IPython. See *Setting configurable options* for details.

Terminal IPython options

**InteractiveShellApp.code_to_run** [Unicode] Default: ‘’

Execute the given command string.
InteractiveShellApp.exec_PYTHONSTARTUP  [Bool] Default: True

Run the file referenced by the PYTHONSTARTUP environment variable at IPython startup.

InteractiveShellApp.exec_files  [List] Default: []

List of files to run at IPython startup.

InteractiveShellApp.exec_lines  [List] Default: []

lines of code to run at IPython startup.

InteractiveShellApp.extensions  [List] Default: []

A list of dotted module names of IPython extensions to load.

InteractiveShellApp.extra_extension  [Unicode] Default: ''

dotted module name of an IPython extension to load.

InteractiveShellApp.file_to_run  [Unicode] Default: ''

A file to be run

InteractiveShellApp.gui  ['glut'|'gtk'|'gtk3'|'osx'|'pyglet'|'qt'|'qt5'|'tk'|'wx'] Default: None

Enable GUI event loop integration with any of ('glut', 'gtk', 'gtk3', 'osx', 'pyglet', 'qt', 'qt5', 'tk', 'wx').

InteractiveShellApp.hide_initial_ns  [Bool] Default: True

Should variables loaded at startup (by startup files, exec_lines, etc.) be hidden from tools like %who?

InteractiveShellApp.matplotlib  ['auto'|'gtk'|'gtk3'|'inline'|'nbagg'|'notebook'|'osx'|'qt'|'qt4'|'qt5'|'tk'|'wx'] Default: None

Configure matplotlib for interactive use with the default matplotlib backend.

InteractiveShellApp.module_to_run  [Unicode] Default: ''

Run the module as a script.

InteractiveShellApp.pylab  ['auto'|'gtk'|'gtk3'|'inline'|'nbagg'|'notebook'|'osx'|'qt'|'qt4'|'qt5'|'tk'|'wx'] Default: None

Pre-load matplotlib and numpy for interactive use, selecting a particular matplotlib backend and loop integration.

InteractiveShellApp.pylab_import_all  [Bool] Default: True

If true, IPython will populate the user namespace with numpy, pylab, etc. and an import * is done from numpy and pylab, when using pylab mode.

When False, pylab mode should not import any names into the user namespace.

InteractiveShellApp.reraise_ipython_extension_failures  [Bool] Default: False

Reraise exceptions encountered loading IPython extensions?

Application.log_datefmt  [Unicode] Default: '%Y-%m-%d %H:%M:%S'

The date format used by logging formatters for %(asctime)s
**Application.log_format** [Unicode] Default: ‘%(name)s%(levelname)s %(message)s’

The Logging format template

**Application.log_level** [0|10|20|30|40|50|‘DEBUG’|‘INFO’|‘WARN’|‘ERROR’|‘CRITICAL’] Default: 30

Set the log level by value or name.

**BaseIPythonApplication.auto_create** [Bool] Default: False

Whether to create profile dir if it doesn’t exist

**BaseIPythonApplication.copy_config_files** [Bool] Default: False

Whether to install the default config files into the profile dir. If a new profile is being created, and IPython contains config files for that profile, then they will be staged into the new directory. Otherwise, default config files will be automatically generated.

**BaseIPythonApplication.extra_config_file** [Unicode] Default: ‘’

Path to an extra config file to load.

If specified, load this config file in addition to any other IPython config.

**BaseIPythonApplication.ipython_dir** [Unicode] Default: ‘’

The name of the IPython directory. This directory is used for logging configuration (through profiles), history storage, etc. The default is usually $HOME/.ipython. This option can also be specified through the environment variable IPYTHONDIR.

**BaseIPythonApplication.overwrite** [Bool] Default: False

Whether to overwrite existing config files when copying

**BaseIPythonApplication.profile** [Unicode] Default: ‘default’

The IPython profile to use.

**BaseIPythonApplication.verbose_crash** [Bool] Default: False

Create a massive crash report when IPython encounters what may be an internal error. The default is to append a short message to the usual traceback

**TerminalIPythonApp.display_banner** [Bool] Default: True

Whether to display a banner upon starting IPython.

**TerminalIPythonApp.force_interact** [Bool] Default: False

If a command or file is given via the command-line, e.g. ‘ipython foo.py’, start an interactive shell after executing the file or command.

**TerminalIPythonApp.quick** [Bool] Default: False

Start IPython quickly by skipping the loading of config files.

**InteractiveShell.ast_node_interactivity** [‘all’|‘last’|‘last_expr’|‘none’] Default: ‘last_expr’

‘all’, ‘last’, ‘last_expr’ or ‘none’, specifying which nodes should be run interactively (displaying output from expressions).

5.1. Configuring IPython
**InteractiveShell.ast_transformers** [List] Default: []

A list of ast.NodeTransformer subclass instances, which will be applied to user input before code is run.

**InteractiveShell.autocall** [0|1|2] Default: 0

Make IPython automatically call any callable object even if you didn’t type explicit parentheses. For example, ‘str 43’ becomes ‘str(43)’ automatically. The value can be ‘0’ to disable the feature, ‘1’ for ‘smart’ autocall, where it is not applied if there are no more arguments on the line, and ‘2’ for ‘full’ autocall, where all callable objects are automatically called (even if no arguments are present).

**InteractiveShell.autoindent** [CBool] Default: True

Autoindent IPython code entered interactively.

**InteractiveShell.automagic** [CBool] Default: True

Enable magic commands to be called without the leading %.

**InteractiveShell.banner1** [Unicode] Default: ‘Python 3.4.3 (default, Oct 14 2015, 20:28:29) \nType "copyright...

The part of the banner to be printed before the profile

**InteractiveShell.banner2** [Unicode] Default: ‘’

The part of the banner to be printed after the profile

**InteractiveShell.cache_size** [Int] Default: 1000

Set the size of the output cache. The default is 1000, you can change it permanently in your config file. Setting it to 0 completely disables the caching system, and the minimum value accepted is 20 (if you provide a value less than 20, it is reset to 0 and a warning is issued). This limit is defined because otherwise you’ll spend more time re-flushing a too small cache than working

**InteractiveShell.color_info** [CBool] Default: True

Use colors for displaying information about objects. Because this information is passed through a pager (like ‘less’), and some pagers get confused with color codes, this capability can be turned off.

**InteractiveShell.colors** [‘NoColor’|’LightBG’|’Linux’] Default: ‘Linux’

Set the color scheme (NoColor, Linux, or LightBG).

**InteractiveShell.debug** [CBool] Default: False

No description

**InteractiveShell.deep_reload** [CBool] Default: False

Deprecated

Will be removed in IPython 6.0

Enable deep (recursive) reloading by default. IPython can use the deep_reload module which reloads changes in modules recursively (it replaces the reload() function, so you don’t need to change anything to use it). deep_reload forces a full reload of modules whose code may have changed, which the
default reload() function does not. When deep_reload is off, IPython will use the normal reload(), but
deep_reload will still be available as dreload().

**InteractiveShell.disable_failing_post_execute** [CBool] Default: False

Don’t call post-execute functions that have failed in the past.

**InteractiveShell.display_page** [Bool] Default: False

If True, anything that would be passed to the pager will be displayed as regular output instead.

**InteractiveShell.history_length** [Int] Default: 10000

No description

**InteractiveShell.history_load_length** [Int] Default: 1000

The number of saved history entries to be loaded into the readline buffer at startup.

**InteractiveShell.ipython_dir** [Unicode] Default: ‘’

No description

**InteractiveShell.logappend** [Unicode] Default: ‘’

Start logging to the given file in append mode. Use logfile to specify a log file to overwrite logs to.

**InteractiveShelllogfile** [Unicode] Default: ‘’

The name of the logfile to use.

**InteractiveShell.logstart** [CBool] Default: False

Start logging to the default log file in overwrite mode. Use logappend to specify a log file to append logs to.

**InteractiveShell.multiline_history** [CBool] Default: True

Save multi-line entries as one entry in readline history

**InteractiveShell.object_info_string_level** [0|1|2] Default: 0

No description

**InteractiveShell.pdb** [CBool] Default: False

Automatically call the pdb debugger after every exception.

**InteractiveShell.prompt_in1** [Unicode] Default: ‘In [\#]: ’

Deprecated, will be removed in IPython 5.0, use PromptManager.in_template

**InteractiveShell.prompt_in2** [Unicode] Default: ‘ \D.: ’

Deprecated, will be removed in IPython 5.0, use PromptManager.in2_template

**InteractiveShell.prompt_out** [Unicode] Default: ‘Out[\#]: ’

Deprecated, will be removed in IPython 5.0, use PromptManager.out_template

---

5.1. Configuring IPython
InteractiveShell.prompts_pad_left [CBool] Default: True

Deprecated, will be removed in IPython 5.0, use PromptManager.justify

InteractiveShell.quiet [CBool] Default: False

No description

InteractiveShell.readline_parse_and_bind [List] Default: ['tab: complete', '"\C-l":
clear-screen', 'set show-all-if-a..."

No description

InteractiveShell.readline_remove_delims [Unicode] Default: '−/~'

No description

InteractiveShell.readline_use [CBool] Default: True

No description

InteractiveShell.separate_in [SeparateUnicode] Default: '\n'

No description

InteractiveShell.separate_out [SeparateUnicode] Default: ''

No description

InteractiveShell.separate_out2 [SeparateUnicode] Default: ''

No description

InteractiveShell.show_rewritten_input [CBool] Default: True

Show rewritten input, e.g. for autocall.

InteractiveShell.wildcards_case_sensitive [CBool] Default: True

No description

InteractiveShell.xmode ['Context'|'Plain'|'Verbose'] Default: 'Context'

No description

TerminalInteractiveShell.autoedit_syntax [CBool] Default: False

auto editing of files with syntax errors.

TerminalInteractiveShell.confirm_exit [CBool] Default: True

Set to confirm when you try to exit IPython with an EOF (Control-D in Unix, Control-Z/Enter in Windows). By typing ‘exit’ or ‘quit’, you can force a direct exit without any confirmation.

TerminalInteractiveShell.editor [Unicode] Default: 'vi'

Set the editor used by IPython (default to $EDITOR/vi/notepad).

TerminalInteractiveShell.pager [Unicode] Default: 'less'

The shell program to be used for paging.
TerminalInteractiveShell.screen_length  [Int] Default: 0

Number of lines of your screen, used to control printing of very long strings. Strings longer than this
number of lines will be sent through a pager instead of directly printed. The default value for this is 0,
which means IPython will auto-detect your screen size every time it needs to print certain potentially
long strings (this doesn’t change the behavior of the ‘print’ keyword, it’s only triggered internally). If
for some reason this isn’t working well (it needs curses support), specify it yourself. Otherwise don’t
change the default.

TerminalInteractiveShell.term_title  [CBool] Default: False

Enable auto setting the terminal title.

PromptManager.color_scheme  [Unicode] Default: ‘Linux’

No description

PromptManager.in2_template  [Unicode] Default: ‘ .\\D.: ’

Continuation prompt.

PromptManager.in_template  [Unicode] Default: ‘In [\\#]: ’

Input prompt. ‘#’ will be transformed to the prompt number

PromptManager.justify  [Bool] Default: True

If True (default), each prompt will be right-aligned with the preceding one.

PromptManager.out_template  [Unicode] Default: ‘Out[\\#]: ’

Output prompt. ‘#’ will be transformed to the prompt number

HistoryAccessor.connection_options  [Dict] Default: {}

Options for configuring the SQLite connection

These options are passed as keyword args to sqlite3.connect when establishing database conenctions.

HistoryAccessor.enabled  [Bool] Default: True

enable the SQLite history

set enabled=False to disable the SQLite history, in which case there will be no stored history, no
SQLite connection, and no background saving thread. This may be necessary in some threaded envi-
ronments where IPython is embedded.

HistoryAccessor.hist_file  [Unicode] Default: ‘’

Path to file to use for SQLite history database.

By default, IPython will put the history database in the IPython profile directory. If you would rather
share one history among profiles, you can set this value in each, so that they are consistent.

Due to an issue with fcntl, SQLite is known to misbehave on some NFS mounts. If you see IPython
hanging, try setting this to something on a local disk, e.g:

```bash
ipython --HistoryManager.hist_file=/tmp/ipython_hist.sqlite
```
HistoryManager.db_cache_size  [Int] Default: 0
    Write to database every x commands (higher values save disk access & power). Values of 1 or less
effectively disable caching.

HistoryManager.db_log_output  [Bool] Default: False
    Should the history database include output? (default: no)

ProfileDir.location  [Unicode] Default: ‘’
    Set the profile location directly. This overrides the logic used by the profile option.

BaseFormatter.deferred_printers  [Dict] Default: { }
    No description

BaseFormatter.enabled  [Bool] Default: True
    No description

BaseFormatter.singleton_printers  [Dict] Default: { }
    No description

BaseFormatter.type_printers  [Dict] Default: { }
    No description

PlainTextFormatter.float_precision  [CUnicode] Default: ‘’
    No description

PlainTextFormatter.max_seq_length  [Int] Default: 1000
    Truncate large collections (lists, dicts, tuples, sets) to this size.
    Set to 0 to disable truncation.

PlainTextFormatter.max_width  [Int] Default: 79
    No description

PlainTextFormatter.newline  [Unicode] Default: ‘\n’
    No description

PlainTextFormatter.pprint  [Bool] Default: True
    No description

PlainTextFormatter.verbose  [Bool] Default: False
    No description

Completer.greedy  [CBool] Default: False
    Activate greedy completion
    This will enable completion on elements of lists, results of function calls, etc., but can be unsafe
because the code is actually evaluated on TAB.
**IPCompleter.limit_to__all__** [CBool] Default: False

Instruct the completer to use __all__ for the completion
Specifically, when completing on object.<tab>.
When True: only those names in obj.__all__ will be included.
When False [default]: the __all__ attribute is ignored

**IPCompleter.merge_completions** [CBool] Default: True

Whether to merge completion results into a single list
If False, only the completion results from the first non-empty completer will be returned.

**IPCompleter.omit__names** [0|1|2] Default: 2

Instruct the completer to omit private method names
Specifically, when completing on object.<tab>.
When 2 [default]: all names that start with ‘_’ will be excluded.
When 1: all ‘magic’ names (__foo__) will be excluded.
When 0: nothing will be excluded.

**ScriptMagics.script_magics** [List] Default: []

Extra script cell magics to define
This generates simple wrappers of %script foo as %foo.
If you want to add script magics that aren’t on your path, specify them in script_paths

**ScriptMagics.script_paths** [Dict] Default: {}

Dict mapping short ‘ruby’ names to full paths, such as ‘/opt/secret/bin/ruby’
Only necessary for items in script_magics where the default path will not find the right interpreter.

**StoreMagics.autorestore** [Bool] Default: False

If True, any %store-d variables will be automatically restored when IPython starts.

**IPython kernel options**

These options can be used in ipython_kernel_config.py. The kernel also respects any options in ipython_config.py

**ConnectionFileMixin.connection_file** [Unicode] Default: ‘’

JSON file in which to store connection info [default: kernel-<pid>.json]
This file will contain the IP, ports, and authentication key needed to connect clients to this kernel.
By default, this file will be created in the security dir of the current profile, but can be specified by absolute path.
ConnectionFileMixin.control_port [Int] Default: 0
set the control (ROUTER) port [default: random]

ConnectionFileMixin.hb_port [Int] Default: 0
set the heartbeat port [default: random]

ConnectionFileMixin.iopub_port [Int] Default: 0
set the iopub (PUB) port [default: random]

ConnectionFileMixin.ip [Unicode] Default: '
Set the kernel’s IP address [default localhost]. If the IP address is something other than localhost, then Consoles on other machines will be able to connect to the Kernel, so be careful!

ConnectionFileMixin.shell_port [Int] Default: 0
set the shell (ROUTER) port [default: random]

ConnectionFileMixin.stdin_port [Int] Default: 0
set the stdin (ROUTER) port [default: random]

ConnectionFileMixin.transport [‘tcp’|’ipc’] Default: ‘tcp’
No description

InteractiveShellApp.code_to_run [Unicode] Default: '
Execute the given command string.

InteractiveShellApp.exec_PYTHONSTARTUP [Bool] Default: True
Run the file referenced by the PYTHONSTARTUP environment variable at IPython startup.

InteractiveShellApp.exec_files [List] Default: []
List of files to run at IPython startup.

InteractiveShellApp.exec_lines [List] Default: []
lines of code to run at IPython startup.

InteractiveShellApp.extensions [List] Default: []
A list of dotted module names of IPython extensions to load.

InteractiveShellApp.extra_extension [Unicode] Default: '
dotted module name of an IPython extension to load.

InteractiveShellApp.file_to_run [Unicode] Default: '
A file to be run

InteractiveShellApp.gui [‘glut’|’gtk’|’gtk3’|’osx’|’pyglet’|’qt’|’qt5’|’tk’|’wx’] Default: None
**InteractiveShellApp.hide_initial_ns** [Bool] Default: True

Should variables loaded at startup (by startup files, exec_lines, etc.) be hidden from tools like %who?

**InteractiveShellApp.matplotlib** [‘auto’|’gtk’|’gtk3’|’inline’|’nbagg’|’notebook’|’osx’|’qt’|’qt4’|’qt5’|’tk’|’wx’]
Default: None

Configure matplotlib for interactive use with the default matplotlib backend.

**InteractiveShellApp.module_to_run** [Unicode] Default: ''

Run the module as a script.

**InteractiveShellApp.pylab** [‘auto’|’gtk’|’gtk3’|’inline’|’nbagg’|’notebook’|’osx’|’qt’|’qt4’|’qt5’|’tk’|’wx’]
Default: None

Pre-load matplotlib and numpy for interactive use, selecting a particular matplotlib backend and loop integration.

**InteractiveShellApp.pylab_import_all** [Bool] Default: True

If true, IPython will populate the user namespace with numpy, pylab, etc. and an import * is done from numpy and pylab, when using pylab mode.

When False, pylab mode should not import any names into the user namespace.

**InteractiveShellApp.reraise_ipython_extension_failures** [Bool] Default: False

Reraise exceptions encountered loading IPython extensions?

**Application.log_datefmt** [Unicode] Default: '%Y-%m-%d %H:%M:%S'

The date format used by logging formatters for %(asctime)s

**Application.log_format** [Unicode] Default: '%(name)s|%(levelname)s %(message)s'

The Logging format template

**Application.log_level** [0|10|20|30|40|50|'DEBUG'|'INFO'|'WARNING'|'ERROR'|'CRITICAL'] Default: 30

Set the log level by value or name.

**BasePythonApplication.auto_create** [Bool] Default: False

Whether to create profile dir if it doesn’t exist

**BasePythonApplication.copy_config_files** [Bool] Default: False

Whether to install the default config files into the profile dir. If a new profile is being created, and IPython contains config files for that profile, then they will be staged into the new directory. Otherwise, default config files will be automatically generated.

**BasePythonApplication.extra_config_file** [Unicode] Default: ''

Path to an extra config file to load.

If specified, load this config file in addition to any other IPython config.

**BasePythonApplication.ipython_dir** [Unicode] Default: ''

5.1. Configuring IPython
The name of the IPython directory. This directory is used for logging configuration (through profiles),
history storage, etc. The default is usually $HOME/.ipython. This option can also be specified through
the environment variable IPYTHONDIR.

**BaseIPythonApplication.overwrite** [Bool] Default: False

Whether to overwrite existing config files when copying

**BaseIPythonApplication.profile** [Unicode] Default: 'default'

The IPython profile to use.

**BaseIPythonApplication.verbose_crash** [Bool] Default: False

Create a massive crash report when IPython encounters what may be an internal error. The default is
to append a short message to the usual traceback

**IPKernelApp.displayhook_class** [DottedObjectName] Default: 'ipykernel.displayhook.ZMQDisplayHook'

The importstring for the DisplayHook factory

**IPKernelApp.interrupt** [Int] Default: 0

ONLY USED ON WINDOWS Interrupt this process when the parent is signaled.

**IPKernelApp.kernel_class** [Type] Default: 'ipykernel.ipkernel.IPythonKernel'

The Kernel subclass to be used.

This should allow easy re-use of the IPKernelApp entry point to configure and launch kernels other
than IPython’s own.

**IPKernelApp.no_stderr** [Bool] Default: False

redirect stderr to the null device

**IPKernelApp.no_stdout** [Bool] Default: False

redirect stdout to the null device

**IPKernelApp.outstream_class** [DottedObjectName] Default: 'ipykernel.iostream.OutStream'

The importstring for the OutStream factory

**IPKernelApp.parent_handle** [Int] Default: 0

kill this process if its parent dies. On Windows, the argument specifies the HANDLE of the parent
process, otherwise it is simply boolean.

**Kernel._darwin_app_nap** [Bool] Default: True

Whether to use appnope for compatiblity with OS X App Nap.

Only affects OS X >= 10.9.

**Kernel._execute_sleep** [Float] Default: 0.0005

No description

**Kernel._poll_interval** [Float] Default: 0.05

No description
InteractiveShell.ast_node_interactivity ['all'|'last'|'last_expr'|'none'] Default: 'last_expr'

'all', 'last', 'last_expr' or 'none', specifying which nodes should be run interactively (displaying output from expressions).

InteractiveShell.ast_transformers [List] Default: []

A list of ast.NodeTransformer subclass instances, which will be applied to user input before code is run.

InteractiveShell.autocall [0|1|2] Default: 0

Make IPython automatically call any callable object even if you didn’t type explicit parentheses. For example, 'str 43' becomes 'str(43)' automatically. The value can be '0' to disable the feature, '1' for 'smart' autocall, where it is not applied if there are no more arguments on the line, and '2' for 'full' autocall, where all callable objects are automatically called (even if no arguments are present).

InteractiveShell.autoindent [CBool] Default: True

Autoindent IPython code entered interactively.

InteractiveShell.automagic [CBool] Default: True

Enable magic commands to be called without the leading %.

InteractiveShell.banner1 [Unicode] Default: 'Python 3.4.3 (default, Oct 14 2015, 20:28:29) \nType "copyri...

The part of the banner to be printed before the profile

InteractiveShell.banner2 [Unicode] Default: ''

The part of the banner to be printed after the profile

InteractiveShell.cache_size [Int] Default: 1000

Set the size of the output cache. The default is 1000, you can change it permanently in your config file. Setting it to 0 completely disables the caching system, and the minimum value accepted is 20 (if you provide a value less than 20, it is reset to 0 and a warning is issued). This limit is defined because otherwise you’ll spend more time re-flushing a too small cache than working

InteractiveShell.color_info [CBool] Default: True

Use colors for displaying information about objects. Because this information is passed through a pager (like ‘less’), and some pagers get confused with color codes, this capability can be turned off.

InteractiveShell.colors ['NoColor'|'LightBG'|'Linux'] Default: 'Linux'

Set the color scheme (NoColor, Linux, or LightBG).

InteractiveShell.debug [CBool] Default: False

No description

InteractiveShell.deep_reload [CBool] Default: False

Deprecated

Will be removed in IPython 6.0
Enable deep (recursive) reloading by default. IPython can use the deep_reload module which reloads changes in modules recursively (it replaces the reload() function, so you don’t need to change anything to use it). deep_reload forces a full reload of modules whose code may have changed, which the default reload() function does not. When deep_reload is off, IPython will use the normal reload(), but deep_reload will still be available as dreload().

**InteractiveShell.disable_failing_post_execute** [CBool] Default: False

Don’t call post-execute functions that have failed in the past.

**InteractiveShell.display_page** [Bool] Default: False

If True, anything that would be passed to the pager will be displayed as regular output instead.

**InteractiveShell.history_length** [Int] Default: 10000

No description

**InteractiveShell.history_load_length** [Int] Default: 1000

The number of saved history entries to be loaded into the readline buffer at startup.

**InteractiveShell.ipython_dir** [Unicode] Default: '

No description

**InteractiveShell.logappend** [Unicode] Default: '

Start logging to the given file in append mode. Use logfile to specify a log file to overwrite logs to.

**InteractiveShelllogfile** [Unicode] Default: '

The name of the logfile to use.

**InteractiveShell.logstart** [CBool] Default: False

Start logging to the default log file in overwrite mode. Use logappend to specify a log file to append logs to.

**InteractiveShell.multiline_history** [CBool] Default: True

Save multi-line entries as one entry in readline history

**InteractiveShell.object_info_string_level** [0|1|2] Default: 0

No description

**InteractiveShell.pdb** [CBool] Default: False

Automatically call the pdb debugger after every exception.

**InteractiveShell.prompt_in1** [Unicode] Default: 'In [\#]: '

Deprecated, will be removed in IPython 5.0, use PromptManager.in_template

**InteractiveShell.prompt_in2** [Unicode] Default: '.\D.: '

Deprecated, will be removed in IPython 5.0, use PromptManager.in2_template
InteractiveShell.prompt_out [Unicode] Default: 'Out[\#]: '  
Deprecated, will be removed in IPython 5.0, use PromptManager.out_template

InteractiveShell.prompts_pad_left [CBool] Default: True  
Deprecated, will be removed in IPython 5.0, use PromptManager.justify

InteractiveShell.quiet [CBool] Default: False  
No description

InteractiveShell.readline_parse_and_bind [List] Default: ['tab: complete', '\"\C-l": clear-screen', 'set show-all-if-a...  
No description

InteractiveShell.readline_remove_delims [Unicode] Default: '\-/~'  
No description

InteractiveShell.readline_use [CBool] Default: True  
No description

InteractiveShell.separate_in [SeparateUnicode] Default: '\n'  
No description

InteractiveShell.separate_out [SeparateUnicode] Default: ''  
No description

InteractiveShell.separate_out2 [SeparateUnicode] Default: ''  
No description

InteractiveShell.show_rewritten_input [CBool] Default: True  
Show rewritten input, e.g. for autocall.

InteractiveShell.wildcards_case_sensitive [CBool] Default: True  
No description

InteractiveShell.xmode ['Context'|'Plain'|'Verbose'] Default: 'Context'  
No description

ProfileDir.location [Unicode] Default: ''  
Set the profile location directly. This overrides the logic used by the profile option.

Session.buffer_threshold [Int] Default: 1024  
Threshold (in bytes) beyond which an object’s buffer should be extracted to avoid pickling.

Session.check_pid [Bool] Default: True  
Whether to check PID to protect against calls after fork.  
This check can be disabled if fork-safety is handled elsewhere.
Session.copy_threshold  [Int] Default: 65536

Threshold (in bytes) beyond which a buffer should be sent without copying.

Session.debug  [Bool] Default: False

Debug output in the Session

Session.digest_history_size  [Int] Default: 65536

The maximum number of digests to remember.

The digest history will be culled when it exceeds this value.

Session.item_threshold  [Int] Default: 64

The maximum number of items for a container to be introspected for custom serialization. Containers larger than this are pickled outright.

Session.key  [CBytes] Default: b''

execution key, for signing messages.

Session.keyfile  [Unicode] Default: ''

path to file containing execution key.

Session.metadata  [Dict] Default: {}

Metadata dictionary, which serves as the default top-level metadata dict for each message.

Session.packer  [DottedObjectName] Default: 'json'

The name of the packer for serializing messages. Should be one of 'json', 'pickle', or an import name for a custom callable serializer.

Session.session  [CUnicode] Default: ''

The UUID identifying this session.

Session.signature_scheme  [Unicode] Default: 'hmac-sha256'

The digest scheme used to construct the message signatures. Must have the form 'hmac-HASH'.

Session.unpacker  [DottedObjectName] Default: 'json'

The name of the unpacker for unserializing messages. Only used with custom functions for packer.

Session.username  [Unicode] Default: 'username'

Username for the Session. Default is your system username.

5.1.3 Specific config details

Prompts

In the terminal, the format of the input and output prompts can be customised. This does not currently affect other frontends.
The following codes in the prompt string will be substituted into the prompt string:

<table>
<thead>
<tr>
<th>Short</th>
<th>Long</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%n,#</td>
<td>{color.number}{count}{color.prompt}</td>
<td>history counter with bolding</td>
</tr>
<tr>
<td>\N</td>
<td>{count}</td>
<td>history counter without bolding</td>
</tr>
<tr>
<td>\D</td>
<td>{dots}</td>
<td>series of dots the same width as the history counter</td>
</tr>
<tr>
<td>\T</td>
<td>{time}</td>
<td>current time</td>
</tr>
<tr>
<td>\w</td>
<td>{cwd}</td>
<td>current working directory</td>
</tr>
<tr>
<td>\W</td>
<td>{cwd_last}</td>
<td>basename of CWD</td>
</tr>
<tr>
<td>\Xn</td>
<td>{cwd_x[n]}</td>
<td>Show the last n terms of the CWD. n=0 means show all.</td>
</tr>
<tr>
<td>\Yn</td>
<td>{cwd_y[n]}</td>
<td>Like Xn, but show ‘~’ for $HOME</td>
</tr>
<tr>
<td>\h</td>
<td>hostname, up to the first ‘.’</td>
<td></td>
</tr>
<tr>
<td>\H</td>
<td>full hostname</td>
<td></td>
</tr>
<tr>
<td>\u</td>
<td>username (from the $USER environment variable)</td>
<td></td>
</tr>
<tr>
<td>\v</td>
<td>IPython version</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>root symbol (“$” for normal user or “#” for root)</td>
<td></td>
</tr>
<tr>
<td>\</td>
<td>escaped ‘\’</td>
<td></td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
<td></td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
<td></td>
</tr>
<tr>
<td>n/a</td>
<td>{color.&lt;Name&gt;}</td>
<td>set terminal colour - see below for list of names</td>
</tr>
</tbody>
</table>

Available colour names are: Black, BlinkBlack, BlinkBlue, BlinkCyan, BlinkGreen, BlinkLightGray, BlinkPurple, BlinkRed, BlinkYellow, Blue, Brown, Cyan, DarkGray, Green, LightBlue, LightCyan, LightGray, LightGreen, LightPurple, LightRed, Purple, Red, White, Yellow. The selected colour scheme also defines the names prompt and number. Finally, the name normal resets the terminal to its default colour.

So, this config:

```
c.PromptManager.in_template = "{color.LightGreen}{time}{color.Yellow} \u{color.normal}>>>"
```

will produce input prompts with the time in light green, your username in yellow, and a >>> prompt in the default terminal colour.

### Terminal Colors

The default IPython configuration has most bells and whistles turned on (they’re pretty safe). But there’s one that may cause problems on some systems: the use of color on screen for displaying information. This is very useful, since IPython can show prompts and exception tracebacks with various colors, display syntax-highlighted source code, and in general make it easier to visually parse information.

The following terminals seem to handle the color sequences fine:

- Linux main text console, KDE Konsole, Gnome Terminal, E-term, rxvt, xterm.
- CDE terminal (tested under Solaris). This one boldfaces light colors.
- (X)Emacs buffers. See the (X)Emacs section for more details on using IPython with (X)Emacs.
- A Windows (XP/2k) command prompt with pyreadline.
• A Windows (XP/2k) CygWin shell. Although some users have reported problems; it is not clear whether there is an issue for everyone or only under specific configurations. If you have full color support under cygwin, please post to the IPython mailing list so this issue can be resolved for all users.

These have shown problems:

• Windows command prompt in WinXP/2k logged into a Linux machine via telnet or ssh.

• Windows native command prompt in WinXP/2k, without Gary Bishop’s extensions. Once Gary’s readline library is installed, the normal WinXP/2k command prompt works perfectly.

Currently the following color schemes are available:

• NoColor: uses no color escapes at all (all escapes are empty ‘ ’ strings). This ‘scheme’ is thus fully safe to use in any terminal.

• Linux: works well in Linux console type environments: dark background with light fonts. It uses bright colors for information, so it is difficult to read if you have a light colored background.

• LightBG: the basic colors are similar to those in the Linux scheme but darker. It is easy to read in terminals with light backgrounds.

IPython uses colors for two main groups of things: prompts and tracebacks which are directly printed to the terminal, and the object introspection system which passes large sets of data through a pager.

If you are seeing garbage sequences in your terminal and no colour, you may need to disable colours: run %colors NoColor inside IPython, or add this to a config file:

```python
c.InteractiveShell.colors = 'NoColor'
```

**Colors in the pager**

On some systems, the default pager has problems with ANSI colour codes. To configure your default pager to allow these:

1. Set the environment PAGER variable to `less`.

2. Set the environment LESS variable to `-r` (plus any other options you always want to pass to less by default). This tells less to properly interpret control sequences, which is how color information is given to your terminal.

**Editor configuration**

IPython can integrate with text editors in a number of different ways:

• Editors (such as (X)Emacs, vim and TextMate) can send code to IPython for execution.

• IPython’s `%edit` magic command can open an editor of choice to edit a code block.

The `%edit` command (and its alias `%ed`) will invoke the editor set in your environment as `EDITOR`. If this variable is not set, it will default to vi under Linux/Unix and to notepad under Windows. You may want to set this variable properly and to a lightweight editor which doesn’t take too long to start (that is, something
other than a new instance of Emacs). This way you can edit multi-line code quickly and with the power of a real editor right inside IPython.

You can also control the editor by setting `TerminalInteractiveShell.editor` in `ipython_config.py`.

**Vim**

Paul Ivanov’s [vim-ipython](https://github.com/ivp/vim-ipython) provides powerful IPython integration for vim.

**(X)Emacs**

If you are a dedicated Emacs user, and want to use Emacs when IPython’s `%edit` magic command is called you should set up the Emacs server so that new requests are handled by the original process. This means that almost no time is spent in handling the request (assuming an Emacs process is already running). For this to work, you need to set your EDITOR environment variable to `emacsclient`. The code below, supplied by Francois Pinard, can then be used in your `.emacs` file to enable the server:

```lisp
(defun fp-kill-server-with-buffer-routine ()
  (and server-buffer-clients (server-done)))
(add-hook 'kill-buffer-hook 'fp-kill-server-with-buffer-routine))
```

Thanks to the work of Alexander Schmolck and Prabhu Ramachandran, currently (X)Emacs and IPython get along very well in other ways.

With (X)EMacs >= 24, You can enable IPython in python-mode with:

```lisp
(require 'python)
(setq python-shell-interpreter "ipython")
```

See also:

- [Overview of the IPython configuration system](#)
- [Technical details of the config system](#)

## 5.2 Extending and integrating with IPython

### 5.2.1 IPython extensions

A level above configuration are IPython extensions, Python modules which modify the behaviour of the shell. They are referred to by an importable module name, and can be placed anywhere you’d normally import from, or in `.ipython/extensions/`.
Getting extensions

A few important extensions are bundled with IPython. Others can be found on the extensions index on the wiki, and the Framework :: IPython tag on PyPI.

Extensions on PyPI can be installed using pip, like any other Python package.

Using extensions

To load an extension while IPython is running, use the %load_ext magic:

```python
In [1]: %load_ext myextension
```

To load it each time IPython starts, list it in your configuration file:

```python
c.InteractiveShellApp.extensions = [ 'myextension'
]
```

Writing extensions

An IPython extension is an importable Python module that has a couple of special functions to load and unload it. Here is a template:

```python
# myextension.py

def load_ipython_extension(ipython):
    # The `ipython` argument is the currently active `InteractiveShell`
    # instance, which can be used in any way. This allows you to register
    # new magics or aliases, for example.

def unload_ipython_extension(ipython):
    # If you want your extension to be unloadable, put that logic here.
```

This load_ipython_extension() function is called after your extension is imported, and the currently active InteractiveShell instance is passed as the only argument. You can do anything you want with IPython at that point.

load_ipython_extension() will not be called again if the user use %load_extension. The user have to explicitly ask the extension to be reloaded (with %reload_extension). In case where the use ask the extension to be reloaded, the extension will be unloaded (with unload_ipython_extension), and loaded again.

Useful InteractiveShell methods include register_magic_function(), push() (to add variables to the user namespace) and drop_by_id() (to remove variables on unloading).

See also:

Defining custom magics
You can put your extension modules anywhere you want, as long as they can be imported by Python’s standard import mechanism. However, to make it easy to write extensions, you can also put your extensions in `extensions/` within the IPython directory. This directory is added to `sys.path` automatically.

When your extension is ready for general use, please add it to the extensions index. We also encourage you to upload it to PyPI and use the Framework :: IPython classifier, so that users can install it with standard packaging tools.

**Extensions bundled with IPython**

**autoreload**

IPython extension to reload modules before executing user code.

autoreload reloads modules automatically before entering the execution of code typed at the IPython prompt.

This makes for example the following workflow possible:

```
In [1]: %load_ext autoreload

In [2]: %autoreload 2

In [3]: from foo import some_function

In [4]: some_function()
Out[4]: 42

In [5]: # open foo.py in an editor and change some_function to return 43

In [6]: some_function()
Out[6]: 43
```

The module was reloaded without reloading it explicitly, and the object imported with `from foo import ...` was also updated.

**Usage** The following magic commands are provided:

%autoreload

Reload all modules (except those excluded by `%aimport`) automatically now.

%autoreload 0

Disable automatic reloading.

%autoreload 1

Reload all modules imported with `%aimport` every time before executing the Python code typed.

%autoreload 2
Reload all modules (except those excluded by `%aimport`) every time before executing the Python code typed.

`%aimport`

List modules which are to be automatically imported or not to be imported.

`%aimport foo`

Import module ‘foo’ and mark it to be autoreloaded for `%autoreload 1`

`%aimport -foo`

Mark module ‘foo’ to not be autoreloaded.

**Caveats**  Reloading Python modules in a reliable way is in general difficult, and unexpected things may occur. `%autoreload` tries to work around common pitfalls by replacing function code objects and parts of classes previously in the module with new versions. This makes the following things to work:

- Functions and classes imported via ‘from xxx import foo’ are upgraded to new versions when ‘xxx’ is reloaded.
- Methods and properties of classes are upgraded on reload, so that calling ‘c.foo()’ on an object ‘c’ created before the reload causes the new code for ‘foo’ to be executed.

Some of the known remaining caveats are:

- Replacing code objects does not always succeed: changing a @property in a class to an ordinary method or a method to a member variable can cause problems (but in old objects only).
- Functions that are removed (eg. via monkey-patching) from a module before it is reloaded are not upgraded.
- C extension modules cannot be reloaded, and so cannot be autoreloaded.

**storemagic**

`%store magic` for lightweight persistence.

Stores variables, aliases and macros in IPython’s database.

To automatically restore stored variables at startup, add this to your `ipython_config.py` file:

```
c.StoreMagics.autorestore = True
```

```
StoreMagics.store(parameter_s='')
```

Lightweight persistence for python variables.

Example:

```
In [1]: l = ['hello',10,'world']
In [2]: %store l
In [3]: exit

(IPython session is closed and started again...)
```
Usage:

• %store - Show list of all variables and their current values
• %store spam - Store the current value of the variable spam to disk
• %store -d spam - Remove the variable and its value from storage
• %store -z - Remove all variables from storage
• %store -r - Refresh all variables from store (overwrite current vals)
• %store -r spam bar - Refresh specified variables from store (delete current val)
• %store foo >a.txt - Store value of foo to new file a.txt
• %store foo >>a.txt - Append value of foo to file a.txt

It should be noted that if you change the value of a variable, you need to %store it again if you want to persist the new value.

Note also that the variables will need to be pickleable; most basic python types can be safely %store’d. Also aliases can be %store’d across sessions.

• octavemagic used to be bundled, but is now part of oct2py. Use %load_ext oct2py.ipython to load it.
• rmagic is now part of rpy2. Use %load_ext rpy2.ipython to load it, and see rpy2.ipython.rmagic for details of how to use it.
• cythonmagic used to be bundled, but is now part of cython. Use %load_ext Cython to load it.
• sympyprinting used to be a bundled extension, but you should now use sympy.init_printing() instead.

5.2.2 Integrating your objects with IPython

Tab completion

To change the attributes displayed by tab-completing your object, define a __dir__(self) method for it. For more details, see the documentation of the built-in dir() function.

Rich display

The notebook and the Qt console can display richer representations of objects. To use this, you can define any of a number of _repr_*_() methods. Note that these are surrounded by single, not double
underscores.

Both the notebook and the Qt console can display svg, png and jpeg representations. The notebook can also display html, javascript, and latex. If the methods don’t exist, or return None, it falls back to a standard repr().

For example:

```python
class Shout(object):
    def __init__(self, text):
        self.text = text

    def __repr_html__(self):
        return "<h1>" + self.text + "</h1>"
```

Custom exception tracebacks

Rarely, you might want to display a different traceback with an exception - IPython’s own parallel computing framework does this to display errors from the engines. To do this, define a _render_traceback_(self) method which returns a list of strings, each containing one line of the traceback.

Please be conservative in using this feature; by replacing the default traceback you may hide important information from the user.

5.2.3 Defining custom magics

There are two main ways to define your own magic functions: from standalone functions and by inheriting from a base class provided by IPython: IPython.core.magic.Magics. Below we show code you can place in a file that you load from your configuration, such as any file in the startup subdirectory of your default IPython profile.

First, let us see the simplest case. The following shows how to create a line magic, a cell one and one that works in both modes, using just plain functions:

```python
from IPython.core.magic import (register_line_magic, register_cell_magic,
                                 register_line_cell_magic)

@register_line_magic
def lmagic(line):
    "my line magic"
    return line

@register_cell_magic
def cmagic(line, cell):
    "my cell magic"
    return line, cell

@register_line_cell_magic
def lcmagic(line, cell=None):
    "Magic that works both as %lcmagic and as %%lcmagic"
```
if cell is None:
    print("Called as line magic")
    return line
else:
    print("Called as cell magic")
    return line, cell

# In an interactive session, we need to delete these to avoid
# name conflicts for automagic to work on line magics.
del lmagic, lcmagic

You can also create magics of all three kinds by inheriting from the `IPython.core.magic.Magics` class. This lets you create magics that can potentially hold state in between calls, and that have full access to the main IPython object:

```python
from __future__ import print_function
from IPython.core.magic import (Magics, magics_class, line_magic, cell_magic, line_cell_magic)

@magics_class
class MyMagics(Magics):
    @line_magic
    def lmagic(self, line):
        "my line magic"
        print("Full access to the main IPython object:", self.shell)
        print("Variables in the user namespace:", list(self.shell.user_ns.keys()))
        return line

    @cell_magic
    def cmagic(self, line, cell):
        "my cell magic"
        return line, cell

    @line_cell_magic
    def lcmagic(self, line, cell=None):
        "Magic that works both as %lcmagic and as %%lcmagic"
        if cell is None:
            print("Called as line magic")
            return line
        else:
            print("Called as cell magic")
            return line, cell
```

# In order to actually use these magics, you must register them with a
# running IPython. This code must be placed in a file that is loaded once
# IPython is up and running:
ip = get_ipython()
You can register the class itself without instantiating it. IPython will call the default constructor on it.

```
ip.register_magics(MyMagics)
```

If you want to create a class with a different constructor that holds additional state, then you should always call the parent constructor and instantiate the class yourself before registration:

```python
@magics_class
class StatefulMagics(Magics):
    "Magics that hold additional state"
    def __init__(self, shell, data):
        # You must call the parent constructor
        super(StatefulMagics, self).__init__(shell)
        self.data = data

    # etc...
```

This class must then be registered with a manually created instance, since its constructor has different arguments from the default:

```python
ip = get_ipython()
magics = StatefulMagics(ip, some_data)
ip.register_magics(magics)
```

In earlier versions, IPython had an API for the creation of line magics (cell magics did not exist at the time) that required you to create functions with a method-looking signature and to manually pass both the function and the name. While this API is no longer recommended, it remains indefinitely supported for backwards compatibility purposes. With the old API, you’d create a magic as follows:

```python
def func(self, line):
    print("Line magic called with line:", line)
    print("IPython object:", self.shell)

ip = get_ipython()
# Declare this function as the magic %mycommand
ip.define_magic('mycommand', func)
```

### 5.2.4 Custom input transformation

IPython extends Python syntax to allow things like magic commands, and help with the ? syntax. There are several ways to customise how the user’s input is processed into Python code to be executed.

These hooks are mainly for other projects using IPython as the core of their interactive interface. Using them carelessly can easily break IPython!

#### String based transformations

When the user enters a line of code, it is first processed as a string. By the end of this stage, it must be valid Python syntax.
These transformers all subclass `IPython.core.inputtransformer.InputTransformer`, and are used by `IPython.core.inputsplitter.IPythonInputSplitter`.

These transformers act in three groups, stored separately as lists of instances in attributes of `IPythonInputSplitter`:

- **physical_line_transforms** act on the lines as the user enters them. For example, these strip Python prompts from examples pasted in.

- **logical_line_transforms** act on lines as connected by explicit line continuations, i.e. \ at the end of physical lines. They are skipped inside multiline Python statements. This is the point where IPython recognises `%magic` commands, for instance.

- **python_line_transforms** act on blocks containing complete Python statements. Multi-line strings, lists and function calls are reassembled before being passed to these, but note that function and class `definitions` are still a series of separate statements. IPython does not use any of these by default.

An InteractiveShell instance actually has two `IPythonInputSplitter` instances, as the attributes `input_splitter`, to tell when a block of input is complete, and `input_transformer_manager`, to transform complete cells. If you add a transformer, you should make sure that it gets added to both, e.g.:

```python
ip.input_splitter.logical_line_transforms.append(my_transformer())
ip.input_transformer_manager.logical_line_transforms.append(my_transformer())
```

These transformers may raise `SyntaxError` if the input code is invalid, but in most cases it is clearer to pass unrecognised code through unmodified and let Python’s own parser decide whether it is valid.

Changed in version 2.0: Added the option to raise `SyntaxError`.

### Stateless Transformations

The simplest kind of transformations work one line at a time. Write a function which takes a line and returns a line, and decorate it with `StatelessInputTransformer.wrap()`:

```python
@StatelessInputTransformer.wrap
def my_special_commands(line):
    if line.startswith("-"):
        return "specialcommand(" + repr(line) + ")"
    return line
```

The decorator returns a factory function which will produce instances of `StatelessInputTransformer` using your function.

### Coroutine Transformers

More advanced transformers can be written as coroutines. The coroutine will be sent each line in turn, followed by `None` to reset it. It can yield lines, or `None` if it is accumulating text to yield at a later point. When reset, it should give up any code it has accumulated.

This code in IPython strips a constant amount of leading indentation from each line in a cell:
@CoroutineInputTransformer.wrap

def leading_indent():
    """Remove leading indentation.

    If the first line starts with a spaces or tabs, the same whitespace will be
    removed from each following line until it is reset.
    """
    space_re = re.compile(r'^[ 	]+')
    line = ''
    while True:
        line = (yield line)

        if line is None:
            continue

        m = space_re.match(line)
        if m:
            space = m.group(0)
            while line is not None:
                if line.startswith(space):
                    line = line[len(space):]
                line = (yield line)
        else:
            # No leading spaces - wait for reset
            while line is not None:
                line = (yield line)

    leading_indent.look_in_string = True

Token-based transformers

There is an experimental framework that takes care of tokenizing and untokenizing lines of code. Define a function that accepts a list of tokens, and returns an iterable of output tokens, and decorate it with `TokenInputTransformer.wrap()`. These should only be used in `python_line_transforms`.

AST transformations

After the code has been parsed as Python syntax, you can use Python’s powerful Abstract Syntax Tree tools to modify it. Subclass `ast.NodeTransformer`, and add an instance to `shell.ast_transformers`.

This example wraps integer literals in an `Integer` class, which is useful for mathematical frameworks that want to handle e.g. 1/3 as a precise fraction:

class IntegerWrapper(ast.NodeTransformer):
    """Wraps all integers in a call to Integer()"""
    def visit_Num(self, node):
        if isinstance(node.n, int):
            return ast.Call(func=ast.Name(id='Integer', ctx=ast.Load()),
                             args=[node], keywords=[])
        return node
5.2.5 IPython Events

Extension code can register callbacks functions which will be called on specific events within the IPython code. You can see the current list of available callbacks, and the parameters that will be passed with each, in the callback prototype functions defined in `IPython.core.callbacks`.

To register callbacks, use `IPython.core.events.EventManager.register()`. For example:

```python
class VarWatcher(object):
    def __init__(self, ip):
        self.shell = ip
        self.last_x = None

    def pre_execute(self):
        self.last_x = self.shell.user_ns.get('x', None)

    def post_execute(self):
        if self.shell.user_ns.get('x', None) != self.last_x:
            print("x changed!")

def load_ipython_extension(ip):
    vw = VarWatcher(ip)
    ip.events.register('pre_execute', vw.pre_execute)
    ip.events.register('post_execute', vw.post_execute)
```

### Events

These are the events IPython will emit. Callbacks will be passed no arguments, unless otherwise specified.

**shell_initialized**

```python
def shell_initialized(ipython):
    ...
```

This event is triggered only once, at the end of setting up IPython. Extensions registered to load by default as part of configuration can use this to execute code to finalize setup. Callbacks will be passed the `InteractiveShell` instance.

**pre_run_cell**

`pre_run_cell` fires prior to interactive execution (e.g. a cell in a notebook). It can be used to note the state prior to execution, and keep track of changes.

**pre_execute**

`pre_execute` is like `pre_run_cell`, but is triggered prior to any execution. Sometimes code can be executed by libraries, etc. which skipping the history/display mechanisms, in which cases `pre_run_cell` will not fire.
post_run_cell

post_run_cell runs after interactive execution (e.g. a cell in a notebook). It can be used to cleanup or notify or perform operations on any side effects produced during execution. For instance, the inline matplotlib backend uses this event to display any figures created but not explicitly displayed during the course of the cell.

post_execute

The same as pre_execute, post_execute is like post_run_cell, but fires for all executions, not just interactive ones.

See also:

Module IPython.core.hooks The older ‘hooks’ system allows end users to customise some parts of IPython’s behaviour.

Custom input transformation By registering input transformers that don’t change code, you can monitor what is being executed.

5.2.6 Integrating with GUI event loops

When the user types %gui qt, IPython integrates itself with the Qt event loop, so you can use both a GUI and an interactive prompt together. IPython supports a number of common GUI toolkits, but from IPython 3.0, it is possible to integrate other event loops without modifying IPython itself.

Terminal IPython handles event loops very differently from the IPython kernel, so different steps are needed to integrate with each.

Event loops in the terminal

In the terminal, IPython uses a blocking Python function to wait for user input. However, the Python C API provides a hook, PyOS_InputHook(), which is called frequently while waiting for input. This can be set to a function which briefly runs the event loop and then returns.

IPython provides Python level wrappers for setting and resetting this hook. To use them, subclass IPython.lib.inpthook.InputHookBase, and define an enable(app=None) method, which initialises the event loop and calls self.manager.set_inpthook(f) with a function which will briefly run the event loop before exiting. Decorate the class with a call to IPython.lib.inpthook.register():

```python
from IPython.lib.inpthook import register, InputHookBase

@register('clutter')
class ClutterInputHook(InputHookBase):
    def enable(self, app=None):
        self.manager.set_inpthook(inputhook_clutter)
```
You can also optionally define a `disable()` method, taking no arguments, if there are extra steps needed to clean up. IPython will take care of resetting the hook, whether or not you provide a disable method.

The simplest way to define the hook function is just to run one iteration of the event loop, or to run until no events are pending. Most event loops provide some mechanism to do one of these things. However, the GUI may lag slightly, because the hook is only called every 0.1 seconds. Alternatively, the hook can keep running the event loop until there is input ready on stdin. IPython provides a function to facilitate this:

```python
IPython.lib.inpthook.stdin_ready()
```

Returns True if there is something ready to read on stdin.

If this is the case, the hook function should return immediately.

This is implemented for Windows and POSIX systems - on other platforms, it always returns True, so that the hook always gives Python a chance to check for input.

### Event loops in the kernel

The kernel runs its own event loop, so it’s simpler to integrate with others. IPython allows the other event loop to take control, but it must call `IPython.kernel.zmq.kernelbase.Kernel.do_one_iteration()` periodically.

To integrate with this, write a function that takes a single argument, the IPython kernel instance, arranges for your event loop to call `kernel.do_one_iteration()` at least every `kernel._poll_interval` seconds, and starts the event loop.

Decorate this function with `IPython.kernel.zmq.eventloops.register_integration()`, passing in the names you wish to register it for. Here is a slightly simplified version of the Tkinter integration already included in IPython:

```python
@register_integration('tk')
def loop_tk(kernel):
    """Start a kernel with the Tk event loop."""
    from tkinter import Tk

    # Tk uses milliseconds
    poll_interval = int(1000*kernel._poll_interval)
    # For Tkinter, we create a Tk object and call its withdraw method.
    class Timer(object):
        def __init__(self, func):
            self.app = Tk()
            self.app.withdraw()
            self.func = func

        def on_timer(self):
            self.func()
            self.app.after(poll_interval, self.on_timer)

        def start(self):
            self.on_timer()  # Call it once to get things going.
            self.app.mainloop()
```

5.2. Extending and integrating with IPython
```python
kernel.timer = Timer(kernel.do_one_iteration)
kernel.timer.start()
```

Some event loops can go one better, and integrate checking for messages on the kernel’s ZMQ sockets, making the kernel more responsive than plain polling. How to do this is outside the scope of this document; if you are interested, look at the integration with Qt in `IPython.kernel.zmq.eventloops`. 
6.1 How IPython works

6.1.1 Terminal IPython

When you type `ipython`, you get the original IPython interface, running in the terminal. It does something like this:

```python
while True:
    code = input(">>> ")
    exec(code)
```

Of course, it’s much more complex, because it has to deal with multi-line code, tab completion using `readline`, magic commands, and so on. But the model is like that: prompt the user for some code, and when they’ve entered it, exec it in the same process. This model is often called a REPL, or Read-Eval-Print-Loop.

6.1.2 The IPython Kernel

All the other interfaces—the Notebook, the Qt console, `ipython console` in the terminal, and third party interfaces—use the IPython Kernel. This is a separate process which is responsible for running user code, and things like computing possible completions. Frontends communicate with it using JSON messages sent over ZeroMQ sockets; the protocol they use is described in Messaging in Jupyter.

The core execution machinery for the kernel is shared with terminal IPython:
A kernel process can be connected to more than one frontend simultaneously. In this case, the different frontends will have access to the same variables.

This design was intended to allow easy development of different frontends based on the same kernel, but it also made it possible to support new languages in the same frontends, by developing kernels in those languages, and we are refining IPython to make that more practical.

Today, there are two ways to develop a kernel for another language. Wrapper kernels reuse the communications machinery from IPython, and implement only the core execution part. Native kernels implement execution and communications in the target language:
Wrapper kernels are easier to write quickly for languages that have good Python wrappers, like octave_kernel, or languages where it’s impractical to implement the communications machinery, like bash_kernel. Native kernels are likely to be better maintained by the community using them, like IJulia or IHaskell.

See also:
Making kernels for Jupyter
Making simple Python wrapper kernels

6.2 Making simple Python wrapper kernels

New in version 3.0.
You can now re-use the kernel machinery in IPython to easily make new kernels. This is useful for languages
that have Python bindings, such as Octave (via Oct2Py), or languages where the REPL can be controlled in a tty using pexpect, such as bash.

See also:

bash_kernel  A simple kernel for bash, written using this machinery

6.2.1 Required steps

Subclass ipykernel.kernelbase.Kernel, and implement the following methods and attributes:

class MyKernel

implementation
implementation_version
language
language_version
banner

Information for Kernel info replies. ‘Implementation’ refers to the kernel (e.g. IPython), and ‘language’ refers to the language it interprets (e.g. Python). The ‘banner’ is displayed to the user in console UIs before the first prompt. All of these values are strings.

language_info

Language information for Kernel info replies, in a dictionary. This should contain the key mimetype with the mimetype of code in the target language (e.g. ‘text/x-python’), and file_extension (e.g. ‘py’). It may also contain keys codemirror_mode and pygments_lexer if they need to differ from language.

Other keys may be added to this later.

do_execute (code, silent, store_history=True, user_expressions=None, allow_stdin=False)

Execute user code.

Parameters

• code (str) – The code to be executed.
• silent (bool) – Whether to display output.
• store_history (bool) – Whether to record this code in history and increase the execution count. If silent is True, this is implicitly False.
• user_expressions (dict) – Mapping of names to expressions to evaluate after the code has run. You can ignore this if you need to.
• allow_stdin (bool) – Whether the frontend can provide input on request (e.g. for Python’s raw_input ()).

Your method should return a dict containing the fields described in Execution results. To display output, it can send messages using send_response (). See messaging for details of the different message types.

To launch your kernel, add this at the end of your module:
if __name__ == '__main__':
    from ipykernel.kernelapp import IPKernelApp
    IPKernelApp.launch_instance(kernel_class=MyKernel)

6.2.2 Example

echokernel.py will simply echo any input it’s given to stdout:

```python
from ipykernel.kernelbase import Kernel
class EchoKernel(Kernel):
    implementation = 'Echo'
    implementation_version = '1.0'
    language = 'no-op'
    language_version = '0.1'
    language_info = {'mimetype': 'text/plain'}
    banner = "Echo kernel - as useful as a parrot"

def do_execute(self, code, silent, store_history=True, user_expressions=None,
                  allow_stdin=False):
    if not silent:
        stream_content = {'name': 'stdout', 'text': code}
        self.send_response(self.iopub_socket, 'stream', stream_content)
    return {'status': 'ok',
            # The base class increments the execution count
            'execution_count': self.execution_count,
            'payload': [],
            'user_expressions': {}}

if __name__ == '__main__':
    from ipykernel.kernelapp import IPKernelApp
    IPKernelApp.launch_instance(kernel_class=EchoKernel)
```

Here’s the Kernel spec kernel.json file for this:

```json
{"argv": ["python", "-m", "echokernel", "-f", "{connection_file}"],
"display_name": "Echo"}
```

6.2.3 Optional steps

You can override a number of other methods to improve the functionality of your kernel. All of these methods should return a dictionary as described in the relevant section of the messaging spec.

class MyKernel

    do_complete (code, cursor_pos)
    Code completion

6.2. Making simple Python wrapper kernels
Parameters

- **code** (*str*) – The code already present
- **cursor_pos** (*int*) – The position in the code where completion is requested

See also:

Completion messages

**do_inspect** (*code*, *cursor_pos*, *detail_level=0*)

Object introspection

Parameters

- **code** (*str*) – The code
- **cursor_pos** (*int*) – The position in the code where introspection is requested
- **detail_level** (*int*) – 0 or 1 for more or less detail. In IPython, 1 gets the source code.

See also:

Introspection messages

**do_history** (*hist_access_type*, *output*, *raw*, *session=None*, *start=None*, *stop=None*, *n=None*, *pattern=None*, *unique=False*)

History access. Only the relevant parameters for the type of history request concerned will be passed, so your method definition must have defaults for all the arguments shown with defaults here.

See also:

History messages

**do_is_complete** (*code*)

Is code entered in a console-like interface complete and ready to execute, or should a continuation prompt be shown?

Parameters **code** (*str*) – The code entered so far - possibly multiple lines

See also:

Code completeness messages

**do_shutdown** (*restart*)

Shutdown the kernel. You only need to handle your own clean up - the kernel machinery will take care of cleaning up its own things before stopping.

Parameters **restart** (*bool*) – Whether the kernel will be started again afterwards

See also:

Kernel shutdown messages
6.3 Execution semantics in the IPython kernel

The execution of use code consists of the following phases:

1. Fire the `pre_execute` event.
2. Fire the `pre_run_cell` event unless silent is True.
3. Execute the `code` field, see below for details.
4. If execution succeeds, expressions in `user_expressions` are computed. This ensures that any error in the expressions don’t affect the main code execution.
5. Fire the `post_execute` event.

See also:

IPython Events

To understand how the `code` field is executed, one must know that Python code can be compiled in one of three modes (controlled by the `mode` argument to the `compile()` builtin):

- **single** Valid for a single interactive statement (though the source can contain multiple lines, such as a for loop). When compiled in this mode, the generated bytecode contains special instructions that trigger the calling of `sys.displayhook()` for any expression in the block that returns a value. This means that a single statement can actually produce multiple calls to `sys.displayhook()`, if for example it contains a loop where each iteration computes an unassigned expression would generate 10 calls:

```python
for i in range(10):
    i**2
```

- **exec** An arbitrary amount of source code, this is how modules are compiled. `sys.displayhook()` is *never* implicitly called.

- **eval** A single expression that returns a value. `sys.displayhook()` is *never* implicitly called.

The `code` field is split into individual blocks each of which is valid for execution in ‘single’ mode, and then:

- If there is only a single block: it is executed in ‘single’ mode.
- If there is more than one block:
  - if the last one is a single line long, run all but the last in ‘exec’ mode and the very last one in ‘single’ mode. This makes it easy to type simple expressions at the end to see computed values.
  - if the last one is no more than two lines long, run all but the last in ‘exec’ mode and the very last one in ‘single’ mode. This makes it easy to type simple expressions at the end to see computed values.
  - otherwise (last one is also multiline), run all in ‘exec’ mode
  - otherwise (last one is also multiline), run all in ‘exec’ mode as a single unit.
6.4 New IPython Console Lexer

New in version 2.0.0.

The IPython console lexer has been rewritten and now supports tracebacks and customized input/output prompts. An entire suite of lexers is now available at `IPython.lib.lexers`. These include:

**IPythonLexer & IPython3Lexer** Lexers for pure IPython (python + magic/shell commands)

**IPythonPartialTracebackLexer & IPythonTracebackLexer** Supports 2.x and 3.x via the keyword `python3`. The partial traceback lexer reads everything but the Python code appearing in a traceback. The full lexer combines the partial lexer with an IPython lexer.

**IPythonConsoleLexer** A lexer for IPython console sessions, with support for tracebacks. Supports 2.x and 3.x via the keyword `python3`.

**IPyLexer** A friendly lexer which examines the first line of text and from it, decides whether to use an IPython lexer or an IPython console lexer. Supports 2.x and 3.x via the keyword `python3`.

Previously, the `IPythonConsoleLexer` class was available at `IPython.sphinxext.ipython_console_highlight`. It was inserted into Pygments’ list of available lexers under the name `ipython`. It should be mentioned that this name is inaccurate, since an IPython console session is not the same as IPython code (which itself is a superset of the Python language).

Now, the Sphinx extension inserts two console lexers into Pygments’ list of available lexers. Both are IPyLexer instances under the names: `ipython` and `ipython3`. Although the names can be confusing (as mentioned above), their continued use is, in part, to maintain backwards compatibility and to aid typical usage. If a project needs to make Pygments aware of more than just the IPyLexer class, then one should not make the IPyLexer class available under the name `ipython` and use `ipy` or some other non-conflicting value.

Code blocks such as:

```
.. code-block:: ipython

    In [1]: 2**2
    Out[1]: 4
```

will continue to work as before, but now, they will also properly highlight tracebacks. For pure IPython code, the same lexer will also work:

```
.. code-block:: ipython

    x = '' .join(map(str, range(10)))
    !echo $x
```

Since the first line of the block did not begin with a standard IPython console prompt, the entire block is assumed to consist of IPython code instead.
6.5 Writing code for Python 2 and 3

IPython.utils.py3compat.PY3
Boolean indicating whether we’re currently in Python 3.

6.5.1 Iterators

Many built-in functions and methods in Python 2 come in pairs, one returning a list, and one returning an iterator (e.g. `range()` and `xrange`). In Python 3, there is usually only the iterator form, but it has the name which gives a list in Python 2 (e.g. `range()`).

The way to write compatible code depends on what you need:

- A list, e.g. for serialisation, or to test if something is in it.
- Iteration, but it will never be used for very many items, so efficiency isn’t especially important.
- Iteration over many items, where efficiency is important.

<table>
<thead>
<tr>
<th>list</th>
<th>iteration (small)</th>
<th>iteration(large)</th>
</tr>
</thead>
<tbody>
<tr>
<td>list(range(n))</td>
<td>range(n)</td>
<td>py3compat.xrange(n)</td>
</tr>
<tr>
<td>list(map(f, it))</td>
<td>map(f, it)</td>
<td>–</td>
</tr>
<tr>
<td>list(zip(a, b))</td>
<td>zip(a, b)</td>
<td>–</td>
</tr>
<tr>
<td>list(d.items())</td>
<td>d.items()</td>
<td>py3compat.iteritems(d)</td>
</tr>
<tr>
<td>list(d.values())</td>
<td>d.values()</td>
<td>py3compat.itervalues(d)</td>
</tr>
</tbody>
</table>

Iterating over a dictionary yields its keys, so there is rarely a need to use `dict.keys()` or `dict.iterkeys()`.

Avoid using `map()` to cause function side effects. This is more clearly written with a simple for loop.

IPython.utils.py3compat.xrange
A reference to `range` on Python 3, and `xrange` on Python 2.

IPython.utils.py3compat.iteritems(d)
IPython.utils.py3compat.itervalues(d)
Iterate over (key, value) pairs of a dictionary, or just over values. `iterkeys` is not defined: iterating over the dictionary yields its keys.

6.5.2 Changed standard library locations

Several parts of the standard library have been renamed and moved. This is a short list of things that we’re using. A couple of them have names in `IPython.utils.py3compat`, so you don’t need both imports in each module that uses them.
Be careful with StringIO: `io.StringIO` is available in Python 2.7, but it behaves differently from `StringIO.StringIO`, and much of our code assumes the use of the latter on Python 2. So a try/except on the import may cause problems.

```python
IPython.utils.py3compat.input()
```

Behaves like `raw_input` on Python 2.

```python
IPython.utils.py3compat.builtin_mod
IPython.utils.py3compat.builtin_mod_name
```

A reference to the module containing `builtins`, and its name as a string.

### 6.5.3 Unicode

Always be explicit about what is text (unicode) and what is bytes. Encoding goes from unicode to bytes, and decoding goes from bytes to unicode.

To open files for reading or writing text, use `io.open()`, which is the Python 3 built-in `open` function, available on Python 2 as well. We almost always need to specify the encoding parameter, because the default is platform dependent.

We have several helper functions for converting between string types. They all use the encoding from `IPython.utils.encoding.getdefaultencoding()` by default, and the `errors='replace'` option to do best-effort conversions for the user's system.

```python
IPython.utils.py3compat.unicode_to_str(u, encoding=None)
IPython.utils.py3compat.str_to_unicode(s, encoding=None)
```

Convert between unicode and the native str type. No-ops on Python 3.

```python
IPython.utils.py3compat.str_to_bytes(s, encoding=None)
IPython.utils.py3compat.bytes_to_str(u, encoding=None)
```

Convert between bytes and the native str type. No-ops on Python 2.

```python
IPython.utils.py3compat.cast_unicode(s, encoding=None)
IPython.utils.py3compat.cast_bytes(s, encoding=None)
```

Convert strings to unicode/bytes when they may be of either type.

```python
IPython.utils.py3compat.cast_unicode_py2(s, encoding=None)
```
IPython.utils.py3compat.cast_bytes_py2\((s, encoding=None)\)
Convert strings to unicode/bytes when they may be of either type on Python 2, but return them unaltered on Python 3 (where string types are more predictable).

IPython.utils.py3compat.unicode_type
A reference to \texttt{str} on Python 3, and to \texttt{unicode} on Python 2.

IPython.utils.py3compat.string_types
A tuple for \texttt{isinstance} checks: \{(\texttt{str},)\} on Python 3, \{(\texttt{str}, \texttt{unicode})\} on Python 2.

### 6.5.4 Relative imports

```python
# This makes Python 2 behave like Python 3:
from __future__ import absolute_import

import io # Imports the standard library io module
from . import io # Import the io module from the package containing the current module
from .io import foo # foo from the io module next to this module
from IPython.utils import io # This still works
```

### 6.5.5 Print function

```python
# Support the print function on Python 2:
from __future__ import print_function

print(a, b)
print(foo, file=sys.stderr)
print(bar, baz, sep='\t', end='')
```

### 6.5.6 Metaclasses

The syntax for declaring a class with a metaclass is different in Python 2 and 3. A helper function works for most cases:

```python
IPython.utils.py3compat.with_metaclass()
Create a base class with a metaclass. Copied from the six library.

Used like this:
```
class FormatterABC(with_metaclass(abc.ABCMeta, object)):
    ...
```

Combining inheritance between Qt and the traitlets system, however, does not work with this. Instead, we do this:

```python
class QtKernelClientMixin(MetaQObjectHasTraits('NewBase', (HasTraits, SuperQObject), {})):
    ...
```

### 6.5. Writing code for Python 2 and 3
This gives the new class a metaclass of `MetaQObjectHasTraits`, and the parent classes `HasTraits` and `SuperQObject`.

### 6.5.7 Doctests

**IPython.utils.py3compat.doctest_refactor_print(func_or_str)**

Refactors print statements in doctests in Python 3 only. Accepts a string or a function, so it can be used as a decorator.

**IPython.utils.py3compat.u_format(func_or_str)**

Handle doctests written with `{u}'abc\p'`, replacing the `{u}` with `u` for Python 2, and removing it for Python 3.

Accepts a string or a function, so it can be used as a decorator.

### 6.5.8 Execfile

**IPython.utils.py3compat.execfile(fname, glob, loc=None)**

Equivalent to the Python 2 `execfile` builtin. We redefine it in Python 2 to better handle non-ascii filenames.

### 6.5.9 Miscellaneous

**IPython.utils.py3compat.safe_unicode(e)**

`unicode(e)` with various fallbacks. Used for exceptions, which may not be safe to call `unicode()` on.

**IPython.utils.py3compat.isidentifier(s, dotted=False)**

Checks whether the string `s` is a valid identifier in this version of Python. In Python 3, non-ascii characters are allowed. If `dotted` is True, it allows dots (i.e. attribute access) in the string.

**IPython.utils.py3compat.getcwd()**

Return the current working directory as `unicode`, like `os.getcwd()` on Python 2.

**IPython.utils.py3compat.MethodType()**

Constructor for `types.MethodType` that takes two arguments, like the real constructor on Python 3.

### 6.6 Overview of the IPython configuration system

This section describes the IPython configuration system. This is based on `traitlets.config`; see that documentation for more information about the overall architecture.

#### 6.6.1 Configuration file location

So where should you put your configuration files? IPython uses “profiles” for configuration, and by default, all profiles will be stored in the so called “IPython directory”. The location of this directory is determined...
by the following algorithm:

- If the `ipython-dir` command line flag is given, its value is used.
- If not, the value returned by `IPython.paths.get_ipython_dir()` is used. This function will first look at the `IPYTHONDIR` environment variable and then default to `~/.ipython`. Historical support for the `IPYTHON_DIR` environment variable will be removed in a future release.

For most users, the configuration directory will be `~/.ipython`.

Previous versions of IPython on Linux would use the XDG config directory, creating `~/.config/ipython` by default. We have decided to go back to `~/.ipython` for consistency among systems. IPython will issue a warning if it finds the XDG location, and will move it to the new location if there isn’t already a directory there.

Once the location of the IPython directory has been determined, you need to know which profile you are using. For users with a single configuration, this will simply be ‘default’, and will be located in `<IPYTHONDIR>/profile_default`.

The next thing you need to know is what to call your configuration file. The basic idea is that each application has its own default configuration filename. The default named used by the `ipython` command line program is `ipython_config.py`, and all IPython applications will use this file. Other applications, such as the parallel `ipcluster` scripts or the QtConsole will load their own config files after `ipython_config.py`. To load a particular configuration file instead of the default, the name can be overridden by the `config_file` command line flag.

To generate the default configuration files, do:

```
$ ipython profile create
```

and you will have a default `ipython_config.py` in your IPython directory under `profile_default`. If you want the default config files for the IPython.parallel applications, add `--parallel` to the end of the command-line args.

### Locating these files

From the command-line, you can quickly locate the `IPYTHONDIR` or a specific profile with:

```
$ ipython locate
/home/you/.ipython

$ ipython locate profile foo
/home/you/.ipython/profile_foo
```

These map to the utility functions: `IPython.utils.path.get_ipython_dir()` and `IPython.utils.path.locate_profile()` respectively.

### 6.6.2 Profiles

A profile is a directory containing configuration and runtime files, such as logs, connection info for the parallel apps, and your IPython command history.
The idea is that users often want to maintain a set of configuration files for different purposes: one for doing numerical computing with NumPy and SciPy and another for doing symbolic computing with SymPy. Profiles make it easy to keep a separate configuration files, logs, and histories for each of these purposes.

Let’s start by showing how a profile is used:

```
$ ipython --profile=sympy
```

This tells the `ipython` command line program to get its configuration from the “sympy” profile. The file names for various profiles do not change. The only difference is that profiles are named in a special way. In the case above, the “sympy” profile means looking for `ipython_config.py` in `<IPYTHONDIR>/profile_sympy`.

The general pattern is this: simply create a new profile with:

```
$ ipython profile create <name>
```

which adds a directory called `profile_<name>` to your IPython directory. Then you can load this profile by adding `--profile=<name>` to your command line options. Profiles are supported by all IPython applications.

IPython ships with some sample profiles in `IPython/config/profile`. If you create profiles with the name of one of our shipped profiles, these config files will be copied over instead of starting with the automatically generated config files.

IPython extends the config loader for Python files so that you can inherit config from another profile. To do this, use a line like this in your Python config file:

```
load_subconfig('ipython_config.py', profile='default')
```

### 6.7 IPython GUI Support Notes

IPython allows GUI event loops to be run in an interactive IPython session. This is done using Python’s `PyOS_InputHook` hook which Python calls when the `raw_input()` function is called and waiting for user input. IPython has versions of this hook for wx, pyqt4 and pygtk.

When a GUI program is used interactively within IPython, the event loop of the GUI should not be started. This is because, the `PyOS_Inputhook` itself is responsible for iterating the GUI event loop.

IPython has facilities for installing the needed input hook for each GUI toolkit and for creating the needed main GUI application object. Usually, these main application objects should be created only once and for some GUI toolkits, special options have to be passed to the application object to enable it to function properly in IPython.

We need to answer the following questions:

- Who is responsible for creating the main GUI application object, IPython or third parties (matplotlib, enthought.traits, etc.)?
- What is the proper way for third party code to detect if a GUI application object has already been created? If one has been created, how should the existing instance be retrieved?
• In a GUI application object has been created, how should third party code detect if the GUI event loop is running. It is not sufficient to call the relevant function methods in the GUI toolkits (like IsMainLoopRunning) because those don’t know if the GUI event loop is running through the input hook.

• We might need a way for third party code to determine if it is running in IPython or not. Currently, the only way of running GUI code in IPython is by using the input hook, but eventually, GUI based versions of IPython will allow the GUI event loop in the more traditional manner. We will need a way for third party code to distinguish between these two cases.

Here is some sample code I have been using to debug this issue:

```python
from matplotlib import pyplot as plt
from enthought.traits import api as traits

class Foo(traits.HasTraits):
    a = traits.Float()

f = Foo()
f.configure_traits()

plt.plot(range(10))
```
Developer’s guide to core IPython

This guide documents the development of core IPython. Alternatively, developers of third party tools and libraries that use IPython should see the Developer’s guide for third party tools and libraries.

Developers working on core IPython should also consult the developer information on the IPython GitHub wiki.

7.1 IPython release process

This document contains the process that is used to create an IPython release.

Conveniently, the release script in the tools directory of the IPython repository automates most of the release process. This document serves as a handy reminder and checklist for the release manager.

7.1.1 1. Set Environment variables

Set environment variables to document previous release tag, current release milestone, current release version, and git tag:

```shell
PREV_RELEASE=4.0.0
MILESTONE=4.1
VERSION=4.1.0
BRANCH=master
```

These variables may be used later to copy/paste as answers to the script questions instead of typing the appropriate command when the time comes. These variables are not used by the scripts directly; therefore, there is no need to export the variables.

7.1.2 2. Create GitHub stats and finish release note

Note: Before generating the GitHub stats, verify that all closed issues and pull requests have appropriate milestones. This search should return no results before creating the GitHub stats.
If a major release:

- merge any pull request notes into what’s new:

```python
tools/update_whatsnew.py
```

- update `docs/source/whatsnew/development.rst`, to ensure it covers the major release features

- move the contents of `development.rst` to `versionX.rst` where `X` is the numerical release version

- generate summary of GitHub contributions, which can be done with:

```python
tools/github_stats.py --milestone $MILESTONE > stats.rst
```

which may need some manual cleanup of `stats.rst`. Add the cleaned `stats.rst` results to `docs/source/whatsnew/github-stats-X.rst` where `X` is the numerical release version. If creating a major release, make a new `github-stats-X.rst` file; if creating a minor release, the content from `stats.rst` may simply be added to the top of an existing `github-stats-X.rst` file.

To find duplicates and update `.mailmap`, use:

```bash
git log --format="%aN <%aE>" $PREV_RELEASE... | sort -u -f
```

### 7.1.3 3. Make sure the repository is clean

of any file that could be problematic. Remove all non-tracked files with:

```bash
git clean -xfdi
```

This will ask for confirmation before removing all untracked files. Make sure the `dist/` folder is clean to avoid any stale builds from previous build attempts.

### 7.1.4 4. Run the `tools/build_release` script

Running `tools/build_release` does all the file checking and building that the real release script will do. This makes test installations, checks that the build procedure runs OK, and tests other steps in the release process.

We encourage creating a test build of the docs as well.

### 7.1.5 5. Create and push the new tag

Edit `IPython/core/release.py` to have the current version.

in particular, update version number and `_version_extra` content in `IPython/core/release.py`. 
Make sure the version number matches pep440, in particular, rc and beta are not separated by . or the sdist and bdist will appear as different releases. For example, a valid version number for a release candidate (rc) release is: 1.3rc1. Notice that there is no separator between the ‘3’ and the ‘r’.

Commit the changes to release.py:

```
git commit -am "release $VERSION"
git push origin $BRANCH
```

Create and push the tag:

```
git tag -am "release $VERSION" "$VERSION"
git push origin --tags
```

Update release.py back to x.y-dev or x.y-maint, and push:

```
git commit -am "back to development"
git push origin $BRANCH
```

### 7.1.6 6. Get a fresh clone

Get a fresh clone of the tag for building the release:

```
cd /tmp
git clone --depth 1 https://github.com/ipython/ipython.git -b "$VERSION"
```

### 7.1.7 7. Run the release script

Run the release script, this step requires having a current wheel, Python >=3.4 and Python 2.7.:

```
cd tools && ./release
```

This makes the tarballs, zipfiles, and wheels, and put them under the dist/ folder. Be sure to test the wheel and the sdist locally before uploading them to PyPI.

Use the following to actually upload the result of the build:

```
./release upload
```

It should posts them to archive.ipython.org and registers the release with PyPI if you have the various authorisations.

You might need to use twine (twine upload dist/*) manually to actually upload on PyPI. Unlike setuptools, twine is able to upload packages over SSL.

### 7.1.8 8. Draft a short release announcement

The announcement should include:

- release highlights
- a link to the html version of the What’s new section of the documentation

7.1. IPython release process 397
• a link to upgrade or installation tips (if necessary)

Post the announcement to the mailing list and/or blog, and link from Twitter.

7.1.9 9. Update milestones on GitHub

These steps will bring milestones up to date:

• close the just released milestone
• open a new milestone for the next release (x, y+1), if the milestone doesn’t exist already

7.1.10 10. Update the IPython website

The IPython website should document the new release:

• add release announcement (news, announcements)
• update current version and download links
• update links on the documentation page (especially if a major release)

7.1.11 11. Celebrate!

Celebrate the release and please thank the contributors for their work. Great job!
<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPython</td>
<td>IPython: tools for interactive and parallel computing in Python.</td>
</tr>
<tr>
<td>IPython.core.alias</td>
<td>System command aliases.</td>
</tr>
<tr>
<td>IPython.core.application</td>
<td>An application for IPython.</td>
</tr>
<tr>
<td>IPython.core.autocall</td>
<td>Autocall capabilities for IPython.core.</td>
</tr>
<tr>
<td>IPython.core.builtin_trap</td>
<td>A context manager for managing things injected into <strong>builtin</strong>.</td>
</tr>
<tr>
<td>IPython.core.compilerop</td>
<td>Compiler tools with improved interactive support.</td>
</tr>
<tr>
<td>IPython.core.completer</td>
<td>Word completion for IPython.</td>
</tr>
<tr>
<td>IPython.core.completerlib</td>
<td>Implementations for various useful completers.</td>
</tr>
<tr>
<td>IPython.core.crashhandler</td>
<td>sys.excepthook for IPython itself, leaves a detailed report on disk.</td>
</tr>
<tr>
<td>IPython.core.debugger</td>
<td>Pdb debugger class.</td>
</tr>
<tr>
<td>IPython.core.error</td>
<td>Global exception classes for IPython.core.</td>
</tr>
<tr>
<td>IPython.core.events</td>
<td>Infrastructure for registering and firing callbacks on application events.</td>
</tr>
<tr>
<td>IPython.core.extensions</td>
<td>A class for managing IPython extensions.</td>
</tr>
<tr>
<td>IPython.core.formatters</td>
<td>Display formatters.</td>
</tr>
<tr>
<td>IPython.core.getipython</td>
<td>Simple function to call to get the current InteractiveShell instance</td>
</tr>
<tr>
<td>IPython.core.history</td>
<td>History related magics and functionality</td>
</tr>
<tr>
<td>IPython.core.historyapp</td>
<td>An application for managing IPython history.</td>
</tr>
<tr>
<td>IPython.core.hooks</td>
<td>Hooks for IPython.</td>
</tr>
<tr>
<td>IPython.core.inputsplitter</td>
<td>Input handling and transformation machinery.</td>
</tr>
<tr>
<td>IPython.core.inputtransformer</td>
<td>Input transformer classes to support IPython special syntax.</td>
</tr>
<tr>
<td>IPython.core.interactiveshell</td>
<td>Main IPython class.</td>
</tr>
<tr>
<td>IPython.core.logger</td>
<td>Logger class for IPython’s logging facilities.</td>
</tr>
<tr>
<td>IPython.core.macro</td>
<td>Support for interactive macros in IPython</td>
</tr>
<tr>
<td>IPython.core.magic</td>
<td>Magic functions for InteractiveShell.</td>
</tr>
<tr>
<td>IPython.core.magic_arguments</td>
<td>A decorator-based method of constructing IPython magics with argparse.</td>
</tr>
<tr>
<td>IPython.core.oinspect</td>
<td>Tools for inspecting Python objects.</td>
</tr>
<tr>
<td>IPython.core.page</td>
<td>Paging capabilities for IPython.core</td>
</tr>
<tr>
<td>IPython.core.payload</td>
<td>Payload system for IPython.</td>
</tr>
<tr>
<td>IPython.core.payloadpage</td>
<td>A payload based version of page.</td>
</tr>
<tr>
<td>IPython.core.prefilter</td>
<td>Prefiltering components.</td>
</tr>
<tr>
<td>IPython.core.profileapp</td>
<td>An application for managing IPython profiles.</td>
</tr>
</tbody>
</table>
Table 8.1 – continued from previous page

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPython.core.profiledir</td>
<td>An object for managing IPython profile directories.</td>
</tr>
<tr>
<td>IPython.core.prompts</td>
<td>Classes for handling input/output prompts.</td>
</tr>
<tr>
<td>IPython.core.pylabtools</td>
<td>Pylab (matplotlib) support utilities.</td>
</tr>
<tr>
<td>IPython.core.shellapp</td>
<td>A mixin for Application classes that launch InteractiveShell instances.</td>
</tr>
<tr>
<td>IPython.core.splitinput</td>
<td>Simple utility for splitting user input.</td>
</tr>
<tr>
<td>IPython.core.ultratb</td>
<td>Verbose and colourful traceback formatting.</td>
</tr>
<tr>
<td>IPython.core.usage</td>
<td>Usage information for the main IPython applications.</td>
</tr>
<tr>
<td>IPython.display</td>
<td>Public API for display tools in IPython.</td>
</tr>
<tr>
<td>IPython.lib.backgroundjobs</td>
<td>Manage background (threaded) jobs conveniently from an interactive shell.</td>
</tr>
<tr>
<td>IPython.lib.clipboard</td>
<td>Utilities for accessing the platform's clipboard.</td>
</tr>
<tr>
<td>IPython.lib.deepreload</td>
<td>Provides a reload() function that acts recursively.</td>
</tr>
<tr>
<td>IPython.lib.demo</td>
<td>Module for interactive demos using IPython.</td>
</tr>
<tr>
<td>IPython.lib.editorhooks</td>
<td>‘editor’ hooks for common editors that work well with ipython.</td>
</tr>
<tr>
<td>IPython.lib.guisupport</td>
<td>Support for creating GUI apps and starting event loops.</td>
</tr>
<tr>
<td>IPython.lib.inpthook</td>
<td>Inpthook management for GUI event loop integration.</td>
</tr>
<tr>
<td>IPython.lib.latextools</td>
<td>Tools for handling LaTeX.</td>
</tr>
<tr>
<td>IPython.lib.pretty</td>
<td>Python advanced pretty printer.</td>
</tr>
<tr>
<td>IPython.lib.security</td>
<td>Password generation for the IPython notebook.</td>
</tr>
<tr>
<td>IPython.paths</td>
<td>Find files and directories which IPython uses.</td>
</tr>
<tr>
<td>IPython.terminal.embed</td>
<td>An embedded IPython shell.</td>
</tr>
<tr>
<td>IPython.terminal.interactiveshell</td>
<td>Subclass of InteractiveShell for terminal based frontends.</td>
</tr>
<tr>
<td>IPython.terminal.ipapp</td>
<td>The Application object for the command line ipython program.</td>
</tr>
<tr>
<td>IPython.testing</td>
<td>Testing support (tools to test IPython itself).</td>
</tr>
<tr>
<td>IPython.testing.decorators</td>
<td>Decorators for labeling test objects.</td>
</tr>
<tr>
<td>IPython.testing.globalsource</td>
<td>Global IPython app to support test running.</td>
</tr>
<tr>
<td>IPython.testing.iputil</td>
<td>Experimental code for cleaner support of IPython syntax with unittest.</td>
</tr>
<tr>
<td>IPython.testing.skipdoctest</td>
<td>Decorators marks that a doctest should be skipped, for both python 2 and 3.</td>
</tr>
<tr>
<td>IPython.utils.PyColorize</td>
<td>Generic testing tools.</td>
</tr>
<tr>
<td>IPython.utils.capture</td>
<td>Class and program to colorize python source code for ANSI terminals.</td>
</tr>
<tr>
<td>IPython.utils.coloransi</td>
<td>IO capturing utilities.</td>
</tr>
<tr>
<td>IPython.utils.contexts</td>
<td>Tools for coloring text in ANSI terminals.</td>
</tr>
<tr>
<td>IPython.utils.data</td>
<td>Miscellaneous context managers.</td>
</tr>
<tr>
<td>IPython.utils.decorators</td>
<td>Utilities for working with data structures like lists, dicts and tuples.</td>
</tr>
<tr>
<td>IPython.utils.dir2</td>
<td>Decorators that don’t go anywhere else.</td>
</tr>
<tr>
<td>IPython.utils.encoding</td>
<td>A fancy version of Python’s built-in dir() function.</td>
</tr>
<tr>
<td>IPython.utils.encoding</td>
<td>Utilities for dealing with text encodings.</td>
</tr>
<tr>
<td>IPython.utils.frame</td>
<td>Utilities for working with stack frames.</td>
</tr>
<tr>
<td>IPython.utils.generics</td>
<td>Generic functions for extending IPython.</td>
</tr>
<tr>
<td>IPython.utils.importjson</td>
<td>A simple utility to import something by its string name.</td>
</tr>
<tr>
<td>IPython.utils.io</td>
<td>IO related utilities.</td>
</tr>
<tr>
<td>IPython.utils.ipstruct</td>
<td>A dict subclass that supports attribute style access.</td>
</tr>
</tbody>
</table>
Table 8.1 – continued from previous page

<table>
<thead>
<tr>
<th>Module Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IPython.utils.module_paths</code></td>
<td>Utility functions for finding modules</td>
</tr>
<tr>
<td><code>IPython.utils.openpy</code></td>
<td>Tools to open .py files as Unicode, using the encoding specified within the file.</td>
</tr>
<tr>
<td><code>IPython.utils.path</code></td>
<td>Utilities for path handling.</td>
</tr>
<tr>
<td><code>IPython.utils.process</code></td>
<td>Utilities for working with external processes.</td>
</tr>
<tr>
<td><code>IPython.utils.rlineimpl</code></td>
<td>Imports and provides the ‘correct’ version of readline for the platform.</td>
</tr>
<tr>
<td><code>IPython.utils.sentinel</code></td>
<td>Sentinel class for constants with useful reprs</td>
</tr>
<tr>
<td><code>IPython.utils.shimmodule</code></td>
<td>A shim module for deprecated imports</td>
</tr>
<tr>
<td><code>IPython.utils.strdispatch</code></td>
<td>String dispatch class to match regexps and dispatch commands.</td>
</tr>
<tr>
<td><code>IPython.utils.sysinfo</code></td>
<td>Utilities for getting information about IPython and the system it’s running on.</td>
</tr>
<tr>
<td><code>IPython.utils.syspathcontext</code></td>
<td>Context managers for adding things to sys.path temporarily.</td>
</tr>
<tr>
<td><code>IPython.utils.tempdir</code></td>
<td>TemporaryDirectory class, copied from Python 3.2.</td>
</tr>
<tr>
<td><code>IPython.utils.terminal</code></td>
<td>Utilities for working with terminals.</td>
</tr>
<tr>
<td><code>IPython.utils.text</code></td>
<td>Utilities for working with strings and text.</td>
</tr>
<tr>
<td><code>IPython.utils.timing</code></td>
<td>Utilities for timing code execution.</td>
</tr>
<tr>
<td><code>IPython.utils.tokenutil</code></td>
<td>Token-related utilities</td>
</tr>
<tr>
<td><code>IPython.utils.tz</code></td>
<td>Timezone utilities</td>
</tr>
<tr>
<td><code>IPython.utils.ulinecache</code></td>
<td>Wrapper around linecache which decodes files to unicode according to PEP 263.</td>
</tr>
<tr>
<td><code>IPython.utils.version</code></td>
<td>Utilities for version comparison</td>
</tr>
<tr>
<td><code>IPython.utils.warn</code></td>
<td>Utilities for warnings.</td>
</tr>
<tr>
<td><code>IPython.utils.wildcard</code></td>
<td>Support for wildcard pattern matching in object inspection.</td>
</tr>
</tbody>
</table>

8.1 IPython

IPython: tools for interactive and parallel computing in Python.
http://ipython.org

8.1.1 1 Class

class IPython.Config(*args, **kwargs)
    Bases: traitlets.config.loader.Config
    __init__(*args, **kwargs)

8.1.2 3 Functions

IPython.embed_kernel(module=None, local_ns=None, **kwargs)
    Embed and start an IPython kernel in a given scope.

    If you don’t want the kernel to initialize the namespace from the scope of the surrounding function, and/or you want to load full IPython configuration, you probably want IPython.start_kernel() instead.

Parameters
• **module** (*ModuleType, optional*) – The module to load into IPython globals (default: caller)

• **local_ns** (*dict, optional*) – The namespace to load into IPython user namespace (default: caller)

• **kwargs** (*various, optional*) – Further keyword args are relayed to the IPKernelApp constructor, allowing configuration of the Kernel. Will only have an effect on the first embed_kernel call for a given process.

IPython.start_ipython(*argv=None, **kwargs*)
Launch a normal IPython instance (as opposed to embedded)

IPython.embed() puts a shell in a particular calling scope, such as a function or method for debugging purposes, which is often not desirable.

start_ipython() does full, regular IPython initialization, including loading startup files, configuration, etc. much of which is skipped by embed().

This is a public API method, and will survive implementation changes.

**Parameters**

• **argv** (*list or None, optional*) – If unspecified or None, IPython will parse command-line options from sys.argv. To prevent any command-line parsing, pass an empty list: argv=[].

• **user_ns** (*dict, optional*) – specify this dictionary to initialize the IPython user namespace with particular values.

• **kwargs** (*various, optional*) – Any other kwargs will be passed to the Application constructor, such as config.

IPython.start_kernel(*argv=None, **kwargs*)
Launch a normal IPython kernel instance (as opposed to embedded)

IPython.embed_kernel() puts a shell in a particular calling scope, such as a function or method for debugging purposes, which is often not desirable.

start_kernel() does full, regular IPython initialization, including loading startup files, configuration, etc. much of which is skipped by embed().

**Parameters**

• **argv** (*list or None, optional*) – If unspecified or None, IPython will parse command-line options from sys.argv. To prevent any command-line parsing, pass an empty list: argv=[].

• **user_ns** (*dict, optional*) – specify this dictionary to initialize the IPython user namespace with particular values.

• **kwargs** (*various, optional*) – Any other kwargs will be passed to the Application constructor, such as config.
8.2 Module: `core.alias`

System command aliases.

Authors:
- Fernando Perez
- Brian Granger

8.2.1 4 Classes

class `IPython.core.alias.AliasError`
Bases: `Exception`

class `IPython.core.alias.InvalidAliasError`
Bases: `IPython.core.alias.AliasError`

class `IPython.core.alias.Alias`(shell, name, cmd)
Bases: `object`

Callable object storing the details of one alias.
Instances are registered as magic functions to allow use of aliases.

__init__(shell, name, cmd)

validate()

Validate the alias, and return the number of arguments.

class `IPython.core.alias.AliasManager`(shell=None, **kwargs)
Bases: `traitlets.config.configurable.Configurable`

__init__(shell=None, **kwargs)

define_alias(name, cmd)

Define a new alias after validating it.

This will raise an `AliasError` if there are validation problems.

get_alias(name)

Return an alias, or None if no alias by that name exists.

is_alias(name)

Return whether or not a given name has been defined as an alias

retrieve_alias(name)

Retrieve the command to which an alias expands.

soft_define_alias(name, cmd)

Define an alias, but don’t raise on an AliasError.
### 8.2.2 1 Function

```python
IPython.core.alias.default_aliases()
```
Return list of shell aliases to auto-define.

### 8.3 Module: `core.application`

An application for IPython.

All top-level applications should use the classes in this module for handling configuration and creating configurables.

The job of an `Application` is to create the master configuration object and then create the configurable objects, passing the config to them.

#### 8.3.1 2 Classes

```python
class IPython.core.application.ProfileAwareConfigLoader(filename, path=None, **kw)
```
A Python file config loader that is aware of IPython profiles.

```python
class IPython.core.application.BaseIPythonApplication(**kwargs)
```
Bases: `traitlets.config.application.Application`

```python
__init__(**kwargs)
```

```python
excepthook(etype, evalue, tb)
```
this is `sys.excepthook` after `init_crashhandler`

```python
init_config_files()
```
[optionally] copy default config files into profile dir.

```python
init_crash_handler()
```
Create a crash handler, typically setting `sys.excepthook` to it.

```python
init_profile_dir()
```
initialize the profile dir

```python
load_config_file(suppress_errors=True)
```
Load the config file.

By default, errors in loading config are handled, and a warning printed on screen. For testing, the suppress_errors option is set to False, so errors will make tests fail.

```python
python_config_loader_class
```
alias of `ProfileAwareConfigLoader`
stage_default_config_file()
  auto generate default config file, and stage it into the profile.

8.4 Module: core.autocall

Autocall capabilities for IPython.core.

Authors:
  • Brian Granger
  • Fernando Perez
  • Thomas Kluyver

Notes

8.4.1 3 Classes

class IPython.core.autocall.IPyAutocall(ip=None)
  Bases: object

  Instances of this class are always autocalled
  This happens regardless of ‘autocall’ variable state. Use this to develop macro-like mechanisms.

  __init__ (ip=None)

  set_ip (ip)
    Will be used to set _ip point to current ipython instance b/f call
    Override this method if you don’t want this to happen.

class IPython.core.autocall.ExitAutocall(ip=None)
  Bases: IPython.core.autocall.IPyAutocall

  An autocallable object which will be added to the user namespace so that exit, exit(), quit or quit() are all valid ways to close the shell.

class IPython.core.autocall.ZMQExitAutocall(ip=None)
  Bases: IPython.core.autocall.ExitAutocall

  Exit IPython. Autocallable, so it needn’t be explicitly called.

  Parameters keep_kernel (bool) – If True, leave the kernel alive. Otherwise, tell the kernel to exit too (default).

8.5 Module: core.builtin_trap

A context manager for managing things injected into __builtins__.

Authors:
8.5.1 1 Class

class IPython.core.builtin_trap.BuiltinTrap (shell=None)
    Bases: traitlets.config.configurable.Configurable

    __init__ (shell=None)
    activate ()
        Store ipython references in the __builtin__ namespace.

    add_builtin (key, value)
        Add a builtin and save the original.

    deactivate ()
        Remove any builtins which might have been added by add_builtins, or restore overwritten ones to their previous values.

    remove_builtin (key, orig)
        Remove an added builtin and re-set the original.

8.6 Module: core.compilerop

Compiler tools with improved interactive support.

Provides compilation machinery similar to codeop, but with caching support so we can provide interactive tracebacks.

8.6.1 Authors

    • Robert Kern
    • Fernando Perez
    • Thomas Kluyver

8.6.2 1 Class

class IPython.core.compilerop.CachingCompiler
    Bases: codeop.Compile

    A compiler that caches code compiled from interactive statements.

    __init__ ()
ast_parse (source, filename='<unknown>', symbol='exec')

Parse code to an AST with the current compiler flags active.

Arguments are exactly the same as ast.parse (in the standard library), and are passed to the built-in compile function.

cache (code, number=0)

Make a name for a block of code, and cache the code.

Parameters

- code (str) – The Python source code to cache.
- number (int) – A number which forms part of the code’s name. Used for the execution counter.

Returns

- The name of the cached code (as a string). Pass this as the filename argument to compilation, so that tracebacks are correctly hooked up.

compiler_flags

Flags currently active in the compilation process.

reset_compiler_flags()

Reset compiler flags to default state.

8.6.3 2 Functions

IPython.core.compilerop.code_name (code, number=0)

Compute a (probably) unique name for code for caching.

This now expects code to be unicode.

IPython.core.compilerop.check_linecache_ipython (*args)

Call linecache.checkcache() safely protecting our cached values.

8.7 Module: core.completer

Word completion for IPython.

This module is a fork of the rlcompleter module in the Python standard library. The original enhancements made to rlcompleter have been sent upstream and were accepted as of Python 2.3, but we need a lot more functionality specific to IPython, so this module will continue to live as an IPython-specific utility.

Original rlcompleter documentation:

This requires the latest extension to the readline module (the completes keywords, built-ins and globals in __main__: when completing NAME.NAME..., it evaluates (!) the expression up to the last dot and completes its attributes.

It’s very cool to do “import string” type “string.”, hit the completion key (twice), and see the list of names defined by the string module!
Tip: to use the tab key as the completion key, call

```python
readline.parse_and_bind("tab: complete")
```

Notes:

- Exceptions raised by the completer function are ignored (and generally cause the completion to fail). This is a feature – since readline sets the tty device in raw (or cbreak) mode, printing a traceback wouldn’t work well without some complicated hoopla to save, reset and restore the tty state.

- The evaluation of the `NAME.NAME...` form may cause arbitrary application defined code to be executed if an object with a `__getattr__` hook is found. Since it is the responsibility of the application (or the user) to enable this feature, I consider this an acceptable risk. More complicated expressions (e.g. function calls or indexing operations) are not evaluated.

- GNU readline is also used by the built-in functions `input()` and `raw_input()`, and thus these also benefit/suffer from the completer features. Clearly an interactive application can benefit by specifying its own completer function and using `raw_input()` for all its input.

- When the original stdin is not a tty device, GNU readline is never used, and this module (and the readline module) are silently inactive.

### 8.7.1 3 Classes

```python
class CompletionSplitter delims=None
Bases: object

An object to split an input line in a manner similar to readline.

By having our own implementation, we can expose readline-like completion in a uniform manner to all frontends. This object only needs to be given the line of text to be split and the cursor position on said line, and it returns the ‘word’ to be completed on at the cursor after splitting the entire line.

What characters are used as splitting delimiters can be controlled by setting the `delims` attribute (this is a property that internally automatically builds the necessary regular expression)

```python
__init__(delims=None)
```

```python
delims
Return the string of delimiter characters.
```

```python
split_line(line, cursor_pos=None)
Split a line of text with a cursor at the given position.
```

```python
class Completer namespace=None, global_namespace=None, **kwargs
Bases: traitlets.config.configurable.Configurable

__init__(namespace=None, global_namespace=None, **kwargs)
Create a new completer for the command line.

Completer(namespace=ns,global_namespace=ns2) -> completer instance.

If unspecified, the default namespace where completions are performed is `__main__` (technically, `__main__.dict`). Namespaces should be given as dictionaries.
```
An optional second namespace can be given. This allows the completer to handle cases where both the local and global scopes need to be distinguished.

Completer instances should be used as the completion mechanism of readline via the set_completer() call:

```python
code = Completer(my_namespace).complete
```

### attr_matches (text)

Compute matches when text contains a dot.

Assuming the text is of the form `NAME.NAME...[NAME]`, and is evaluable in `self.namespace` or `self.global_namespace`, it will be evaluated and its attributes (as revealed by `dir()`) are used as possible completions. (For class instances, class members are also considered.)

**WARNING:** this can still invoke arbitrary C code, if an object with a `__getattr__` hook is evaluated.

### complete (text, state)

Return the next possible completion for `'text'`.

This is called successively with `state == 0, 1, 2, ...` until it returns `None`. The completion should begin with `text`.

### global_matches (text)

Compute matches when text is a simple name.

Return a list of all keywords, built-in functions and names currently defined in `self.namespace` or `self.global_namespace` that match.

### class IPython.core.completer.IPCompleter (shell=None, namespace=None, global_namespace=None, use_readline=True, config=None, **kwargs)

**Bases:** `IPython.core.completer.Completer`  

Extension of the completer class with IPython-specific features

```python
completer = IPCompleter() -> completer
```

Return a completer object suitable for use by the readline library via `set_completer()`.  

**Inputs:**

- `shell`: `None`.
- `namespace`: an optional dict where completions are performed.
- `global_namespace`: `None`.
- `use_readline`: `True`.
- `config`: `None`.
- `**kwargs`.

8.7. Module: core.completer
call about the current line.

all_completions(text)
Wrapper around the complete method for the benefit of emacs and pydb.

complete(text=None, line_buffer=None, cursor_pos=None)
Find completions for the given text and line context.

Note that both the text and the line_buffer are optional, but at least one of them must be given.

Parameters

- **text** (string, optional) – Text to perform the completion on. If not given, the line buffer is split using the instance’s CompletionSplitter object.
- **line_buffer** (string, optional) – If not given, the completer attempts to obtain the current line buffer via readline. This keyword allows clients which are requesting for text completions in non-readline contexts to inform the completer of the entire text.
- **cursor_pos** (int, optional) – Index of the cursor in the full line buffer. Should be provided by remote frontends where kernel has no access to frontends state.

Returns

- **text** (str) – Text that was actually used in the completion.
- **matches** (list) – A list of completion matches.

dict_key_matches(text)
Match string keys in a dictionary, after e.g. ‘foo’

file_matches(text)
Match filenames, expanding ~USER type strings.

Most of the seemingly convoluted logic in this completer is an attempt to handle filenames with spaces in them. And yet it’s not quite perfect, because Python’s readline doesn’t expose all of the GNU readline details needed for this to be done correctly.

For a filename with a space in it, the printed completions will be only the parts after what’s already been typed (instead of the full completions, as is normally done). I don’t think with the current (as of Python 2.3) Python readline it’s possible to do better.

latex_matches(text)
Match Latex syntax for unicode characters.

This does both alp -> alpha and alpha -> $\alpha$

Used on Python 3 only.

magic_matches(text)
Match magics

python_func_kw_matches(text)
Match named parameters (kwargs) of the last open function
python_matches (text)
    Match attributes or global python names

rlcomplete (text, state)
    Return the state-th possible completion for ‘text’.

    This is called successively with state == 0, 1, 2, ... until it returns None. The completion should
    begin with ‘text’.

    Parameters
    • text (string) – Text to perform the completion on.
    • state (int) – Counter used by readline.

unicode_name_matches (text)
    Match Latex-like syntax for unicode characters base on the name of the character.

    This does GREEK SMALL LETTER ETA -> \eta

    Works only on valid python 3 identifier, or on combining characters that will combine to form a
    valid identifier.

    Used on Python 3 only.

8.7.2 9 Functions

IPython.core.completer.has_open_quotes (s)
    Return whether a string has open quotes.

    This simply counts whether the number of quote characters of either type in the string is odd.

    Returns
    • If there is an open quote, the quote character is returned. Else, return
      False.

IPython.core.completer.protect_filename (s)
    Escape a string to protect certain characters.

IPython.core.completer.expand_user (path)
    Expand ‘~’-style usernames in strings.

    This is similar to os.path.expanduser(), but it computes and returns extra information that
    will be useful if the input was being used in computing completions, and you wish to return the
    completions with the original ‘~’ instead of its expanded value.

    Parameters path (str) – String to be expanded. If no ~ is present, the output is the
    same as the input.

    Returns
    • newpath (str) – Result of ~ expansion in the input path.
    • tilde_expanded (bool) – Whether any expansion was performed or not.
• `tilde_val (str)` – The value that ~ was replaced with.

IPython.core.completer.compress_user(path, tilde_expand, tilde_val)

Does the opposite of expand_user, with its outputs.

IPython.core.completer.penalize_magics_key(word)

key for sorting that penalizes magic commands in the ordering

Normal words are left alone.

Magic commands have the initial % moved to the end, e.g. %matplotlib is transformed as follows:

```
%matplotlib -> matplotlib%
```

[The choice of the final % is arbitrary.]

Since “matplotlib” < “matplotlib%” as strings, “timeit” will appear before the magic “%timeit” in the ordering

For consistency, move “%%” to the end, so cell magics appear after line magics with the same name.

A check is performed that there are no other “%” in the string; if there are, then the string is not a magic command and is left unchanged.

IPython.core.completer.get__all__entries(obj)

returns the strings in the __all__ attribute

IPython.core.completer.match_dict_keys(keys, prefix, delims)

Used by dict_key_matches, matching the prefix to a list of keys

IPython.core.completer.back_unicode_name_matches(text)

Match unicode characters back to unicode name

This does -> snowman

Note that snowman is not a valid python3 combining character but will be expanded. Though it will not recombine back to the snowman character by the completion machinery.

This will not either back-complete standard sequences like n, b ...

Used on Python 3 only.

IPython.core.completer.back_latex_name_matches(text)

Match latex characters back to unicode name

This does -> sqrt

Used on Python 3 only.

8.8 Module: core.completerlib

Implementations for various useful completers.

These are all loaded by default by IPython.
8.8.1 10 Functions

IPython.core.completerlib.module_list(path)
Return the list containing the names of the modules available in the given folder.

IPython.core.completerlib.get_root_modules()
Returns a list containing the names of all the modules available in the folders of the pythonpath.

    ip.db['rootmodules_cache'] maps sys.path entries to list of modules.

IPython.core.completerlib.is_importable(module, attr, only_modules)

IPython.core.completerlib.try_import(mod, only_modules=False)

IPython.core.completerlib.quick_completer(cmd, completions)
Easily create a trivial completer for a command.

    Takes either a list of completions, or all completions in string (that will be split on whitespace).

    Example:

    [d:\ipython]|1> import ipy_completers
    [d:\ipython]|2> ipy_completers.quick_completer('foo', ['bar','baz'])
    [d:\ipython]|3> foo b<TAB>
    bar baz
    [d:\ipython]|3> foo ba

IPython.core.completerlib.module_completion(line)
Returns a list containing the completion possibilities for an import line.

    The line looks like this: ‘import xml.d’ ‘from xml.dom import’

IPython.core.completerlib.module_completer(self, event)
Give completions after user has typed ‘import ...’ or ‘from ...’

IPython.core.completerlib.magic_run_completer(self, event)
Complete files that end in .py or .ipy or .ipynb for the %run command.

IPython.core.completerlib.cd_completer(self, event)
Completer function for cd, which only returns directories.

IPython.core.completerlib.reset_completer(self, event)
A completer for %reset magic

8.9 Module: core.crashhandler

sys.excepthook for IPython itself, leaves a detailed report on disk.

Authors:

    • Fernando Perez
    • Brian E. Granger
### 8.9.1 Class

```python
class IPython.core.crashhandler.CrashHandler(app,
                                           contact_name=None,
                                           contact_email=None,
                                           bug_tracker=None,
                                           show_crash_traceback=True,
                                           call_pdb=False)
```

**Bases:** `object`

Customizable crash handlers for IPython applications.

Instances of this class provide a `__call__()` method which can be used as a `sys.excepthook`. The `__call__()` signature is:

```python
def __call__(self, etype, evalue, etb)
```

**Parameters**

- `app` *(Application)* – A running `Application` instance, which will be queried at crash time for internal information.
- `contact_name` *(str)* – A string with the name of the person to contact.
- `contact_email` *(str)* – A string with the email address of the contact.
- `bug_tracker` *(str)* – A string with the URL for your project’s bug tracker.
- `show_crash_traceback` *(bool)* – If false, don’t print the crash traceback on stderr, only generate the on-disk report
- `instance attributes` *(Non-argument)* –
- `instances contain some non-argument attributes which allow for` *(These)* –
  - `customization of the crash handler’s behavior.
    Please see the` *(further)* –
  - `for further details` *(source)* –

```python
__call__(etype, evalue, etb)
```

Handle an exception, call for compatible with `sys.excepthook`

```python
make_report(traceback)
```

Return a string containing a crash report.

### 8.9.2 Function

```python
IPython.core.crashhandler.crash_handler_lite(etype, evalue, tb)
```

A light excepthook, adding a small message to the usual traceback
8.10 Module: core.debugger

Pdb debugger class.
Modified from the standard pdb.Pdb class to avoid including readline, so that the command line completion of other programs which include this isn’t damaged.

In the future, this class will be expanded with improvements over the standard pdb.
The code in this file is mainly lifted out of cmd.py in Python 2.2, with minor changes. Licensing should therefore be under the standard Python terms. For details on the PSF (Python Software Foundation) standard license, see:
http://www.python.org/2.2.3/license.html

8.10.1 2 Classes

```python
class IPython.core.debugger.Tracer(colors=None)
    Bases: object

    Class for local debugging, similar to pdb.set_trace.
    
    Instances of this class, when called, behave like pdb.set_trace, but providing IPython’s enhanced capabilities.
    
    This is implemented as a class which must be initialized in your own code and not as a standalone function because we need to detect at runtime whether IPython is already active or not. That detection is done in the constructor, ensuring that this code plays nicely with a running IPython, while functioning acceptably (though with limitations) if outside of it.

    __init__(colors=None)
    Create a local debugger instance.
    
    Parameters colors (str, optional) -- The name of the color scheme to use, it must be one of IPython’s valid color schemes. If not given, the function will default to the current IPython scheme when running inside IPython, and to ‘NoColor’ otherwise.
```

Examples

```python
from IPython.core.debugger import Tracer; debug_here = Tracer()

Later in your code:

download here()  # -> will open up the debugger at that point.
```

Once the debugger activates, you can use all of its regular commands to step through code, set breakpoints, etc. See the pdb documentation from the Python standard library for usage details.

```python
__call__()
Starts an interactive debugger at the point where called.
```
This is similar to the pdb.set_trace() function from the std lib, but using IPython’s enhanced debugger.

```python
class IPython.core.debugger.Pdb(
    color_scheme='NoColor',
    completekey=None,
    stdin=None, stdout=None, context=5
)
```

Bases: pdb.Pdb

Modified Pdb class, does not load readline.

```python
__init__(color_scheme='NoColor', completekey=None, stdin=None, stdout=None, context=5)
```

```python
down(*args, **kw)
    d(own) [count] Move the current frame count (default one) levels down in the stack trace (to a newer frame).
```

```python
down(*args, **kw)
    d(own) [count] Move the current frame count (default one) levels down in the stack trace (to a newer frame).
```

```python
d_pdef(arg)
    Print the call signature for any callable object.
    The debugger interface to %pdef
```

```python
d_pdoc(arg)
    Print the docstring for an object.
    The debugger interface to %pdoc.
```

```python
d_pfile(arg)
    Print (or run through pager) the file where an object is defined.
    The debugger interface to %pfile.
```

```python
d_pinfo(arg)
    Provide detailed information about an object.
    The debugger interface to %pinfo, i.e., obj?.
```

```python
d_pinfo2(arg)
    Provide extra detailed information about an object.
    The debugger interface to %pinfo2, i.e., obj??.
```

```python
d_psource(arg)
    Print (or run through pager) the source code for an object.
```

```python
d_q(*args, **kw)
    q uit exit
    Quit from the debugger. The program being executed is aborted.
```

```python
d_quit(*args, **kw)
    q uit exit
    Quit from the debugger. The program being executed is aborted.
```
do_u(*args, **kw)
    u(p) [count] Move the current frame count (default one) levels up in the stack trace (to an older frame).

do_up(*args, **kw)
    u(p) [count] Move the current frame count (default one) levels up in the stack trace (to an older frame).

do_w(arg)
    w(here) Print a stack trace, with the most recent frame at the bottom. An arrow indicates the “current frame”, which determines the context of most commands. ‘bt’ is an alias for this command.

    Take a number as argument as an (optional) number of context line to print

do_where(arg)
    w(here) Print a stack trace, with the most recent frame at the bottom. An arrow indicates the “current frame”, which determines the context of most commands. ‘bt’ is an alias for this command.

    Take a number as argument as an (optional) number of context line to print

list_command_pydb(arg)
    List command to use if we have a newer pydb installed

new_do_restart(arg)
    Restart command. In the context of ipython this is exactly the same thing as ‘quit’.

print_list_lines(filename, first, last)
    The printing (as opposed to the parsing part of a ‘list’ command.

set_colors(scheme)
    Shorthand access to the color table scheme selector method.

8.10.2 3 Functions

IPython.core.debugger.BdbQuit_excepthook(et, ev, tb, excepthook=None)
    Exception hook which handles BdbQuit exceptions.

    All other exceptions are processed using the excepthook parameter.

IPython.core.debugger.BdbQuit_IPython_excepthook(self, et, ev, tb, tb_offset=None)

IPython.core.debugger.decorate_fn_with_doc(new_fn, old_fn, additional_text='''
    Make new_fn have old_fn’s doc string. This is particularly useful for the do_. . . commands that hook into the help system. Adapted from from a comp.lang.python posting by Duncan Booth.

8.11 Module: core.error

Global exception classes for IPython.core.
8.11.1 5 Classes

**class** IPython.core.error.IPythonCoreError  
**Bases:** Exception  

**class** IPython.core.error.TryNext  
**Bases:** IPython.core.error.IPythonCoreError  
Try next hook exception.  
Raise this in your hook function to indicate that the next hook handler should be used to handle the operation.

**class** IPython.core.error.UsageError  
**Bases:** IPython.core.error.IPythonCoreError  
Error in magic function arguments, etc.  
Something that probably won’t warrant a full traceback, but should nevertheless interrupt a macro / batch file.

**class** IPython.core.error.StdinNotImplementedError  
**Bases:** IPython.core.error.IPythonCoreError, NotImplementedError  
raw_input was requested in a context where it is not supported  
For use in IPython kernels, where only some frontends may support stdin requests.

**class** IPython.core.error.InputRejected  
**Bases:** Exception  
Input rejected by ast transformer.  
Raise this in your NodeTransformer to indicate that InteractiveShell should not execute the supplied input.

8.12 Module: core.events

Infrastructure for registering and firing callbacks on application events.

Unlike IPython.core.hooks, which lets end users set single functions to be called at specific times, or a collection of alternative methods to try, callbacks are designed to be used by extension authors. A number of callbacks can be registered for the same event without needing to be aware of one another.
The functions defined in this module are no-ops indicating the names of available events and the arguments which will be passed to them.

Note: This API is experimental in IPython 2.0, and may be revised in future versions.

### 8.12.1 Class

```python
class IPython.core.events.EventManager(shell, available_events)

Bases: object

Manage a collection of events and a sequence of callbacks for each.

This is attached to `InteractiveShell` instances as an `events` attribute.

Note: This API is experimental in IPython 2.0, and may be revised in future versions.
```

```python
__init__(shell, available_events)

Initialise the CallbackManager.

Parameters

• **shell** – The `InteractiveShell` instance

• **available_callbacks** – An iterable of names for callback events.

register(event, function)

Register a new event callback

Parameters

• **event** (*str*) – The event for which to register this callback.

• **function** (*callable*) – A function to be called on the given event. It should take the same parameters as the appropriate callback prototype.

Raises

- **TypeError** – If function is not callable.

- **KeyError** – If event is not one of the known events.

trigger(event, *args, **kwargs)

Call callbacks for event.

Any additional arguments are passed to all callbacks registered for this event. Exceptions raised by callbacks are caught, and a message printed.

unregister(event, function)

Remove a callback from the given event.
```
8.12.2 5 Functions

IPython.core.events.pre_execute()
Fires before code is executed in response to user/frontend action.
This includes comm and widget messages and silent execution, as well as user code cells.

IPython.core.events.pre_run_cell()
Fires before user-entered code runs.

IPython.core.events.post_execute()
Fires after code is executed in response to user/frontend action.
This includes comm and widget messages and silent execution, as well as user code cells.

IPython.core.events.post_run_cell()
Fires after user-entered code runs.

IPython.core.events.shell_initialized(ip)
Fires after initialisation of InteractiveShell.
This is before extensions and startup scripts are loaded, so it can only be set by subclassing.

Parameters ip (InteractiveShell) – The newly initialised shell.

8.13 Module: core.excolors

Color schemes for exception handling code in IPython.

8.13.1 1 Class

class IPython.core.excolors.Deprec(wrapped_obj)
Bases: object
__init__(wrapped_obj)

8.13.2 1 Function

IPython.core.excolors.exception_colors()
Return a color table with fields for exception reporting.
The table is an instance of ColorSchemeTable with schemes added for ‘Linux’, ‘LightBG’ and ‘No-Color’ and fields for exception handling filled in.

Examples:
>>> ec = exception_colors()
>>> ec.active_scheme_name''
>>> print(ec.active_colors)
None
Now we activate a color scheme: >>> ec.set_active_scheme('NoColor') >>> ec.active_scheme_name 'NoColor' >>> sorted(ec.active_colors.keys()) ['Normal', 'caret', 'em', 'excName', 'filename', 'filenameEm', 'line', 'lineno', 'linenoEm', 'name', 'nameEm', 'normalEm', 'topline', 'vName', 'val', 'valEm']

8.14 Module: core.extensions

A class for managing IPython extensions.

8.14.1 1 Class

class IPython.core.extensions.ExtensionManager (shell=None, **kwargs)

Bases: traitlets.config.configurable.Configurable

A class to manage IPython extensions.

An IPython extension is an importable Python module that has a function with the signature:

    def load_ipython_extension(ipython):
        # Do things with ipython

This function is called after your extension is imported and the currently active InteractiveShell instance is passed as the only argument. You can do anything you want with IPython at that point, including defining new magic and aliases, adding new components, etc.

You can also optionally define an unload_ipython_extension(ipython)() function, which will be called if the user unloads or reloads the extension. The extension manager will only call load_ipython_extension() again if the extension is reloaded.

You can put your extension modules anywhere you want, as long as they can be imported by Python’s standard import mechanism. However, to make it easy to write extensions, you can also put your extensions in os.path.join(self.ipython_dir, 'extensions'). This directory is added to sys.path automatically.

__init__ (shell=None, **kwargs)

install_extension (url, filename=None)

Download and install an IPython extension.

If filename is given, the file will be so named (inside the extension directory). Otherwise, the name from the URL will be used. The file must have a .py or .zip extension; otherwise, a ValueError will be raised.

Returns the full path to the installed file.

load_extension (module_str)

Load an IPython extension by its module name.

Returns the string “already loaded” if the extension is already loaded, “no load function” if the module doesn’t have a load_ipython_extension function, or None if it succeeded.
reload_extension (module_str)
Reload an IPython extension by calling reload.

If the module has not been loaded before, InteractiveShell.load_extension() is called. Otherwise reload() is called and then the load_ipython_extension() function of the module, if it exists is called.

unload_extension (module_str)
Unload an IPython extension by its module name.

This function looks up the extension’s name in sys.modules and simply calls mod.unload_ipython_extension(self).

Returns the string “no unload function” if the extension doesn’t define a function to unload itself, “not loaded” if the extension isn’t loaded, otherwise None.

8.15 Module: core.formatters
Display formatters.

Inheritance diagram:

8.15.1 15 Classes

class IPython.core.formatters.DisplayFormatter(**kwargs)
Bases: traitlets.config.configurable.Configurable

format (obj, include=None, exclude=None)
Return a format data dict for an object.
By default all format types will be computed.

The following MIME types are currently implemented:

- text/plain
- text/html
- text/markdown
- text/latex
- application/json
- application/javascript
- application/pdf
- image/png
- image/jpeg
- image/svg+xml

Parameters

- **obj** *(object)* – The Python object whose format data will be computed.
- **include** *(list or tuple, optional)* – A list of format type strings (MIME types) to include in the format data dict. If this is set only the format types included in this list will be computed.
- **exclude** *(list or tuple, optional)* – A list of format type string (MIME types) to exclude in the format data dict. If this is set all format types will be computed, except for those included in this argument.

Returns

*(format_dict, metadata_dict)* –

format_dict is a dictionary of key/value pairs, one of each format that was generated for the object. The keys are the format types, which will usually be MIME type strings and the values and JSON’able data structure containing the raw data for the representation in that format.

metadata_dict is a dictionary of metadata about each mime-type output. Its keys will be a strict subset of the keys in format_dict.

Return type tuple of two dicts

**format_types**

Return the format types (MIME types) of the active formatters.

class IPython.core.formatters.FormatterWarning

Bases: UserWarning

Warning class for errors in formatters
class IPython.core.formatters.FormatterABC
    Bases: abc._NewBase

Abstract base class for Formatters.

A formatter is a callable class that is responsible for computing the raw format data for a particular format type (MIME type). For example, an HTML formatter would have a format type of text/html and would return the HTML representation of the object when called.

    __call__(obj)
    Return a JSON’able representation of the object.

    If the object cannot be formatted by this formatter, warn and return None.

class IPython.core.formatters.BaseFormatter(**kwargs)
    Bases: traitlets.config.configurable.Configurable

A base formatter class that is configurable.

This formatter should usually be used as the base class of all formatters. It is a traited Configurable class and includes an extensible API for users to determine how their objects are formatted. The following logic is used to find a function to format an given object.

1. The object is introspected to see if it has a method with the name print_method. If it does, that object is passed to that method for formatting.

2. If no print method is found, three internal dictionaries are consulted to find print method: singleton_printers, type_printers and deferred_printers.

Users should use these dictionaries to register functions that will be used to compute the format data for their objects (if those objects don’t have the special print methods). The easiest way of using these dictionaries is through the for_type() and for_type_by_name() methods.

If no function/callable is found to compute the format data, None is returned and this format type is not used.

    __contains__(typ)
    map in to lookup_by_type

for_type (typ, func=None)
    Add a format function for a given type.

Parameters

    • typ (type or '__module__.__name__' string for a type) –
        The class of the object that will be formatted using func.

    • func (callable) – A callable for computing the format data. func will be called with the object to be formatted, and will return the raw data in this formatter’s format. Subclasses may use a different call signature for the func argument.

        If func is None or not specified, there will be no change, only returning the current value.

Returns oldfunc – The currently registered callable. If you are registering a new formatter, this will be the previous value (to enable restoring later).
Return type: callable

**for_type_by_name**(type_module, type_name, func=None)

Add a format function for a type specified by the full dotted module and name of the type, rather than the type of the object.

Parameters

- **type_module**(str) – The full dotted name of the module the type is defined in, like numpy.
- **type_name**(str) – The name of the type (the class name), like dtype
- **func**(callable) – A callable for computing the format data. func will be called with the object to be formatted, and will return the raw data in this formatter's format. Subclasses may use a different call signature for the func argument.

If func is None or unspecified, there will be no change, only returning the current value.

Returns oldfunc – The currently registered callable. If you are registering a new formatter, this will be the previous value (to enable restoring later).

Return type: callable

**lookup**(obj)

Look up the formatter for a given instance.

Parameters **obj**(object instance) –

Returns f – The registered formatting callable for the type.

Return type: callable

Raises: KeyError if the type has not been registered.

**lookup_by_type**(typ)

Look up the registered formatter for a type.

Parameters **typ**(type or '__module__.__name__' string for a type) –

Returns f – The registered formatting callable for the type.

Return type: callable

Raises: KeyError if the type has not been registered.

**pop**(typ, default=IPython.core.formatters._raise_key_error)

Pop a formatter for the given type.

Parameters

- **typ**(type or '__module__.__name__' string for a type) –
- **default**(object) – value to be returned if no formatter is registered for typ.

Returns obj – The last registered object for the type.
Return type object

Raises KeyError if the type is not registered and default is not specified.

class IPython.core.formatters.PlainTextFormatter(**kwargs)

Bases: IPython.core.formatters.BaseFormatter

The default pretty-printer.

This uses IPython.lib.pretty to compute the format data of the object. If the object cannot be pretty printed, repr() is used. See the documentation of IPython.lib.pretty for details on how to write pretty printers. Here is a simple example:

```python
def dtype_pprinter(obj, p, cycle):
    if cycle:
        return p.text('dtype(...)')
    if hasattr(obj, 'fields'):
        if obj.fields is None:
            p.text(repr(obj))
        else:
            p.begin_group(7, 'dtype([')
            for i, field in enumerate(obj.descr):
                if i > 0:
                    p.text(',
                p.breakable()
                p.pretty(field)
            p.end_group(7, ']'))
```

class IPython.core.formatters.HTMLFormatter(**kwargs)

Bases: IPython.core.formatters.BaseFormatter

An HTML formatter.

To define the callables that compute the HTML representation of your objects, define a _repr_html_() method or use the for_type() or for_type_by_name() methods to register functions that handle this.

The return value of this formatter should be a valid HTML snippet that could be injected into an existing DOM. It should not include the '<html>' or '<body>' tags.

class IPython.core.formatters.MarkdownFormatter(**kwargs)

Bases: IPython.core.formatters.BaseFormatter

A Markdown formatter.

To define the callables that compute the Markdown representation of your objects, define a _repr_markdown_() method or use the for_type() or for_type_by_name() methods to register functions that handle this.

The return value of this formatter should be a valid Markdown.

class IPython.core.formatters.SVGFormatter(**kwargs)

Bases: IPython.core.formatters.BaseFormatter

An SVG formatter.
To define the callables that compute the SVG representation of your objects, define a **_repr_svg_**() method or use the for_type() or for_type_by_name() methods to register functions that handle this.

The return value of this formatter should be valid SVG enclosed in `<svg>` tags, that could be injected into an existing DOM. It should not include the `<html>` or `<body>` tags.

```python
class IPython.core.formatters.PNGFormatter(**kwargs)
Bases: IPython.core.formatters.BaseFormatter
A PNG formatter.
To define the callables that compute the PNG representation of your objects, define a **_repr_png_**() method or use the for_type() or for_type_by_name() methods to register functions that handle this.

The return value of this formatter should be raw PNG data, not base64 encoded.
```

```python
class IPython.core.formatters.JPEGFormatter(**kwargs)
Bases: IPython.core.formatters.BaseFormatter
A JPEG formatter.
To define the callables that compute the JPEG representation of your objects, define a **_repr_jpeg_**() method or use the for_type() or for_type_by_name() methods to register functions that handle this.

The return value of this formatter should be raw JPEG data, not base64 encoded.
```

```python
class IPython.core.formatters.LatexFormatter(**kwargs)
Bases: IPython.core.formatters.BaseFormatter
A LaTeX formatter.
To define the callables that compute the LaTeX representation of your objects, define a **_repr_latex_**() method or use the for_type() or for_type_by_name() methods to register functions that handle this.

The return value of this formatter should be a valid LaTeX equation, enclosed in either `$`, `$$` or another LaTeX equation environment.
```

```python
class IPython.core.formatters.JSONFormatter(**kwargs)
Bases: IPython.core.formatters.BaseFormatter
A JSON string formatter.
To define the callables that compute the JSONable representation of your objects, define a **_repr_json_**() method or use the for_type() or for_type_by_name() methods to register functions that handle this.

The return value of this formatter should be a JSONable list or dict. JSON scalars (None, number, string) are not allowed, only dict or list containers.
```

```python
class IPython.core.formatters.JavascriptFormatter(**kwargs)
Bases: IPython.core.formatters.BaseFormatter
A Javascript formatter.
```
To define the callables that compute the Javascript representation of your objects, define a 

```python
__repr_javascript__()
```

method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

The return value of this formatter should be valid Javascript code and should *not* be enclosed in `<script>` tags.

```python
class IPython.core.formatters.PDFFormatter(**kwargs)
Bases: IPython.core.formatters.BaseFormatter
```

A PDF formatter.

To define the callables that compute the PDF representation of your objects, define a `__repr_pdf__()` method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this.

The return value of this formatter should be raw PDF data, *not* base64 encoded.

```python
class IPython.core.formatters.IPYthonDisplayFormatter(**kwargs)
Bases: IPython.core.formatters.BaseFormatter
```

A Formatter for objects that know how to display themselves.

To define the callables that compute the representation of your objects, define a 

```python
__ipython_display__()
```

method or use the `for_type()` or `for_type_by_name()` methods to register functions that handle this. Unlike mime-type displays, this method should not return anything, instead calling any appropriate display methods itself.

This display formatter has highest priority. If it fires, no other display formatter will be called.

### 8.15.2 2 Functions

```python
IPython.core.formatters.catch_format_error(method, self, *args, **kwargs)
```

Show traceback on failed format call

```python
IPython.core.formatters.format_display_data(obj, include=None, exclude=None)
```

Return a format data dict for an object.

By default all format types will be computed.

The following MIME types are currently implemented:

- `text/plain`
- `text/html`
- `text/markdown`
- `text/latex`
- `application/json`
- `application/javascript`
- `application/pdf`
• image/png
• image/jpeg
• image/svg+xml

Parameters obj (object) – The Python object whose format data will be computed.

Returns

- format_dict (dict) – A dictionary of key/value pairs, one or each format that was generated for the object. The keys are the format types, which will usually be MIME type strings and the values and JSON’able data structure containing the raw data for the representation in that format.

- include (list or tuple, optional) – A list of format type strings (MIME types) to include in the format data dict. If this is set only the format types included in this list will be computed.

- exclude (list or tuple, optional) – A list of format type string (MIME types) to exclude in the format data dict. If this is set all format types will be computed, except for those included in this argument.

8.16 Module: core.getipython

Simple function to call to get the current InteractiveShell instance

8.16.1 1 Function

IPython.core.getipython.get_ipython()

Get the global InteractiveShell instance.

Returns None if no InteractiveShell instance is registered.

8.17 Module: core.history

History related magics and functionality

8.17.1 4 Classes

class IPython.core.history.HistoryAccessorBase (**kwargs)
Bases: traitlets.config.configurable.LoggingConfigurable

An abstract class for History Accessors

class IPython.core.history.HistoryAccessor (profile='default', hist_file='; **traits)
Bases: IPython.core.history.HistoryAccessorBase

Access the history database without adding to it.
This is intended for use by standalone history tools. IPython shells use HistoryManager, below, which is a subclass of this.

```
__init__(profile='default', hist_file='', **traits)

Create a new history accessor.
```

**Parameters**

- `profile (str)` – The name of the profile from which to open history.
- `hist_file (str)` – Path to an SQLite history database stored by IPython. If specified, hist_file overrides profile.
- `config (Config)` – Config object. hist_file can also be set through this.

```
get_last_session_id()

Get the last session ID currently in the database.
```

Within IPython, this should be the same as the value stored in HistoryManager.session_number.

```
get_range (session, start=1, stop=None, raw=True, output=False)

Retrieve input by session.
```

**Parameters**

- `session (int)` – Session number to retrieve.
- `start (int)` – First line to retrieve.
- `stop (int)` – End of line range (excluded from output itself). If None, retrieve to the end of the session.
- `raw (bool)` – If True, return untranslated input
- `output (bool)` – If True, attempt to include output. This will be ‘real’ Python objects for the current session, or text reprs from previous sessions if db_log_output was enabled at the time. Where no output is found, None is used.

**Returns**

An iterator over the desired lines. Each line is a 3-tuple, either (session, line, input) if output is False, or (session, line, (input, output)) if output is True.

```
get_range_by_str (rangestr, raw=True, output=False)

Get lines of history from a string of ranges, as used by magic commands %hist, %save, %macro, etc.
```

**Parameters**

- `rangestr (str)` – A string specifying ranges, e.g. “5 ~2/1-4”. See magic_history() for full details.
- `output (raw,)` – As get_range()

**Returns**

```
Return type  Tuples as get_range()
```
get_session_info(session)
    Get info about a session.

    Parameters session (int) – Session number to retrieve.

    Returns
    • session_id (int) – Session ID number
    • start (datetime) – Timestamp for the start of the session.
    • end (datetime) – Timestamp for the end of the session, or None if IPython crashed.
    • num_cmds (int) – Number of commands run, or None if IPython crashed.
    • remark (unicode) – A manually set description.

get_tail(n=10, raw=True, output=False, include_latest=False)
    Get the last n lines from the history database.

    Parameters
    • n (int) – The number of lines to get
    • output (raw,) – See get_range()
    • include_latest (bool) – If False (default), n+1 lines are fetched, and the latest one is discarded. This is intended to be used where the function is called by a user command, which it should not return.

    Returns
    Return type Tuples as get_range()

init_db()
    Connect to the database, and create tables if necessary.

search(pattern='*', raw=True, search_raw=True, output=False, n=None, unique=False)
    Search the database using unix glob-style matching (wildcards * and ?).

    Parameters
    • pattern (str) – The wildcarded pattern to match when searching
    • search_raw (bool) – If True, search the raw input, otherwise, the parsed input
    • output (raw,) – See get_range()
    • n (None or int) – If an integer is given, it defines the limit of returned entries.
    • unique (bool) – When it is true, return only unique entries.

    Returns
    Return type Tuples as get_range()
writeout_cache()
    Overridden by HistoryManager to dump the cache before certain database lookups.

class IPython.core.history.HistoryManager(shell=None, config=None, **traits)
    Bases: IPython.core.history.HistoryAccessor

    A class to organize all history-related functionality in one place.

    __init__(shell=None, config=None, **traits)
    Create a new history manager associated with a shell instance.

def end_session()
    Close the database session, filling in the end time and line count.

def get_range(session=0, start=1, stop=None, raw=True, output=False)
    Retrieve input by session.

    Parameters
    • session (int) – Session number to retrieve. The current session is 0, and 
      negative numbers count back from current session, so -1 is previous session.
    • start (int) – First line to retrieve.
    • stop (int) – End of line range (excluded from output itself). If None, retrieve 
      to the end of the session.
    • raw (bool) – If True, return untranslated input
    • output (bool) – If True, attempt to include output. This will be ‘real’ 
      Python objects for the current session, or text reprs from previous sessions if 
      db_log_output was enabled at the time. Where no output is found, None is 
      used.

    Returns
    An iterator over the desired lines. Each line is a 3-tuple, either (session, line, 
    input) if output is False, or (session, line, (input, output)) if output is True.

    Return type entries

def get_session_info(session=0)
    Get info about a session.

    Parameters
    session (int) – Session number to retrieve. The current session is 0, and 
    negative numbers count back from current session, so -1 is the previous session.

    Returns
    • session_id (int) – Session ID number
    • start (datetime) – Timestamp for the start of the session.
    • end (datetime) – Timestamp for the end of the session, or None if IPython 
      crashed.
    • num_cmds (int) – Number of commands run, or None if IPython crashed.
    • remark (unicode) – A manually set description.
name_session(name)
    Give the current session a name in the history database.

ew_session(conn=None)
    Get a new session number.
reset(new_session=True)
    Clear the session history, releasing all object references, and optionally open a new session.
store_inputs(line_num, source, source_raw=None)
    Store source and raw input in history and create input cache variables _i*.
    Parameters
        • line_num(int) – The prompt number of this input.
        • source(str) – Python input.
        • source_raw(str, optional) – If given, this is the raw input without any IPython transformations applied to it. If not given, source is used.
store_output(line_num)
    If database output logging is enabled, this saves all the outputs from the indicated prompt number to the database. It’s called by run_cell after code has been executed.
    Parameters
        line_num(int) – The line number from which to save outputs
writeout_cache(conn=None)
    Write any entries in the cache to the database.
class IPython.core.history.HistorySavingThread(history_manager)
    Bases: threading.Thread
    This thread takes care of writing history to the database, so that the UI isn’t held up while that happens. It waits for the HistoryManager’s save_flag to be set, then writes out the history cache. The main thread is responsible for setting the flag when the cache size reaches a defined threshold.
    __init__(history_manager)
    stop()
        This can be called from the main thread to safely stop this thread.
        Note that it does not attempt to write out remaining history before exiting. That should be done by calling the HistoryManager’s end_session method.

8.17.2 3 Functions
IPython.core.history.needs_sqlite(f, self, *a, **kw)
    Decorator: return an empty list in the absence of sqlite.
IPython.core.history.catch_corrupt_db(f, self, *a, **kw)
    A decorator which wraps HistoryAccessor method calls to catch errors from a corrupt SQLite database, move the old database out of the way, and create a new one.
IPython Documentation, Release 4.2.1

IPython.core.history.extract_hist_ranges(ranges_str)
   Turn a string of history ranges into 3-tuples of (session, start, stop).

   Examples

   >>> list(extract_hist_ranges("-8/5--7/4 2"))
   [(-8, 5, None), (-7, 1, 5), (0, 2, 3)]

8.18 Module: core.historyapp

An application for managing IPython history.

To be invoked as the ipython history subcommand.

8.18.1 3 Classes

class IPython.core.historyapp.HistoryTrim(**kwargs)
   Bases: IPython.core.application.BaseIPythonApplication

class IPython.core.historyapp.HistoryClear(**kwargs)
   Bases: IPython.core.historyapp.HistoryTrim

class IPython.core.historyapp.HistoryApp(**kwargs)
   Bases: traitlets.config.application.Application

8.19 Module: core.hooks

Hooks for IPython.

In Python, it is possible to overwrite any method of any object if you really want to. But IPython exposes a few ‘hooks’, methods which are designed to be overwritten by users for customization purposes. This module defines the default versions of all such hooks, which get used by IPython if not overridden by the user.

Hooks are simple functions, but they should be declared with self as their first argument, because when activated they are registered into IPython as instance methods. The self argument will be the IPython running instance itself, so hooks have full access to the entire IPython object.

If you wish to define a new hook and activate it, you can make an extension or a startup script. For example, you could use a startup file like this:

```python
import os

def calljed(self, filename, linenum):
    "My editor hook calls the jed editor directly."
    print "Calling my own editor, jed ..."
    if os.system('jed +%d %s' % (linenum, filename)) != 0:
      ...
```
raise TryNext()

def load_ipython_extension(ip):
    ip.set_hook('editor', calljed)

8.19.1 1 Class

class IPython.core.hooks.CommandChainDispatcher(commands=None)
    Bases: object
    Dispatch calls to a chain of commands until some func can handle it
    Usage: instantiate, execute “add” to add commands (with optional priority), execute normally via f() calling mechanism.
    __init__(commands=None)
    __call__(*args, **kw)
        Command chain is called just like normal func.
        This will call all funcs in chain with the same args as were given to this function, and return the result of first func that didn’t raise TryNext
    __iter__()
        Return all objects in chain.
        Handy if the objects are not callable.
    add(func, priority=0)
        Add a func to the cmd chain with given priority

8.19.2 9 Functions

IPython.core.hooks.editor(self, filename, linenum=None, wait=True)
    Open the default editor at the given filename and linenum.
    This is IPython’s default editor hook, you can use it as an example to write your own modified one. To set your own editor function as the new editor hook, call ip.set_hook(‘editor’, yourfunc).

IPython.core.hooks.fix_error_editor(self, filename, linenum, column, msg)
    Open the editor at the given filename, linenum, column and show an error message. This is used for correcting syntax errors. The current implementation only has special support for the VIM editor, and falls back on the ‘editor’ hook if VIM is not used.
    Call ip.set_hook(‘fix_error_editor’, yourfunc) to use your own function,

IPython.core.hooks.synchronize_with_editor(self, filename, linenum, column)

IPython.core.hooks.shutdown_hook(self)
    default shutdown hook
    Typically, shutdown hooks should raise TryNext so all shutdown ops are done
IPython.core.hooks.late_startup_hook(self)
Executed after ipython has been constructed and configured

IPython.core.hooks.show_in_pager(self, data, start, screen_lines)
Run a string through pager

IPython.core.hooks.pre_prompt_hook(self)
Run before displaying the next prompt
Use this e.g. to display output from asynchronous operations (in order to not mess up text entry)

IPython.core.hooks.pre_run_code_hook(self)
Executed before running the (prefiltered) code in IPython

IPython.core.hooks.clipboard_get(self)
Get text from the clipboard.

8.20 Module: core.inputsplitter

Input handling and transformation machinery.

The first class in this module, InputSplitter, is designed to tell when input from a line-oriented frontend is complete and should be executed, and when the user should be prompted for another line of code instead. The name ‘input splitter’ is largely for historical reasons.

A companion, IPythonInputSplitter, provides the same functionality but with full support for the extended IPython syntax (magics, system calls, etc). The code to actually do these transformations is in IPython.core.inputtransformer. IPythonInputSplitter feeds the raw code to the transformers in order and stores the results.

For more details, see the class docstrings below.

8.20.1 2 Classes

class IPython.core.inputsplitter.InputSplitter
Bases: object

An object that can accumulate lines of Python source before execution.

This object is designed to be fed python source line-by-line, using push(). It will return on each push whether the currently pushed code could be executed already. In addition, it provides a method called push_accepts_more() that can be used to query whether more input can be pushed into a single interactive block.

This is a simple example of how an interactive terminal-based client can use this tool:

isp = InputSplitter()
while isp.push_accepts_more():
    indent = ' ' * isp.indent_spaces
    prompt = '>>> ' + indent
    line = indent + raw_input(prompt)
isp.push(line)
print 'Input source was:\n', isp.source_reset(),

__init__()  
Create a new InputSplitter instance.

check_complete(source)  
Return whether a block of code is ready to execute, or should be continued

Parameters source (string) – Python input code, which can be multilne.

Returns

• status (str) – One of ‘complete’, ‘incomplete’, or ‘invalid’ if source is not a
  prefix of valid code.

• indent_spaces (int or None) – The number of spaces by which to indent the
  next line of code. If status is not ‘incomplete’, this is None.

push(lines)  
Push one or more lines of input.

This stores the given lines and returns a status code indicating whether the code forms a complete
Python block or not.

Any exceptions generated in compilation are swallowed, but if an exception was produced, the
method returns True.

Parameters lines (string) – One or more lines of Python input.

Returns is_complete – True if the current input source (the result of the current input
plus prior inputs) forms a complete Python execution block. Note that this value
is also stored as a private attribute (_is_complete), so it can be queried at any
time.

Return type boolean

push_accepts_more()  
Return whether a block of interactive input can accept more input.

This method is meant to be used by line-oriented frontends, who need to guess whether a block
is complete or not based solely on prior and current input lines. The InputSplitter considers it
has a complete interactive block and will not accept more input when either:

• A SyntaxError is raised

• The code is complete and consists of a single line or a single non-compound statement

• The code is complete and has a blank line at the end

If the current input produces a syntax error, this method immediately returns False but does not
raise the syntax error exception, as typically clients will want to send invalid syntax to
an execution backend which might convert the invalid syntax into valid Python via one of the
dynamic IPython mechanisms.
reset ()
Reset the input buffer and associated state.

source_reset ()
Return the input source and perform a full reset.

class IPython.core.inputsplitter.IPythonInputSplitter (line_input_checker=True,
physical_line_transforms=None,
logical_line_transforms=None,
python_line_transforms=None)

Bases: IPython.core.inputsplitter.InputSplitter
An input splitter that recognizes all of IPython’s special syntax.

__init__ (line_input_checker=True,
physical_line_transforms=None,
logical_line_transforms=None,
python_line_transforms=None)

push (lines)
Push one or more lines of IPython input.

This stores the given lines and returns a status code indicating whether the code forms a complete
Python block or not, after processing all input lines for special IPython syntax.

Any exceptions generated in compilation are swallowed, but if an exception was produced, the
method returns True.

Parameters lines (string) – One or more lines of Python input.

Returns is_complete – True if the current input source (the result of the current input
plus prior inputs) forms a complete Python execution block. Note that this value is
also stored as a private attribute (_is_complete), so it can be queried at any time.

Return type boolean

raw_reset ()
Return raw input only and perform a full reset.

reset ()
Reset the input buffer and associated state.

transform_cell (cell)
Process and translate a cell of input.

transforms
Quick access to all transformers.

transforms_in_use
Transformers, excluding logical line transformers if we’re in a Python line.

8.20.2 5 Functions

IPython.core.inputsplitter.num_init_spaces (s)
Return the number of initial spaces in a string.
Note that tabs are counted as a single space. For now, we do not support mixing of tabs and spaces in the user’s input.

**Parameters** \( s \) (string) –

**Returns** \( n \)

**Return type** int

IPython.core.inputsplitter.last_blank(src)
Determine if the input source ends in a blank.

A blank is either a newline or a line consisting of whitespace.

**Parameters** src (string) – A single or multiline string.

IPython.core.inputsplitter.last_two_blanks(src)
Determine if the input source ends in two blanks.

A blank is either a newline or a line consisting of whitespace.

**Parameters** src (string) – A single or multiline string.

IPython.core.inputsplitter.remove_comments(src)
Remove all comments from input source.

Note: comments are NOT recognized inside of strings!

**Parameters** src (string) – A single or multiline input string.

**Returns**

**Return type** String with all Python comments removed.

IPython.core.inputsplitter.get_input_encoding()
Return the default standard input encoding.

If sys.stdin has no encoding, ‘ascii’ is returned.

### 8.21 Module: core.inputtransformer

Input transformer classes to support IPython special syntax.

This includes the machinery to recognise and transform %magic commands, !system commands, help? querying, prompt stripping, and so forth.

#### 8.21.1 5 Classes

**class** IPython.core.inputtransformer.InputTransformer

**Bases:** abc._NewBase

Abstract base class for line-based input transformers.
push (line)
Send a line of input to the transformer, returning the transformed input or None if the transformer
is waiting for more input.
Must be overridden by subclasses.
Implementations may raise SyntaxError if the input is invalid. No other exceptions may be
raised.
reset ()
Return, transformed any lines that the transformer has accumulated, and reset its internal state.
Must be overridden by subclasses.
classmethod wrap (func)
Can be used by subclasses as a decorator, to return a factory that will allow instantiation with
the decorated object.
class IPython.core.inputtransformer.StatelessInputTransformer (func)
Bases: IPython.core.inputtransformer.InputTransformer
Wrapper for a stateless input transformer implemented as a function.
__init__ (func)
push (line)
Send a line of input to the transformer, returning the transformed input.
reset ()
No-op - exists for compatibility.
class IPython.core.inputtransformer.CoroutineInputTransformer (coro,
**kwargs)
Bases: IPython.core.inputtransformer.InputTransformer
Wrapper for an input transformer implemented as a coroutine.
__init__ (coro, **kwargs)
push (line)
Send a line of input to the transformer, returning the transformed input or None if the transformer
is waiting for more input.
reset ()
Return, transformed any lines that the transformer has accumulated, and reset its internal state.
class IPython.core.inputtransformer.TokenInputTransformer (func)
Bases: IPython.core.inputtransformer.InputTransformer
Wrapper for a token-based input transformer.
func should accept a list of tokens (5-tuples, see tokenize docs), and return an iterable which can be
passed to tokenize.untokenize().
__init__ (func)
class IPython.core.inputtransformer.assemble_python_lines
Bases: IPython.core.inputtransformer.TokenInputTransformer
8.21.2 12 Functions

IPython.core.inputtransformer.assemble_logical_lines()
Join lines following explicit line continuations ()

IPython.core.inputtransformer.escaped_commands(line)
Transform escaped commands - %magic, !system, ?help + various autocalls.

IPython.core.inputtransformer.has_comment(src)
Indicate whether an input line has (i.e. ends in, or is) a comment.
This uses tokenize, so it can distinguish comments from # inside strings.

Parameters src (string) – A single line input string.

Returns comment – True if source has a comment.

Return type bool

IPython.core.inputtransformer.ends_in_comment_or_string(src)
Indicates whether or not an input line ends in a comment or within a multiline string.

Parameters src (string) – A single line input string.

Returns comment – True if source ends in a comment or multiline string.

Return type bool

IPython.core.inputtransformer.help_end(line)
Translate lines with ?/?? at the end

IPython.core.inputtransformer.cellmagic(end_on_blank_line=False)
Captures & transforms cell magics.

After a cell magic is started, this stores up any lines it gets until it is reset (sent None).

IPython.core.inputtransformer.classic_prompt()
Strip the >>>/... prompts of the Python interactive shell.

IPython.core.inputtransformer.ipy_prompt()
Strip IPython’s In [1]:/...: prompts.

IPython.core.inputtransformer.leading_indent()
Remove leading indentation.

If the first line starts with a spaces or tabs, the same whitespace will be removed from each following line until it is reset.

IPython.core.inputtransformer.strip_encoding_cookie()
Remove encoding comment if found in first two lines

If the first or second line has the # coding: utf-8 comment, it will be removed.

IPython.core.inputtransformer.assign_from_system(line)
Transform assignment from system commands (e.g. files = !ls)
IPython.core.inputtransformer.assign_from_magic(line)
    Transform assignment from magic commands (e.g. a = %who_ls)

8.22 Module: core.interactiveshell

Main IPython class.

8.22.1 5 Classes

class IPython.core.interactiveshell.SpaceInInput
    Bases: Exception

class IPython.core.interactiveshell.SeparateUnicode
    Bases: traitlets.traitlets.Unicode
    A Unicode subclass to validate separate_in, separate_out, etc.
    This is a Unicode based trait that converts '0'->” and ‘\n’->’\n’.

class IPython.core.interactiveshell.ExecutionResult
    Bases: object
    The result of a call to InteractiveShell.run_cell()
    Stores information about what took place.
    raise_error()
        Reraises error if success is False, otherwise does nothing

class IPython.core.interactiveshell.InteractiveShell
    Bases: traitlets.config.configurable.SingletonConfigurable
    An enhanced, interactive shell for Python.

    __init__ (ipython_dir=None, profile_dir=None, user_module=None, user_ns=None, custom_exceptions=(), **kwargs)

    all_ns_refs
        Get a list of references to all the namespace dictionaries in which IPython might store a user-
        created object.

        Note that this does not include the displayhook, which also caches objects from the output.
atexit_operations ()
This will be executed at the time of exit.

Cleanup operations and saving of persistent data that is done unconditionally by IPython should be performed here.

For things that may depend on startup flags or platform specifics (such as having readline or not), register a separate atexit function in the code that has the appropriate information, rather than trying to clutter

auto_rewrite_input (cmd)
Print to the screen the rewritten form of the user’s command.

This shows visual feedback by rewriting input lines that cause automatic calling to kick in, like:

```bash
/ f x
```
into:

```bash
------> f(x)
```

after the user’s input prompt. This helps the user understand that the input line was transformed automatically by IPython.

call_pdb
Control auto-activation of pdb at exceptions

clear_main_mod_cache ()
Clear the cache of main modules.
Mainly for use by utilities like %reset.

Examples

In [15]: import IPython
In [16]: m = _ip.new_main_mod(IPython.__file__, 'IPython')
In [17]: len(_ip._main_mod_cache) > 0 Out[17]: True
In [18]: _ip.clear_main_mod_cache()
In [19]: len(_ip._main_mod_cache) == 0 Out[19]: True

complete (text, line=None, cursor_pos=None)
Return the completed text and a list of completions.

Parameters

- **text (string)** – A string of text to be completed on. It can be given as empty and instead a line/position pair are given. In this case, the completer itself will split the line like readline does.
- **line (string, optional)** – The complete line that text is part of.
• **cursor_pos** (*int, optional*) – The position of the cursor on the input line.

**Returns**

• **text** (*string*) – The actual text that was completed.

• **matches** (*list*) – A sorted list with all possible completions.

The optional arguments allow the completion to take more context into account, and are part of the low-level completion API.

This is a wrapper around the completion mechanism, similar to what readline does at the command line when the TAB key is hit. By exposing it as a method, it can be used by other non-readline environments (such as GUIs) for text completion.

Simple usage example:

In[1]: x = ‘hello’
In[2]: _ip.complete('x.l') Out[2]: (‘x.l’, [‘x.ljust’, ‘x.lower’, ‘x.lstrip’])

**debugger** (*force=False*)

Call the pydb/pdb debugger.

**Keywords:**

• force(False): by default, this routine checks the instance call_pdb flag and does not actually invoke the debugger if the flag is false. The ‘force’ option forces the debugger to activate even if the flag is false.

**define_macro** (*name, themacro*)

Define a new macro

**Parameters**

• **name** (*str*) – The name of the macro.

• **themacro** (*str or Macro*) – The action to do upon invoking the macro. If a string, a new Macro object is created by passing the string to it.

**del_var** (*varname, by_name=False*)

Delete a variable from the various namespaces, so that, as far as possible, we’re not keeping any hidden references to it.

**Parameters**

• **varname** (*str*) – The name of the variable to delete.

• **by_name** (*bool*) – If True, delete variables with the given name in each namespace. If False (default), find the variable in the user namespace, and delete references to it.

**drop_by_id** (*variables*)

Remove a dict of variables from the user namespace, if they are the same as the values in the dictionary.
This is intended for use by extensions: variables that they’ve added can be taken back out if they
are unloaded, without removing any that the user has overwritten.

**Parameters**

**variables** *(dict)* – A dictionary mapping object names (as strings) to the objects.

**enable_matplotlib** *(gui= None)*

Enable interactive matplotlib and inline figure support.

This takes the following steps:

1. select the appropriate eventloop and matplotlib backend
2. set up matplotlib for interactive use with that backend
3. configure formatters for inline figure display
4. enable the selected gui eventloop

**Parameters**

**gui** *(optional, string)* – If given, dictates the choice of matplotlib GUI backend to use (should be one of IPython’s supported backends, ‘qt’, ‘osx’, ‘tk’, ‘gtk’, ‘wx’ or ‘inline’), otherwise we use the default chosen by matplotlib (as dictated by the matplotlib build-time options plus the user’s matplotlibrc configuration file). Note that not all backends make sense in all contexts, for example a terminal ipython can’t display figures inline.

**enable_pylab** *(gui= None, import_all=True, welcome_message=False)*

Activate pylab support at runtime.

This turns on support for matplotlib, preloads into the interactive namespace all of numpy and
pylab, and configures IPython to correctly interact with the GUI event loop. The GUI backend
to be used can be optionally selected with the optional gui argument.

This method only adds preloading the namespace to InteractiveShell.enable_matplotlib.

**Parameters**

- **gui** *(optional, string)* – If given, dictates the choice of matplotlib GUI backend to use (should be one of IPython’s supported backends, ‘qt’, ‘osx’, ‘tk’, ‘gtk’, ‘wx’ or ‘inline’), otherwise we use the default chosen by matplotlib (as dictated by the matplotlib build-time options plus the user’s matplotlibrc configuration file). Note that not all backends make sense in all contexts, for example a terminal ipython can’t display figures inline.

- **import_all** *(optional, bool, default: True)* – Whether to do from numpy import * and from pylab import * in addition to module imports.

- **welcome_message** *(deprecated)* – This argument is ignored, no welcome message will be displayed.

**ev** *(expr)*

Evaluate python expression expr in user namespace.

Returns the result of evaluation
**ex (cmd)**
Execute a normal python statement in user namespace.

**excepthook (etype, value, tb)**
One more defense for GUI apps that call sys.excepthook.

GUI frameworks like wxPython trap exceptions and call sys.excepthook themselves. I guess this is a feature that enables them to keep running after exceptions that would otherwise kill their mainloop. This is a bother for IPython which excepts to catch all of the program exceptions with a try: except: statement.

Normally, IPython sets sys.excepthook to a CrashHandler instance, so if any app directly invokes sys.excepthook, it will look to the user like IPython crashed. In order to work around this, we can disable the CrashHandler and replace it with this excepthook instead, which prints a regular traceback using our InteractiveTB. In this fashion, apps which call sys.excepthook will generate a regular-looking exception from IPython, and the CrashHandler will only be triggered by real IPython crashes.

This hook should be used sparingly, only in places which are not likely to be true IPython errors.

**extract_input_lines (range_str, raw=False)**
Return as a string a set of input history slices.

**Parameters**

- **range_str (string)** – The set of slices is given as a string, like “~5/6-~4/2 4:8 9”, since this function is for use by magic functions which get their arguments as strings. The number before the / is the session number: ~n goes n back from the current session.
- **raw (bool, optional)** – By default, the processed input is used. If this is true, the raw input history is used instead.

**Notes**

Slices can be described with two notations:

- N:M -> standard python form, means including items N...(M-1).
- N–M -> include items N..M (closed endpoint).

**find_cell_magic (magic_name)**
Find and return a cell magic by name.

Returns None if the magic isn’t found.

**find_line_magic (magic_name)**
Find and return a line magic by name.

Returns None if the magic isn’t found.

**find_magic (magic_name, magic_kind='line')**
Find and return a magic of the given type by name.

Returns None if the magic isn’t found.
find_user_code(target,  raw=True,  py_only=False,  skip_encoding_cookie=True,  search_ns=False)
Get a code string from history, file, url, or a string or macro.
This is mainly used by magic functions.

Parameters

• target (str) – A string specifying code to retrieve. This will be tried respec-
tively as: ranges of input history (see %history for syntax), url, corresponding
.py file, filename, or an expression evaluating to a string or Macro in the user
namespace.

• raw (bool) – If true (default), retrieve raw history. Has no effect on the other
retrieval mechanisms.

• py_only (bool (default False)) – Only try to fetch python code, do not try alternative methods to decode file if unicode fails.

Returns

• A string of code.

• ValueError is raised if nothing is found, and TypeError if it evaluates
to an object of another type. In each case, .args[0] is a printable
message.

get_exception_only(exc_tuple=None)
Return as a string (ending with a newline) the exception that just occurred, without any trace-
back.

get_ipython()  
Return the currently running IPython instance.

getoutput(cmd, split=True, depth=0)  
Get output (possibly including stderr) from a subprocess.

Parameters

• cmd (str) – Command to execute (can not end in ‘&’, as background processes
are not supported.

• split (bool, optional) – If True, split the output into an IPython SList.
Otherwise, an IPython LSString is returned. These are objects similar to normal
lists and strings, with a few convenience attributes for easier manipulation
of line-based output. You can use ‘?’ on them for details.

• depth (int, optional) – How many frames above the caller are the local
variables which should be expanded in the command string? The default (0)
assumes that the expansion variables are in the stack frame calling this function.

init_completer()  
Initialize the completion machinery.
This creates completion machinery that can be used by client code, either interactively in-process (typically triggered by the readline library), programmatically (such as in test suites) or out-of-process (typically over the network by remote frontends).

```python
init_deprecation_warnings()
```

Register default filter for deprecation warning.

This will allow deprecation warning of function used interactively to show warning to users, and still hide deprecation warning from libraries import.

```python
init_environment()
```

Any changes we need to make to the user’s environment.

```python
init_history()
```

Sets up the command history, and starts regular autosaves.

```python
init_logstart()
```

Initialize logging in case it was requested at the command line.

```python
init_readline()
```

Moved to terminal subclass, here only to simplify the init logic.

```python
init_user_ns()
```

Initialize all user-visible namespaces to their minimum defaults.

Certain history lists are also initialized here, as they effectively act as user namespaces.

**Notes**

All data structures here are only filled in, they are NOT reset by this method. If they were not empty before, data will simply be added to them.

```python
init_virtualenv()
```

Add a virtualenv to sys.path so the user can import modules from it. This isn’t perfect: it doesn’t use the Python interpreter with which the virtualenv was built, and it ignores the –no-site-packages option. A warning will appear suggesting the user installs IPython in the virtualenv, but for many cases, it probably works well enough.

Adapted from code snippets online.


**magic** *(arg_s)*

DEPRECATED. Use run_line_magic() instead.

Call a magic function by name.

Input: a string containing the name of the magic function to call and any additional arguments to be passed to the magic.

`magic('name -opt foo bar')` is equivalent to typing at the ipython prompt:

In[1]: `%name -opt foo bar`

To call a magic without arguments, simply use `magic('name')`. 

448 Chapter 8. The IPython API
This provides a proper Python function to call IPython’s magics in any valid Python code you can type at the interpreter, including loops and compound statements.

**mktempfile** *(data=None, prefix='ipython_edit_')*

Make a new tempfile and return its filename.

This makes a call to tempfile.mkstemp (created in a tempfile.mkdtemp), but it registers the created filename internally so ipython cleans it up at exit time.

Optional inputs:

- *data(None)*: if data is given, it gets written out to the temp file immediately, and the file is closed again.

**new_main_mod** *(filename, modname)*

Return a new ‘main’ module object for user code execution.

*filename* should be the path of the script which will be run in the module. Requests with the same filename will get the same module, with its namespace cleared.

*modname* should be the module name - normally either '__main__' or the basename of the file without the extension.

When scripts are executed via %run, we must keep a reference to their __main__ module around so that Python doesn’t clear it, rendering references to module globals useless.

This method keeps said reference in a private dict, keyed by the absolute path of the script. This way, for multiple executions of the same script we only keep one copy of the namespace (the last one), thus preventing memory leaks from old references while allowing the objects from the last execution to be accessible.

**object_inspect** *(oname, detail_level=0)*

Get object info about oname

**object_inspect_text** *(oname, detail_level=0)*

Get object info as formatted text

**prepare_user_module** *(user_module=None, user_ns=None)*

Prepare the module and namespace in which user code will be run.

When IPython is started normally, both parameters are None: a new module is created automatically, and its __dict__ used as the namespace.

If only user_module is provided, its __dict__ is used as the namespace. If only user_ns is provided, a dummy module is created, and user_ns becomes the global namespace. If both are provided (as they may be when embedding), user_ns is the local namespace, and user_module provides the global namespace.

**Parameters**

- *user_module (module, optional)* – The current user module in which IPython is being run. If None, a clean module will be created.

- *user_ns (dict, optional)* – A namespace in which to run interactive commands.

**Returns**
Return type  A tuple of user_module and user_ns, each properly initialised.

**push**(variables, interactive=True)

Inject a group of variables into the IPython user namespace.

**Parameters**

- **variables**(dict, str or list/tuple of str) – The variables to inject into the user's namespace. If a dict, a simple update is done. If a str, the string is assumed to have variable names separated by spaces. A list/tuple of str can also be used to give the variable names. If just the variable names are give (list/tuple/str) then the variable values looked up in the callers frame.

- **interactive**(bool) – If True (default), the variables will be listed with the who magic.

**register_magic_function**(func, magic_kind='line', magic_name=None)

Expose a standalone function as magic function for IPython.

This will create an IPython magic (line, cell or both) from a standalone function. The functions should have the following signatures:

- For line magics: `def f(line)`
- For cell magics: `def f(line, cell)`
- For a function that does both: `def f(line, cell=None)`

In the latter case, the function will be called with `cell=None` when invoked as `%f`, and with `cell` as a string when invoked as `%%f`.

**Parameters**

- **func**(callable) – Function to be registered as a magic.

- **magic_kind**(str) – Kind of magic, one of ‘line’, ‘cell’ or ‘line_cell’

- **magic_name**(optional str) – If given, the name the magic will have in the IPython namespace. By default, the name of the function itself is used.

**register_post_execute**(func)

DEPRECATED: Use ip.events.register(‘post_run_cell’, func)

Register a function for calling after code execution.

**reset**(new_session=True)

Clear all internal namespaces, and attempt to release references to user objects.

If new_session is True, a new history session will be opened.

**reset_selective**(regex=None)

Clear selective variables from internal namespaces based on a specified regular expression.

**Parameters**

- **regex**(string or compiled pattern, optional) – A regular expression pattern that will be used in searching variable names in the users namespaces.
**restore_sys_module_state()**

Restore the state of the sys module.

**run_ast_nodes(nodelist, cell_name, interactivity='last_expr', compiler=<built-in function compile>, result=None)**

Run a sequence of AST nodes. The execution mode depends on the interactivity parameter.

**Parameters**

- `nodelist (list)` – A sequence of AST nodes to run.
- `cell_name (str)` – Will be passed to the compiler as the filename of the cell. Typically the value returned by ip.compile.cache(cell).
- `interactivity (str)` – ‘all’, ‘last’, ‘last_expr’ or ‘none’, specifying which nodes should be run interactively (displaying output from expressions). ‘last_expr’ will run the last node interactively only if it is an expression (i.e. expressions in loops or other blocks are not displayed. Other values for this parameter will raise a ValueError.
- `compiler (callable)` – A function with the same interface as the built-in compile(), to turn the AST nodes into code objects. Default is the built-in compile().
- `result (ExecutionResult, optional)` – An object to store exceptions that occur during execution.

**Returns**

- `True if an exception occurred while running code, False if it finished`
- `running`

**run_cell(raw_cell, store_history=False, silent=False, shell_futures=True)**

Run a complete IPython cell.

**Parameters**

- `raw_cell (str)` – The code (including IPython code such as %magic functions) to run.
- `store_history (bool)` – If True, the raw and translated cell will be stored in IPython’s history. For user code calling back into IPython’s machinery, this should be set to False.
- `silent (bool)` – If True, avoid side-effects, such as implicit displayhooks and and logging. silent=True forces store_history=False.
- `shell_futures (bool)` – If True, the code will share future statements with the interactive shell. It will both be affected by previous __future__ imports, and any __future__ imports in the code will affect the shell. If False, __future__ imports are not shared in either direction.

**Returns** `result`

**Return type** `ExecutionResult`
run_cell_magic(magic_name, line, cell)
Execute the given cell magic.

Parameters

- magic_name(str) – Name of the desired magic function, without ‘%’ prefix.
- line(str) – The rest of the first input line as a single string.
- cell(str) – The body of the cell as a (possibly multiline) string.

run_code(code_obj, result=None)
Execute a code object.

When an exception occurs, self.showtraceback() is called to display a traceback.

Parameters

- code_obj(code object) – A compiled code object, to be executed
- result(ExecutionResult, optional) – An object to store exceptions that occur during execution.

Returns

- False (successful execution.)
- True (an error occurred.)

run_line_magic(magic_name, line)
Execute the given line magic.

Parameters

- magic_name(str) – Name of the desired magic function, without ‘%’ prefix.
- line(str) – The rest of the input line as a single string.

runcode(code_obj, result=None)
Execute a code object.

When an exception occurs, self.showtraceback() is called to display a traceback.

Parameters

- code_obj(code object) – A compiled code object, to be executed
- result(ExecutionResult, optional) – An object to store exceptions that occur during execution.

Returns

- False (successful execution.)
- True (an error occurred.)

safe_execfile(fname, *where, **kw)
A safe version of the builtin execfile().

This version will never throw an exception, but instead print helpful error messages to the screen. This only works on pure Python files with the .py extension.
Parameters

- **fname** (*string*) – The name of the file to be executed.
- **where** (*tuple*) – One or two namespaces, passed to execfile() as (globals,locals). If only one is given, it is passed as both.
- **exit_ignore** (*bool (False)*) – If True, then silence SystemExit for non-zero status (it is always silenced for zero status, as it is so common).
- **raise_exceptions** (*bool (False)*) – If True raise exceptions everywhere. Meant for testing.
- **shell_futures** (*bool (False)*) – If True, the code will share future statements with the interactive shell. It will both be affected by previous __future__ imports, and any __future__ imports in the code will affect the shell. If False, __future__ imports are not shared in either direction.

**safe_execfile_ipy** (*fname, shell_futures=False, raise_exceptions=False*)

Like safe_execfile, but for .ipy or .ipynb files with IPython syntax.

Parameters

- **fname** (*str*) – The name of the file to execute. The filename must have a .ipy or .ipynb extension.
- **shell_futures** (*bool (False)*) – If True, the code will share future statements with the interactive shell. It will both be affected by previous __future__ imports, and any __future__ imports in the code will affect the shell. If False, __future__ imports are not shared in either direction.
- **raise_exceptions** (*bool (False)*) – If True raise exceptions everywhere. Meant for testing.

**safe_run_module** (*mod_name, where*)

A safe version of runpy.run_module().

This version will never throw an exception, but instead print helpful error messages to the screen.

SystemExit exceptions with status code 0 or None are ignored.

Parameters

- **mod_name** (*string*) – The name of the module to be executed.
- **where** (*dict*) – The globals namespace.

**save_sys_module_state** (**)

Save the state of hooks in the sys module.

This has to be called after self.user_module is created.

**set_autoindent** (*value=None*)

Set the autoindent flag, checking for readline support.

If called with no arguments, it acts as a toggle.
**set_completer_frame** *(frame=None)*  
Set the frame of the completer.

**set_custom_completer** *(completer, pos=0)*  
Adds a new custom completer function.

The position argument (defaults to 0) is the index in the completers list where you want the completer to be inserted.

**set_custom_exc** *(exc_tuple, handler)*  
Set a custom exception handler, which will be called if any of the exceptions in exc_tuple occur in the mainloop (specifically, in the run_code() method).

**Parameters**

- **exc_tuple** *(tuple of exception classes)* – A tuple of exception classes, for which to call the defined handler. It is very important that you use a tuple, and NOT a LIST here, because of the way Python’s except statement works. If you only want to trap a single exception, use a singleton tuple:

  ```
  exc_tuple == (MyCustomException,)
  ```

- **handler** *(callable)* – handler must have the following signature:

  ```python
def my_handler(self, etype, value, tb, tb_offset=None):
    ...
    return structured_traceback
  ```

  Your handler must return a structured traceback (a list of strings), or None.

  This will be made into an instance method (via types.MethodType) of IPython itself, and it will be called if any of the exceptions listed in the exc_tuple are caught. If the handler is None, an internal basic one is used, which just prints basic info.

  To protect IPython from crashes, if your handler ever raises an exception or returns an invalid result, it will be immediately disabled.

- **WARNING** *(by putting in your own exception handler into IPython’s main)* –

  - **loop, you run a very good chance of nasty crashes.**

  - **This (execution)** –

  - **should only be used if you really know what you are doing.** *(facility)* –

**set_hook** *(name, hook) → sets an internal IPython hook.*

IPython exposes some of its internal API as user-modifiable hooks. By adding your function to one of these hooks, you can modify IPython’s behavior to call at runtime your own routines.

**set_next_input** *(s, replace=False)*  
Sets the ‘default’ input string for the next command line.

Example:
In [1]: _ip.set_next_input("Hello Word")
In [2]: Hello Word_ # cursor is here

**show_usage()**
Show a usage message

**show_usage_error(exc)**
Show a short message for UsageErrors

These are special exceptions that shouldn’t show a traceback.

**showindentationerror()**
Called by run_cell when there’s an IndentationError in code entered at the prompt.

This is overridden in TerminalInteractiveShell to show a message about the %paste magic.

**showsyntaxerror(filename=None)**
Display the syntax error that just occurred.

This doesn’t display a stack trace because there isn’t one.

If a filename is given, it is stuffed in the exception instead of what was there before (because Python’s parser always uses “<string>” when reading from a string).

**showtraceback(exctuple=None, filename=None, tb_offset=None, exception_only=False)**
Display the exception that just occurred.

If nothing is known about the exception, this is the method which should be used throughout the code for presenting user backtraces, rather than directly invoking the InteractiveTB object.

A specific showsyntaxerror() also exists, but this method can take care of calling it if needed, so unless you are explicitly catching a SyntaxError exception, don’t try to analyze the stack manually and simply call this method.

**system(cmd)**
Call the given cmd in a subprocess, piping stdout/err

**system_piped(cmd)**
Call the given cmd in a subprocess, piping stdout/err

**system_raw(cmd)**
Call the given cmd in a subprocess using os.system on Windows or subprocess.call using the system shell on other platforms.

**transform_ast(node)**
Apply the AST transformations from self.ast_transformers
Parameters `node` (*ast.Node*) – The root node to be transformed. Typically called with the `ast.Module` produced by parsing user input.

Returns

- An `ast.Node` corresponding to the node it was called with. Note that it may also modify the passed object, so don’t rely on references to the original AST.

`user_expressions` (*expressions*)
Evaluate a dict of expressions in the user’s namespace.

Parameters `expressions` (*dict*) – A dict with string keys and string values. The expression values should be valid Python expressions, each of which will be evaluated in the user namespace.

Returns

- A dict, keyed like the input expressions dict, with the rich mime-typed display_data of each value.

`var_expand` (*cmd*, `depth=0`, `formatter=<IPython.utils.text.DollarFormatter object>*)
Expand python variables in a string.

The depth argument indicates how many frames above the caller should be walked to look for the local namespace where to expand variables.

The global namespace for expansion is always the user’s interactive namespace.

`write` (*data*)
Write a string to the default output

`write_err` (*data*)
Write a string to the default error output

class `IPython.core.interactiveshell.InteractiveShellABC`
Bases: `abc._NewBase`

An abstract base class for InteractiveShell.

8.22.2 1 Function

`IPython.core.interactiveshell.get_default_colors()`

8.23 Module: `core.logger`

Logger class for IPython’s logging facilities.
8.23.1 1 Class

class IPython.core.logger.Logger(home_dir, logfname='Logger.log', loghead='', logmode='over')

Bases: object

A Logfile class with different policies for file creation
__init__(home_dir, logfname='Logger.log', loghead='', logmode='over')

close_log()

Fully stop logging and close log file.

In order to start logging again, a new logstart() call needs to be made, possibly (though not necessarily) with a new filename, mode and other options.

log(line_mod, line_ori)

Write the sources to a log.

Inputs:

• line_mod: possibly modified input, such as the transformations made by input prefilters or input handlers of various kinds. This should always be valid Python.

• line_ori: unmodified input line from the user. This is not necessarily valid Python.

log_write(data, kind='input')

Write data to the log file, if active

logstart(logfname=None, loghead=None, logmode=None, log_output=False, timestamps=False, log_raw_input=False)

Generate a new log-file with a default header.

Raises RuntimeError if the log has already been started

logstate()

Print a status message about the logger.

logstop()

Fully stop logging and close log file.

In order to start logging again, a new logstart() call needs to be made, possibly (though not necessarily) with a new filename, mode and other options.

switch_log(val)

Switch logging on/off. val should be ONLY a boolean.

8.24 Module: core.macro

Support for interactive macros in IPython
8.24.1 1 Class

class IPython.core.macro.Macro(code)
   Bases: object
   Simple class to store the value of macros as strings.
   Macro is just a callable that executes a string of IPython input when called.
   __init__(code)
      store the macro value, as a single string which can be executed
   __getstate__()
      needed for safe pickling via %store

8.25 Module: core.magic

Magic functions for InteractiveShell.

8.25.1 4 Classes

class IPython.core.magic.Bunch
   Bases: object

class IPython.core.magic.MagicsManager(shell=None, config=None, user_magics=None, **traits)
   Bases: traitlets.config.configurable.Configurable
   Object that handles all magic-related functionality for IPython.
   __init__(shell=None, config=None, user_magics=None, **traits)
   auto_status()
      Return descriptive string with automagic status.
   define_magic(name, func)
      [Deprecated] Expose own function as magic function for IPython.
      Will be removed in IPython 5.0
      Example:
      def foo_impl(self, parameter_s=''):
         'My very own magic!. (Use docstrings, IPython reads them).'
         print 'Magic function. Passed parameter is between < >:'
         print '<%s>' % parameter_s
         print 'The self object is:', self
         ip.define_magic('foo', foo_impl)
   lsmagic()
      Return a dict of currently available magic functions.
The return dict has the keys ‘line’ and ‘cell’, corresponding to the two types of magics we support. Each value is a list of names.

```
lsmagic_docs (brief=False, missing='')
```

Return dict of documentation of magic functions.

The return dict has the keys ‘line’ and ‘cell’, corresponding to the two types of magics we support. Each value is a dict keyed by magic name whose value is the function docstring. If a docstring is unavailable, the value of missing is used instead.

If brief is True, only the first line of each docstring will be returned.

```
register (*magic_objects)
```

Register one or more instances of Magics.

Take one or more classes or instances of classes that subclass the main `core.Magic` class, and register them with IPython to use the magic functions they provide. The registration process will then ensure that any methods that have decorated to provide line and/or cell magics will be recognized with the `%x`/`%%x` syntax as a line/cell magic respectively.

If classes are given, they will be instantiated with the default constructor. If your classes need a custom constructor, you should instantiate them first and pass the instance.

The provided arguments can be an arbitrary mix of classes and instances.

```
Parameters
magic_objects (one or more classes or instances) –
```

```
register_alias (alias_name, magic_name, magic_kind='line')
```

Register an alias to a magic function.

The alias is an instance of `MagicAlias`, which holds the name and kind of the magic it should call. Binding is done at call time, so if the underlying magic function is changed the alias will call the new function.

```
Parameters

  • alias_name (str) – The name of the magic to be registered.
  • magic_name (str) – The name of an existing magic.
  • magic_kind (str) – Kind of magic, one of ‘line’ or ‘cell’
```

```
register_function (func, magic_kind='line', magic_name=None)
```

Expose a standalone function as magic function for IPython.

This will create an IPython magic (line, cell or both) from a standalone function. The functions should have the following signatures:

- For line magics: `def f(line)`
- For cell magics: `def f(line, cell)`
- For a function that does both: `def f(line, cell=None)`

In the latter case, the function will be called with `cell=None` when invoked as `%f`, and with `cell` as a string when invoked as `%%f`.

```
Parameters
```
• **func**(Callable) – Function to be registered as a magic.
• **magic_kind**(str) – Kind of magic, one of ‘line’, ‘cell’ or ‘line_cell’
• **magic_name**(optional str) – If given, the name the magic will have in the IPython namespace. By default, the name of the function itself is used.

```python
class IPython.core.magic.Magics (shell=None, **kwargs)
```

Bases: traitlets.config.configurable.Configurable

Base class for implementing magic functions.

Shell functions which can be reached as `%function_name`. All magic functions should accept a string, which they can parse for their own needs. This can make some functions easier to type, e.g. `%cd ../` vs. `%cd("../")`

Classes providing magic functions need to subclass this class, and they MUST:

• Use the method decorators @line_magic and @cell_magic to decorate individual methods as magic functions, AND

• Use the class decorator @magics_class to ensure that the magic methods are properly registered at the instance level upon instance initialization.

See `magic_functions` for examples of actual implementation classes.

```python
__init__ (shell=None, **kwargs)
arg_err (func)
    Print docstring if incorrect arguments were passed
default_option (fn, optstr)
    Make an entry in the options_table for fn, with value optstr
format_latex (strng)
    Format a string for latex inclusion.
parse_options (arg_str, opt_str, *long_opts, **kw)
    Parse options passed to an argument string.

    The interface is similar to that of `getopt.getopt()`, but it returns a `Struct` with the options as keys and the stripped argument string still as a string.

    arg_str is quoted as a true sys.argv vector by using shlex.split. This allows us to easily expand variables, glob files, quote arguments, etc.
```

Parameters

• **arg_str**(str) – The arguments to parse.
• **opt_str**(str) – The options specification.
• **mode**(str, default ‘string’) – If given as ‘list’, the argument string is returned as a list (split on whitespace) instead of a string.
• **list_all**(bool, default False) – Put all option values in lists. Normally only options appearing more than once are put in a list.
• **posix** *(bool, default True)* – Whether to split the input line in POSIX mode or not, as per the conventions outlined in the *shlex* module from the standard library.

```python
class IPython.core.magic.MagicAlias(shell, magic_name, magic_kind)
Bases: object
```

An alias to another magic function.

An alias is determined by its magic name and magic kind. Lookup is done at call time, so if the underlying magic changes the alias will call the new function.

Use the `MagicsManager.register_alias()` method or the `%alias_magic` magic function to create and register a new alias.

```python
__init__(shell, magic_name, magic_kind)
__call__(*args, **kwargs)
```

Call the magic alias.

### 8.25.2 6 Functions

**IPython.core.magic.on_off(tag)**

Return an ON/OFF string for a 1/0 input. Simple utility function.

**IPython.core.magic.compress_dhist(dh)**

Compress a directory history into a new one with at most 20 entries.

Return a new list made from the first and last 10 elements of `dhist` after removal of duplicates.

**IPython.core.magic.needs_local_scope(func)**

Decorator to mark magic functions which need to local scope to run.

**IPython.core.magic.magics_class(cls)**

Class decorator for all subclasses of the main Magics class.

Any class that subclasses Magics *must* also apply this decorator, to ensure that all the methods that have been decorated as line/cell magics get correctly registered in the class instance. This is necessary because when method decorators run, the class does not exist yet, so they temporarily store their information into a module global. Application of this class decorator copies that global data to the class instance and clears the global.

Obviously, this mechanism is not thread-safe, which means that the creation of subclasses of Magic should only be done in a single-thread context. Instantiation of the classes has no restrictions. Given that these classes are typically created at IPython startup time and before user application code becomes active, in practice this should not pose any problems.

**IPython.core.magic.record_magic(dct, magic_kind, magic_name, func)**

Utility function to store a function as a magic of a specific kind.

**Parameters**

- **dct (dict)** – A dictionary with ‘line’ and ‘cell’ subdicts.
- **magic_kind (str)** – Kind of magic to be stored.
• `magic_name (str)` – Key to store the magic as.
• `func (function)` – Callable object to store.

IPython.core.magic.validate_type(magic_kind)
Ensure that the given magic_kind is valid.

Check that the given magic_kind is one of the accepted spec types (stored in the global magic_spec), raise ValueError otherwise.

8.26 Module: core.magic_arguments

A decorator-based method of constructing IPython magics with argparse option handling.

New magic functions can be defined like so:

```python
from IPython.core.magic_arguments import (argument, magic_arguments,
                                          parse_argstring)

@magic_arguments()
@argument('-o', '--option', help='An optional argument.')
@argument('arg', type=int, help='An integer positional argument.')
def magic_cool(self, arg):
    """ A really cool magic command."
    
    args = parse_argstring(magic_cool, arg)
    ...
```

The @magic_arguments decorator marks the function as having argparse arguments. The @argument decorator adds an argument using the same syntax as argparse’s add_argument() method. More sophisticated uses may also require the @argument_group or @kwds decorator to customize the formatting and the parsing.

Help text for the magic is automatically generated from the docstring and the arguments:

```
In[1]: %cool?
%cool [-o OPTION] arg

A really cool magic command.

positional arguments:
    arg            An integer positional argument.

optional arguments:
    -o OPTION, --option OPTION
              An optional argument.
```

Inheritance diagram:
8.26.1 8 Classes

class IPython.core.magic_arguments.MagicArgumentParser (prog=None, usage=None, description=None, epilog=None, parents=None, formatter_class=<class 'IPython.core.magic_arguments.MagicHelpFormatter'>, prefix_chars='-', argument_default=None, conflict_handler='error', add_help=False)

Bases: argparse.ArgumentParser

An ArgumentParser tweaked for use by IPython magics.

___init___ (prog=None, usage=None, description=None, epilog=None, parents=None, formatter_class=<class 'IPython.core.magic_arguments.MagicHelpFormatter'>, prefix_chars='-', argument_default=None, conflict_handler='error', add_help=False)

error (message)

Raise a catchable error instead of exiting.

parse_argstring (argstring)

Split a string into an argument list and parse that argument list.

class IPython.core.magic_arguments.ArgDecorator
Bases: object

Base class for decorators to add ArgumentParser information to a method.
add_to_parser (parser, group)
   Add this object’s information to the parser, if necessary.

class IPython.core.magic_arguments.magic_arguments (name=None)
   Bases: IPython.core.magic_arguments.ArgDecorator
Mark the magic as having argparse arguments and possibly adjust the name.
   __init__ (name=None)

class IPython.core.magic_arguments.ArgMethodWrapper (*args, **kwds)
   Bases: IPython.core.magic_arguments.ArgDecorator
Base class to define a wrapper for ArgumentParser method.
Child class must define either _method_name or add_to_parser.
   __init__ (*args, **kwds)
   add_to_parser (parser, group)
   Add this object’s information to the parser.

class IPython.core.magic_arguments.argument (*args, **kwds)
   Bases: IPython.core.magic_arguments.ArgMethodWrapper
Store arguments and keywords to pass to add_argument().
Instances also serve to decorate command methods.

class IPython.core.magic_arguments.defaults (*args, **kwds)
   Bases: IPython.core.magic_arguments.ArgMethodWrapper
Store arguments and keywords to pass to set_defaults().
Instances also serve to decorate command methods.

class IPython.core.magic_arguments.argument_group (*args, **kwds)
   Bases: IPython.core.magic_arguments.ArgMethodWrapper
Store arguments and keywords to pass to add_argument_group().
Instances also serve to decorate command methods.
   add_to_parser (parser, group)
   Add this object’s information to the parser.

class IPython.core.magic_arguments.kwds (**kwds)
   Bases: IPython.core.magic_arguments.ArgDecorator
Provide other keywords to the sub-parser constructor.
   __init__ (**kwds)

8.26.2 3 Functions

IPython.core.magic_arguments.construct_parser (magic_func)
   Construct an argument parser using the function decorations.
IPython.core.magic_arguments.parse_argstring(magic_func, argstring)
    Parse the string of arguments for the given magic function.

IPython.core.magic_arguments.real_name(magic_func)
    Find the real name of the magic.

8.27 Module: core.oinspect

Tools for inspecting Python objects.
Uses syntax highlighting for presenting the various information elements.
Similar in spirit to the inspect module, but all calls take a name argument to reference the name under which
an object is being read.

8.27.1 1 Class

class IPython.core.oinspect.Inspector(color_table={'':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583320>, 'LightBG':<IPython.utils.coloransi.ColorScheme object at 0x7f44b25833c8>, 'NoColor':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583320>, 'Linux':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583390>}, code_color_table={'':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583320>, 'LightBG':<IPython.utils.coloransi.ColorScheme object at 0x7f44b25833c8>, 'NoColor':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583320>, 'Linux':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583390>}, scheme='NoColor', str_detail_level=0)

Bases: object

__init__(color_table={'':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583320>, 'LightBG':<IPython.utils.coloransi.ColorScheme object at 0x7f44b25833c8>, 'NoColor':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583320>, 'Linux':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583390>}, code_color_table={'':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583320>, 'LightBG':<IPython.utils.coloransi.ColorScheme object at 0x7f44b25833c8>, 'NoColor':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583320>, 'Linux':<IPython.utils.coloransi.ColorScheme object at 0x7f44b2583390>}, scheme='NoColor', str_detail_level=0)
**info** (*obj, oname='', formatter=None, info=None, detail_level=0*)
Compute a dict with detailed information about an object.

Optional arguments:
- `oname`: name of the variable pointing to the object.
- `formatter`: special formatter for docstrings (see `pdoc`)
- `info`: a structure with some information fields which may have been precomputed already.
- `detail_level`: if set to 1, more information is given.

**noinfo** (*msg, oname*)
Generic message when no information is found.

**pdef** (*obj, oname='*')
Print the call signature for any callable object.

If the object is a class, print the constructor information.

**pdoc** (*obj, oname='*', formatter=None*)
Print the docstring for any object.

Optional: `-formatter`: a function to run the docstring through for specially formatted docstrings.

**Examples**

```
In [1]: class NoInit: ...
   ...: pass
In [2]: class NoDoc: ...
   ...: def __init__(self): ...
   ...: pass
In [3]: %pdoc NoDoc No documentation found for NoDoc
In [4]: %pdoc NoInit No documentation found for NoInit
In [5]: obj = NoInit()
In [6]: %pdoc obj No documentation found for obj
In [5]: obj2 = NoDoc()
In [6]: %pdoc obj2 No documentation found for obj2
```

**pfile** (*obj, oname='*')
Show the whole file where an object was defined.

**pinfo** (*obj, oname='*', formatter=None, info=None, detail_level=0*)
Show detailed information about an object.

Optional arguments:
- `oname`: name of the variable pointing to the object.
- `formatter`: special formatter for docstrings (see `pdoc`)
- `info`: a structure with some information fields which may have been precomputed already.
• `detail_level`: if set to 1, more information is given.

**psearch** *(pattern, ns_table, ns_search=[], ignore_case=False, show_all=False)*

Search namespaces with wildcards for objects.

**Arguments:**

• `pattern`: string containing shell-like wildcards to use in namespace searches and optionally a type specification to narrow the search to objects of that type.

• `ns_table`: dict of name->namespaces for search.

**Optional arguments:**

• `ns_search`: list of namespace names to include in search.

• `ignore_case(False)`: make the search case-insensitive.

• `show_all(False)`: show all names, including those starting with underscores.

**psource** *(obj, oname='')*

Print the source code for an object.

### 8.27.2 10 Functions

**IPython.core.oinspect.object_info(**`kw`**)**

Make an object info dict with all fields present.

**IPython.core.oinspect.get_encoding**(obj)

Get encoding for python source file defining obj

Returns None if obj is not defined in a sourcefile.

**IPython.core.oinspect.getdoc**(obj)

Stable wrapper around inspect.getdoc.

This can’t crash because of attribute problems.

It also attempts to call a getdoc() method on the given object. This allows objects which provide their docstrings via non-standard mechanisms (like Pyro proxies) to still be inspected by ipython’s ? system.

**IPython.core.oinspect.getsource**(obj, oname='')

Wrapper around inspect.getsource.

This can be modified by other projects to provide customized source extraction.

**Parameters**

• `obj` *(object)* – an object whose source code we will attempt to extract

• `oname` *(str)* – (optional) a name under which the object is known

**Returns** `src`

**Return type** unicode or None
IPython.core.oinspect.is_simple_callable(obj)
True if obj is a function

IPython.core.oinspect.getargspec(obj)
In addition to functions and methods, this can also handle objects with a __call__ attribute.

IPython.core.oinspect.format_argspec(argspec)
Format argspec, convenience wrapper around inspect’s.
This takes a dict instead of ordered arguments and calls inspect.format_argspec with the arguments in the necessary order.

IPython.core.oinspect.call_tip(oinfo, format_call=True)
Extract call tip data from an oinfo dict.

Parameters
• oinfo(dict) –
• format_call (bool, optional) – If True, the call line is formatted and returned as a string. If not, a tuple of (name, argspec) is returned.

Returns
• call_info (None, str or (str, dict) tuple.) – When format_call is True, the whole call information is formatted as a single string. Otherwise, the object’s name and its argspec dict are returned. If no call information is available, None is returned.
• docstring (str or None) – The most relevant docstring for calling purposes is returned, if available. The priority is: call docstring for callable instances, then constructor docstring for classes, then main object’s docstring otherwise (regular functions).

IPython.core.oinspect.find_file(obj)
Find the absolute path to the file where an object was defined.
This is essentially a robust wrapper around inspect.getabsfile.
Returns None if no file can be found.

Parameters
• obj (any Python object) –

Returns
• fname – The absolute path to the file where the object was defined.

Return type
str

IPython.core.oinspect.find_source_lines(obj)
Find the line number in a file where an object was defined.
This is essentially a robust wrapper around inspect.getsourcelines.
Returns None if no file can be found.

Parameters
• obj (any Python object) –

Returns
• lineno – The line number where the object definition starts.
8.28 Module: core.page

Paging capabilities for IPython.core

Notes

For now this uses IPython hooks, so it can’t be in IPython.utils. If we can get rid of that dependency, we could move it there. ——

8.28.1 10 Functions

IPython.core.page.display_page(strng, start=0, screen_lines=25)

Just display, no paging. screen_lines is ignored.

IPython.core.page.as_hook(page_func)

Wrap a pager func to strip the self arg

so it can be called as a hook.

IPython.core.page.page_dumb(strng, start=0, screen_lines=25)

Very dumb ‘pager’ in Python, for when nothing else works.

Only moves forward, same interface as page(), except for pager_cmd and mode.

IPython.core.page.pager_page(strng, start=0, screen_lines=0, pager_cmd=None)

Display a string, piping through a pager after a certain length.

strng can be a mime-bundle dict, supplying multiple representations, keyed by mime-type.

The screen_lines parameter specifies the number of usable lines of your terminal screen (total lines minus lines you need to reserve to show other information).

If you set screen_lines to a number <=0, page() will try to auto-determine your screen size and will only use up to (screen_size+screen_lines) for printing, paging after that. That is, if you want auto-detection but need to reserve the bottom 3 lines of the screen, use screen_lines = -3, and for auto-detection without any lines reserved simply use screen_lines = 0.

If a string won’t fit in the allowed lines, it is sent through the specified pager command. If none given, look for PAGER in the environment, and ultimately default to less.

If no system pager works, the string is sent through a ‘dumb pager’ written in python, very simplistic.

IPython.core.page.page(data, start=0, screen_lines=0, pager_cmd=None)

Display content in a pager, piping through a pager after a certain length.

data can be a mime-bundle dict, supplying multiple representations, keyed by mime-type, or text.

Pager is dispatched via the show_in_pager IPython hook. If no hook is registered, pager_page will be used.
IPython.core.page.page_file(fname, start=0, pager_cmd=None)
    Page a file, using an optional pager command and starting line.

IPython.core.page.get_pager_cmd(pager_cmd=None)
    Return a pager command.

    Makes some attempts at finding an OS-correct one.

IPython.core.page.get_pager_start(pager, start)
    Return the string for paging files with an offset.

    This is the ‘+N’ argument which less and more (under Unix) accept.

IPython.core.page.page_more()

IPython.core.page.snip_print(str, width=75, print_full=0, header=' ')
    Print a string snipping the midsection to fit in width.

    print_full: mode control:
        • 0: only snip long strings
        • 1: send to page() directly.
        • 2: snip long strings and ask for full length viewing with page()

    Return 1 if snipping was necessary, 0 otherwise.

8.29 Module: core.payload

Payload system for IPython.

Authors:
    • Fernando Perez
    • Brian Granger

8.29.1 1 Class

class IPython.core.payload.PayloadManager(**kwargs)
    Bases: traitlets.config.configurable.Configurable

    write_payload(data, single=True)
    Include or update the specified data payload in the PayloadManager.

    If a previous payload with the same source exists and single is True, it will be overwritten
    with the new one.

8.30 Module: core.payloadpage

A payload based version of page.
8.30.1 2 Functions

IPython.core.payloadpage.page(strng, start=0, screen_lines=0, pager_cmd=None)

Print a string, piping through a pager.

This version ignores the screen_lines and pager_cmd arguments and uses IPython’s payload system instead.

**Parameters**

- **strng (str or mime-dict)** – Text to page, or a mime-type keyed dict of already formatted data.
- **start (int)** – Starting line at which to place the display.

IPython.core.payloadpage.install_payload_page()

DEPRECATED, use show_in_pager hook

Install this version of page as IPython.core.page.page.

8.31 Module: core.prefilter

Prefiltering components.

Prefilters transform user input before it is exec’d by Python. These transforms are used to implement additional syntax such as !ls and %magic.

Authors:

- Brian Granger
- Fernando Perez
- Dan Milstein
- Ville Vainio

8.31.1 16 Classes

class IPython.core.prefilter.PrefilterError

Bases: Exception

class IPython.core.prefilter.PrefilterManager(shell=None, **kwargs)

Bases: traitlets.config.configurable.Configurable

Main prefilter component.

The IPython prefilter is run on all user input before it is run. The prefilter consumes lines of input and produces transformed lines of input.

The implementation consists of two phases:

1. Transformers
2. Checkers and handlers

8.31. Module: core.prefilter
Over time, we plan on deprecating the checkers and handlers and doing everything in the transformers.

The transformers are instances of PrefilterTransformer and have a single method transform() that takes a line and returns a transformed line. The transformation can be accomplished using any tool, but our current ones use regular expressions for speed.

After all the transformers have been run, the line is fed to the checkers, which are instances of PrefilterChecker. The line is passed to the check() method, which either returns None or a PrefilterHandler instance. If None is returned, the other checkers are tried. If an PrefilterHandler instance is returned, the line is passed to the handle() method of the returned handler and no further checkers are tried.

Both transformers and checkers have a priority attribute, that determines the order in which they are called. Smaller priorities are tried first.

Both transformers and checkers also have enabled attribute, which is a boolean that determines if the instance is used.

Users or developers can change the priority or enabled attribute of transformers or checkers, but they must call the sort_checkers() or sort_transformers() method after changing the priority.

__init__ (shell=None, **kwargs)

checkers
Return a list of checkers, sorted by priority.

find_handler (line_info)
Find a handler for the line_info by trying checkers.

get_handler_by_esc (esc_str)
Get a handler by its escape string.

get_handler_by_name (name)
Get a handler by its name.

handlers
Return a dict of all the handlers.

init_checkers ()
Create the default checkers.

init_handlers ()
Create the default handlers.

init_transformers ()
Create the default transformers.

prefilter_line (line, continue_prompt=False)
Prefilter a single input line as text.

This method prefilters a single line of text by calling the transformers and then the checkers/handlers.

prefilter_line_info (line_info)
Prefilter a line that has been converted to a LineInfo object.
This implements the checker/handler part of the prefilter pipe.

**prefilter_lines** *(lines, continue_prompt=False)*

Prefilter multiple input lines of text.

This is the main entry point for prefiltering multiple lines of input. This simply calls `prefilter_line()` for each line of input.

This covers cases where there are multiple lines in the user entry, which is the case when the user goes back to a multiline history entry and presses enter.

**register_checker** *(checker)*

Register a checker instance.

**register_handler** *(name, handler, esc_strings)*

Register a handler instance by name with esc_strings.

**register_transformer** *(transformer)*

Register a transformer instance.

**sort_checkers** *

Sort the checkers by priority.

This must be called after the priority of a checker is changed. The `register_checker()` method calls this automatically.

**sort_transformers** *

Sort the transformers by priority.

This must be called after the priority of a transformer is changed. The `register_transformer()` method calls this automatically.

**transform_line** *(line, continue_prompt)*

Calls the enabled transformers in order of increasing priority.

**transformers**

Return a list of checkers, sorted by priority.

**unregister_checker** *(checker)*

Unregister a checker instance.

**unregister_handler** *(name, handler, esc_strings)*

Unregister a handler instance by name with esc_strings.

**unregister_transformer** *(transformer)*

Unregister a transformer instance.

```python
class IPython.core.prefilter.PrefilterTransformer (shell=None, pre-
filtr_manager=None, **kwargs)

Bases: traitlets.config.configurable.Configurable

Transform a line of user input.

__init__ (shell=None, prefilter_manager=None, **kwargs)
```
transform (line, continue_prompt)
Transform a line, returning the new one.

class IPython.core.prefilter.PrefilterChecker (shell=None, prefilter_manager=None, **kwargs)
Bases: traitlets.config.configurable.Configurable
Inspect an input line and return a handler for that line.
__init__ (shell=None, prefilter_manager=None, **kwargs)
check (line_info)
Inspect line_info and return a handler instance or None.

class IPython.core.prefilter.EmacsChecker (shell=None, prefilter_manager=None, **kwargs)
Bases: IPython.core.prefilter.PrefilterChecker
check (line_info)
Emacs ipython-mode tags certain input lines.

class IPython.core.prefilter.MacroChecker (shell=None, prefilter_manager=None, **kwargs)
Bases: IPython.core.prefilter.PrefilterChecker

class IPython.core.prefilter.IPyAutocallChecker (shell=None, prefilter_manager=None, **kwargs)
Bases: IPython.core.prefilter.PrefilterChecker
check (line_info)
Instances of IPyAutocall in user_ns get autocalled immediately.

class IPython.core.prefilter.AssignmentChecker (shell=None, prefilter_manager=None, **kwargs)
Bases: IPython.core.prefilter.PrefilterChecker
check (line_info)
Check to see if user is assigning to a var for the first time, in which case we want to avoid any sort of automagic / autocall games.

This allows users to assign to either alias or magic names true python variables (the magic/alias systems always take second seat to true python code). E.g. ls='hi’, or ls,that=1,2

class IPython.core.prefilter.AutoMagicChecker (shell=None, prefilter_manager=None, **kwargs)
Bases: IPython.core.prefilter.PrefilterChecker
check (line_info)
If the ifun is magic, and automagic is on, run it. Note: normal, non-auto magic would already have been triggered via ‘%’ in check_esc_chars. This just checks for automagic. Also, before triggering the magic handler, make sure that there is nothing in the user namespace which could shadow it.
class IPython.core.prefilter.PythonOpsChecker (shell=None, pre-
filter_manager=None, **kwargs)

check (line_info)
If the ‘rest’ of the line begins with a function call or pretty much any python operator, we should
simply execute the line (regardless of whether or not there’s a possible autocall expansion). This
avoids spurious (and very confusing) geattr() accesses.

class IPython.core.prefilter.AutocallChecker (shell=None, pre-
filter_manager=None, **kwargs)

check (line_info)
Check if the initial word/function is callable and autocall is on.

class IPython.core.prefilter.PrefilterHandler (shell=None, pre-
filter_manager=None, **kwargs)

__init__ (shell=None, prefilter_manager=None, **kwargs)

handle (line_info)
Handle normal input lines. Use as a template for handlers.

class IPython.core.prefilter.MacroHandler (shell=None, pre-
filter_manager=None, **kwargs)

Bases: IPython.core.prefilter.PrefilterHandler

class IPython.core.prefilter.MagicHandler (shell=None, pre-
filter_manager=None, **kwargs)

Bases: IPython.core.prefilter.PrefilterHandler

handle (line_info)
Execute magic functions.

class IPython.core.prefilter.AutoHandler (shell=None, pre-
filter_manager=None, **kwargs)

Bases: IPython.core.prefilter.PrefilterHandler

handle (line_info)
Handle lines which can be auto-executed, quoting if requested.

class IPython.core.prefilter.EmacsHandler (shell=None, pre-
filter_manager=None, **kwargs)

Bases: IPython.core.prefilter.PrefilterHandler

handle (line_info)
Handle input lines marked by python-mode.

8.31.2 1 Function

IPython.core.prefilter.is_shadowed (identifier, ip)
Is the given identifier defined in one of the namespaces which shadow the alias and magic names-
paces? Note that an identifier is different than ifun, because it can not contain a ‘.’ character.
8.32 Module: core.profileapp

An application for managing IPython profiles.
To be invoked as the ipython profile subcommand.

Authors:
- Min RK

8.32.1 4 Classes

class IPython.core.profileapp.ProfileLocate(**kwargs)
   Bases: IPython.core.application.BaseIPythonApplication

class IPython.core.profileapp.ProfileList(**kwargs)
   Bases: traitlets.config.application.Application

class IPython.core.profileapp.ProfileCreate(**kwargs)
   Bases: IPython.core.application.BaseIPythonApplication

class IPython.core.profileapp.ProfileApp(**kwargs)
   Bases: traitlets.config.application.Application

8.32.2 2 Functions

IPython.core.profileapp.list_profiles_in(path)
   list profiles in a given root directory

IPython.core.profileapp.list_bundled_profiles()
   list profiles that are bundled with IPython.

8.33 Module: core.profiledir

An object for managing IPython profile directories.

8.33.1 2 Classes

class IPython.core.profiledir.ProfileDirError
   Bases: Exception

class IPython.core.profiledir.ProfileDir(**kwargs)
   Bases: traitlets.config.configurable.LoggingConfigurable

   An object to manage the profile directory and its resources.
   The profile directory is used by all IPython applications, to manage configuration, logging and security.
This object knows how to find, create and manage these directories. This should be used by any code that wants to handle profiles.

```python
copy_config_file(config_file, path=None, overwrite=False)
```
Copy a default config file into the active profile directory.

Default configuration files are kept in `IPython.core.profile`. This function moves these from that location to the working profile directory.

```python
classmethod create_profile_dir(profile_dir, config=None)
```
Create a new profile directory given a full path.

**Parameters**

- `profile_dir` (*str*) – The full path to the profile directory. If it does exist, it will be used. If not, it will be created.

```python
classmethod create_profile_dir_by_name(path, name='default', config=None)
```
Create a profile dir by profile name and path.

**Parameters**

- `path` (*unicode*) – The path (directory) to put the profile directory in.
- `name` (*unicode*) – The name of the profile. The name of the profile directory will be “profile_<profile>”.

```python
classmethod find_profile_dir(profile_dir, config=None)
```
Find/create a profile dir and return its ProfileDir.

This will create the profile directory if it doesn’t exist.

**Parameters**

- `profile_dir` (*unicode or str*) – The path of the profile directory.

```python
classmethod find_profile_dir_by_name(ipython_dir, name='default', config=None)
```
Find an existing profile dir by profile name, return its ProfileDir.

This searches through a sequence of paths for a profile dir. If it is not found, a `ProfileDirError` exception will be raised.

The search path algorithm is: 1. `py3compat.getcwd()` 2. `ipython_dir`

**Parameters**

- `ipython_dir` (*unicode or str*) – The IPython directory to use.
- `name` (*unicode or str*) – The name of the profile. The name of the profile directory will be “profile_<profile>”.

### 8.34 Module: core.prompts

Classes for handling input/output prompts.

Authors:

- Fernando Perez
• Brian Granger
• Thomas Kluyver

8.34.1 3 Classes

class IPython.core.prompts.LazyEvaluate(func, *args, **kwargs)
   Bases: object
   This is used for formatting strings with values that need to be updated at that time, such as the current
time or working directory.
   __init__(func, *args, **kwargs)

class IPython.core.prompts.UserNSFormatter(shell)
   Bases: string.Formatter
   A Formatter that falls back on a shell’s user_ns and __builtins__ for name resolution
   __init__(shell)

class IPython.core.prompts.PromptManager(shell, **kwargs)
   Bases: traitlets.config.configurable.Configurable
   This is the primary interface for producing IPython’s prompts.
   __init__(shell, **kwargs)
   render(name, color=True, just=None, **kwargs)
      Render the selected prompt.
         Parameters
         • name (str) – Which prompt to render. One of ‘in’, ‘in2’, ‘out’, ‘rewrite’
         • color (bool) – If True (default), include ANSI escape sequences for a
coloured prompt.
         • just (bool) – If True, justify the prompt to the width of the last prompt. The
default is stored in self.justify.
         • **kwargs – Additional arguments will be passed to the string formatting op-
eration, so they can override the values that would otherwise fill in the template.

   Returns
   Return type  A string containing the rendered prompt.

   update_prompt(name, new_template=None)
      This is called when a prompt template is updated. It processes abbreviations used in the prompt
template (like #) and calculates how many invisible characters (ANSI colour escapes) the result-
ing prompt contains.

      It is also called for each prompt on changing the colour scheme. In both cases, traitlets should
take care of calling this automatically.
8.34.2 3 Functions

IPython.core.prompts.multiple_replace(dict, text)
Replace in ‘text’ all occurrences of any key in the given dictionary by its corresponding value. Returns
the new string.

IPython.core.prompts.cwd_filt(depth)
Return the last depth elements of the current working directory.
$HOME is always replaced with ‘~’. If depth==0, the full path is returned.

IPython.core.prompts.cwd_filt2(depth)
Return the last depth elements of the current working directory.
$HOME is always replaced with ‘~’. If depth==0, the full path is returned.

8.35 Module: core.pylabtools

Pylab (matplotlib) support utilities.

8.35.1 10 Functions

IPython.core.pylabtools.getfigs(*fig_nums)
Get a list of matplotlib figures by figure numbers.
If no arguments are given, all available figures are returned. If the argument list contains references
to invalid figures, a warning is printed but the function continues pasting further figures.

Parameters figs (tuple) – A tuple of ints giving the figure numbers of the figures to
return.

IPython.core.pylabtools.figsize(sizex, sizey)
Set the default figure size to be [sizex, sizey].
This is just an easy to remember, convenience wrapper that sets:

```
matplotlib.rcParams['figure.figsize'] = [sizex, sizey]
```

IPython.core.pylabtools.print_figure(fig, fmt='png', bbox_inches='tight', **kwargs)
Print a figure to an image, and return the resulting file data
Returned data will be bytes unless fmt='svg', in which case it will be unicode.
Any keyword args are passed to fig.canvas.print_figure, such as quality or bbox_inches.

IPython.core.pylabtools.retina_figure(fig, **kwargs)
Format a figure as a pixel-doubled (retina) PNG

IPython.core.pylabtools.mpl_runner(safe_execfile)
Factory to return a matplotlib-enabled runner for %run.
Parameters **safe_execfile** (*function*) – This must be a function with the same interface as the `safe_execfile()` method of IPython.

Returns

- A function suitable for use as the `runner` argument of the `%run` magic function.

**IPython.core.pylabtools.select_figure Formats** (*shell, formats, **kwargs*)

Select figure formats for the inline backend.

Parameters

- **shell** (*InteractiveShell*) – The main IPython instance.


- ****kwargs – Extra keyword arguments to be passed to fig.canvas.print_figure.

**IPython.core.pylabtools.find_gui_and_backend** (*gui=None, gui_select=None*)

Given a gui string return the gui and mpl backend.

Parameters


- **gui_select** (*str*) – Can be one of (‘tk’, ‘gtk’, ‘wx’, ‘qt’, ‘qt4’, ‘inline’). This is any gui already selected by the shell.

Returns

- A tuple of (gui, backend) where backend is one of (‘TkAgg’, ‘GTKAgg’, ‘WXAgg’, ‘Qt4Agg’, ‘module’ (/ipypython.pylab.backend_inline’).)

**IPython.core.pylabtools.activate_matplotlib** (*backend*)

Activate the given backend and set interactive to True.

**IPython.core.pylabtools.import_pylab** (*user_ns, import_all=True*)

Populate the namespace with pylab-related values.

Imports matplotlib, pylab, numpy, and everything from pylab and numpy.

Also imports a few names from IPython (figsize, display, getfigs)

**IPython.core.pylabtools.configure_inline_support** (*shell, backend*)

Configure an IPython shell object for matplotlib use.

Parameters

- **shell** (*InteractiveShell instance*) –

- **backend** (*matplotlib backend*) –
8.36 Module: core.shellapp

A mixin for Application classes that launch InteractiveShell instances, load extensions, etc.

8.36.1 1 Class

class IPython.core.shellapp.InteractiveShellApp(**kwargs)
    Bases: traitlets.config.configurable.Configurable

A Mixin for applications that start InteractiveShell instances.

Provides configurables for loading extensions and executing files as part of configuring a Shell environment.

The following methods should be called by the initialize() method of the subclass:

• init_path()
• init_shell() (to be implemented by the subclass)
• init_gui_pylab()
• init_extensions()
• init_code()

init_code()
    run the pre-flight code, specified via exec_lines

init_extensions()
    Load all IPython extensions in IPythonApp.extensions.
    This uses the ExtensionManager.load_extensions() to load all the extensions listed in self.extensions.

init_gui_pylab()
    Enable GUI event loop integration, taking pylab into account.

init_path()
    Add current working directory, ‘’ , to sys.path

8.37 Module: core.splitinput

Simple utility for splitting user input. This is used by both inputsplitter and prefilter.

Authors:

• Brian Granger
• Fernando Perez
8.37.1 1 Class

class IPython.core.splitinput.LineInfo (line, continue_prompt=False)
    Bases: object

    A single line of input and associated info.
    Includes the following as properties:

    line The original, raw line

    continue_prompt Is this line a continuation in a sequence of multiline input?

    pre Any leading whitespace.

    esc The escape character(s) in pre or the empty string if there isn’t one. Note that ‘!!’ and ‘??’ are possible values for esc. Otherwise it will always be a single character.

    ifun The ‘function part’, which is basically the maximal initial sequence of valid python identifiers and the ‘.’ character. This is what is checked for alias and magic transformations, used for auto-calling, etc. In contrast to Python identifiers, it may start with “%” and contain “*”.

    the_rest Everything else on the line.

    __init__ (line, continue_prompt=False)

    ofind (ip)
        Do a full, attribute-walking lookup of the ifun in the various namespaces for the given IPython InteractiveShell instance.

        Return a dict with keys: {found, obj, ospace, ismagic}

        Note: can cause state changes because of calling getattr, but should only be run if autocall is on and if the line hasn’t matched any other, less dangerous handlers.

        Does cache the results of the call, so can be called multiple times without worrying about further damaging state.

8.37.2 1 Function

IPython.core.splitinput.split_user_input (line, pattern=None)
    Split user input into initial whitespace, escape character, function part and the rest.

8.38 Module: core.ultratb

Verbose and colourful traceback formatting.

ColorTB

I’ve always found it a bit hard to visually parse tracebacks in Python. The ColorTB class is a solution to that problem. It colors the different parts of a traceback in a manner similar to what you would expect from a syntax-highlighting text editor.

Installation instructions for ColorTB:
```python
import sys, ultratb
sys.excepthook = ultratb.ColorTB()
```

**VerboseTB**

I’ve also included a port of Ka-Ping Yee’s “cgitb.py” that produces all kinds of useful info when a traceback occurs. Ping originally had it spit out HTML and intended it for CGI programmers, but why should they have all the fun? I altered it to spit out colored text to the terminal. It’s a bit overwhelming, but kind of neat, and maybe useful for long-running programs that you believe are bug-free. If a crash *does* occur in that type of program you want details. Give it a shot—you’ll love it or you’ll hate it.

**Note:** The Verbose mode prints the variables currently visible where the exception happened (shortening their strings if too long). This can potentially be very slow, if you happen to have a huge data structure whose string representation is complex to compute. Your computer may appear to freeze for a while with cpu usage at 100%. If this occurs, you can cancel the traceback with Ctrl-C (maybe hitting it more than once).

If you encounter this kind of situation often, you may want to use the Verbose_novars mode instead of the regular Verbose, which avoids formatting variables (but otherwise includes the information and context given by Verbose).

Installation instructions for VerboseTB:

```python
import sys, ultratb
sys.excepthook = ultratb.VerboseTB()
```

Note: Much of the code in this module was lifted verbatim from the standard library module ‘traceback.py’ and Ka-Ping Yee’s ‘cgitb.py’.

### 8.38.1 Color schemes

The colors are defined in the class TBTools through the use of the ColorSchemeTable class. Currently the following exist:

- **NoColor**: allows all of this module to be used in any terminal (the color escapes are just dummy blank strings).
- **Linux**: is meant to look good in a terminal like the Linux console (black or very dark background).
- **LightBG**: similar to Linux but swaps dark/light colors to be more readable in light background terminals.

You can implement other color schemes easily, the syntax is fairly self-explanatory. Please send back new schemes you develop to the author for possible inclusion in future releases.

Inheritance diagram:
8.38.2 7 Classes

class IPython.core.ultratb.TBTools (color_scheme='NoColor', call_pdb=False, ostream=None)

Bases: object

Basic tools used by all traceback printer classes.

__init__ (color_scheme='NoColor', call_pdb=False, ostream=None)

color_toggle ()

Toggle between the currently active color scheme and NoColor.

ostream

Output stream that exceptions are written to.

Valid values are:

• None: the default, which means that IPython will dynamically resolve to io.stdout. This ensures compatibility with most tools, including Windows (where plain stdout doesn’t recognize ANSI escapes).

• Any object with ‘write’ and ‘flush’ attributes.

set_colors (*args, **kw)

Shorthand access to the color table scheme selector method.

stb2text (stb)

Convert a structured traceback (a list) to a string.

structured_traceback (etype, evalue, tb, tb_offset=None, context=5, mode=None)

Return a list of traceback frames.

Must be implemented by each class.

text (etype, value, tb, tb_offset=None, context=5)

Return formatted traceback.

Subclasses may override this if they add extra arguments.

class IPython.core.ultratb.ListTB (color_scheme='NoColor', call_pdb=False, ostream=None)

Bases: IPython.core.ultratb.TBTools

Print traceback information from a traceback list, with optional color.
Calling requires 3 arguments: (etype, evalue, elist) as would be obtained by:

```python
etype, evalue, tb = sys.exc_info()
if tb:
    elist = traceback.extract_tb(tb)
else:
    elist = None
```

It can thus be used by programs which need to process the traceback before printing (such as console replacements based on the code module from the standard library).

Because they are meant to be called without a full traceback (only a list), instances of this class can’t call the interactive pdb debugger.

```python
__init__(color_scheme='NoColor', call_pdb=False, ostream=None)
```

**get_exception_only(etype, value)**

Only print the exception type and message, without a traceback.

**Parameters**

- `etype` *(exception type)*
- `value` *(exception value)*

**show_exception_only(etype, evalue)**

Only print the exception type and message, without a traceback.

**Parameters**

- `etype` *(exception type)*
- `value` *(exception value)*

**structured_traceback(etype, value, elist, tb_offset=None, context=5)**

Return a color formatted string with the traceback info.

**Parameters**

- `etype` *(exception type)* – Type of the exception raised.
- `value` *(object)* – Data stored in the exception
- `elist` *(list)* – List of frames, see class docstring for details.
- `tb_offset` *(int, optional)* – Number of frames in the traceback to skip. If not given, the instance value is used (set in constructor).
- `context` *(int, optional)* – Number of lines of context information to print.

**Returns**

**Return type** String with formatted exception.

```python
class IPython.core.ultratb.VerboseTB(color_scheme='Linux', call_pdb=False, ostream=None, tb_offset=0, long_header=False, include_vars=True, check_cache=None)
Bases: IPython.core.ultratb.TBTools
```
A port of Ka-Ping Yee’s cgitb.py module that outputs color text instead of HTML. Requires inspect
and pydoc. Crazy, man.

Modified version which optionally strips the topmost entries from the traceback, to be used with
alternate interpreters (because their own code would appear in the traceback).

```python
__init__(color_scheme='Linux', call_pdb=False, ostream=None, tb_offset=0,
         long_header=False, include_vars=True, check_cache=None)
```

Specify traceback offset, headers and color scheme.

Define how many frames to drop from the tracebacks. Calling it with tb_offset=1 allows use of
this handler in interpreters which will have their own code at the top of the traceback (VerboseTB
will first remove that frame before printing the traceback info).

```python
__call__(etype=None, evalue=None, etb=None)
```

This hook can replace sys.excepthook (for Python 2.1 or higher).

def debugger(force=False)
    Call up the pdb debugger if desired, always clean up the tb reference.

    Keywords:
    • force(False): by default, this routine checks the instance call_pdb flag and does not actually
      invoke the debugger if the flag is false. The ‘force’ option forces the debugger to activate
      even if the flag is false.

    If the call_pdb flag is set, the pdb interactive debugger is invoked. In all cases, the self.tb refer-
    ence to the current traceback is deleted to prevent lingering references which hamper memory
    management.

    Note that each call to pdb() does an ‘import readline’, so if your app requires a special setup for
    the readline completers, you’ll have to fix that by hand after invoking the exception handler.

```python
format_exception_as_a_whole(etype, evalue, etb, number_of_lines_of_context, tb_offset)
```

Formats the header, traceback and exception message for a single exception.

This may be called multiple times by Python 3 exception chaining (PEP 3134).

```python
format_record(frame, file, lnum, func, lines, index)
```

Format a single stack frame

```python
format_records(records, last_unique, recursion_repeat)
```

Format the stack frames of the traceback

```python
structured_traceback(etype, evalue, etb, tb_offset=None, number_of_lines_of_context=5)
```

Return a nice text document describing the traceback.

```python
class IPython.core.ultratb.FormattedTB (mode='Plain', color_scheme='Linux',
                                          call_pdb=False, ostream=None, tb_offset=0,
                                          long_header=False, include_vars=False,
                                          check_cache=None)
```

Bases: IPython.core.ultratb.VerboseTB, IPython.core.ultratb.ListTB

Subclass ListTB but allow calling with a traceback.
It can thus be used as a sys.excepthook for Python > 2.1.

Also adds ‘Context’ and ‘Verbose’ modes, not available in ListTB.

Allows a tb_offset to be specified. This is useful for situations where one needs to remove a number of topmost frames from the traceback (such as occurs with python programs that themselves execute other python code, like Python shells).

```python
__init__(mode='Plain', color_scheme='Linux', call_pdb=False, ostream=None, tb_offset=0, long_header=False, include_vars=False, check_cache=None)
```

```python
set_mode(mode=None)
```

Switch to the desired mode.

If mode is not specified, cycles through the available modes.

```python
stb2text(stb)
```

Convert a structured traceback (a list) to a string.

```python
class IPython.core.ultratb.AutoFormattedTB(mode='Plain', color_scheme='Linux', call_pdb=False, ostream=None, tb_offset=0, long_header=False, include_vars=False, check_cache=None)
```

Bases: `IPython.core.ultratb.FormattedTB`

A traceback printer which can be called on the fly.

It will find out about exceptions by itself.

A brief example:

```python
AutoTB = AutoFormattedTB(mode = 'Verbose', color_scheme='Linux')
try:
    ...
except:
    AutoTB()  # or AutoTB(out=logfile) where logfile is an open file object
```

```python
__call__(etype=None, evalue=None, etb=None, out=None, tb_offset=None)
```

Print out a formatted exception traceback.

Optional arguments:

- `out`: an open file-like object to direct output to.
- `tb_offset`: the number of frames to skip over in the stack, on a per-call basis (this overrides temporarily the instance’s `tb_offset` given at initialization time.

```python
class IPython.core.ultratb.ColorTB(color_scheme='Linux', call_pdb=0, **kwargs)
```

Bases: `IPython.core.ultratb.FormattedTB`

Shorthand to initialize a FormattedTB in Linux colors mode.

```python
__init__(color_scheme='Linux', call_pdb=0, **kwargs)
```

```python
class IPython.core.ultratb.SyntaxTB(color_scheme='NoColor')
```

Bases: `IPython.core.ultratb.ListTB`

---

8.38. Module: core.ultratb
Extension which holds some state: the last exception value

```python
__init__(color_scheme='NoColor')
```

```python
clear_err_state()
```

Return the current error state and clear it

```python
stb2text(stb)
```

Convert a structured traceback (a list) to a string.

### 8.38.3 10 Functions

**IPython.core.ultratb.inspect_error()**

Print a message about internal inspect errors.

These are unfortunately quite common.

**IPython.core.ultratb.findsource(object)**

Return the entire source file and starting line number for an object.

The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of all the lines in the file and the line number indexes a line in that list. An IOError is raised if the source code cannot be retrieved.

**IPython.core.ultratb.getargs(co)**

Get information about the arguments accepted by a code object.

Three things are returned: (args, varargs, varkw), where ‘args’ is a list of argument names (possibly containing nested lists), and ‘varargs’ and ‘varkw’ are the names of the * and ** arguments or None.

**IPython.core.ultratb.with_patch_inspect(f)**

decorator for monkeypatching inspect.findsource

**IPython.core.ultratb.fix_frame_records_filenames(records)**

Try to fix the filenames in each record from inspect.getinnerframes().

Particularly, modules loaded from within zip files have useless filenames attached to their code object, and inspect.getinnerframes() just uses it.

**IPython.core.ultratb.is_recursion_error(etype, value, records)**

**IPython.core.ultratb.find_recursion(etype, value, records)**

Identify the repeating stack frames from a RecursionError traceback

‘records’ is a list as returned by VerboseTB.get_records()

Returns (last_unique, repeat_length)

**IPython.core.ultratb.text_repr(value)**

Hopefully pretty robust repr equivalent.

**IPython.core.ultratb.eqrepr(value, repr=<function text_repr>)**

**IPython.core.ultratb.nullrepr(value, repr=<function text_repr>)**
8.39 Module: core.usage

Usage information for the main IPython applications.

8.39.1 1 Function

IPython.core.usage.page_guiref(arg_s=None)

Show a basic reference about the GUI Console.

8.40 Module: display

Public API for display tools in IPython.

8.40.1 18 Classes

class IPython.display.Audio(data=None, filename=None, url=None, embed=None, rate=None, autoplay=False)

Bases: IPython.core.display.DisplayObject

Create an audio object.

When this object is returned by an input cell or passed to the display function, it will result in Audio controls being displayed in the frontend (only works in the notebook).

Parameters

- **data** (numpy array, list, unicode, str or bytes) – Can be one of
  - Numpy 1d array containing the desired waveform (mono)
  - Numpy 2d array containing waveforms for each channel. Shape=(NCHAN, NSAMPLES). For the standard channel order, see http://msdn.microsoft.com/en-us/library/windows/hardware/dn653308(v=vs.85).aspx
  - List of float or integer representing the waveform (mono)
  - String containing the filename
  - Bytestring containing raw PCM data or
  - URL pointing to a file on the web.

If the array option is used the waveform will be normalized.

If a filename or url is used the format support will be browser dependent.

- **url** (unicode) – A URL to download the data from.
- **filename** (unicode) – Path to a local file to load the data from.
• **embed**(boolean) – Should the audio data be embedded using a data URI (True) or should the original source be referenced. Set this to True if you want the audio to playable later with no internet connection in the notebook.

  Default is True, unless the keyword argument `url` is set, then default value is False.

• **rate**(integer) – The sampling rate of the raw data. Only required when data parameter is being used as an array

• **autoplay**(bool) – Set to True if the audio should immediately start playing.

  Default is False.

**Examples**

```python
# Generate a sound
import numpy as np
framerate = 44100
t = np.linspace(0, 5, framerate*5)
data = np.sin(2*np.pi*220*t) + np.sin(2*np.pi*224*t))
Audio(data, rate=framerate)

# Can also do stereo or more channels
dataleft = np.sin(2*np.pi*220*t)
dataright = np.sin(2*np.pi*224*t)
Audio([dataleft, dataright], rate=framerate)

Audio("http://www.nch.com.au/acm/8k16bitpcm.wav")  # From URL
Audio(url="http://www.w3schools.com/html/horse.ogg")

Audio('/path/to/sound.wav')  # From file
Audio(filename='/path/to/sound.ogg')

Audio(b'RAW_WAV_DATA..)  # From bytes
Audio(data=b'RAW_WAV_DATA..)
```

```
__init__ (data=None, filename=None, url=None, embed=None, rate=None, autoplay=False)
reload()

Reload the raw data from file or URL.
```

**class** `IPython.display.DisplayObject (data=None, url=None, filename=None)`

**Bases:** `object`

An object that wraps data to be displayed.

```
__init__ (data=None, url=None, filename=None)

Create a display object given raw data.

When this object is returned by an expression or passed to the display function, it will result in the data being displayed in the frontend. The MIME type of the data should match the subclasses
used, so the Png subclass should be used for ‘image/png’ data. If the data is a URL, the data will first be downloaded and then displayed. If

Parameters

- **data** *(unicode, str or bytes)* – The raw data or a URL or file to load the data from
- **url** *(unicode)* – A URL to download the data from.
- **filename** *(unicode)* – Path to a local file to load the data from.

**reload()**
Reload the raw data from file or URL.

```python
class IPython.display.FileLink(path, url_prefix=' ', result_html_prefix=' ', result_html_suffix='<br>')

Bases: object
```

Class for embedding a local file link in an IPython session, based on path
e.g. to embed a link that was generated in the IPython notebook as my/data.txt
you would do:
```
local_file = FileLink("my/data.txt")
display(local_file)
```
or in the HTML notebook, just:
```
FileLink("my/data.txt")
```

```python
__init__(path, url_prefix=' ', result_html_prefix=' ', result_html_suffix='<br>')

Parameters

- **path** *(str)* – path to the file or directory that should be formatted
- **directory_prefix** *(str)* – prefix to be prepended to all files to form a working link [default: ‘files’]
- **result_html_prefix** *(str)* – text to append to beginning to link [default: none]
- **result_html_suffix** *(str)* – text to append at the end of link [default: ‘<br>’]

__repr__()
return absolute path to file
```

```python
class IPython.display.FileLinks(path, url_prefix=' ', included_suffixes=None, result_html_prefix=' ', result_html_suffix='<br>', notebook_display_formatter=None, terminal_display_formatter=None, recursive=True)

Bases: IPython.lib.display.FileLink
```

Class for embedding local file links in an IPython session, based on path
e.g. to embed links to files that were generated in the IPython notebook under `my/data`, you would do:

```python
local_files = FileLinks("my/data")
display(local_files)
```

or in the HTML notebook, just:

```python
FileLinks("my/data")
```

```
__init__ (path, url_prefix='', included_suffixes=None, result_html_prefix='', result_html_suffix='<br>', notebook_display_formatter=None, terminal_display_formatter=None, recursive=True)
```

See `FileLink` for the `path`, `url_prefix`, `result_html_prefix` and `result_html_suffix` parameters.

`included_suffixes` [list] Filename suffixes to include when formatting output [default: include all files]

`notebook_display_formatter` [function] Used to format links for display in the notebook. See discussion of formatter functions below.

`terminal_display_formatter` [function] Used to format links for display in the terminal. See discussion of formatter functions below.

Formatter functions must be of the form:

```python
f(dirname, fnames, included_suffixes)
```

`dirname` [str] The name of a directory

`fnames` [list] The files in that directory

`included_suffixes` [list] The file suffixes that should be included in the output (passing None means to include all suffixes in the output in the built-in formatters)

`recursive` [boolean] Whether to recurse into subdirectories. Default is True.

The function should return a list of lines that will be printed in the notebook (if passing notebook_display_formatter) or the terminal (if passing terminal_display_formatter). This function is iterated over for each directory in self.path. Default formatters are in place, can be passed here to support alternative formatting.

```
__repr__ ()
return newline-separated absolute paths
```

```
class IPython.display.HTML (data=None, url=None, filename=None)
Bases: IPython.core.display.TextDisplayObject
__html__ ()
This method exists to inform other HTML-using modules (e.g. MarkupSafe, htmltag, etc) that this object is HTML and does not need things like special characters (<>&) escaped.
```

```
class IPython.display.IFrame (src, width, height, **kwargs)
Bases: object
```

Chapter 8. The IPython API
Generic class to embed an iframe in an IPython notebook

```python
__init__(src, width, height, **kwargs)
```

### class IPython.display.Image(data=None, url=None, filename=None, format=None, embed=None, width=None, height=None, retina=False, uncon fined=False, metadata=None)

Bases: IPython.core.display.DisplayObject

```python
__init__(data=None, url=None, filename=None, format=None, embed=None, width=None, height=None, retina=False, uncon fined=False, metadata=None)
```

Create a PNG/JPEG image object given raw data.

When this object is returned by an input cell or passed to the display function, it will result in the image being displayed in the frontend.

**Parameters**

- **data** *(unicode, str or bytes)* – The raw image data or a URL or filename to load the data from. This always results in embedded image data.

- **url** *(unicode)* – A URL to download the data from. If you specify `url=`, the image data will not be embedded unless you also specify `embed=True`.

- **filename** *(unicode)* – Path to a local file to load the data from. Images from a file are always embedded.

- **format** *(unicode)* – The format of the image data (png/jpeg/jpg). If a filename or URL is given for `format` will be inferred from the filename extension.

- **embed** *(bool)* – Should the image data be embedded using a data URI (True) or be loaded using an `<img>` tag. Set this to True if you want the image to be viewable later with no internet connection in the notebook.

  Default is True, unless the keyword argument `url` is set, then default value is False.

  Note that QtConsole is not able to display images if `embed` is set to False

- **width** *(int)* – Width to which to constrain the image in html

- **height** *(int)* – Height to which to constrain the image in html

- **retina** *(bool)* – Automatically set the width and height to half of the measured width and height. This only works for embedded images because it reads the width/height from image data. For non-embedded images, you can just set the desired display width and height directly.

- **unconfined** *(bool)* – Set `unconfined=True` to disable max-width confinement of the image.

- **metadata** *(dict)* – Specify extra metadata to attach to the image.
Examples

# embedded image data, works in qtconsole and notebook # when passed positionally, # the first arg can be any of raw image data, # a URL, or a filename from which # to load image data. # The result is always embedding image data for inline images. Image('http://www.google.fr/images/srpr/logo3w.png') Image('/path/to/image.jpg') Image(b'RAW_PNG_DATA...')

# Specifying Image(url=...) does not embed the image data, # it only generates <img> tag with a link to the source. # This will not work in the qtconsole or offline. Image(url='http://www.google.fr/images/srpr/logo3w.png')

text = "example text"

reload()

Reload the raw data from file or URL.

class IPython.display.JSON (data=None, url=None, filename=None)
    Bases: IPython.core.display.DisplayObject

    JSON expects a JSON-able dict or list
    not an already-serialized JSON string.
    Scalar types (None, number, string) are not allowed, only dict or list containers.

class IPython.display.Javascript (data=None, url=None, filename=None, lib=None, css=None)
    Bases: IPython.core.display.TextDisplayObject

    __init__ (data=None, url=None, filename=None, lib=None, css=None)
        Create a Javascript display object given raw data.

        When this object is returned by an expression or passed to the display function, it will result in
        the data being displayed in the frontend. If the data is a URL, the data will first be downloaded
        and then displayed.

        In the Notebook, the containing element will be available as element, and jQuery will be
        available. Content appended to element will be visible in the output area.

    Parameters

    * data (unicode, str or bytes) – The Javascript source code or a URL to
download it from.
    * url (unicode) – A URL to download the data from.
    * filename (unicode) – Path to a local file to load the data from.
    * lib (list or str) – A sequence of Javascript library URLs to load asyn-
        chronously before running the source code. The full URLs of the libraries
        should be given. A single Javascript library URL can also be given as a string.
    * css (: list or str) – A sequence of css files to load before running the
        source code. The full URLs of the css files should be given. A single css URL
        can also be given as a string.
class IPython.display.Latex(data=None, url=None, filename=None)
   Bases: IPython.core.display.TextDisplayObject

class IPython.display.Markdown(data=None, url=None, filename=None)
   Bases: IPython.core.display.TextDisplayObject

class IPython.display.Math(data=None, url=None, filename=None)
   Bases: IPython.core.display.TextDisplayObject

class IPython.display.Pretty(data=None, url=None, filename=None)
   Bases: IPython.core.display.TextDisplayObject

class IPython.display.SVG(data=None, url=None, filename=None)
   Bases: IPython.core.display.DisplayObject

class IPython.display.ScribdDocument(id, width=400, height=300, **kwargs)
   Bases: IPython.lib.display.IFrame
   
   Class for embedding a Scribd document in an IPython session
   
   Use the start_page params to specify a starting point in the document
   Use the view_mode params to
   specify display type
   one off scroll | slideshow | book
   
   e.g to Display Wes' foundational paper about PANDAS in book mode from page 3
   ScribdDocument(71048089, width=800, height=400, start_page=3, view_mode="book")

   __init__ (id, width=400, height=300, **kwargs)

class IPython.display.TextDisplayObject(data=None, url=None, filename=None)
   Bases: IPython.core.display.DisplayObject
   
   Validate that display data is text

class IPython.display.VimeoVideo(id, width=400, height=300, **kwargs)
   Bases: IPython.lib.display.IFrame
   
   Class for embedding a Vimeo video in an IPython session, based on its video id.

   __init__ (id, width=400, height=300, **kwargs)

class IPython.display.YouTubeVideo(id, width=400, height=300, **kwargs)
   Bases: IPython.lib.display.IFrame
   
   Class for embedding a YouTube Video in an IPython session, based on its video id.
   
   e.g. to embed the video from https://www.youtube.com/watch?v=foo, you would do:

   vid = YouTubeVideo("foo")
   display(vid)

   To start from 30 seconds:

   vid = YouTubeVideo("abc", start=30)
   display(vid)

   To calculate seconds from time as hours, minutes, seconds use `datetime.timedelta`:
start=int(timedelta(hours=1, minutes=46, seconds=40).total_seconds())

Other parameters can be provided as documented at https://developers.google.com/youtube/player_parameters#parameter-

subheader

When converting the notebook using nbconvert, a jpeg representation of the video will be inserted in

the document.

__init__ (id, width=400, height=300, **kwargs)

8.40.2 15 Functions

IPython.display.clear_output (wait=False)

Clear the output of the current cell receiving output.

Parameters

**wait** (bool [default: false]) – Wait to clear the output until new

output is available to replace it.

IPython.display.display (*objs, **kwargs)

Display a Python object in all frontends.

By default all representations will be computed and sent to the frontends. Frontends can decide which

representation is used and how.

Parameters

- **objs** (tuple of objects) – The Python objects to display.
- **raw** (bool, optional) – Are the objects to be displayed already mimetype-

keyed dicts of raw display data, or Python objects that need to be formatted before

display? [default: False]
- **include** (list or tuple, optional) – A list of format type strings

(MIME types) to include in the format data dict. If this is set only the format
types included in this list will be computed.
- **exclude** (list or tuple, optional) – A list of format type strings

(MIME types) to exclude in the format data dict. If this is set all format types

will be computed, except for those included in this argument.
- **metadata** (dict, optional) – A dictionary of metadata to associate with

the output. mime-type keys in this dictionary will be associated with the individ-

ual representation formats, if they exist.

IPython.display.display_html (*objs, **kwargs)

Display the HTML representation of an object.

Note: If raw=False and the object does not have a HTML representation, no HTML will be shown.

Parameters

- **objs** (tuple of objects) – The Python objects to display, or if raw=True

raw HTML data to display.
• **raw** (*bool*) – Are the data objects raw data or Python objects that need to be formatted before display? [default: False]

• **metadata** (*dict* (optional)) – Metadata to be associated with the specific mimetype output.

IPython.display.display_javascript (*objs*, **kwargs)
Display the Javascript representation of an object.

**Parameters**

• **objs** (*tuple of objects*) – The Python objects to display, or if raw=True raw javascript data to display.

• **raw** (*bool*) – Are the data objects raw data or Python objects that need to be formatted before display? [default: False]

• **metadata** (*dict* (optional)) – Metadata to be associated with the specific mimetype output.

IPython.display.display_jpeg (*objs*, **kwargs)
Display the JPEG representation of an object.

**Parameters**

• **objs** (*tuple of objects*) – The Python objects to display, or if raw=True raw JPEG data to display.

• **raw** (*bool*) – Are the data objects raw data or Python objects that need to be formatted before display? [default: False]

• **metadata** (*dict* (optional)) – Metadata to be associated with the specific mimetype output.

IPython.display.display_json (*objs*, **kwargs)
Display the JSON representation of an object.

Note that not many frontends support displaying JSON.

**Parameters**

• **objs** (*tuple of objects*) – The Python objects to display, or if raw=True raw json data to display.

• **raw** (*bool*) – Are the data objects raw data or Python objects that need to be formatted before display? [default: False]

• **metadata** (*dict* (optional)) – Metadata to be associated with the specific mimetype output.

IPython.display.display_latex (*objs*, **kwargs)
Display the LaTeX representation of an object.

**Parameters**

• **objs** (*tuple of objects*) – The Python objects to display, or if raw=True raw latex data to display.
• **raw (bool)** – Are the data objects raw data or Python objects that need to be formatted before display? [default: False]

• **metadata (dict (optional))** – Metadata to be associated with the specific mimetype output.

**IPython.display.display_markdown(*objs, **kwargs)**

Displays the Markdown representation of an object.

**Parameters**

• **objs (tuple of objects)** – The Python objects to display, or if raw=True raw markdown data to display.

• **raw (bool)** – Are the data objects raw data or Python objects that need to be formatted before display? [default: False]

• **metadata (dict (optional))** – Metadata to be associated with the specific mimetype output.

**IPython.display.display_pdf(*objs, **kwargs)**

Display the PDF representation of an object.

**Parameters**

• **objs (tuple of objects)** – The Python objects to display, or if raw=True raw javascript data to display.

• **raw (bool)** – Are the data objects raw data or Python objects that need to be formatted before display? [default: False]

• **metadata (dict (optional))** – Metadata to be associated with the specific mimetype output.

**IPython.display.display_png(*objs, **kwargs)**

Display the PNG representation of an object.

**Parameters**

• **objs (tuple of objects)** – The Python objects to display, or if raw=True raw png data to display.

• **raw (bool)** – Are the data objects raw data or Python objects that need to be formatted before display? [default: False]

• **metadata (dict (optional))** – Metadata to be associated with the specific mimetype output.

**IPython.display.display_pretty(*objs, **kwargs)**

Display the pretty (default) representation of an object.

**Parameters**

• **objs (tuple of objects)** – The Python objects to display, or if raw=True raw text data to display.

• **raw (bool)** – Are the data objects raw data or Python objects that need to be formatted before display? [default: False]
**metadata** *(dict (optional))* – Metadata to be associated with the specific mimetype output.

**IPython.display.display_svg(** *objs, **kwargs*)

Display the SVG representation of an object.

**Parameters**

- **objs** *(tuple of objects)* – The Python objects to display, or if raw=True raw svg data to display.
- **raw** *(bool)* – Are the data objects raw data or Python objects that need to be formatted before display? [default: False]
- **metadata** *(dict (optional))* – Metadata to be associated with the specific mimetype output.

**IPython.display.publish_display_data** *(data, metadata=None, source=None)*

Publish data and metadata to all frontends.

See the display_data message in the messaging documentation for more details about this message type.

The following MIME types are currently implemented:

- text/plain
- text/html
- text/markdown
- text/latex
- application/json
- application/javascript
- image/png
- image/jpeg
- image/svg+xml

**Parameters**

- **data** *(dict)* – A dictionary having keys that are valid MIME types (like 'text/plain' or 'image/svg+xml') and values that are the data for that MIME type. The data itself must be a JSON’able data structure. Minimally all data should have the ‘text/plain’ data, which can be displayed by all frontends. If more than the plain text is given, it is up to the frontend to decide which representation to use.
- **metadata** *(dict)* – A dictionary for metadata related to the data. This can contain arbitrary key, value pairs that frontends can use to interpret the data. mimetype keys matching those in data can be used to specify metadata about particular representations.
- **source** *(str, deprecated)* – Unused.
**IPython.display**.set_matplotlib_close(close=True)

Set whether the inline backend closes all figures automatically or not.

By default, the inline backend used in the IPython Notebook will close all matplotlib figures automatically after each cell is run. This means that plots in different cells won’t interfere. Sometimes, you may want to make a plot in one cell and then refine it in later cells. This can be accomplished by:

In [1]: set_matplotlib_close(False)

To set this in your config files use the following:

c.InlineBackend.close_figures = False

**Parameters**

- close (bool) – Should all matplotlib figures be automatically closed after each cell is run?

**IPython.display**.set_matplotlib_formats(*formats, **kwargs)

Select figure formats for the inline backend. Optionally pass quality for JPEG.

For example, this enables PNG and JPEG output with a JPEG quality of 90%:

In [1]: set_matplotlib_formats('png', 'jpeg', quality=90)

To set this in your config files use the following:

c.InlineBackend.figure_formats = {'png', 'jpeg'}
c.InlineBackend.print_figure_kwarg.update({'quality' : 90})

**Parameters**

- **kwargs – Keyword args will be relayed to figure.canvas.print_figure.

### 8.41 Module: lib.backgroundjobs

Manage background (threaded) jobs conveniently from an interactive shell.

This module provides a BackgroundJobManager class. This is the main class meant for public usage, it implements an object which can create and manage new background jobs.

It also provides the actual job classes managed by these BackgroundJobManager objects, see their docstrings below.

This system was inspired by discussions with B. Granger and the BackgroundCommand class described in the book Python Scripting for Computational Science, by H. P. Langtangen:

http://folk.uio.no/hpl/scripting

(although ultimately no code from this text was used, as IPython’s system is a separate implementation).
An example notebook is provided in our documentation illustrating interactive use of the system.

### 8.41.1 4 Classes

#### class IPython.lib.backgroundjobs.BackgroundJobManager

**Bases:** object

Class to manage a pool of backgrounded threaded jobs.

Below, we assume that ‘jobs’ is a BackgroundJobManager instance.

Usage summary (see the method docstrings for details):

- `jobs.new(...) -> start a new job`
- `jobs() or jobs.status() -> print status summary of all jobs`
- `jobs[N] -> returns job number N.`
- `foo = jobs[N].result -> assign to variable foo the result of job N`
- `jobs[N].traceback() -> print the traceback of dead job N`
- `jobs.remove(N) -> remove (finished) job N`
- `jobs.flush() -> remove all finished jobs`

As a convenience feature, BackgroundJobManager instances provide the utility result and traceback methods which retrieve the corresponding information from the jobs list:

- `jobs.result(N) <-> jobs[N].result`
- `jobs.traceback(N) <-> jobs[N].traceback()`

While this appears minor, it allows you to use tab completion interactively on the job manager instance.

#### __init__()

#### __call__()

An alias to self.status(),

This allows you to simply call a job manager instance much like the Unix `jobs` shell command.

#### flush()

Flush all finished jobs (completed and dead) from lists.

Running jobs are never flushed.

It first calls `_status_new()`, to update info. If any jobs have completed since the last `_status_new()` call, the flush operation aborts.

#### new(func_or_exp, *args, **kwargs)

Add a new background job and start it in a separate thread.

There are two types of jobs which can be created:

1. Jobs based on expressions which can be passed to an eval() call. The expression must be given as a string. For example:
job_manager.new('myfunc(x,y,z=1)',[glob],[loc])

The given expression is passed to `eval()`, along with the optional global/local dicts provided. If no dicts are given, they are extracted automatically from the caller’s frame.

A Python statement is NOT a valid `eval()` expression. Basically, you can only use as an `eval()` argument something which can go on the right of an ‘=` sign and be assigned to a variable.

For example,”print ‘hello’” is not valid, but ‘2+3’ is.

2. Jobs given a function object, optionally passing additional positional arguments:

    job_manager.new(myfunc, x, y)

The function is called with the given arguments.

If you need to pass keyword arguments to your function, you must supply them as a dict named `kw`:

    job_manager.new(myfunc, x, y, kw=dict(z=1))

The reason for this asymmetry is that the `new()` method needs to maintain access to its own keywords, and this prevents name collisions between arguments to `new()` and arguments to your own functions.

In both cases, the result is stored in the `job.result` field of the background job object.

You can set `daemon` attribute of the thread by giving the keyword argument `daemon`.

Notes and caveats:

1. All threads running share the same standard output. Thus, if your background jobs generate output, it will come out on top of whatever you are currently writing. For this reason, background jobs are best used with silent functions which simply return their output.

2. Threads also all work within the same global namespace, and this system does not lock interactive variables. So if you send job to the background which operates on a mutable object for a long time, and start modifying that same mutable object interactively (or in another backgrounded job), all sorts of bizarre behaviour will occur.

3. If a background job is spending a lot of time inside a C extension module which does not release the Python Global Interpreter Lock (GIL), this will block the IPython prompt. This is simply because the Python interpreter can only switch between threads at Python bytecodes. While the execution is inside C code, the interpreter must simply wait unless the extension module releases the GIL.

4. There is no way, due to limitations in the Python threads library, to kill a thread once it has started.

    `remove(num)`

    Remove a finished (completed or dead) job.

    `result(N)` → return the result of job N.

    `status(verbose=0)`

    Print a status of all jobs currently being managed.
class IPython.lib.backgroundjobs.BackgroundJobBase
    Bases: threading.Thread

    Base class to build BackgroundJob classes.

    The derived classes must implement:
    • Their own __init__, since the one here raises NotImplementedError. The derived constructor
      must call self._init() at the end, to provide common initialization.
    • A strform attribute used in calls to __str__.
    • A call() method, which will make the actual execution call and must return a value to be held in
      the ‘result’ field of the job object.

    __init__()
      Must be implemented in subclasses.

      Subclasses must call __init__() for standard initialisation.

class IPython.lib.backgroundjobs.BackgroundJobExpr(expression, glob=None, loc=None)
    Bases: IPython.lib.backgroundjobs.BackgroundJobBase

    Evaluate an expression as a background job (uses a separate thread).

    __init__(expression, glob=None, loc=None)
      Create a new job from a string which can be fed to eval().

      global/locals dicts can be provided, which will be passed to the eval call.

class IPython.lib.backgroundjobs.BackgroundJobFunc(func, *args, **kwargs)
    Bases: IPython.lib.backgroundjobs.BackgroundJobBase

    Run a function call as a background job (uses a separate thread).

    __init__(func, *args, **kwargs)
      Create a new job from a callable object.

      Any positional arguments and keyword args given to this constructor after the initial callable are
      passed directly to it.

8.42 Module: lib.clipboard

Utilities for accessing the platform’s clipboard.

8.42.1 1 Class

class IPython.lib.clipboard.ClipboardEmpty
    Bases: ValueError
8.42.2 3 Functions

IPython.lib.clipboard.win32_clipboard_get()
Get the current clipboard’s text on Windows.
Requires Mark Hammond’s pywin32 extensions.

IPython.lib.clipboard.osx_clipboard_get()
Get the clipboard’s text on OS X.

IPython.lib.clipboard.tkinter_clipboard_get()
Get the clipboard’s text using Tkinter.
This is the default on systems that are not Windows or OS X. It may interfere with other UI toolkits and should be replaced with an implementation that uses that toolkit.

8.43 Module: lib.deepreload

Provides a reload() function that acts recursively.

Python’s normal reload function only reloads the module that it’s passed. The reload() function in this module also reloads everything imported from that module, which is useful when you’re changing files deep inside a package.

To use this as your default reload function, type this for Python 2:

```python
import __builtin__
from IPython.lib import deepreload
__builtin__.reload = deepreload.reload
```

Or this for Python 3:

```python
import builtins
from IPython.lib import deepreload
builtins.reload = deepreload.reload
```

A reference to the original reload is stored in this module as original_reload, so you can restore it later.

This code is almost entirely based on knee.py, which is a Python re-implementation of hierarchical module import.

8.43.1 9 Functions

IPython.lib.deepreload.replace_import_hook(new_import)

IPython.lib.deepreload.get_parent(globals, level)
parent, name = get_parent(globals, level)

Return the package that an import is being performed in. If globals comes from the module foo.bar.bat (not itself a package), this returns the sys.modules entry for foo.bar. If globals is from a package’s __init__.py, the package’s entry in sys.modules is returned.
If globals doesn’t come from a package or a module in a package, or a corresponding entry is not
found in sys.modules, None is returned.

```
IPython.lib.deepreload.load_next (mod, altmod, name, buf)
    mod, name, buf = load_next(mod, altmod, name, buf)
    altmod is either None or same as mod

IPython.lib.deepreload.import_submodule (mod, subname, fullname)
    m = import_submodule(mod, subname, fullname)

IPython.lib.deepreload.add_submodule (mod, submod, fullname, subname)
    mod.{subname} = submod

IPython.lib.deepreload.ensure_fromlist (mod, fromlist, buf, recursive)
    Handle ‘from module import a, b, c’ imports.

IPython.lib.deepreload.deep_import_hook (name, globals=None, locals=None,
                                           fromlist=None, level=-1)
    Replacement for __import__()

IPython.lib.deepreload.deep_reload_hook (m)
    Replacement for reload().

IPython.lib.deepreload.reload (module, exclude=('sys', 'os.path', 'builtins',
                                              '__main__'))
    Recursively reload all modules used in the given module. Optionally takes a list of modules to exclude
    from reloading. The default exclude list contains sys, __main__, and __builtin__, to prevent, e.g.,
    resetting display, exception, and io hooks.
```

### 8.44 Module: `lib.demo`

Module for interactive demos using IPython.

This module implements a few classes for running Python scripts interactively in IPython for demonstrations. With very simple markup (a few tags in comments), you can control points where the script stops executing and returns control to IPython.

#### 8.44.1 Provided classes

The classes are (see their docstrings for further details):

- Demo: pure python demos
- IPythonDemo: demos with input to be processed by IPython as if it had been typed interactively (so
  magics work, as well as any other special syntax you may have added via input prefilters).
- LineDemo: single-line version of the Demo class. These demos are executed one line at a time, and
  require no markup.
- IPythonLineDemo: IPython version of the LineDemo class (the demo is executed a line at a time, but
  processed via IPython).
• ClearMixin: mixin to make Demo classes with less visual clutter. It declares an empty marquee and a pre_cmd that clears the screen before each block (see Subclassing below).

• ClearDemo, ClearIPDemo: mixin-enabled versions of the Demo and IPythonDemo classes.

Inheritance diagram:

![Inheritance Diagram](image)

### 8.44.2 Subclassing

The classes here all include a few methods meant to make customization by subclassing more convenient. Their docstrings below have some more details:

- marquee(): generates a marquee to provide visible on-screen markers at each block start and end.
- pre_cmd(): run right before the execution of each block.
- post_cmd(): run right after the execution of each block. If the block raises an exception, this is NOT called.

### 8.44.3 Operation

The file is run in its own empty namespace (though you can pass it a string of arguments as if in a command line environment, and it will see those as sys.argv). But at each stop, the global IPython namespace is updated with the current internal demo namespace, so you can work interactively with the data accumulated so far.

By default, each block of code is printed (with syntax highlighting) before executing it and you have to confirm execution. This is intended to show the code to an audience first so you can discuss it, and only proceed with execution once you agree. There are a few tags which allow you to modify this behavior.

The supported tags are:

```
# <demo> stop
```

Defines block boundaries, the points where IPython stops execution of the file and returns to the interactive prompt.
You can optionally mark the stop tag with extra dashes before and after the word ‘stop’, to help visually distinguish the blocks in a text editor:

```markdown
# <demo> — stop —
# <demo> silent

Make a block execute silently (and hence automatically). Typically used in cases where you have some boilerplate or initialization code which you need executed but do not want to be seen in the demo.

# <demo> auto

Make a block execute automatically, but still being printed. Useful for simple code which does not warrant discussion, since it avoids the extra manual confirmation.

# <demo> auto_all

This tag can only be in the first block, and if given it overrides the individual auto tags to make the whole demo fully automatic (no block asks for confirmation). It can also be given at creation time (or the attribute set later) to override what’s in the file.

While _any_ python file can be run as a Demo instance, if there are no stop tags the whole file will run in a single block (no different that calling first %pycat and then %run). The minimal markup to make this useful is to place a set of stop tags; the other tags are only there to let you fine-tune the execution.

This is probably best explained with the simple example file below. You can copy this into a file named ex_demo.py, and try running it via:

```python
from IPython.demo import Demo
d = Demo('ex_demo.py')
d()
```

Each time you call the demo object, it runs the next block. The demo object has a few useful methods for navigation, like again(), edit(), jump(), seek() and back(). It can be reset for a new run via reset() or reloaded from disk (in case you’ve edited the source) via reload(). See their docstrings below.

Note: To make this simpler to explore, a file called “demo-exercizer.py” has been added to the “docs/examples/core” directory. Just cd to this directory in an IPython session, and type:

```bash
%run demo-exercizer.py
```

and then follow the directions.

**Example**

The following is a very simple example of a valid demo file.

```python
print 'Hello, welcome to an interactive IPython demo.'
# The mark below defines a block boundary, which is a point where IPython will
```

---

**8.44. Module: lib.demo**

---
# stop execution and return to the interactive prompt. The dashes are actually
# optional and used only as a visual aid to clearly separate blocks while
# editing the demo code.
# <demo> stop
x = 1
y = 2
# <demo> stop
# the mark below makes this block as silent
# <demo> silent
print 'This is a silent block, which gets executed but not printed.'
# <demo> stop
# <demo> auto
print 'This is an automatic block.'
print 'It is executed without asking for confirmation, but printed.'
z = x+y
print 'z=',z
# <demo> stop
# This is just another normal block.
print 'z is now: ', z
print 'bye!

############################ END EXAMPLE DEMO <ex_demo.py> ################################

8.44.4 8 Classes

```python
class IPython.lib.demo.DemoError
    Bases: Exception
class IPython.lib.demo.Demo(src, title='', arg_str='', auto_all=None)
    Bases: object
    __init__ (src, title='', arg_str='', auto_all=None)
        Make a new demo object. To run the demo, simply call the object.
        See the module docstring for full details and an example (you can use IPython.Demo? in IPython
to see it).
        Inputs:

        • **src** is either a file, or file-like object, or a string that can be resolved to a filename.

        Optional inputs:

        • **title**: a string to use as the demo name. Of most use when the demo you are making comes
          from an object that has no filename, or if you want an alternate denotation distinct from the
          filename.
```
• `arg_str('')`: a string of arguments, internally converted to a list just like `sys.argv`, so the demo script can see a similar environment.

• `auto_all(None)`: global flag to run all blocks automatically without confirmation. This attribute overrides the block-level tags and applies to the whole demo. It is an attribute of the object, and can be changed at runtime simply by reassigning it to a boolean value.

```python
__call__ (index=None)
run a block of the demo.

If index is given, it should be an integer >=1 and <= nblocks. This means that the calling convention is one off from typical Python lists. The reason for the inconsistency is that the demo always prints ‘Block n/N, and N is the total, so it would be very odd to use zero-indexing here.
```

```python
again()
Move the seek pointer back one block and re-execute.
```

```python
back (num=1)
Move the seek pointer back num blocks (default is 1).
```

```python
edit (index=None)
Edit a block.

If no number is given, use the last block executed.

This edits the in-memory copy of the demo, it does NOT modify the original source file. If you want to do that, simply open the file in an editor and use reload() when you make changes to the file. This method is meant to let you change a block during a demonstration for explanatory purposes, without damaging your original script.
```

```python
fload()
Load file object.
```

```python
jump (num=1)
Jump a given number of blocks relative to the current one.

The offset can be positive or negative, defaults to 1.
```

```python
marquee (txt='', width=78, mark='*')
Return the input string centered in a ‘marquee’.
```

```python
post_cmd()
Method called after executing each block.
```

```python
pre_cmd()
Method called before executing each block.
```

```python
reload()
Reload source from disk and initialize state.
```

```python
reset()
Reset the namespace and seek pointer to restart the demo
```

```python
run_cell (source)
Execute a string with one or more lines of code
```
seek (index)
Move the current seek pointer to the given block.
You can use negative indices to seek from the end, with identical semantics to those of Python lists.

show (index=None)
Show a single block on screen

show_all ()
Show entire demo on screen, block by block

class IPython.lib.demo.IPythonDemo (src, title='', arg_str='', auto_all=None)
Bases: IPython.lib.demo.Demo
Class for interactive demos with IPython’s input processing applied.
This subclasses Demo, but instead of executing each block by the Python interpreter (via exec), it actually calls IPython on it, so that any input filters which may be in place are applied to the input block.
If you have an interactive environment which exposes special input processing, you can use this class instead to write demo scripts which operate exactly as if you had typed them interactively. The default Demo class requires the input to be valid, pure Python code.

run_cell (source)
Execute a string with one or more lines of code

class IPython.lib.demo.LineDemo (src, title='', arg_str='', auto_all=None)
Bases: IPython.lib.demo.Demo
Demo where each line is executed as a separate block.
The input script should be valid Python code.
This class doesn’t require any markup at all, and it’s meant for simple scripts (with no nesting or any kind of indentation) which consist of multiple lines of input to be executed, one at a time, as if they had been typed in the interactive prompt.
Note: the input can not have any indentation, which means that only single-lines of input are accepted, not even function definitions are valid.

reload ()
Reload source from disk and initialize state.

class IPython.lib.demo.IPythonLineDemo (src, title='', arg_str='', auto_all=None)
Bases: IPython.lib.demo.IPythonDemo, IPython.lib.demo.LineDemo
Variant of the LineDemo class whose input is processed by IPython.

class IPython.lib.demo.ClearMixin
Bases: object
Use this mixin to make Demo classes with less visual clutter.
Demos using this mixin will clear the screen before every block and use blank marquees.
Note that in order for the methods defined here to actually override those of the classes it’s mixed with, it must go /first/ in the inheritance tree. For example:

```python
class ClearIPDemo(ClearMixin, IPythonDemo): pass
```

will provide an IPythonDemo class with the mixin’s features.

```python
marquee (txt='', width=78, mark='*')
```
Blank marquee that returns ‘ ‘ no matter what the input.

```python
pre_cmd ()
```
Method called before executing each block.

This one simply clears the screen.

```python
class IPython.lib.demo.ClearDemo(src, title='', arg_str='', auto_all=None)
```

```python
class IPython.lib.demo.ClearIPDemo(src, title='', arg_str='', auto_all=None)
```

8.44.5 1 Function

```python
IPython.lib.demo.re_mark (mark)
```

8.45 Module: lib.editorhooks

‘editor’ hooks for common editors that work well with ipython

They should honor the line number argument, at least.

Contributions are very welcome.

8.45.1 11 Functions

```python
IPython.lib.editorhooks.install_editor (template, wait=False)
```
Installs the editor that is called by IPython for the %edit magic.

This overrides the default editor, which is generally set by your EDITOR environment variable or is notepad (windows) or vi (linux). By supplying a template string run_template, you can control how the editor is invoked by IPython – (e.g. the format in which it accepts command line options).

**Parameters**

- **template** *(basestring)* – run_template acts as a template for how your editor is invoked by the shell. It should contain ‘{filename}’, which will be replaced on invocation with the file name, and ‘{line}’, $line by line number (or 0) to invoke the file with.

- **wait** *(bool)* – If wait is true, wait until the user presses enter before returning, to facilitate non-blocking editors that exit immediately after the call.
Support for creating GUI apps and starting event loops.

IPython’s GUI integration allows interactive plotting and GUI usage in IPython session. IPython has two different types of GUI integration:

1. The terminal based IPython supports GUI event loops through Python’s PyOS_InputHook. PyOS_InputHook is a hook that Python calls periodically whenever raw_input is waiting for a user to type code. We implement GUI support in the terminal by setting PyOS_InputHook to a function that iterates the event loop for a short while. It is important to note that in this situation, the real GUI event loop is NOT run in the normal manner, so you can’t use the normal means to detect that it is running.

2. In the two process IPython kernel/frontend, the GUI event loop is run in the kernel. In this case, the event loop is run in the normal manner by calling the function or method of the GUI toolkit that starts the event loop.

In addition to starting the GUI event loops in one of these two ways, IPython will always create an appropriate GUI application object when GUI integration is enabled.

If you want your GUI apps to run in IPython you need to do two things:

1. Test to see if there is already an existing main application object. If there is, you should use it. If there is not an existing application object you should create one.
2. Test to see if the GUI event loop is running. If it is, you should not start it. If the event loop is not running you may start it.

This module contains functions for each toolkit that perform these things in a consistent manner. Because of how PyOS_InputHook runs the event loop you cannot detect if the event loop is running using the traditional calls (such as `wx.GetApp.IsMainLoopRunning()` in wxPython). If PyOS_InputHook is set These methods will return a false negative. That is, they will say the event loop is not running, when is actually is. To work around this limitation we proposed the following informal protocol:

- Whenever someone starts the event loop, they *must* set the `_in_event_loop` attribute of the main application object to `True`. This should be done regardless of how the event loop is actually run.

- Whenever someone stops the event loop, they *must* set the `_in_event_loop` attribute of the main application object to `False`.

- If you want to see if the event loop is running, you *must* use `hasattr` to see if `_in_event_loop` attribute has been set. If it is set, you *must* use its value. If it has not been set, you can query the toolkit in the normal manner.

- If you want GUI support and no one else has created an application or started the event loop you *must* do this. We don’t want projects to attempt to defer these things to someone else if they themselves need it.

The functions below implement this logic for each GUI toolkit. If you need to create custom application subclasses, you will likely have to modify this code for your own purposes. This code can be copied into your own project so you don’t have to depend on IPython.

### 8.46.1 6 Functions

**IPython.lib.guisupport.get_app_wx(*args, **kwargs)**

Create a new wx app or return an exiting one.

**IPython.lib.guisupport.is_event_loop_running_wx(app=None)**

Is the wx event loop running.

**IPython.lib.guisupport.start_event_loop_wx(app=None)**

Start the wx event loop in a consistent manner.

**IPython.lib.guisupport.get_app_qt4(*args, **kwargs)**

Create a new qt4 app or return an existing one.

**IPython.lib.guisupport.is_event_loop_running_qt4(app=None)**

Is the qt4 event loop running.

**IPython.lib.guisupport.start_event_loop_qt4(app=None)**

Start the qt4 event loop in a consistent manner.

### 8.47 Module: lib.inputhook

Inputhook management for GUI event loop integration.
8.47.1 11 Classes

class IPython.lib.inputhook.InputHookManager
    Bases: object

    Manage PyOS_InputHook for different GUI toolkits.
    This class installs various hooks under PyOSInputHook to handle GUI event loop integration.

    __init__()

    clear_app_refs(gui=None)
        Clear IPython’s internal reference to an application instance.
        Whenever we create an app for a user on qt4 or wx, we hold a reference to the app. This is needed
        because in some cases bad things can happen if a user doesn’t hold a reference themselves. This
        method is provided to clear the references we are holding.

        Parameters gui (None or str) – If None, clear all app references. If (‘wx’, ‘qt4’) clear the app for that toolkit. References are not held for gtk or tk as those toolkits don’t have the notion of an app.

    clear_inputhook(app=None)
        Set PyOS_InputHook to NULL and return the previous one.

        Parameters app (optional, ignored) – This parameter is allowed only so that clear_inputhook() can be called with a similar interface as all the enable_* methods. But the actual value of the parameter is ignored. This uniform interface makes it easier to have user-level entry points in the main IPython app like enable_gui().

    current_gui()
        Return a string indicating the currently active GUI or None.

    disable_gui()
        Disable GUI event loop integration.
        If an application was registered, this sets its __in_event_loop attribute to False. It then calls clear_inputhook().

    enable_gui(gui=None, app=None)
        Switch amongst GUI input hooks by name.
        This is a higher level method than set_inputhook() - it uses the GUI name to look up a registered object which enables the input hook for that GUI.

        Parameters

        • gui (optional, string or None) – If None (or ‘none’), clears input hook, otherwise it must be one of the recognized GUI names (see GUI_* constants in module).

        • app (optional, existing application object.) – For toolkits that have the concept of a global app, you can supply an existing one. If not given, the toolkit will be probed for one, and if none is found, a new one will
be created. Note that GTK does not have this concept, and passing an app if `gui="GTK"` will raise an error.

**Returns**

- *The output of the underlying gui switch routine, typically the actual*
- *PyOS_InputHook wrapper object or the GUI toolkit app created, if there was*
- *one.*

`get_pyos_inpthook()`  
Return the current PyOS_InputHook as a ctypes.c_void_p.

`get_pyos_inpthook_as_func()`  
Return the current PyOS_InputHook as a ctypes.PYFUNCYPE.

`register(toolkitname, *aliases)`  
Register a class to provide the event loop for a given GUI.

This is intended to be used as a class decorator. It should be passed the names with which to register this GUI integration. The classes themselves should subclass `InputHookBase`.

```python
@inputhook_manager.register('qt')
class QtInputHook(InputHookBase):
    def enable(self, app=None):
        ...
```

`set_inpthook(callback)`  
Set PyOS_InputHook to callback and return the previous one.

```python
class IPython.lib.inputhook.InputHookBase(manager)
Bases: object
Base class for input hooks for specific toolkits.
Subclasses should define an `enable()` method with one argument, `app`, which will either be an instance of the toolkit’s application class, or None. They may also define a `disable()` method with no arguments.
__init__(manager)

class IPython.lib.inputhook.NullInputHook(manager)
Bases: IPython.lib.inputhook.InputHookBase
A null inpthook that doesn’t need to do anything

class IPython.lib.inputhook.WxInputHook(manager)
Bases: IPython.lib.inputhook.InputHookBase

disable()
    Disable event loop integration with wxPython.
    This restores appnapp on OS X

enable(app=None)
    Enable event loop integration with wxPython.
```


**Parameters** app *(WX Application, optional.)* – Running application to use. If not given, we probe WX for an existing application object, and create a new one if none is found.

**Notes**

This method sets the `PyOS_InputHook` for wxPython, which allows the wxPython to integrate with terminal based applications like IPython.

If `app` is not given we probe for an existing one, and return it if found. If no existing app is found, we create an `wx.App` as follows:

```python
import wx
app = wx.App(redirect=False, clearSigInt=False)
```

**class** `IPython.lib.inputhook.Qt4InputHook` *(manager)*

**Bases:** `IPython.lib.inputhook.InputHookBase`

**disable_qt4** ()

Disable event loop integration with PyQt4.

This restores appnapp on OS X

**enable** *(app=None)*

Enable event loop integration with PyQt4.

**Parameters** app *(Qt Application, optional.)* – Running application to use. If not given, we probe Qt for an existing application object, and create a new one if none is found.

**Notes**

This method sets the `PyOS_InputHook` for PyQt4, which allows the PyQt4 to integrate with terminal based applications like IPython.

If `app` is not given we probe for an existing one, and return it if found. If no existing app is found, we create an `QApplication` as follows:

```python
from PyQt4 import QtCore
app = QtGui.QApplication(sys.argv)
```

**class** `IPython.lib.inputhook.Qt5InputHook` *(manager)*

**Bases:** `IPython.lib.inputhook.Qt4InputHook`

**class** `IPython.lib.inputhook.GtkInputHook` *(manager)*

**Bases:** `IPython.lib.inputhook.InputHookBase`

**enable** *(app=None)*

Enable event loop integration with PyGTK.
**Parameters**  
**app** *(ignored)* – Ignored, it’s only a placeholder to keep the call signature of all gui activation methods consistent, which simplifies the logic of supporting magics.

**Notes**

This methods sets the PyOS_InputHook for PyGTK, which allows the PyGTK to integrate with terminal based applications like IPython.

```python
class IPython.lib.inputhook.TkInputHook(manager)
Bases: IPython.lib.inputhook.InputHookBase

enable(app=None)
Enable event loop integration with Tk.

Parameters **app** *(toplevel Tkinter.Tk widget, optional.)* – Running toplevel widget to use. If not given, we probe Tk for an existing one, and create a new one if none is found.

**Notes**

If you have already created a Tkinter.Tk object, the only thing done by this method is to register with the InputHookManager, since creating that object automatically sets PyOS_InputHook.

```python
class IPython.lib.inputhook.GlutInputHook(manager)
Bases: IPython.lib.inputhook.InputHookBase

disable()
Disable event loop integration with glut.

This sets PyOS_InputHook to NULL and set the display function to a dummy one and set the timer to a dummy timer that will be triggered very far in the future.

enable(app=None)
Enable event loop integration with GLUT.

Parameters **app** *(ignored)* – Ignored, it’s only a placeholder to keep the call signature of all gui activation methods consistent, which simplifies the logic of supporting magics.

**Notes**

This methods sets the PyOS_InputHook for GLUT, which allows the GLUT to integrate with terminal based applications like IPython. Due to GLUT limitations, it is currently not possible to start the event loop without first creating a window. You should thus not create another window but use instead the created one. See ‘gui-glut.py’ in the docs/examples/lib directory.

The default screen mode is set to: glut.GLUT_DOUBLE | glut.GLUT_RGBA | glut.GLUT_DEPTH
class IPython.lib.inputhook.PygletInputHook (manager)
Bases: IPython.lib.inputhook.InputHookBase

enable (app=None)
Enable event loop integration with pyglet.

Parameters app (ignored) – Ignored, it’s only a placeholder to keep the call signature of all gui activation methods consistent, which simplifies the logic of supporting magics.

Notes
This methods sets the PyOS_InputHook for pyglet, which allows pyglet to integrate with terminal based applications like IPython.

class IPython.lib.inputhook.Gtk3InputHook (manager)
Bases: IPython.lib.inputhook.InputHookBase

enable (app=None)
Enable event loop integration with Gtk3 (gir bindings).

Parameters app (ignored) – Ignored, it’s only a placeholder to keep the call signature of all gui activation methods consistent, which simplifies the logic of supporting magics.

Notes
This methods sets the PyOS_InputHook for Gtk3, which allows the Gtk3 to integrate with terminal based applications like IPython.

8.48 Module: lib.latextools

Tools for handling LaTeX.

8.48.1 1 Class

class IPython.lib.latextools.LaTeXTool (**kwargs)
Bases: traitlets.config.configurable.SingletonConfigurable

An object to store configuration of the LaTeX tool.

8.48.2 6 Functions

IPython.lib.latextools.latex_to_png (s, encode=False, backend=None, wrap=False)

Render a LaTeX string to PNG.
Parameters

- **s** *(text)* – The raw string containing valid inline LaTeX.
- **encode** *(bool, optional)* – Should the PNG data be base64 encoded to make it JSON’able.
- **backend** *(matplotlib, dvipng)* – Backend for producing PNG data.
- **wrap** *(bool)* – If true, automatically wrap s as a LaTeX equation.
- **is returned when the backend cannot be used.** *(None)* –


```
IPython.lib.latextools.latex_to_png_mpl(s, wrap)
IPython.lib.latextools.latex_to_png_dvipng(s, wrap)
```

Invoke kpswhich command with an argument filename.

```
IPython.lib.latextools.genelatex(body, wrap)
```
Generate LaTeX document for dvipng backend.

```
IPython.lib.latextools.latex_to_html(s, alt='image')
```
Render LaTeX to HTML with embedded PNG data using data URIs.

**Parameters**

- **s** *(str)* – The raw string containing valid inline LaTeX.
- **alt** *(str)* – The alt text to use for the HTML.

## 8.49 Module: `lib.lexers`

Defines a variety of Pygments lexers for highlighting IPython code.

This includes:

- **IPythonLexer, IPython3Lexer** Lexers for pure IPython (python + magic/shell commands)
- **IPythonPartialTracebackLexer, IPythonTracebackLexer** Supports 2.x and 3.x via keyword python3. The partial traceback lexer reads everything but the Python code appearing in a traceback. The full lexer combines the partial lexer with an IPython lexer.
- **IPythonConsoleLexer** A lexer for IPython console sessions, with support for tracebacks.
- **IPyLexer** A friendly lexer which examines the first line of text and from it, decides whether to use an IPython lexer or an IPython console lexer. This is probably the only lexer that needs to be explicitly added to Pygments.

### 8.49.1 4 Classes

**class** IPython.lib.lexers.IPythonPartialTracebackLexer(**options**)

Bases: pygments.lexer.RegexLexer
Partial lexer for IPython tracebacks.
Handles all the non-python output. This works for both Python 2.x and 3.x.

```python
class IPython.lib.lexers.IPythonTracebackLexer(**options)
    Bases: pygments.lexer.DelegatingLexer

IPython traceback lexer.
```

For doctests, the tracebacks can be snipped as much as desired with the exception to the lines that designate a traceback. For non-syntax error tracebacks, this is the line of hyphens. For syntax error tracebacks, this is the line which lists the File and line number.

```python
__init__(**options)
```

```python
class IPython.lib.lexers.IPythonConsoleLexer(**options)
    Bases: pygments.lexer.Lexer

An IPython console lexer for IPython code-blocks and doctests, such as:
```

```python
.. code-block:: ipythonconsole

    In [1]: a = 'foo'

    In [2]: a
    Out[2]: 'foo'

    In [3]: print a
       foo

    In [4]: 1 / 0
```

Support is also provided for IPython exceptions:

```python
.. code-block:: ipythonconsole

    In [1]: raise Exception

```

```python
---

Exception

Traceback (most recent call last)
<ipython-input-1-fca2ab0ca76b> in <module>()
----> 1 raise Exception

Exception

```

```python
__init__(**options)
```

Initialize the IPython console lexer.

**Parameters**

- **python3 (bool)** – If True, then the console inputs are parsed using a Python 3 lexer. Otherwise, they are parsed using a Python 2 lexer.

- **in1_regex (RegexObject)** – The compiled regular expression used to detect the start of inputs. Although the IPython configuration setting may have
a trailing whitespace, do not include it in the regex. If None, then the default input prompt is assumed.

- **in2_regex** (*RegexObject*) – The compiled regular expression used to detect the continuation of inputs. Although the IPython configuration setting may have a trailing whitespace, do not include it in the regex. If None, then the default input prompt is assumed.

- **out_regex** (*RegexObject*) – The compiled regular expression used to detect outputs. If None, then the default output prompt is assumed.

`buffered_tokens()`

Generator of unprocessed tokens after doing insertions and before changing to a new state.

`get_mci(line)`

Parses the line and returns a 3-tuple: (mode, code, insertion).

- **mode** is the next mode (or state) of the lexer, and is always equal to ‘input’, ‘output’, or ‘tb’.

- **code** is a portion of the line that should be added to the buffer corresponding to the next mode and eventually lexed by another lexer. For example, code could be Python code if mode were ‘input’.

- **insertion** is a 3-tuple (index, token, text) representing an unprocessed “token” that will be inserted into the stream of tokens that are created from the buffer once we change modes. This is usually the input or output prompt.

In general, the next mode depends on current mode and on the contents of `line`.

`ipythb_start = re.compile('(^\^C)?(-+\n)|^( File)(.*)(, line )(\d+\n)')`

The regex to determine when a traceback starts.

**class** IPython.lib.lexers.IPyLexer (**options**)  
**Bases:** pygments.lexer.Lexer

Primary lexer for all IPython-like code.

This is a simple helper lexer. If the first line of the text begins with “In [[0-9]+]:”, then the entire text is parsed with an IPython console lexer. If not, then the entire text is parsed with an IPython lexer.

The goal is to reduce the number of lexers that are registered with Pygments.

`__init__(**options)`

### 8.49.2 1 Function

**IPython.lib.lexers.build_ipy_lex(ers)** (*python3*)  
Builds IPython lexers depending on the value of `python3`.

The lexer inherits from an appropriate Python lexer and then adds information about IPython specific keywords (i.e. magic commands, shell commands, etc.)

**Parameters** `python3` (bool) – If True, then build an IPython lexer from a Python 3 lexer.
8.50 Module: lib.pretty

Python advanced pretty printer. This pretty printer is intended to replace the old `pprint` python module which does not allow developers to provide their own pretty print callbacks.

This module is based on ruby’s `prettyprint.rb` library by Tanaka Akira.

8.50.1 Example Usage

To directly print the representation of an object use `pprint`:

```python
from pretty import pprint
pprint(complex_object)
```

To get a string of the output use `pretty`:

```python
from pretty import pretty
string = pretty(complex_object)
```

8.50.2 Extending

The pretty library allows developers to add pretty printing rules for their own objects. This process is straightforward. All you have to do is to add a `_repr_pretty_` method to your object and call the methods on the pretty printer passed:

```python
class MyObject(object):
    def _repr_pretty_(self, p, cycle):
        ...  
```

Here is an example implementation of a `_repr_pretty_` method for a list subclass:

```python
class MyList(list):
    def _repr_pretty_(self, p, cycle):
        if cycle:
            p.text('MyList(...)')
        else:
            with p.group(8, 'MyList([', ']'):
                for idx, item in enumerate(self):
                    if idx:
                        p.text(',')
p.breakable()
p.pretty(item)
```

The `cycle` parameter is `True` if pretty detected a cycle. You have to react to that or the result is an infinite loop. `p.text()` just adds non breaking text to the output, `p.breakable()` either adds a whitespace or breaks here. If you pass it an argument it’s used instead of the default space. `p.pretty` prettyprints another object using the pretty print method.
The first parameter to the `group` function specifies the extra indentation of the next line. In this example the next item will either be on the same line (if the items are short enough) or aligned with the right edge of the opening bracket of `MyList`.

If you just want to indent something you can use the `group` function without open / close parameters. You can also use this code:

```python
with p.indent(2):
    ...
```

Inheritance diagram:

![Inheritance Diagram](image)

---

**copyright** 2007 by Armin Ronacher. Portions (c) 2009 by Robert Kern.

**license** BSD License.

### 8.50.3 8 Classes

**IPython.lib.pretty.CUnicodeIO**

alias of `StringIO`

**class IPython.lib.pretty.PrettyPrinter**

| (output, max_width=79, newline='n', max_seq_length=1000)
| Bases: IPython.lib.pretty._PrettyPrinterBase

Baseclass for the `RepresentationPrinter` prettyprinter that is used to generate pretty reprs of objects. Contrary to the `RepresentationPrinter` this printer knows nothing about the default pprinters or the `_repr_pretty_` callback method.

```python
__init__(output, max_width=79, newline='n', max_seq_length=1000)
```

**begin_group** *(indent=0, open='')*

Begin a group. If you want support for python < 2.5 which doesn’t has the with statement this is the preferred way:

```python
p.begin_group(1, '{') ... p.end_group(1, '}')
```
The python 2.5 expression would be this:

```python
with p.group(1, '{', '}):
    ...
```

The first parameter specifies the indentation for the next line (usually the width of the opening
text), the second the opening text. All parameters are optional.

**break_()**

Explicitly insert a newline into the output, maintaining correct indentation.

**breakable (sep=' ')**

Add a breakable separator to the output. This does not mean that it will automatically break
here. If no breaking on this position takes place the `sep` is inserted which default to one space.

**end_group (dedent=0, close='')**

End a group. See `begin_group` for more details.

**flush()**

Flush data that is left in the buffer.

**text (obj)**

Add literal text to the output.

**class IPython.lib.pretty.RepresentationPrinter (output, verbose=False, max_width=79, newline='n', singleton_pprinters=None, type_pprinters=None, deferred_pprinters=None, max_seq_length=1000)**

Bases: `IPython.lib.pretty.PrettyPrinter`

Special pretty printer that has a `pretty` method that calls the pretty printer for a python object.

This class stores processing data on `self` so you must never use this class in a threaded environment. Always lock it or reinstanciate it.

Instances also have a verbose flag callbacks can access to control their output. For example the default instance repr prints all attributes and methods that are not prefixed by an underscore if the printer is in verbose mode.

```python
__init__ (output, verbose=False, max_width=79, newline='n', singleton_pprinters=None, type_pprinters=None, deferred_pprinters=None, max_seq_length=1000)
```

**pretty (obj)**

Pretty print the given object.

**class IPython.lib.pretty.Printable**

Bases: `object`

**class IPython.lib.pretty.Text**

Bases: `IPython.lib.pretty.Printable`

```python
__init__()
```

**class IPython.lib.pretty.Breakable (seq, width, pretty)**

Bases: `IPython.lib.pretty.Printable`
__init__ (seq, width, pretty)

class IPython.lib.pretty.Group (depth)
    Bases: IPython.lib.pretty.Printable
    __init__ (depth)

class IPython.lib.pretty.GroupQueue (*groups)
    Bases: object
    __init__ (*groups)

8.50.4 4 Functions

IPython.lib.pretty.pretty (obj, verbose=False, max_width=79, newline='\n', max_seq_length=1000)
    Pretty print the object’s representation.

IPython.lib.pretty.pprint (obj, verbose=False, max_width=79, newline='\n', max_seq_length=1000)
    Like pretty but print to stdout.

IPython.lib.pretty.for_type (typ, func)
    Add a pretty printer for a given type.

IPython.lib.pretty.for_type_by_name (type_module, type_name, func)
    Add a pretty printer for a type specified by the module and name of a type rather than the type object itself.

8.51 Module: lib.security

Password generation for the IPython notebook.

8.51.1 2 Functions

IPython.lib.securitypasswd (passphrase=None, algorithm='sha1')
    Generate hashed password and salt for use in notebook configuration.

    In the notebook configuration, set c.NotebookApp.password to the generated string.

    Parameters

        • passphrase (str) – Password to hash. If unspecified, the user is asked to input and verify a password.

        • algorithm (str) – Hashing algorithm to use (e.g. ‘sha1’ or any argument supported by hashlib.new()).

    Returns hashed_password – Hashed password, in the format ‘hash_algorithm:salt:passphrase_hash’.

    Return type str
Examples

```python
>>> passwd('mypassword')
'sha1:7cf3:b7d6da294ea9592a9480c8f52e63cd42c9f9dd12'
```

IPython.lib.security passwd_check (hashed_passphrase, passphrase)

Verify that a given passphrase matches its hashed version.

Parameters

- **hashed_passphrase** *(str)* – Hashed password, in the format returned by `passwd`.
- **passphrase** *(str)* – Passphrase to validate.

Returns `valid` – True if the passphrase matches the hash.

Return type `bool`

Examples

```python
>>> from IPython.lib.security import passwd_check
>>> passwd_check('sha1:0e112c3ddfce:a68df677475c2b47b6e86d0467ec97ac5f4b83a', ...
...    'mypassword')
True

>>> passwd_check('sha1:0e112c3ddfce:a68df677475c2b47b6e86d0467ec97ac5f4b83a', ...
...    'anotherpassword')
False
```

8.52 Module: paths

Find files and directories which IPython uses.

8.52.1 5 Functions

IPython.paths.get_ipython_dir()

Get the IPython directory for this platform and user.

This uses the logic in `get_home_dir` to find the home directory and then adds `.ipython` to the end of the path.

IPython.paths.get_ipython_cache_dir()

Get the cache directory it is created if it does not exist.

IPython.paths.get_ipython_package_dir()

Get the base directory where IPython itself is installed.
IPython.paths.get_ipython_module_path(module_str)
Find the path to an IPython module in this version of IPython.

This will always find the version of the module that is in this importable IPython package. This will always return the path to the .py version of the module.

IPython.paths.locate_profile(profile='default')
Find the path to the folder associated with a given profile.
I.e. find $IPYTHONDIR/profile_whatever.

8.53 Module: terminal.embed

An embedded IPython shell.

8.53.1 3 Classes

class IPython.terminal.embed.KillEmbeded
Bases: Exception

class IPython.terminal.embed.EmbeddedMagics(shell=None, **kwargs)
Bases: IPython.core.magic.Magics

exit_raise(parameter_s='')
%%exit_raise Make the current embedded kernel exit and raise and exception.

This function sets an internal flag so that an embedded IPython will raise a IPython.terminal.embed.KillEmbeded Exception on exit, and then exit the current I. This is useful to permanently exit a loop that create IPython embed instance.

kill_embedded(parameter_s='')
%%kill_embedded : deactivate for good the current embedded IPython.

This function (after asking for confirmation) sets an internal flag so that an embedded IPython will never activate again. This is useful to permanently disable a shell that is being called inside a loop: once you’ve figured out what you needed from it, you may then kill it and the program will then continue to run without the interactive shell interfering again.

class IPython.terminal.embed.InteractiveShellEmbed(**kw)
Bases: IPython.terminal.interactiveshell.TerminalInteractiveShell

__init__(**kw)

__call__(header='', local_ns=None, module=None, dummy=None, stack_depth=1, global_ns=None, compile_flags=None)
Activate the interactive interpreter.

__call__(self,header='',local_ns=None,module=None,dummy=None) -> Start the interpreter shell with the given local and global namespaces, and optionally print a header string at startup.

The shell can be globally activated/deactivated using the dummy_mode attribute. This allows you to turn off a shell used for debugging globally.
However, each time you call the shell you can override the current state of dummy_mode with the optional keyword parameter ‘dummy’. For example, if you set dummy mode on with IPShell.dummy_mode = True, you can still have a specific call work by making it as IP-Shell(dummy=False).

**mainloop** (local_ns=None, module=None, stack_depth=0, display_banner=None, global_ns=None, compile_flags=None)

Embeds IPython into a running python program.

**Parameters**

- **module** *(local_ns,)* – Working local namespace (a dict) and module (a module or similar object). If given as None, they are automatically taken from the scope where the shell was called, so that program variables become visible.

- **stack_depth** *(int)* – How many levels in the stack to go to looking for namespaces (when local_ns or module is None). This allows an intermediate caller to make sure that this function gets the namespace from the intended level in the stack. By default (0) it will get its locals and globals from the immediate caller.

- **compile_flags** – A bit field identifying the __future__ features that are enabled, as passed to the builtin `compile()` function. If given as None, they are automatically taken from the scope where the shell was called.

### 8.53.2 1 Function

IPython.terminal.embed.embed(**kwargs)

Call this to embed IPython at the current point in your program.

The first invocation of this will create an `InteractiveShellEmbed` instance and then call it. Consecutive calls just call the already created instance.

If you don’t want the kernel to initialize the namespace from the scope of the surrounding function, and/or you want to load full IPython configuration, you probably want `IPython.start_ipython()` instead.

Here is a simple example:

```python
from IPython import embed
a = 10
b = 20
embed(header='First time')
c = 30
d = 40
embed()
```

Full customization can be done by passing a `Config` in as the config argument.
8.54 Module: `terminal.interactiveshell`

Subclass of InteractiveShell for terminal based frontends.

### 8.54.1 3 Classes

**class** `IPython.terminal.interactiveshell.ReadlineNoRecord(shell)`

Bases: `object`

Context manager to execute some code, then reload readline history so that interactive input to the code doesn’t appear when pressing up.

```python
def __init__(self, shell)
```

**get_readline_tail**(n=10)

Get the last n items in readline history.

**class** `IPython.terminal.interactiveshell.TerminalMagics(shell)`

Bases: `IPython.core.magic.Magics`

```python
def __init__(self, shell)
```

**autoindent**(parameter_s='')

Toggle autoindent on/off (if available).

**cpaste**(parameter_s='')

Paste & execute a pre-formatted code block from clipboard.

You must terminate the block with ‘–’ (two minus-signs) or Ctrl-D alone on the line. You can also provide your own sentinel with ‘%paste -s %%%’ (‘%%%’ is the new sentinel for this operation).

The block is dedented prior to execution to enable execution of method definitions. ‘>’ and ‘+’ characters at the beginning of a line are ignored, to allow pasting directly from e-mails, diff files and doctests (the ‘...’ continuation prompt is also stripped). The executed block is also assigned to variable named ‘pasted_block’ for later editing with ‘%edit pasted_block’.

You can also pass a variable name as an argument, e.g. ‘%cpaste foo’. This assigns the pasted block to variable ‘foo’ as string, without dedenting or executing it (preceding >>> and + is still stripped)

‘%cpaste -r’ re-executes the block previously entered by cpaste. ‘%cpaste -q’ suppresses any additional output messages.

Do not be alarmed by garbled output on Windows (it’s a readline bug). Just press enter and type – (and press enter again) and the block will be what was just pasted.

IPython statements (magics, shell escapes) are not supported (yet).

**See also:**

```
paste() automatically pull code from clipboard.
```
Examples

In [8]: %cpaste
Pasting code; enter '--' alone on the line to stop.
>>> a = ["world!", "Hello"]
>>> print " ".join(sorted(a))
--
Hello world!

```
paste (parameter_s='')
Paste & execute a pre-formatted code block from clipboard.

The text is pulled directly from the clipboard without user intervention and printed back on the
screen before execution (unless the -q flag is given to force quiet mode).

The block is dedented prior to execution to enable execution of method definitions. ‘>’ and ‘+’
characters at the beginning of a line are ignored, to allow pasting directly from e-mails, diff files
and doctests (the ‘...’ continuation prompt is also stripped). The executed block is also assigned
to variable named ‘pasted_block’ for later editing with ‘%edit pasted_block’.

You can also pass a variable name as an argument, e.g. ‘%paste foo’. This assigns the pasted
block to variable ‘foo’ as string, without executing it (preceding >>> and + is still stripped).

Options:

-r: re-executes the block previously entered by cpaste.
-q: quiet mode: do not echo the pasted text back to the terminal.

IPython statements (magics, shell escapes) are not supported (yet).

See also:

cpaste() manually paste code into terminal until you mark its end.

erun_pasted(name='pasted_block')
Rerun a previously pasted command.

store_or_execute (block, name)
Execute a block, or store it in a variable, per the user’s request.

class IPython.terminal.interactiveshell.TerminalInteractiveShell (ipython_dir=None,
profile_dir=None,
user_module=None,
user_ns=None,
custom_exceptions=(((), None), **kwargs)

Bases: IPython.core.interactiveshell.InteractiveShell

ask_exit ()
Ask the shell to exit. Can be overiden and used as a callback.
edit_syntax_error()
The bottom half of the syntax error handler called in the main loop.

Loop until syntax error is fixed or user cancels.

static enable_gui (gui=None, app=None)
Switch amongst GUI input hooks by name.

exit()
Handle interactive exit.

This method calls the ask_exit callback.

init_readline()
Command history completion/saving/reloading.

interact (display_banner=None)
Closely emulate the interactive Python console.

mainloop (display_banner=None)
Start the mainloop.

If an optional banner argument is given, it will override the internally created default banner.

pre_readline()
readline hook to be used at the start of each line.

It handles auto-indent and text from set_next_input.

raw_input (prompt=''
Write a prompt and read a line.

The returned line does not include the trailing newline. When the user enters the EOF key sequence, EOFError is raised.

Parameters prompt (str, optional) – A string to be printed to prompt the user.

set_readline_completer()
Reset readline’s completer to be our own.

system (cmd)
Call the given cmd in a subprocess using os.system on Windows or subprocess.call using the system shell on other platforms.

Parameters cmd (str) – Command to execute.

8.54.2 2 Functions

IPython.terminal.interactiveshell.get_default_editor()

IPython.terminal.interactiveshell.get_pasted_lines (sentinel, l_input=<function input>, quiet=False)

Yield pasted lines until the user enters the given sentinel value.
8.55 Module: `terminal.ipapp`

The `Application` object for the command line `ipython` program.

8.55.1 3 Classes

class `IPython.terminal.ipapp.IPAppCrashHandler(app)`
   Bases: `IPython.core.crashhandler.CrashHandler`

   sys.excepthook for IPython itself, leaves a detailed report on disk.

   `__init__(app)`
   `make_report(traceback)`
      Return a string containing a crash report.

class `IPython.terminal.ipapp.LocateIPythonApp(**kwargs)`
   Bases: `IPython.core.application.BaseIPythonApplication`

class `IPython.terminal.ipapp.TerminalIPythonApp(**kwargs)`
   Bases: `IPython.core.application.BaseIPythonApplication`, `IPython.core.shellapp.InteractiveShellApp`

   `crash_handler_class`
      alias of `IPAppCrashHandler`

   `init_banner()`
      optionally display the banner

   `init_shell()`
      initialize the InteractiveShell instance

   `initialize(argv=none)`
      Do actions after construct, but before starting the app.

   `parse_command_line(argv=none)`
      override to allow old `-pylab` flag with deprecation warning

8.55.2 1 Function

`IPython.terminal.ipapp.load_default_config(ipython_dir=None)`
   Load the default config file from the default ipython_dir.

   This is useful for embedded shells.

8.56 Module: `testing`

Testing support (tools to test IPython itself).
8.56.1 1 Function

IPython.testing.test(**kwargs)
Run the entire IPython test suite.
Any of the options for run_iptestall() may be passed as keyword arguments.
For example:

```python
IPython.test(testgroups=['lib', 'config', 'utils'], fast=2)
```

will run those three sections of the test suite, using two processes.

8.57 Module: testing.decorators

Decorators for labeling test objects.

Decorators that merely return a modified version of the original function object are straightforward. Decorators that return a new function object need to use nose.tools.make_decorator(original_function)(decorator) in returning the decorator, in order to preserve metadata such as function name, setup and teardown functions and so on - see nose.tools for more information.

This module provides a set of useful decorators meant to be ready to use in your own tests. See the bottom of the file for the ready-made ones, and if you find yourself writing a new one that may be of generic use, add it here.

Included decorators:

Lightweight testing that remains unittest-compatible.

• An @as_unittest decorator can be used to tag any normal parameter-less function as a unittest TestCase. Then, both nose and normal unittest will recognize it as such. This will make it easier to migrate away from Nose if we ever need/want to while maintaining very lightweight tests.

NOTE: This file contains IPython-specific decorators. Using the machinery in IPython.external.decorators, we import either numpy.testing.decorators if numpy is available, OR use equivalent code in IPython.external._decorators, which we’ve copied verbatim from numpy.

8.57.1 11 Functions

IPython.testing.decorators.as_unittest(func)
Decorator to make a simple function into a normal test via unittest.

IPython.testing.decorators.apply_wrapper(wrapper, func)
Apply a wrapper to a function for decoration.

This mixes Michele Simionato’s decorator tool with nose’s make_decorator, to apply a wrapper in a decorator so that all nose attributes, as well as function signature and other properties, survive the decoration cleanly. This will ensure that wrapped functions can still be well introspected via IPython, for example.
IPython Documentation, Release 4.2.1

IPython.testing.decorators.make_label_dec(label, ds=None)
Factory function to create a decorator that applies one or more labels.

Parameters

- **label** (string or sequence) –
  or more labels that will be applied by the decorator to the functions

it decorates. Labels are attributes of the decorated function with their value set to True.

- **ds** : string An optional docstring for the resulting decorator. If not given, a default docstring is auto-generated.

Returns

**Return type** A decorator.

Examples

A simple labeling decorator:

```python
>>> slow = make_label_dec('slow')
>>> slow.__doc__
"Labels a test as 'slow'."
```

And one that uses multiple labels and a custom docstring:

```python
>>> rare = make_label_dec(['slow', 'hard'],
...                       "Mix labels 'slow' and 'hard' for rare tests.")
>>> rare.__doc__
"Mix labels 'slow' and 'hard' for rare tests."
```

Now, let’s test using this one: `>>> @rare ... def f(): pass ... >>> >>> f.slow True >>> f.hard True`

IPython.testing.decorators.skipif(skip_condition, msg=None)
Make function raise SkipTest exception if skip_condition is true

Parameters

- **skip_condition** (bool or callable) – Flag to determine whether to skip test. If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed.

- **msg** (string) – Message to give on raising a SkipTest exception.

Returns **decorator** – Decorator, which, when applied to a function, causes SkipTest to be raised when the skip_condition was True, and the function to be called normally otherwise.

**Return type** function
Notes

You will see from the code that we had to further decorate the decorator with the nose.tools.make_decorator function in order to transmit function name, and various other metadata.

```python
IPython.testing.decorators.skip(msg=None)

Decorator factory - mark a test function for skipping from test suite.

Parameters msg (string) – Optional message to be added.

Returns decorator – Decorator, which, when applied to a function, causes SkipTest to be raised, with the optional message added.

Return type function
```

```python
IPython.testing.decorators.onlyif(condition, msg)

The reverse from skipif, see skipif for details.
```

```python
IPython.testing.decorators.module_not_available(module)

Can module be imported? Returns true if module does NOT import.

This is used to make a decorator to skip tests that require module to be available, but delay the ‘import numpy’ to test execution time.
```

```python
IPython.testing.decorators.decorated_dummy(dec, name)

Return a dummy function decorated with dec, with the given name.
```

Examples

```python
import IPython.testing.decorators as dec
setup = dec.decorated_dummy(dec.skip_if_no_x11, __name__)
```

```python
IPython.testing.decorators.skip_file_no_x11(name)
```

```python
IPython.testing.decorators.onlyif_cmds_exist(*commands)

Decorator to skip test when at least one of commands is not found.
```

```python
IPython.testing.decorators.onlyif_any_cmd_exists(*commands)

Decorator to skip test unless at least one of commands is found.
```

8.58 Module: testing.globalipapp

Global IPython app to support test running.

We must start our own ipython object and heavily muck with it so that all the modifications IPython makes to system behavior don’t send the doctest machinery into a fit. This code should be considered a gross hack, but it gets the job done.
8.58.1 1 Class

class IPython.testing.globalipapp.StreamProxy(name)
   Bases: IPython.utils.io.IOStream

   Proxy for sys.stdout/err. This will request the stream at call time allowing for nose’s Capture plugin’s redirection of sys.stdout/err.

   Parameters name (str) – The name of the stream. This will be requested anew at every call

   __init__ (name)

8.58.2 3 Functions

IPython.testing.globalipapp.get_ipython()

IPython.testing.globalipapp.xsys(self, cmd)
   Replace the default system call with a capturing one for doctest.

IPython.testing.globalipapp.start_ipython()
   Start a global IPython shell, which we need for IPython-specific syntax.

8.59 Module: testing.iptest

8.59.1 4 Classes

8.59.2 5 Functions

8.60 Module: testing.iptestcontroller

8.60.1 3 Classes

8.60.2 7 Functions

8.61 Module: testing.ipunittest

Experimental code for cleaner support of IPython syntax with unittest.

In IPython up until 0.10, we’ve used very hacked up nose machinery for running tests with IPython special syntax, and this has proved to be extremely slow. This module provides decorators to try a different approach, stemming from a conversation Brian and I (FP) had about this problem Sept/09.

The goal is to be able to easily write simple functions that can be seen by unittest as tests, and ultimately for these to support doctests with full IPython syntax. Nose already offers this based on naming conventions and our hackish plugins, but we are seeking to move away from nose dependencies if possible.

This module follows a different approach, based on decorators.
• A decorator called @ipdoctest can mark any function as having a docstring that should be viewed as a doctest, but after syntax conversion.

### 8.61.1 Authors

• Fernando Perez <Fernando.Perez@berkeley.edu>

### 8.61.2 2 Classes

```python
class IPython.testing.ipunittest.IPython2PythonConverter
    Bases: object
    Convert IPython ‘syntax’ to valid Python.
    Eventually this code may grow to be the full IPython syntax conversion implementation, but for now it only does prompt conversion.
    __init__()
    __call__(ds)
    Convert IPython prompts to python ones in a string.
```

```python
class IPython.testing.ipunittest.Doc2UnitTester(verbos=False)
    Bases: object
    Class whose instances act as a decorator for docstring testing.
    In practice we’re only likely to need one instance ever, made below (though no attempt is made at turning it into a singleton, there is no need for that).
    __init__(verbos=False)
    New decorator.
    Parameters verbose(boolean, optional (False))– Passed to the doctest finder and runner to control verbosity.
    __call__(func)
    Use as a decorator: doctest a function’s docstring as a unittest.
    This version runs normal doctests, but the idea is to make it later run ipython syntax instead.
```

### 8.61.3 2 Functions

```python
IPython.testing.ipunittest.count_failures(runner)
    Count number of failures in a doctest runner.
    Code modeled after the summarize() method in doctest.
```

```python
IPython.testing.ipunittest.ipdocstring(func)
    Change the function docstring via ip2py.
```
8.62 Module: testing.skipdoctest

Decorators marks that a doctest should be skipped, for both python 2 and 3.
The IPython.testing.decorators module triggers various extra imports, including numpy and sympy if they’re present. Since this decorator is used in core parts of IPython, it’s in a separate module so that running IPython doesn’t trigger those imports.

8.62.1 2 Functions

IPython.testing.skipdoctest.skip_doctest\(f\)

Decorator - mark a function or method for skipping its doctest.

This decorator allows you to mark a function whose docstring you wish to omit from testing, while preserving the docstring for introspection, help, etc.

IPython.testing.skipdoctest.skip_doctest_py3\(f\)

Decorator - skip the doctest under Python 3.

8.63 Module: testing.tools

Generic testing tools.

8.63.1 Authors

- Fernando Perez <Fernando.Perez@berkeley.edu>

8.63.2 4 Classes

class IPython.testing.tools.TempFileMixin

Bases: object

Utility class to create temporary Python/IPython files.

Meant as a mixin class for test cases.

\texttt{mktmp}(src, ext=’.py’)

Make a valid python temp file.

IPython.testing.tools.MyStringIO

alias of StringIO

class IPython.testing.tools.AssertPrints\(s, channel=’stdout’, suppress=True\)

Bases: object

Context manager for testing that code prints certain text.
Examples

```python
>>> with AssertPrints("abc", suppress=False):
...     print("abcd")
...     print("def")
...  
abcd
def
```

```python
__init__ (s, channel='stdout', suppress=True)
```

class IPython.testing.tools.AssertNotPrints (s, channel='stdout', suppress=True)

Bases: IPython.testing.tools.AssertPrints

Context manager for checking that certain output isn’t produced.

Counterpart of AssertPrints

8.63.3 12 Functions

IPython.testing.tools.full_path (startPath, files)

Make full paths for all the listed files, based on startPath.

Only the base part of startPath is kept, since this routine is typically used with a script’s __file__ variable as startPath. The base of startPath is then prepended to all the listed files, forming the output list.

Parameters

- **startPath**(string) – Initial path to use as the base for the results. This path is split using os.path.split() and only its first component is kept.

- **files**(string or list) – One or more files.

Examples

```python
>>> full_path('/foo/bar.py',['a.txt','b.txt'])
[’/foo/a.txt’, ’/foo/b.txt’]
```

```python
>>> full_path('/foo',['a.txt','b.txt'])
[’/a.txt’, ’/b.txt’]
```

If a single file is given, the output is still a list:

```python
>>> full_path('/foo','a.txt')
[’/a.txt’]
```

IPython.testing.tools.parse_test_output (txt)

Parse the output of a test run and return errors, failures.

Parameters **txt** (str) – Text output of a test run, assumed to contain a line of one of the following forms:
Returns  number of errors and failures.

Return type  nerr, nfail

IPython.testing.tools.default_argv()
Return a valid default argv for creating testing instances of ipython

IPython.testing.tools.default_config()
Return a config object with good defaults for testing.

IPython.testing.tools.get_ipython_cmd(as_string=False)
Return appropriate IPython command line name. By default, this will return a list that can be used with subprocess.Popen, for example, but passing as_string=True allows for returning the IPython command as a string.

Parameters  as_string (bool) – Flag to allow to return the command as a string.

IPython.testing.tools.ipexec(fname, options=None, commands=())
Utility to call ‘ipython filename’.
Starts IPython with a minimal and safe configuration to make startup as fast as possible.
Note that this starts IPython in a subprocess!

Parameters

•  **fname** (str) – Name of file to be executed (should have .py or .ipy extension).

•  **options** (optional, list) – Extra command-line flags to be passed to IPython.

•  **commands** (optional, list) – Commands to send in on stdin

Returns

Return type  (stdout, stderr) of ipython subprocess.

IPython.testing.tools.ipexec_validate(fname, expected_out, expected_err='', options=None, commands=())
Utility to call ‘ipython filename’ and validate output/error.
This function raises an AssertionError if the validation fails.
Note that this starts IPython in a subprocess!

Parameters

•  **fname** (str) – Name of the file to be executed (should have .py or .ipy extension).

•  **expected_out** (str) – Expected stdout of the process.

•  **expected_err** (optional, str) – Expected stderr of the process.
• **options** (*optional*, *list*) – Extra command-line flags to be passed to IPython.

Returns

Return type None

### IPython.testing.tools.check_pairs(func, pairs)

Utility function for the common case of checking a function with a sequence of input/output pairs.

#### Parameters

- **func** (*callable*) – The function to be tested. Should accept a single argument.
- **pairs** (*iterable*) – A list of (input, expected_output) tuples.

#### Returns

- **None.** Raises an *AssertionError* if any output does not match the expected value.

### IPython.testing.tools.mute_warn()

### IPython.testing.tools.make_tempfile(name)

Create an empty, named, temporary file for the duration of the context.

### IPython.testing.tools.help_output_test(subcommand='')

test that `ipython [subcommand] -h` works

### IPython.testing.tools.help_all_output_test(subcommand='')

test that `ipython [subcommand] --help-all` works

### 8.64 Module: utils.PyColorize

Class and program to colorize python source code for ANSI terminals.

Based on an HTML code highlighter by Jurgen Hermann found at: [http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52298](http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52298)

Modifications by Fernando Perez ([fperez@colorado.edu](mailto:fperez@colorado.edu)).

Information on the original HTML highlighter follows:

MoinMoin - Python Source Parser

Title: Colorize Python source using the built-in tokenizer

Submitter: Jurgen Hermann Last Updated:2001/04/06

Version no:1.2

Description:

This code is part of MoinMoin ([http://moin.sourceforge.net/](http://moin.sourceforge.net/)) and converts Python source code to HTML markup, rendering comments, keywords, operators, numeric and string literals in different colors.
It shows how to use the built-in keyword, token and tokenize modules to scan Python source code and re-emit it with no changes to its original formatting (which is the hard part).

### 8.64.1 1 Class

```python
class IPython.utils.PyColorize.Parser(color_table=None, out=<_io.TextIOWrapper name='<stdout>' mode='w'
name='\<stdout\>' encoding='ANSI_X3.4-1968'>)

Bases: object

Format colored Python source.
```

```python
__init__ (color_table=None, out=<_io.TextIOWrapper name='<stdout>' mode='w'
name='\<stdout\>' encoding='ANSI_X3.4-1968'>)

Create a parser with a specified color table and output channel.
```

```python
__call__ (toktype, toktext, start_pos, end_pos, line)

Token handler, with syntax highlighting.
```

```python
format2 (raw, out=None, scheme='')

Parse and send the colored source.
```

If out and scheme are not specified, the defaults (given to constructor) are used.

out should be a file-type object. Optionally, out can be given as the string ‘str’ and the parser will automatically return the output in a string.

### 8.64.2 1 Function

```python
IPython.utils.PyColorize.main(argv=None)

Run as a command-line script: colorize a python file or stdin using ANSI color escapes and print to stdout.
```

Inputs:

*argv(None): a list of strings like sys.argv[1:] giving the command-line arguments. If None, use sys.argv[1:].

### 8.65 Module: `utils.capture`

IO capturing utilities.

### 8.65.1 3 Classes

```python
class IPython.utils.capture.RichOutput (data=None, metadata=None)

Bases: object
```

```python
__init__ (data=None, metadata=None)
```
class IPython.utils.capture.CapturedIO (stdout, stderr, outputs=None)

Bases: object

Simple object for containing captured stdout/err and rich display StringIO objects

Each instance $c$ has three attributes:

• $c$.stdout: standard output as a string
• $c$.stderr: standard error as a string
• $c$.outputs: a list of rich display outputs

Additionally, there’s a $c$.show() method which will print all of the above in the same order, and can be invoked simply via $c()$.

__init__ (stdout, stderr, outputs=None)

__call__ ()

write my output to sys.stdout/err as appropriate

outputs

A list of the captured rich display outputs, if any.

If you have a CapturedIO object $c$, these can be displayed in IPython using:

```python
from IPython.display import display
for o in c.outputs:
    display(o)
```

show ()

write my output to sys.stdout/err as appropriate

stderr

Captured standard error

stdout

Captured standard output

class IPython.utils.capture.capture_output (stdout=True, stderr=True, display=True)

Bases: object

context manager for capturing stdout/err

__init__ (stdout=True, stderr=True, display=True)

8.66 Module: utils.coloransi

Tools for coloring text in ANSI terminals.
8.66.1 5 Classes

class IPython.utils.coloransi.TermColors
    Bases: object
    Color escape sequences.

    This class defines the escape sequences for all the standard (ANSI?) colors in terminals. Also defines a
    NoColor escape which is just the null string, suitable for defining ‘dummy’ color schemes in terminals
    which get confused by color escapes.

    This class should be used as a mixin for building color schemes.

class IPython.utils.coloransi.InputTermColors
    Bases: object
    Color escape sequences for input prompts.

    This class is similar to TermColors, but the escapes are wrapped in “ and ” so that readline can properly
    know the length of each line and can wrap lines accordingly. Use this class for any colored text which
    needs to be used in input prompts, such as in calls to raw_input().

    This class defines the escape sequences for all the standard (ANSI?) colors in terminals. Also defines a
    NoColor escape which is just the null string, suitable for defining ‘dummy’ color schemes in terminals
    which get confused by color escapes.

    This class should be used as a mixin for building color schemes.

class IPython.utils.coloransi.NoColors
    Bases: object
    This defines all the same names as the colour classes, but maps them to empty strings, so it can easily
    be substituted to turn off colours.

class IPython.utils.coloransi.ColorScheme(_ColorScheme__scheme_name_, color-
dict=NoNone, **colormap)
    Bases: object
    Generic color scheme class. Just a name and a Struct.

    __init__(_ColorScheme__scheme_name_, colordict=NoNone, **colormap)

    copy(name=None)
    Return a full copy of the object, optionally renaming it.

class IPython.utils.coloransi.ColorSchemeTable(scheme_list=NoNone, default_scheme='')
    Bases: dict
    General class to handle tables of color schemes.

    It’s basically a dict of color schemes with a couple of shorthand attributes and some convenient methods.

    active_scheme_name -> obvious active_colors -> actual color table of the active scheme
__init__ (scheme_list=None, default_scheme='')
Create a table of color schemes.

The table can be created empty and manually filled or it can be created with a list of valid color schemes AND the specification for the default active scheme.

add_scheme (new_scheme)
Add a new color scheme to the table.

copy()
Return full copy of object

set_active_scheme (scheme, case_sensitive=0)
Set the currently active scheme.

Names are by default compared in a case-insensitive way, but this can be changed by setting the parameter case_sensitive to true.

8.66.2 1 Function

IPython.utils.coloransi.make_color_table (in_class)
Build a set of color attributes in a class.

Helper function for building the TermColors and :class:`InputTermColors`.

8.67 Module: utils.contexts

Miscellaneous context managers.

8.67.1 2 Classes

class IPython.utils.contexts.preserve_keys (dictionary, *keys)
Bases: object

Preserve a set of keys in a dictionary.

Upon entering the context manager the current values of the keys will be saved. Upon exiting, the dictionary will be updated to restore the original value of the preserved keys. Preserved keys which did not exist when entering the context manager will be deleted.

Examples

```python
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> with preserve_keys(d, 'b', 'c', 'd'):
...    del d['a']
...    del d['b']  # will be reset to 2
...    d['c'] = None  # will be reset to 3
...    d['d'] = 4  # will be deleted
...    d['e'] = 5
```
```python
...  print(sorted(d.items()))
...
[('c', None), ('d', 4), ('e', 5)]
>>> print(sorted(d.items()))
[('b', 2), ('c', 3), ('e', 5)]
```

**__init__(dictionary, *keys)**

class IPython.utils.contexts.NoOpContext

Bases: object

Context manager that does nothing.

### 8.68 Module: **utils.data**

Utilities for working with data structures like lists, dicts and tuples.

#### 8.68.1 3 Functions

**IPython.utils.data.uniq_stable**(elems) → list

Return from an iterable, a list of all the unique elements in the input, but maintaining the order in which they first appear.

Note: All elements in the input must be hashable for this routine to work, as it internally uses a set for efficiency reasons.

**IPython.utils.data.flatten**(seq)

Flatten a list of lists (NOT recursive, only works for 2d lists).

**IPython.utils.data.chop**(seq, size)

Chop a sequence into chunks of the given size.

### 8.69 Module: **utils.decorators**

Decorators that don’t go anywhere else.

This module contains misc. decorators that don’t really go with another module in IPython.utils. Before putting something here please see if it should go into another topical module in IPython.utils.

#### 8.69.1 2 Functions

**IPython.utils.decorators.flag_calls**(func)

Wrap a function to detect and flag when it gets called.

This is a decorator which takes a function and wraps it in a function with a ‘called’ attribute. wrapper.called is initialized to False.
The wrapper.called attribute is set to False right before each call to the wrapped function, so if the call fails it remains False. After the call completes, wrapper.called is set to True and the output is returned.

Testing for truth in wrapper.called allows you to determine if a call to func() was attempted and succeeded.

```python
IPython.utils.decorators.undoc(func)
```

Mark a function or class as undocumented.

This is found by inspecting the AST, so for now it must be used directly as @undoc, not as e.g. @decorators.undoc

## 8.70 Module: `utils.dir2`

A fancy version of Python’s builtin `dir()` function.

### 8.70.1 2 Functions

```python
IPython.utils.dir2.safe_hasattr(obj, attr)
```

In recent versions of Python, `hasattr()` only catches `AttributeError`. This catches all errors.

```python
IPython.utils.dir2.dir2(obj) → list of strings
```

Extended version of the Python builtin `dir()`, which does a few extra checks.

This version is guaranteed to return only a list of true strings, whereas `dir()` returns anything that objects inject into themselves, even if they are later not really valid for attribute access (many extension libraries have such bugs).

## 8.71 Module: `utils.encoding`

Utilities for dealing with text encodings

### 8.71.1 2 Functions

```python
IPython.utils.encoding.get_stream_enc(stream, default=None)
```

Return the given stream’s encoding or a default.

There are cases where `sys.std*` might not actually be a stream, so check for the encoding attribute prior to returning it, and return a default if it doesn’t exist or evaluates as False. `default` is `None` if not provided.

```python
IPython.utils.encoding.getdefaultencoding(prefer_stream=True)
```

Return IPython’s guess for the default encoding for bytes as text.

If `prefer_stream` is True (default), asks for `stdin.encoding` first, to match the calling Terminal, but that is often `None` for subprocesses.
Then fall back on locale.getpreferredencoding(), which should be a sensible platform default (that respects LANG environment), and finally to sys.getdefaultencoding() which is the most conservative option, and usually ASCII on Python 2 or UTF8 on Python 3.

8.72 Module: `utils.frame`

Utilities for working with stack frames.

8.72.1 4 Functions

`IPython.utils.frame.extract_vars(*names, **kw)`

Extract a set of variables by name from another frame.

Parameters

- `*names` – One or more variable names which will be extracted from the caller’s frame.
- `depth` (integer, optional) – How many frames in the stack to walk when looking for your variables. The default is 0, which will use the frame where the call was made.

Examples

```python
In [2]: def func(x):
   ...:     y = 1
   ...:     print(sorted(extract_vars('x','y').items()))
   ...:

In [3]: func('hello')
[['x', 'hello'], ('y', 1)]
```

`IPython.utils.frame.extract_vars_above(*names)`

Extract a set of variables by name from another frame.

Similar to extractVars(), but with a specified depth of 1, so that names are extracted exactly from above the caller.

This is simply a convenience function so that the very common case (for us) of skipping exactly 1 frame doesn’t have to construct a special dict for keyword passing.

`IPython.utils.frame.debugx(expr, pre_msg='')`  

Print the value of an expression from the caller’s frame.

Takes an expression, evaluates it in the caller’s frame and prints both the given expression and the resulting value (as well as a debug mark indicating the name of the calling function. The input must be of a form suitable for eval()).

An optional message can be passed, which will be prepended to the printed expr->value pair.
IPython.utils.frame.extract_module_locals(depth=0)
Returns (module, locals) of the function depth frames away from the caller

8.73 Module: utils.generics

Generic functions for extending IPython.
See http://pypi.python.org/pypi/simplegeneric.

8.73.1 2 Functions

IPython.utils.generics.inspect_object(*args, **kw)
Called when you do obj?

IPython.utils.generics.complete_object(*args, **kw)
Custom completer dispatching for python objects.

Parameters

- obj(object) – The object to complete.
- prev_completions(list) – List of attributes discovered so far.
- should return the list of attributes in obj. If you only wish to(This) –
- to the attributes already discovered normally, return (add) –
- + prev_completions.(own_atts)-

8.74 Module: utils.importstring

A simple utility to import something by its string name.

8.74.1 1 Function

IPython.utils.importstring.import_item(name)
Import and return bar given the string foo.bar.

Calling bar = import_item("foo.bar") is the functional equivalent of executing the code from foo import bar.

Parameters name(string) – The fully qualified name of the module/package being imported.

Returns mod – The module that was imported.

Return type module object
8.75 Module: utils.io

IO related utilities.

8.75.1 2 Classes

class IPython.utils.io.IOStream(stream, fallback=None)
Bases: object

__init__(stream, fallback=None)

class IPython.utils.io.Tee(file_or_name, mode='w', channel='stdout')
Bases: object

A class to duplicate an output stream to stdout/err.

This works in a manner very similar to the Unix ‘tee’ command.

When the object is closed or deleted, it closes the original file given to it for duplication.

__init__(file_or_name, mode='w', channel='stdout')

Construct a new Tee object.

Parameters

- file_or_name (filename or open filehandle (writable)) – File that will be duplicated
- mode (optional, valid mode for open()) – If a filename was give, open with this mode.
- channel (str, one of ['stdout', 'stderr']) –

close()

Close the file and restore the channel.

flush()

Flush both channels.

write(data)

Write data to both channels.

8.75.2 6 Functions

IPython.utils.io.ask_yes_no(prompt, default=None, interrupt=None)

Asks a question and returns a boolean (y/n) answer.

If default is given (one of ‘y’, ’n’), it is used if the user input is empty. If interrupt is given (one of ‘y’, ’n’), it is used if the user presses Ctrl-C. Otherwise the question is repeated until an answer is given.

An EOF is treated as the default answer. If there is no default, an exception is raised to prevent infinite loops.
Valid answers are: y/yes/n/no (match is not case sensitive).

IPython.utils.io.temp_pyfile(src, ext='.py')

Make a temporary python file, return filename and filehandle.

Parameters

• src (string or list of strings (no need for ending newlines if list)) – Source code to be written to the file.

• ext (optional, string) – Extension for the generated file.

Returns It is the caller’s responsibility to close the open file and unlink it.

Return type (filename, open filehandle)

IPython.utils.io.atomic_writing(*args, **kwargs)

DEPRECATED: moved to notebook.services.contents.fileio

IPython.utils.io.raw_print(*args, **kw)

Raw print to sys.__stdout__, otherwise identical interface to print().

IPython.utils.io.raw_print_err(*args, **kw)

Raw print to sys.__stderr__, otherwise identical interface to print().

IPython.utils.io.unicode_std_stream(stream='stdout')

DEPRECATED, moved to nbconvert.utils.io

8.76 Module: utils.ipstruct

A dict subclass that supports attribute style access.

Authors:

• Fernando Perez (original)

• Brian Granger (refactoring to a dict subclass)

8.76.1 1 Class

class IPython.utils.ipstruct.Struct(*args, **kw)

Bases: dict

A dict subclass with attribute style access.

This dict subclass has a a few extra features:

• Attribute style access.

• Protection of class members (like keys, items) when using attribute style access.

• The ability to restrict assignment to only existing keys.

• Intelligent merging.

• Overloaded operators.
__init__(*args, **kw)
Initialize with a dictionary, another Struct, or data.

Parameters

- **args** (*dict, Struct*) – Initialize with one dict or Struct
- **kw** (*dict*) – Initialize with key, value pairs.

Examples

```python
>>> s = Struct(a=10, b=30)
>>> s.a
10
>>> s.b
30
>>> s2 = Struct(s, c=30)
>>> sorted(s2.keys())
['a', 'b', 'c']
```

__add__(other)

`s + s2` -> New Struct made from `s.merge(s2)`.

Examples

```python
>>> s1 = Struct(a=10, b=30)
>>> s2 = Struct(a=20, c=40)
>>> s = s1 + s2
>>> sorted(s.keys())
['a', 'b', 'c']
```

__getattr__(key)

Get an attr by calling `dict.__getitem__()`. Like __setattr__(), this method converts `KeyError` to `AttributeError`.

Examples

```python
>>> s = Struct(a=10)
>>> s.a
10
>>> type(s.get)
<... 'builtin_function_or_method'>
>>> try:
...     s.b
... except AttributeError:
...     print("I don't have that key")
I don't have that key
```
__iadd__(other)
    s += s2 is a shorthand for s.merge(s2).

Examples

```python
>>> s = Struct(a=10, b=30)
>>> s2 = Struct(a=20, c=40)
>>> s += s2
>>> sorted(s.keys())
['a', 'b', 'c']
```

__isub__(other)
    Inplace remove keys from self that are in other.

Examples

```python
>>> s1 = Struct(a=10, b=30)
>>> s2 = Struct(a=40)
>>> s1 -= s2
>>> s1
{'b': 30}
```

__setattr__(key, value)
    Set an attr with protection of class members.
    This calls self.__setitem__() but convert KeyError to AttributeError.

Examples

```python
>>> s = Struct()
>>> s.a = 10
10
>>> try:
...     s.get = 10
... except AttributeError:
...     print("you can't set a class member")
...
you can't set a class member
```

__setitem__(key, value)
    Set an item with check for allownew.

Examples

```python
>>> s = Struct()
>>> s['a'] = 10
```
>>> s.allow_new_attr(False)
>>> s['a'] = 10
>>> s['a']
10
>>> try:
...      s['b'] = 20
... except KeyError:
...      print('this is not allowed')
...  
this is not allowed

__sub__(other)
s1 - s2 -> remove keys in s2 from s1.

Examples

>>> s1 = Struct(a=10,b=30)
>>> s2 = Struct(a=40)
>>> s = s1 - s2
>>> s
{'b': 30}

allow_new_attr(allow=True)
Set whether new attributes can be created in this Struct.
This can be used to catch typos by verifying that the attribute user tries to change already exists in this Struct.

copy()
Return a copy as a Struct.

Examples

>>> s = Struct(a=10,b=30)
>>> s2 = s.copy()
>>> type(s2) is Struct
True

hasattr(key)
hasattr function available as a method.
Implemented like has_key.

Examples

>>> s = Struct(a=10)
>>> s.hasattr('a')
True
>>> s.hasattr('b')
False

>>> s.hasattr('get')
False

merge(__loc_data=None, _Struct__conflict_solve=None, **kw)
Merge two Structs with customizable conflict resolution.

This is similar to update(), but much more flexible. First, a dict is made from data+key=value pairs. When merging this dict with the Struct S, the optional dictionary ‘conflict’ is used to decide what to do.

If conflict is not given, the default behavior is to preserve any keys with their current value (the opposite of the update() method’s behavior).

Parameters

- __loc_data (dict, Struct) – The data to merge into self
- __conflict_solve (dict) – The conflict policy dict. The keys are binary functions used to resolve the conflict and the values are lists of strings naming the keys the conflict resolution function applies to. Instead of a list of strings a space separated string can be used, like ‘a b c’.
- kw (dict) – Additional key, value pairs to merge in

Notes

The __conflict_solve dict is a dictionary of binary functions which will be used to solve key conflicts. Here is an example:

```python
__conflict_solve = dict(
    func1=['a','b','c'],
    func2=['d','e']
)
```

In this case, the function func1() will be used to resolve keys ‘a’, ‘b’ and ‘c’ and the function func2() will be used for keys ‘d’ and ‘e’. This could also be written as:

```python
__conflict_solve = dict(func1='a b c', func2='d e')
```

These functions will be called for each key they apply to with the form:

```python
func1(self['a'], other['a'])
```

The return value is used as the final merged value.

As a convenience, merge() provides five (the most commonly needed) pre-defined policies: preserve, update, add, add_flip and add_s. The easiest explanation is their implementation:

```python
preserve = lambda old,new: old
update = lambda old,new: new
add = lambda old,new: old + new
add_flip = lambda old,new: new + old  # note change of order!
add_s = lambda old,new: old + ' ' + new  # only for str!
```
You can use those four words (as strings) as keys instead of defining them as functions, and the merge method will substitute the appropriate functions for you.

For more complicated conflict resolution policies, you still need to construct your own functions.

**Examples**

This show the default policy:

```python
>>> s = Struct(a=10,b=30)
>>> s2 = Struct(a=20,c=40)
>>> s.merge(s2)
>>> sorted(s.items())
[('a', 10), ('b', 30), ('c', 40)]
```

Now, show how to specify a conflict dict:

```python
>>> s = Struct(a=10,b=30)
>>> s2 = Struct(a=20,b=40)
>>> conflict = {'update':'a','add':'b'}
>>> s.merge(s2,conflict)
>>> sorted(s.items())
[('a', 20), ('b', 70)]
```

8.77 Module: **utils.module_paths**

Utility functions for finding modules

Utility functions for finding modules on sys.path.

`find_mod` finds named module on sys.path.

`get_init` helper function that finds __init__ file in a directory.

`find_module` variant of imp.find_module in std_lib that only returns path to module and not an open file object as well.

8.77.1 3 Functions

**IPython.utils.module_paths.find_module** *(name, path=None)*

imp.find_module variant that only return path of module.

The `imp.find_module` returns a filehandle that we are not interested in. Also we ignore any bytecode files that `imp.find_module` finds.

**Parameters**

- **name** *(str)* – name of module to locate
- **path** *(list of str)* – list of paths to search for name. If path=None then search sys.path
Returns **filename** – Return full path of module or None if module is missing or does not have .py or .pyw extension

**Return type** *str*

IPython.utils.module_paths.get_init(dirname)

Get __init__ file path for module directory

**Parameters** *dirname* *(str)* – Find the __init__ file in directory *dirname*

**Returns** *init_path* – Path to __init__ file

**Return type** *str*

IPython.utils.module_paths.find_mod(module_name)

Find module *module_name* on sys.path

Return the path to module *module_name*. If *module_name* refers to a module directory then return path to __init__ file. Return full path of module or None if module is missing or does not have .py or .pyw extension. We are not interested in running bytecode.

**Parameters** *module_name* *(str)* –

**Returns** *modulepath* – Path to module *module_name*.

**Return type** *str*

### 8.78 Module: *utils.openpy*

Tools to open .py files as Unicode, using the encoding specified within the file, as per PEP 263.

Much of the code is taken from the tokenize module in Python 3.2.

#### 8.78.1 8 Functions

IPython.utils.openpy.detect_encoding(readline)

The detect_encoding() function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, readline, in the same way as the tokenize() generator.

It will call readline a maximum of twice, and return the encoding used (as a string) and a list of any lines (left as bytes) it has read in.

It detects the encoding from the presence of a utf-8 bom or an encoding cookie as specified in pep-0263. If both a bom and a cookie are present, but disagree, a SyntaxError will be raised. If the encoding cookie is an invalid charset, raise a SyntaxError. Note that if a utf-8 bom is found, ‘utf-8-sig’ is returned.

If no encoding is specified, then the default of ‘utf-8’ will be returned.

IPython.utils.openpy.open(filename)

Open a file in read only mode using the encoding detected by detect_encoding().
IPython.utils.openpy.source_to_unicode(txt, errors='replace', skip_encoding_cookie=True)

Converts a bytes string with python source code to unicode.

Unicode strings are passed through unchanged. Byte strings are checked for the python source file encoding cookie to determine encoding. `txt` can be either a bytes buffer or a string containing the source code.

IPython.utils.openpy.strip_encoding_cookie(filelike)

Generator to pull lines from a text-mode file, skipping the encoding cookie if it is found in the first two lines.

IPython.utils.openpy.read_py_file(filename, skip_encoding_cookie=True)

Read a Python file, using the encoding declared inside the file.

**Parameters**

- `filename` (*str*) – The path to the file to read.
- `skip_encoding_cookie` (*bool*) – If True (the default), and the encoding declaration is found in the first two lines, that line will be excluded from the output - compiling a unicode string with an encoding declaration is a SyntaxError in Python 2.

**Returns**

**Return type** A unicode string containing the contents of the file.

IPython.utils.openpy.read_py_url(url, errors='replace', skip_encoding_cookie=True)

Read a Python file from a URL, using the encoding declared inside the file.

**Parameters**

- `url` (*str*) – The URL from which to fetch the file.
- `errors` (*str*) – How to handle decoding errors in the file. Options are the same as for bytes.decode(), but here ‘replace’ is the default.
- `skip_encoding_cookie` (*bool*) – If True (the default), and the encoding declaration is found in the first two lines, that line will be excluded from the output - compiling a unicode string with an encoding declaration is a SyntaxError in Python 2.

**Returns**

**Return type** A unicode string containing the contents of the file.

IPython.utils.openpy.source_from_cache(path)

Given the path to a .pyc/.pyo file, return the path to its .py file.

The .pyc/.pyo file does not need to exist; this simply returns the path to the .py file calculated to correspond to the .pyc/.pyo file. If path does not conform to PEP 3147 format, ValueError will be raised. If sys.implementation.cache_tag is None then NotImplementedError is raised.

IPython.utils.openpy.cache_from_source(path, debug_override=None)

Given the path to a .py file, return the path to its .pyc/.pyo file.
The .py file does not need to exist; this simply returns the path to the .pyc/.pyo file calculated as if the .py file were imported. The extension will be .pyc unless sys.flags.optimize is non-zero, then it will be .pyo.

If debug_override is not None, then it must be a boolean and is used in place of sys.flags.optimize.

If sys.implementation.cache_tag is None then NotImplementedError is raised.

### 8.79 Module: `utils.path`

Utilities for path handling.

#### 8.79.1 1 Class

```python
class IPython.utils.path.HomeDirError
    Bases: Exception
```

#### 8.79.2 16 Functions

```python
IPython.utils.path.get_long_path_name(path)
    Expand a path into its long form.
    On Windows this expands any ~ in the paths. On other platforms, it is a null operation.

IPython.utils.path.unquote_filename(name, win32=False)
    On Windows, remove leading and trailing quotes from filenames.

IPython.utils.path.compress_user(path)
    Reverse of os.path.expanduser()

IPython.utils.path.get_py_filename(name, force_win32=None)
    Return a valid python filename in the current directory.
    If the given name is not a file, it adds `.py` and searches again. Raises IOError with an informative message if the file isn’t found.
    On Windows, apply Windows semantics to the filename. In particular, remove any quoting that has been applied to it. This option can be forced for testing purposes.

IPython.utils.path.filefind(filename, path_dirs=None)
    Find a file by looking through a sequence of paths.
    This iterates through a sequence of paths looking for a file and returns the full, absolute path of the first occurrence of the file. If no set of path dirs is given, the filename is tested as is, after running through expandvars() and expanduser(). Thus a simple call:

    filefind('myfile.txt')

    will find the file in the current working dir, but:
filefind('~/myfile.txt')

Will find the file in the users home directory. This function does not automatically try any paths, such as the cwd or the user's home directory.

Parameters

- **filename** *(str)* – The filename to look for.

- **path_dirs** *(str, None or sequence of str)* – The sequence of paths to look for the file in. If None, the filename need to be absolute or be in the cwd. If a string, the string is put into a sequence and the searched. If a sequence, walk through each element and join with filename, calling expandvars() and expanduser() before testing for existence.

Returns

Return type Raises IOError or returns absolute path to file.

IPython.utils.path.get_home_dir*(require_writable=False)*

Return the ‘home’ directory, as a unicode string.

Uses os.path.expanduser('~'), and checks for writability.

See stdlib docs for how this is determined. $HOME is first priority on ALL platforms.

Parameters **require_writable** *(bool [default: False])-*

if True: guarantees the return value is a writable directory, otherwise raises HomeDirError

if False: The path is resolved, but it is not guaranteed to exist or be writable.

IPython.utils.path.get_xdg_dir()

Return the XDG_CONFIG_HOME, if it is defined and exists, else None.

This is only for non-OS X posix (Linux,Unix,etc.) systems.

IPython.utils.path.get_xdg_cache_dir()

Return the XDG_CACHE_HOME, if it is defined and exists, else None.

This is only for non-OS X posix (Linux,Unix,etc.) systems.

IPython.utils.path.expand_path(s)

Expand $V ARS and ~names in a string, like a shell

Examples In [2]: os.environ[‘FOO’]=’test’

In [3]: expand_path(‘variable FOO is $FOO’) Out[3]: ‘variable FOO is test’

IPython.utils.path.unescape_glob*(string)*

Unescape glob pattern in string.

IPython.utils.path.shellglob*(args)*

Do glob expansion for each element in args and return a flattened list.

Unmatched glob pattern will remain as-is in the returned list.
IPython.utils.path.target_outdated(target, deps)
    Determine whether a target is out of date.
    target_outdated(target,deps) -> 1/0
    deps: list of filenames which MUST exist. target: single filename which may or may not exist.
    If target doesn’t exist or is older than any file listed in deps, return true, otherwise return false.

IPython.utils.path.target_update(target, deps, cmd)
    Update a target with a given command given a list of dependencies.
    target_update(target,deps,cmd) -> runs cmd if target is outdated.
    This is just a wrapper around target_outdated() which calls the given command if target is outdated.

IPython.utils.path.link(src, dst)
    Hard links src to dst, returning 0 or errno.
    Note that the special errno ENOLINK will be returned if os.link isn’t supported by the operating system.

IPython.utils.path.link_or_copy(src, dst)
    Attempts to hardlink src to dst, copying if the link fails.
    Attempts to maintain the semantics of shutil.copy.
    Because os.link does not overwrite files, a unique temporary file will be used if the target already exists, then that file will be moved into place.

IPython.utils.path.ensure_dir_exists(path, mode=493)
    ensure that a directory exists
    If it doesn’t exist, try to create it and protect against a race condition if another process is doing the same.
    The default permissions are 755, which differ from os.makedirs default of 777.

8.80 Module: utils.process

Utilities for working with external processes.

8.80.1 1 Class

class IPython.utils.process.FindCmdError
    Bases: Exception

8.80.2 4 Functions

IPython.utils.process.find_cmd(cmd)
    Find absolute path to executable cmd in a cross platform manner.
This function tries to determine the full path to a command line program using `which` on Unix/Linux/OS X and `win32api` on Windows. Most of the time it will use the version that is first on the users PATH.

Warning, don’t use this to find IPython command line programs as there is a risk you will find the wrong one. Instead find those using the following code and looking for the application itself:

```python
from IPython.utils.path import get_ipython_module_path
from IPython.utils.process import pycmd2argv
argv = pycmd2argv(get_ipython_module_path('IPython.terminal.ipapp'))
```

**Parameters**

```python
Parameters **cmd**(str) – The command line program to look for.
```

**IPython.utils.process.is_cmd_found(cmd)**

Check whether executable `cmd` exists or not and return a bool.

**IPython.utils.process.pycmd2argv(cmd)**

Take the path of a python command and return a list (argv-style).

This only works on Python based command line programs and will find the location of the `python` executable using `sys.executable` to make sure the right version is used.

For a given path `cmd`, this returns `[cmd]` if `cmd`’s extension is `.exe`, `.com` or `.bat`, and `[, cmd]` otherwise.

**Parameters**

```python
Parameters **cmd**(string) – The path of the command.
```

**Returns**

**Return type** argv-style list.

**IPython.utils.process.abbrev_cwd()**

Return abbreviated version of cwd, e.g. `d:mydir`

---

## 8.81 Module: `utils.rlineimpl`

Imports and provides the ‘correct’ version of readline for the platform.

Readline is used throughout IPython as:

```python
import IPython.utils.rlineimpl as readline
```

In addition to normal readline stuff, this module provides `have_readline` boolean and `_outputfile` variable used in IPython.utils.

### 8.81.1 1 Function

**IPython.utils.rlineimpl.clear_history()** → None

Clear the current readline history.
8.82 Module: **utils.sentinel**

Sentinel class for constants with useful reprs

### 8.82.1 1 Class

```python
class IPython.utils.sentinel.Sentinel(name, module, docstring=None)
```

Bases: `object`

```python
__init__(name, module, docstring=None)
```

8.83 Module: **utils.shimmodule**

A shim module for deprecated imports

### 8.83.1 3 Classes

```python
class IPython.utils.shimmodule.ShimWarning
```

Bases: `Warning`

A warning to show when a module has moved, and a shim is in its place.

```python
class IPython.utils.shimmodule.ShimImporter(src, mirror)
```

Bases: `object`

Import hook for a shim.

This ensures that submodule imports return the real target module, not a clone that will confuse `is` and `isinstance` checks.

```python
__init__(src, mirror)
```

```python
find_module(fullname, path=None)
```

Return self if we should be used to import the module.

```python
load_module(fullname)
```

Import the mirrored module, and insert it into `sys.modules`

```python
class IPython.utils.shimmodule.ShimModule(*args, **kwargs)
```

Bases: `module`

```python
__init__(*args, **kwargs)
```

8.84 Module: **utils.strdispatch**

String dispatch class to match regexps and dispatch commands.
8.84.1 Class

class IPython.utils.strdispatch.StrDispatch
    Bases: object

Dispatch (lookup) a set of strings / regexps for match.

Example:

```python
>>> dis = StrDispatch()
>>> dis.add_s('hei', 34, priority = 4)
>>> dis.add_s('hei', 123, priority = 2)
>>> dis.add_re('h.i', 686)
>>> print(list(dis.flat_matches('hei')))
[123, 34, 686]
```

__init__()  

add_re (regex, obj, priority=0)  
Adds a target regexp for dispatching  

add_s (s, obj, priority=0)  
Adds a target 'string' for dispatching  

dispatch (key)  
Get a seq of Commandchain objects that match key  

flat_matches (key)  
Yield all 'value' targets, without priority

8.85 Module: utils.sysinfo

Utilities for getting information about IPython and the system it's running in.

8.85.1 Functions

IPython.utils.sysinfo.pkg_commit_hash (pkg_path)  
Get short form of commit hash given directory pkg_path  

We get the commit hash from (in order of preference):

• IPython.utils.sysinfo.commit  
• git output, if we are in a git repository  

If these fail, we return a not-found placeholder tuple

Parameters pkg_path (str) – directory containing package only used for getting commit from active repo  

Returns

• hash_from (str) – Where we got the hash from - description
• `hash_str` (`str`) – short form of hash

IPython.utils.sysinfo.pkg_info(pkg_path)

Return dict describing the context of this package

**Parameters**

- `pkg_path` (`str`) – path containing __init__.py for package

**Returns**

- context – with named parameters of interest

**Return type**

dict

IPython.utils.sysinfo.get_sys_info()

Return useful information about IPython and the system, as a dict.

IPython.utils.sysinfo.sys_info()

Return useful information about IPython and the system, as a string.

### Examples

```
In [2]: print(sys_info())
{'commit_hash': '144fdae',  # random
   'commit_source': 'repository',
   'ipython_path': '/home/fperez/usr/lib/python2.6/site-packages/IPython',
   'ipython_version': '0.11.dev',
   'os_name': 'posix',
   'platform': 'Linux-2.6.35-22-generic-i686-with-Ubuntu-10.10-maverick',
   'sys_executable': '/usr/bin/python',
   'sys_platform': 'linux2',
   'sys_version': '2.6.6 (r266:84292, Sep 15 2010, 15:52:39) 
   \n   [GCC 4.4.5]')
```

IPython.utils.sysinfo.num_cpus()

Return the effective number of CPUs in the system as an integer.

This cross-platform function makes an attempt at finding the total number of available CPUs in the system, as returned by various underlying system and python calls.

If it can’t find a sensible answer, it returns 1 (though an error may make it return a large positive number that’s actually incorrect).

### 8.86 Module: `utils.syspathcontext`

Context managers for adding things to sys.path temporarily.

**Authors:**

- Brian Granger

### 8.86.1 2 Classes

class IPython.utils.syspathcontext.appended_to_syspath(dir)

Bases: `object`
A context for appending a directory to sys.path for a second.

```python
__init__(dir)
```

class IPython.utils.syspathcontext.prepended_to_syspath(dir)

A context for prepending a directory to sys.path for a second.

```python
__init__(dir)
```

## 8.87 Module: `utils.tempdir`

TemporaryDirectory class, copied from Python 3.2.

This is copied from the stdlib and will be standard in Python 3.2 and onwards.

### 8.87.1 3 Classes

class IPython.utils.tempdir.TemporaryDirectory(suffix='', prefix='tmp', dir=None)

Bases: object

Create and return a temporary directory. This has the same behavior as mkdtemp but can be used as a context manager. For .. rubric:: example

```python
with TemporaryDirectory() as tmpdir: ...
```

Upon exiting the context, the directory and everything contained in it are removed.

```python
__init__(suffix='', prefix='tmp', dir=None)
```

class IPython.utils.tempdir.NamedFileInTemporaryDirectory(filename, mode='w+b', bufsize=-1, **kwds)

Bases: object

```python
__init__(filename, mode='w+b', bufsize=-1, **kwds)
```

Open a file named `filename` in a temporary directory.

This context manager is preferred over NamedTemporaryFile in stdlib tempfile when one needs to reopen the file.

Arguments `mode` and `bufsize` are passed to `open`. Rest of the arguments are passed to TemporaryDirectory.

```python
with TemporaryWorkingDirectory() as tmpdir: ...
```

class IPython.utils.tempdir.TemporaryWorkingDirectory(suffix='', prefix='tmp', dir=None)

Bases: tempfile.TemporaryDirectory

Creates a temporary directory and sets the cwd to that directory. Automatically reverts to previous cwd upon cleanup. Usage example:

```python
with TemporaryWorkingDirectory() as tmpdir: ...
```
8.88 Module: **utils.terminal**

Utilities for working with terminals.

Authors:

- Brian E. Granger
- Fernando Perez
- Alexander Belchenko (e-mail: bialix AT ukr.net)

8.88.1 4 Functions

IPython.utils.terminal.toggle_set_term_title(val)

Control whether set_term_title is active or not.

set_term_title() allows writing to the console titlebar. In embedded widgets this can cause problems, so this call can be used to toggle it on or off as needed.

The default state of the module is for the function to be disabled.

**Parameters** val (bool) – If True, set_term_title() actually writes to the terminal (using the appropriate platform-specific module). If False, it is a no-op.

IPython.utils.terminal.set_term_title(title)

Set terminal title using the necessary platform-dependent calls.

IPython.utils.terminal.freeze_term_title()

IPython.utils.terminal.get_terminal_size(defaultx=80, defaulty=25)

8.89 Module: **utils.text**

Utilities for working with strings and text.

Inheritance diagram:

![Inheritance Diagram for utils.text](image)

8.88. Module: **utils.terminal**
8.89.1 5 Classes

class IPython.utils.text.LSString
    Bases: str

String derivative with a special access attributes.

These are normal strings, but with the special attributes:

    .l (or .list): value as list (split on newlines).
    .n (or .nlstr): original value (the string itself).
    .s (or .spstr): value as whitespace-separated string.
    .p (or .paths): list of path objects (requires path.py package)

Any values which require transformations are computed only once and cached.

Such strings are very useful to efficiently interact with the shell, which typically only understands whitespace-separated options for commands.

class IPython.utils.text.SList
    Bases: list

List derivative with a special access attributes.

These are normal lists, but with the special attributes:

    • .l (or .list): value as list (the list itself).
    • .n (or .nlstr): value as a string, joined on newlines.
    • .s (or .spstr): value as a string, joined on spaces.
    • .p (or .paths): list of path objects (requires path.py package)

Any values which require transformations are computed only once and cached.

fields(*fields)
    Collect whitespace-separated fields from string list

Allows quick awk-like usage of string lists.

Example data (in var a, created by `a = !ls -l`):

    -rwxrwxrwx 1 ville None 18 Dec 14 2006 ChangeLog
    drwxrwxrwx+ 6 ville None 0 Oct 24 18:05 IPython

    • a.fields(0) is `[-rwxrwxrwx', 'drwxrwxrwx+']`
    • a.fields(1,0) is `['1 -rwxrwxrwx', '6 drwxrwxrwx+']` (note the joining by space).
    • a.fields(-1) is `['ChangeLog', 'IPython']`

IndexErrors are ignored.

Without args, fields() just split()’s the strings.
grep *(pattern, prune=False, field=None)*

Return all strings matching ‘pattern’ (a regex or callable)

This is case-insensitive. If prune is true, return all items NOT matching the pattern.

If field is specified, the match must occur in the specified whitespace-separated field.

Examples:

```py
a.grep( lambda x: x.startswith('C') )
a.grep('Cha.*log', prune=True)
a.grep('chm', field=-1)
```

sort *(field=None, nums=False)*

sort by specified fields (see fields())

Example:

```py
a.sort(1, nums=True)
```

Sorts a by second field, in numerical order (so that 21 > 3)

class IPython.utils.text.EvalFormatter

Bases: string.Formatter

A String Formatter that allows evaluation of simple expressions.

Note that this version interprets a : as specifying a format string (as per standard string formatting), so if slicing is required, you must explicitly create a slice.

This is to be used in templating cases, such as the parallel batch script templates, where simple arithmetic on arguments is useful.

Examples

```py
In [1]: f = EvalFormatter()
In [2]: f.format('{n//4}', n=8)
Out[2]: '2'

In [3]: f.format("{greeting[slice(2,4)]}", greeting="Hello")
Out[3]: 'll'
```

class IPython.utils.text.FullEvalFormatter

Bases: string.Formatter

A String Formatter that allows evaluation of simple expressions.

Any time a format key is not found in the kwargs, it will be tried as an expression in the kwargs namespace.

Note that this version allows slicing using [1:2], so you cannot specify a format string. Use *EvalFormatter* to permit format strings.
Examples

```
In [1]: f = FullEvalFormatter()
In [2]: f.format('{n//4}', n=8)
Out[2]: u'2'

In [3]: f.format('{list(range(5))[2:4]}')
Out[3]: u'[2, 3]'

In [4]: f.format('{3*2}')
Out[4]: u'6'
```

class `IPython.utils.text.DollarFormatter`

Bases: `IPython.utils.text.FullEvalFormatter`

Formatter allowing Itpl style $foo replacement, for names and attribute access only. Standard {foo} replacement also works, and allows full evaluation of its arguments.

Examples

```
In [1]: f = DollarFormatter()
In [2]: f.format('{n//4}', n=8)
Out[2]: u'2'

In [3]: f.format('23 * 76 is $result', result=23*76)
Out[3]: u'23 * 76 is 1748'

In [4]: f.format('$a or {b}', a=1, b=2)
Out[4]: u'1 or 2'
```

8.89.2 13 Functions

`IPython.utils.text.indent (instr, nspaces=4, ntabs=0, flatten=False)`

Indent a string a given number of spaces or tabstops.

indent(str,nspaces=4,ntabs=0) -> indent str by ntabs+nspaces.

Parameters

- `instr` (basestring) – The string to be indented.
- `nspaces` (int (default: 4)) – The number of spaces to be indented.
- `ntabs` (int (default: 0)) – The number of tabs to be indented.
- `flatten` (bool (default: False)) – Whether to scrub existing indentation. If True, all lines will be aligned to the same indentation. If False, existing indentation will be strictly increased.

Returns str|unicode

Return type string indented by ntabs and nspaces.
**IPython.utils.text**

### list_strings (arg)

Always return a list of strings, given a string or list of strings as input.

#### Examples

```python
In [7]: list_strings('A single string')
Out[7]: ['A single string']

In [8]: list_strings(['A single string in a list'])
Out[8]: ['A single string in a list']

In [9]: list_strings(['A','list','of','strings'])
Out[9]: ['A', 'list', 'of', 'strings']
```

### marquee (txt='', width=78, mark='*')

Return the input string centered in a ‘marquee’.

#### Examples

```python
In [16]: marquee('A test',40)
Out[16]: '**************** A test ****************'

In [17]: marquee('A test',40,'-')
Out[17]: '---------------- A test ----------------'

In [18]: marquee('A test',40,' ')
Out[18]: ' A test '
```

### num_ini_spaces (strng)

Return the number of initial spaces in a string

### format_screen (strng)

Format a string for screen printing.

This removes some latex-type format codes.

### dedent (text)

Equivalent of textwrap.dedent that ignores unindented first line.

This means it will still dedent strings like: ‘‘foo is a bar ‘‘.

For use in wrap_paragraphs.

### wrap_paragraphs (text, ncols=80)

Wrap multiple paragraphs to fit a specified width.

This is equivalent to textwrap.wrap, but with support for multiple paragraphs, as separated by empty lines.

#### Returns

**Return type** list of complete paragraphs, wrapped to fill ncols columns.
IPython.utils.text.long_substr(data)

Return the longest common substring in a list of strings.

Credit: http://stackoverflow.com/questions/2892931/longest-common-substring-from-more-than-two-strings-python

IPython.utils.text.strip_email_quotes(text)

Strip leading email quotation characters (‘>’).

Removes any combination of leading ‘>’ interspersed with whitespace that appears identically in all lines of the input text.

Parameters text (str) –

Examples

Simple uses:

In [2]: strip_email_quotes('> > text')
Out[2]: 'text'

In [3]: strip_email_quotes('> > text
> > more')
Out[3]: 'text
more'

Note how only the common prefix that appears in all lines is stripped:

In [4]: strip_email_quotes('> > text
> > more
> more...')
Out[4]: '> text
> more
more...'

So if any line has no quote marks (‘>’), then none are stripped from any of them

In [5]: strip_email_quotes('> > text
> > more
last different')
Out[5]: '> > text
> > more
last different'

IPython.utils.text.strip_ansi(source)

Remove ansi escape codes from text.

Parameters source (str) – Source to remove the ansi from

IPython.utils.text.compute_item_matrix(items, row_first=False, empty=None, *args, **kwargs)

Returns a nested list, and info to columnize items

Parameters

• items – list of strings to columnize

• row_first (default False) – Whether to compute columns for a row-first matrix instead of column-first (default).

• empty (default None) – default value to fill list if needed

• separator_size (int (default=2)) – How much caracters will be used as a separation between each columns.
- **displaywidth** *(int (default=80)) – The width of the area onto which the columns should enter*

**Returns**

- **strings_matrix** –
  nested list of string, the outer most list contains as many list as rows, the innermost lists have each as many element as columns. If the total number of elements in *items* does not equal the product of rows*columns, the last element of some lists are filled with None.

- **dict_info** – some info to make columnize easier:
  - **num_columns** number of columns
  - **max_rows** maximum number of rows (final number may be less)
  - **column_widths** list of width of each columns
  - **optimal_separator_width** best separator width between columns

**Examples**

```python
In [1]: l = ['aaa','b','cc','d','eeeeee','f','g','h','i','j','k','l']
...: compute_item_matrix(l, displaywidth=12)
Out[1]:
(((['aaa', 'f', 'k'],
  ['b', 'g', 'i'],
  ['cc', 'h', None],
  ['d', 'i', None],
  ['eeeeee', 'j', None]),
  {'num_columns': 3,
   'column_widths': [5, 1, 1],
   'optimal_separator_width': 2,
   'max_rows': 5})
```

**IPython.utils.text.columnize** *(items, row_first=False, separator=' ', displaywidth=80, spread=False)*

Transform a list of strings into a single string with columns.

**Parameters**

- **items** *(sequence of strings)* – The strings to process.

- **row_first** *(default False)* – Whether to compute columns for a row-first matrix instead of column-first (default).

- **separator** *(str, optional [default is two spaces])* – The string that separates columns.

- **displaywidth** *(int, optional [default is 80])* – Width of the display in number of characters.

**Returns**
Return type The formatted string.

IPython.utils.text.get_text_list(list_, last_sep=' and ', sep=' ,', wrap_item_with='')

Return a string with a natural enumeration of items

```python
>>> get_text_list(['a', 'b', 'c', 'd'])
'a, b, c and d'
>>> get_text_list(['a', 'b', 'c'], ' or ')
'a, b or c'
>>> get_text_list(['a', 'b', 'c'], ', ')
'a, b, c'
>>> get_text_list(['a', 'b'], ' or ')
'a or b'
>>> get_text_list(['a'])
'a'
>>> get_text_list([])
''
>>> get_text_list(['a', 'b'], wrap_item_with='`')
'a` and `b`
>>> get_text_list(['a', 'b', 'c', 'd'], " " = "", sep=" + ")
'a + b + c = d'
```

### 8.90 Module: utils.timing

Utilities for timing code execution.

#### 8.90.1 7 Functions

IPython.utils.timing.clocku() → floating point number

Return the USER CPU time in seconds since the start of the process. This is done via a call to resource.getrusage, so it avoids the wraparound problems in time.clock().

IPython.utils.timing.clocks() → floating point number

Return the SYSTEM CPU time in seconds since the start of the process. This is done via a call to resource.getrusage, so it avoids the wraparound problems in time.clock().

IPython.utils.timing.clock() → floating point number

Return the TOTAL USER+SYSTEM CPU time in seconds since the start of the process. This is done via a call to resource.getrusage, so it avoids the wraparound problems in time.clock().

IPython.utils.timing.clock2() -> (t_user, t_system)

Similar to clock(), but return a tuple of user/system times.

IPython.utils.timing.timings_out(reps, func, *args, **kw) -> (t_total, t_per_call, output)

Execute a function reps times, return a tuple with the elapsed total CPU time in seconds, the time per call and the function’s output.

Under Unix, the return value is the sum of user+system time consumed by the process, computed via the resource module. This prevents problems related to the wraparound effect which the time.clock()
function has.

Under Windows the return value is in wall clock seconds. See the documentation for the time module for more details.

IPython.utils.timing.timings (reps, func, *args, **kw) -> (t_total, t_per_call)

Execute a function reps times, return a tuple with the elapsed total CPU time in seconds and the time per call. These are just the first two values in timings_out().

IPython.utils.timing.timing (func, *args, **kw) -> t_total

Execute a function once, return the elapsed total CPU time in seconds. This is just the first value in timings_out().

### 8.91 Module: `utils.tokenutil`

Token-related utilities

#### 8.91.1 3 Functions

IPython.utils.tokenutil.generate_tokens (readline)

wrap generate_tokens to catch EOF errors

IPython.utils.tokenutil.line_at_cursor (cell, cursor_pos=0)

Return the line in a cell at a given cursor position

Used for calling line-based APIs that don’t support multi-line input, yet.

**Parameters**

- **cell** (*text*) – multiline block of text
- **cursor_pos** (*integer*) – the cursor position

**Returns** (*line, offset*) – The line with the current cursor, and the character offset of the start of the line.

**Return type** (*text, integer*)

IPython.utils.tokenutil.token_at_cursor (cell, cursor_pos=0)

Get the token at a given cursor

Used for introspection.

Function calls are prioritized, so the token for the callable will be returned if the cursor is anywhere inside the call.

**Parameters**

- **cell** (*unicode*) – A block of Python code
- **cursor_pos** (*int*) – The location of the cursor in the block where the token should be found
8.92 Module: utils.tz

Timezone utilities
Just UTC-awareness right now

8.92.1 1 Class

```
class IPython.utils.tz.tzUTC
    Bases: datetime.tzinfo
    tzinfo object for UTC (zero offset)
```

8.92.2 1 Function

```
IPython.utils.tz.utc_aware(unaware)
    decorator for adding UTC tzinfo to datetime’s utcfoo methods
```

8.93 Module: utils.ulinecache

Wrapper around linecache which decodes files to unicode according to PEP 263.
This is only needed for Python 2 - linecache in Python 3 does the same thing itself.

8.93.1 2 Functions

```
IPython.utils.ulinecache.getlines(filename, module_globals=None)
    Get the lines for a file from the cache. Update the cache if it doesn’t contain an entry for this file already.

IPython.utils.ulinecache.getline(filename, lineno, module_globals=None)
```

8.94 Module: utils.version

Utilities for version comparison
It is a bit ridiculous that we need these.

8.94.1 1 Function

```
IPython.utils.version.check_version(v, check)
    check version string v >= check
```
If dev/prerelease tags result in TypeError for string-number comparison, it is assumed that the dependency is satisfied. Users on dev branches are responsible for keeping their own packages up to date.

### 8.95 Module: `utils.warn`

Utilities for warnings. Shouldn’t we just use the built in warnings module.

#### 8.95.1 4 Functions

**IPython.utils.warn.**

- **warn**
  
  ```python
  warn(msg, level=2, exit_val=1)
  ```
  
  Standard warning printer. Gives formatting consistency. Output is sent to io.stderr (sys.stderr by default).

  Options:

  - **-level(2): allows finer control:** 0 -> Do nothing. dummy function. 1 -> Print message. 2 -> Print ‘WARNING:’ + message. (Default level). 3 -> Print ‘ERROR:’ + message. 4 -> Print ‘FATAL ERROR:’ + message and trigger a sys.exit(exit_val).

  - **-exit_val (1):** exit value returned by sys.exit() for a level 4 warning. Ignored for all other levels.

- **info**
  
  ```python
  info(msg)
  ```
  
  Equivalent to warn(msg,level=1).

- **error**
  
  ```python
  error(msg)
  ```
  
  Equivalent to warn(msg,level=3).

- **fatal**
  
  ```python
  fatal(msg, exit_val=1)
  ```
  
  Equivalent to warn(msg,exit_val=exit_val,level=4).

### 8.96 Module: `utils.wildcard`

Support for wildcard pattern matching in object inspection.

#### 8.96.1 Authors

- Jörgen Stenarson <jorgen.stenarson@bostream.nu>
- Thomas Kluyver

#### 8.96.2 6 Functions

**IPython.utils.wildcard.**

- **create_typestr2type_dicts**
  
  ```python
  create_typestr2type_dicts(dont_include_in_type2typestr=['lambda'])
  ```
  
  Return dictionaries mapping lower case typename (e.g. ‘tuple’) to type objects from the types package, and vice versa.
IPython.utils.wildcard.is_type(obj, typestr_or_type)

is_type(obj, typestr_or_type) verifies if obj is of a certain type. It can take strings or actual python types for the second argument, i.e. 'tuple'<>TupleType. 'all' matches all types.

TODO: Should be extended for choosing more than one type.

IPython.utils.wildcard.show_hidden(str, show_all=False)

Return true for strings starting with single _ if show_all is true.

IPython.utils.wildcard.dict_dir(obj)

Produce a dictionary of an object’s attributes. Builds on dir2 by checking that a getattr() call actually succeeds.

IPython.utils.wildcard.filter_ns(ns, name_pattern='*', type_pattern='all', ignore_case=True, show_all=True)

Filter a namespace dictionary by name pattern and item type.

IPython.utils.wildcard.list_namespace(namespace, type_pattern, filter, ignore_case=False, show_all=False)

Return dictionary of all objects in a namespace dictionary that match type_pattern and filter.
9.1 History

9.1.1 Origins

IPython was starting in 2001 by Fernando Perez while he was a graduate student at the University of Colorado, Boulder. IPython as we know it today grew out of the following three projects:

- ipython by Fernando Pérez. Fernando began using Python and ipython began as an outgrowth of his desire for things like Mathematica-style prompts, access to previous output (again like Mathematica’s % syntax) and a flexible configuration system (something better than PYTHONSTARTUP).
- IPP by Janko Hauser. Very well organized, great usability. Had an old help system. IPP was used as the “container” code into which Fernando added the functionality from ipython and LazyPython.
- LazyPython by Nathan Gray. Simple but very powerful. The quick syntax (auto parens, auto quotes) and verbose/colored tracebacks were all taken from here.

Here is how Fernando describes the early history of IPython:

When I found out about IPP and LazyPython I tried to join all three into a unified system. I thought this could provide a very nice working environment, both for regular programming and scientific computing: shell-like features, IDL/Matlab numerics, Mathematica-type prompt history and great object introspection and help facilities. I think it worked reasonably well, though it was a lot more work than I had initially planned.

9.2 Licenses and Copyright

9.2.1 Licenses

IPython source code and examples are licensed under the terms of the new or revised BSD license, as follows:

Copyright (c) 2011, IPython Development Team

All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of the IPython Development Team nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

IPython documentation, examples and other materials are licensed under the terms of the Attribution 4.0 International (CC BY 4.0) license, as follows:

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 -- Definitions.

a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.
b. Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. Licensor means the individual(s) or entity(ies) granting rights under this Public License.

i. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2 -- Scope.
a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

   a. reproduce and Share the Licensed Material, in whole or in part; and

   b. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a) never produces Adapted Material.

5. Downstream recipients.

   a. Offer from the Licensor -- Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

   b. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).
b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 -- License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

   a. retain the following if it is supplied by the Licensor with the Licensed Material:

      i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

      ii. a copyright notice;

      iii. a notice that refers to this Public License;

      iv. a notice that refers to the disclaimer of warranties;

      v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

   b. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and
c. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 -- Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 -- Disclaimer of Warranties and Limitation of Liability.

a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT
ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.

b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 -- Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 -- Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the
Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 -- Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

9.2.2 About the IPython Development Team

Fernando Perez began IPython in 2001 based on code from Janko Hauser <jhauser-AT-zscout.de> and Nathaniel Gray <n8gray-AT-caltech.edu>. Fernando is still the project lead.

The IPython Development Team is the set of all contributors to the IPython project. This includes all of the IPython subprojects. See the release notes for a list of people who have contributed to each release.

9.2.3 Our Copyright Policy

IPython uses a shared copyright model. Each contributor maintains copyright over their contributions to IPython. But, it is important to note that these contributions are typically only changes (diffs/commits) to the repositories. Thus, the IPython source code, in its entirety is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire IPython Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change, when they commit the change to one of the IPython repositories.

Any new code contributed to IPython must be licensed under the BSD license or a similar (MIT) open source license.
9.2.4 Miscellaneous

Some files (DPyGetOpt.py, for example) may be licensed under different conditions. Ultimately each file indicates clearly the conditions under which its author/authors have decided to publish the code.

Versions of IPython up to and including 0.6.3 were released under the GNU Lesser General Public License (LGPL), available at http://www.gnu.org/copyleft/lesser.html.

Online versions of the Creative Commons licenses can be found at:

- http://creativecommons.org/licenses/by/4.0/
- http://creativecommons.org/licenses/by/4.0/legalcode.txt
C
- IPython.core.alias, 403
- IPython.core.application, 404
- IPython.core.autocall, 405
- IPython.core.builtin_trap, 405
- IPython.core.compilerop, 406
- IPython.core.completer, 407
- IPython.core.completerlib, 412
- IPython.core.crashhandler, 413
- IPython.core.debugger, 415
- IPython.core.error, 417
- IPython.core.events, 418
- IPython.core.excolors, 420
- IPython.core.extensions, 421
- IPython.core.formatters, 422
- IPython.core.getipython, 429
- IPython.core.history, 429
- IPython.core.historyapp, 434
- IPython.core.hooks, 434
- IPython.core.inputsplitter, 436
- IPython.core.inputtransformer, 439
- IPython.core.interactiveshell, 442
- IPython.core.logger, 456
- IPython.core.macro, 457
- IPython.core.magic, 458
- IPython.core.magic_arguments, 462
- IPython.core.oinspect, 465
- IPython.core.page, 469
- IPython.core.payload, 470
- IPython.core.payloadpage, 470
- IPython.core.prefilter, 471
- IPython.core.profileapp, 476
- IPython.core.profiledir, 476
- IPython.core.prompts, 477
- IPython.core.pylabtools, 479
- IPython.core.shellapp, 481
- IPython.core.splitinput, 481
- IPython.core.ultratb, 482
- IPython.core.usage, 489

D
- IPython.display, 489

E
- IPython.extensions.autoreload, 367
- IPython.extensions.storemagic, 368

I
- IPython, 401

L
- IPython.lib.backgroundjobs, 500
- IPython.lib.clipboard, 503
- IPython.lib.deepreload, 504
- IPython.lib.demo, 505
- IPython.lib.editorhooks, 511
- IPython.lib.guisupport, 512
- IPython.lib.inputhook, 513
- IPython.lib.latextools, 518
- IPython.lib.lexers, 519
- IPython.lib.pretty, 522
- IPython.lib.security, 525

P
- IPython.paths, 526

T
- IPython.terminal.embed, 527
- IPython.terminal.interactiveshell, 529
- IPython.terminal.ipapp, 532
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPython.testing</td>
<td>532</td>
</tr>
<tr>
<td>IPython.testing.decorators</td>
<td>533</td>
</tr>
<tr>
<td>IPython.testing.globalipapp</td>
<td>535</td>
</tr>
<tr>
<td>IPython.testing.ipunittest</td>
<td>536</td>
</tr>
<tr>
<td>IPython.testing.skipdoctest</td>
<td>538</td>
</tr>
<tr>
<td>IPython.testing.tools</td>
<td>538</td>
</tr>
<tr>
<td>IPython.utils.capture</td>
<td>542</td>
</tr>
<tr>
<td>IPython.utils.coloransi</td>
<td>543</td>
</tr>
<tr>
<td>IPython.utils.contexts</td>
<td>545</td>
</tr>
<tr>
<td>IPython.utils.data</td>
<td>546</td>
</tr>
<tr>
<td>IPython.utils.decorators</td>
<td>546</td>
</tr>
<tr>
<td>IPython.utils.dir2</td>
<td>547</td>
</tr>
<tr>
<td>IPython.utils.encoding</td>
<td>547</td>
</tr>
<tr>
<td>IPython.utils.frame</td>
<td>548</td>
</tr>
<tr>
<td>IPython.utils.generics</td>
<td>549</td>
</tr>
<tr>
<td>IPython.utils.importstring</td>
<td>549</td>
</tr>
<tr>
<td>IPython.utils.io</td>
<td>550</td>
</tr>
<tr>
<td>IPython.utils.ipstruct</td>
<td>551</td>
</tr>
<tr>
<td>IPython.utils.module_paths</td>
<td>556</td>
</tr>
<tr>
<td>IPython.utils.openpy</td>
<td>557</td>
</tr>
<tr>
<td>IPython.utils.path</td>
<td>559</td>
</tr>
<tr>
<td>IPython.utils.process</td>
<td>561</td>
</tr>
<tr>
<td>IPython.utils.py3compat</td>
<td>387</td>
</tr>
<tr>
<td>IPython.utils.PyColorize</td>
<td>541</td>
</tr>
<tr>
<td>IPython.utils.rlineimpl</td>
<td>562</td>
</tr>
<tr>
<td>IPython.utils.sentinel</td>
<td>563</td>
</tr>
<tr>
<td>IPython.utils.shimmodule</td>
<td>563</td>
</tr>
<tr>
<td>IPython.utils.strdispatch</td>
<td>563</td>
</tr>
<tr>
<td>IPython.utils.sysinfo</td>
<td>564</td>
</tr>
<tr>
<td>IPython.utils.syspathcontext</td>
<td>565</td>
</tr>
<tr>
<td>IPython.utils.tempdir</td>
<td>566</td>
</tr>
<tr>
<td>IPython.utils.terminal</td>
<td>567</td>
</tr>
<tr>
<td>IPython.utils.text</td>
<td>567</td>
</tr>
<tr>
<td>IPython.utils.timing</td>
<td>574</td>
</tr>
<tr>
<td>IPython.utils.tokenutil</td>
<td>575</td>
</tr>
<tr>
<td>IPython.utils.tz</td>
<td>576</td>
</tr>
<tr>
<td>IPython.utils.unicodecache</td>
<td>576</td>
</tr>
<tr>
<td>IPython.utils.version</td>
<td>576</td>
</tr>
<tr>
<td>IPython.utils.warn</td>
<td>577</td>
</tr>
<tr>
<td>IPython.utils.wildcard</td>
<td>577</td>
</tr>
</tbody>
</table>
Symbols

ipython-dir=<path>
command line option, 347
__add__ () (IPython.utils.ipstruct.Struct method), 552
__call__ () (IPython.core.crashhandler.CrashHandler method), 414
__call__ () (IPython.core.debugger.Tracer method), 415
__call__ () (IPython.core.formatters.FormatterABC method), 424
__call__ () (IPython.core.hooks.CommandChainDispatcher method), 435
__call__ () (IPython.core.magic.MagicAlias method), 461
__call__ () (IPython.core.ultratb.AutoFormattedTB method), 487
__call__ () (IPython.core.ultratb.VerboseTB method), 486
__call__ () (IPython.lib.backgroundjobs.BackgroundJobManager method), 501
__call__ () (IPython.lib.demo.Demo method), 509
__call__ () (IPython.terminal.embed.InteractiveShellEmbed method), 527
__call__ () (IPython.testing.ipunittest.Doc2UnitTester method), 537
__call__ () (IPython.testing.ipunittest.IPython2PythonConverter method), 537
__call__ () (IPython.utils.PyColorize.Parser method), 542
__call__ () (IPython.utils.capture.CapturedIO method), 543
__contains__ () (IPython.core.formatters.BaseFormatter method), 424
__getattr__ () (IPython.utils.ipstruct.Struct method), 552
__getstate__ () (IPython.core.macro.Macro method), 458
__html__ () (IPython.display.HTML method), 492
__iadd__ () (IPython.utils.ipstruct.Struct method), 552
__init__ () (IPython.Config method), 401
__init__ () (IPython.core.alias.Alias method), 403
__init__ () (IPython.core.alias.AliasManager method), 403
__init__ () (IPython.core.application.BaseIPythonApplication method), 404
__init__ () (IPython.core.autocall.IPyAutocall method), 405
__init__ () (IPython.core.builtin_trap.BuiltinTrap method), 406
__init__ () (IPython.core.compilerop.CachingCompiler method), 406
__init__ () (IPython.core.completer.Completer method), 408
__init__ () (IPython.core.completer.CompletionSplitter method), 408
__init__ () (IPython.core.completer.IPCompleter method), 409
__init__ () (IPython.core.crashhandler.CrashHandler method), 414
__init__ () (IPython.core.debugger.Pdb method), 416
__init__ () (IPython.core.debugger.Tracer method), 415
__init__ () (IPython.core.events.EventManager method), 419
__init__ () (IPython.core.excolors.Deprec method), 420
__init__ () (IPython.core.extensions.ExtensionManager method), 421
__init__ () (IPython.core.history.HistoryAccessor method), 421
__init__() (IPython.lib.lexers.IPythonConsoleLexer method), 520
__init__() (IPython.lib.lexers.IPythonTracebackLexer method), 520
__init__() (IPython.lib.pretty.Breakable method), 524
__init__() (IPython.lib.pretty.Group method), 525
__init__() (IPython.lib.pretty.GroupQueue method), 525
__init__() (IPython.lib.pretty.PrettyPrinter method), 523
__init__() (IPython.lib.pretty.RepresentationPrinter method), 524
__init__() (IPython.terminal.embed.InteractiveShellEmbed method), 527
__init__() (IPython.terminal.interactiveshell.ReadlineNoRecord method), 529
__init__() (IPython.terminal.interactiveshell.TerminalMagics method), 527
__init__() (IPython.terminal.interactiveshell.ReadlineNoRecord method), 529
__init__() (IPython.terminal.ipapp.IPAppCrashHandler method), 532
__init__() (IPython.terminal.ipapp.IAppCrashHandler method), 532
__init__() (IPython.terminal.ipapp.IAppCrashHandler method), 532
__init__() (IPython.testing.globalipapp.StreamProxy method), 536
__init__() (IPython.testing.globalipapp.StreamProxy method), 536
__init__() (IPython.testing.globalipapp.StreamProxy method), 536
__init__() (IPython.testing.globalipapp.StreamProxy method), 536
__init__() (IPython.testing.ipunit.test.Doc2UnitTester method), 537
__init__() (IPython.testing.ipunit.test.IPython2PythonConverter method), 537
__init__() (IPython.testing.tools.AssertPrints method), 539
__init__() (IPython.utils.PyColorize.Parser method), 542
__init__() (IPython.utils.capture.CapturedIO method), 543
__init__() (IPython.utils.capture.CapturedIO method), 543
__init__() (IPython.utils.capture.CapturedIO method), 543
__init__() (IPython.utils.coloransi.ColorScheme method), 544
__init__() (IPython.utils.coloransi.ColorSchemeTable method), 544
__init__() (IPython.utils.coloransi.ColorSchemeTable method), 544
__init__() (IPython.utils.contexts.preserve_keys method), 546
__init__() (IPython.utils.io.IOStream method), 550
__init__() (IPython.utils.io.Tee method), 550
__init__() (IPython.utils.ipstruct.Struct method), 552
__init__() (IPython.utils.sentinel.Sentinel method), 563
__init__() (IPython.utils.shimmodule.ShimImporter method), 563
__init__() (IPython.utils.shimmodule.ShimModule method), 563
__init__() (IPython.utils.syspathcontext.appended_to_syspath method), 566
__init__() (IPython.utils.syspathcontext.prepended_to_syspath method), 566
__init__() (IPython.utils.tempdir.NamedFileInTemporaryDirectory method), 566
__init__() (IPython.utils.tempdir.TemporaryDirectory method), 566
__isub__() (IPython.utils.ipstruct.Struct method), 553
__iter__() (IPython.core.hooks.CommandChainDispatcher method), 435
__repr__() (IPython.display.FileLink method), 491
__repr__() (IPython.display.FileLinks method), 492
__setattr__() (IPython.utils.ipstruct.Struct method), 553
__setitem__() (IPython.utils.ipstruct.Struct method), 553
__sub__() (IPython.utils.ipstruct.Struct method), 554
A
abbrev_cwd() (in module IPython.utils.process), 562
activate() (IPython.core.builtin_trap.BuiltinTrap method), 406
activate_matplotlib() (in module IPython.core.pylabtools), 406
add() (IPython.core.hooks.CommandChainDispatcher method), 435
add_builtin() (IPython.core.builtin_trap.BuiltinTrap method), 406
add_re() (IPython.utils.strdispatch.StrDispatch method), 564
add_scheme() (IPython.utils.coloransi.ColorSchemeTable method), 545
add_submodule() (in module IPython.lib.deepreload), 505
add_to_parser() (IPython.core.magic_arguments.ArgDecorator IPython.core.inputtransformer), 441
AddAssignmentChecker (class in IPython.core.prefilter), 441
AddMethodWrapper (class in IPython.core.magic_arguments), 464
ast_parse() (IPython.core.compilerop.CachingCompiler method), 406
atexit_operations() (IPython.core.interactiveshell.InteractiveShell method), 442
atomic_writing() (in module IPython.utils.io), 551
attr_matches() (IPython.core.completer.Completer method), 409
Audio (class in IPython.display), 489
auto_rewrite_input() (IPython.core.interactiveshell.InteractiveShell method), 443
AutocallChecker (class in IPython.core.prefilter), 475
AutoFormattedTB (class in IPython.core.ultratb), 487
AutoHandler (class in IPython.core.prefilter), 475
autoindent() (IPython.terminal.interactiveshell.TerminalMagics method), 529
automagic magic command, 284
AutoMagicChecker (class in IPython.core.prefilter), 474
back() (IPython.lib.demo.Demo method), 509
back_latex_name_matches() (in module IPython.core.completer), 412
back_unicode_name_matches() (in module IPython.core.completer), 412
BackgroundJobBase (class in IPython.lib.backgroundjobs), 502
BackgroundJobExpr (class in IPython.lib.backgroundjobs), 502
BackgroundJobFunc (class in IPython.lib.backgroundjobs), 503
BackgroundJobManager (class in IPython.lib.backgroundjobs), 501
banner (MyKernel attribute), 382
BaseFormatter (class in IPython.core.formatters), 424
BaseIPythonApplication (class in IPython.core.application), 404
bash
cell magic, 316
BdbQuit_excepthook() (in module IPython.core.debugger), 417
BdbQuit_IPython_excepthook() (in module IPython.core.debugger), 417
begin_group() (IPython.lib.pretty.PrettyPrinter method), 523
cd
magic command, 285
cd_completer() (in module IPython.core.completerlib), 413
bookmark
magic command, 284
break_() (IPython.lib.pretty.PrettyPrinter method), 524
Breakable (class in IPython.lib.pretty), 524
breakable() (IPython.lib.pretty.PrettyPrinter method), 524
buffered_tokens() (IPython.lib.lexers.IPythonConsoleLexer method), 521
build_ipylexer() (in module IPython.lib.lexers), 521
builtin_mod (in module IPython.utils.py3compat), 388
builtin_mod_name (in module IPython.utils.py3compat), 388
BuiltinTrap (class in IPython.core.builtin_trap), 406
Bunch (class in IPython.core.magic), 458
bytes_to_str() (in module IPython.utils.py3compat), 388
C

check() (IPython.core.prefilter.AssignmentChecker method), 474
check() (IPython.core.prefilter.AutocallChecker method), 475
check() (IPython.core.prefilter.AutoMagicChecker method), 474
check() (IPython.core.prefilter.PyAutocallChecker method), 474
check() (IPython.core.prefilter.PyPrefilterChecker method), 474
check_complete() (IPython.core.inputsplitter.InputSplitter method), 437
check_linecache_ipython() (in module IPython.core.compilerop), 407
check_linecache() (IPython.core.compilerop method), 407
class method), 477
create_profile_dir_by_name() (IPython.core.profiledir.ProfileDir class method), 477
create_typestr2type_dicts() (in module IPython.utils.wildcard), 577

crimson_editor() (in module IPython.lib.editorhooks), 512
CUnicodeIO (in module IPython.lib.pretty), 523
current_gui() (IPython.lib.inputhook.InputHookManager method), 514
cwd_filt() (in module IPython.core.prompts), 479
cwd_filt2() (in module IPython.core.prompts), 479

D
decorate_fn_with_doc() (in module IPython.core.debugger), 417
decorated_dummy() (in module IPython.testing.decorators), 535
dedent() (in module IPython.utils.text), 571
deprecated() (in module IPython.core.reloadapi), 461
deprecated() (in module IPython.core.magic.MagicsManager), 458
default_aliases() (in module IPython.core.alias), 404
default_argv() (in module IPython.testing.tools), 540
default_config() (in module IPython.testing.tools), 540
default_option() (IPython.core.magic.MagicsManager method), 460
defaults (class in IPython.core.magic_arguments), 464
define_alias() (IPython.core.alias.AliasManager method), 403

Index 597
define_macro() (IPython.core.interactiveshell.InteractiveShell method), 444
define_magic() (IPython.core.magic.MagicsManager method), 458
del_var() (IPython.core.interactiveshell.InteractiveShell method), 444
delims (IPython.core.completer.CompletionSplitter attribute), 408
dict_dir() (in module IPython.utils.wildcard), 578
dict_key_matches() (IPython.core.completer.IPCompleter method), 410
dispatch() (IPython.utils.strdispatch.StrDispatch method), 564
disable() (in module IPython.display), 496
disable_toolbar() (in module IPython.display), 497
display() (in module IPython.display), 496
display_html() (in module IPython.display), 496
display_javascript() (in module IPython.display), 497
display_jpeg() (in module IPython.display), 497
display_json() (in module IPython.display), 497
display latent() (in module IPython.display), 497
display_markdown() (in module IPython.display), 498
display_pretty() (in module IPython.display), 499
display_page() (in module IPython.core.page), 469
display_pdf() (in module IPython.display), 498
display_png() (in module IPython.display), 498
display_pretty() (in module IPython.display), 498
display_svg() (in module IPython.display), 499
display_formatter (class in IPython.core.formatters), 422
display_object (class in IPython.display), 490
do_complete() (MyKernel method), 383
do_d() (IPython.core.debugger.Pdb method), 416
do_down() (IPython.core.debugger.Pdb method), 416
do_execute() (MyKernel method), 382
do_history() (MyKernel method), 384
do_inspect() (MyKernel method), 384
do_is_complete() (MyKernel method), 384
do_pdef() (IPython.core.debugger.Pdb method), 416
do_pdoc() (IPython.core.debugger.Pdb method), 416
do_pfile() (IPython.core.debugger.Pdb method), 416
do_pinfo() (IPython.core.debugger.Pdb method), 416
do_pinfo2() (IPython.core.debugger.Pdb method), 416
do_psource() (IPython.core.debugger.Pdb method), 416
do_q() (IPython.core.debugger.Pdb method), 416
do_quit() (IPython.core.debugger.Pdb method), 416
do_shutdown() (MyKernel method), 384
do_u() (IPython.core.debugger.Pdb method), 416
do_up() (IPython.core.debugger.Pdb method), 417
do_w() (IPython.core.debugger.Pdb method), 417
do_where() (IPython.core.debugger.Pdb method), 417
Doc2UnitTester (class in IPython.testing.ipunittest), 537
doctest mode
  magic command, 287
doctest_refactor_print() (in module IPython.utils.py3compat), 390
DollarFormatter (class in IPython.utils.text), 570
drop_by_id() (IPython.core.interactiveshell.InteractiveShell method), 444
E
dict
  magic command, 288
dict() (IPython.lib.demo.Demo method), 509
dict_syntax_error() (IPython.terminal.interactiveshell.TerminalInteractiveShell method), 530
EDITOR, 364
dict() (in module IPython.core.hooks), 435
dict() (in module IPython.lib.editorhooks), 512
EmacsChecker (class in IPython.core.prefilter), 474
EmacsHandler (class in IPython.core.prefilter), 475
embed() (in module IPython.terminal.embed), 528
embed_kernel() (in module IPython), 401
EmbeddedMagics (class in IPython.terminal.embed), 527
enable() (IPython.lib.ipinputhook.Gtk3InputHook method), 518
enable() (IPython.lib.ipinputhook.GtkInputHook method), 516
enable() (IPython.lib.ipinputhook.PygetInputHook method), 518
enable() (IPython.lib.ipinputhook.Qt4InputHook method), 516
enable() (IPython.lib.ipinputhook.TkInputHook method), 517
enable() (IPython.lib.ipinputhook.WxInputHook method), 515
enable_gui() (IPython.lib.ipinputhook.InputHookManager method), 514
enable_gui() (IPython.terminal.interactiveshell.TerminalInteractiveShell static method), 531
enable_matplotlib() (IPython.core.interactiveshell.InteractiveShell method), 445
enable_pylab() (IPython.core.interactiveshell.InteractiveShell method), 445
end_group() (IPython.lib.pretty.PrettyPrinter method), 524
end_session() (IPython.core.history.HistoryManager method), 432
ends_in_comment_or_string() (in module IPython.core.inputtransformer), 441
ensure_dir_exists() (in module IPython.utils.path), 561
ensure_fromlist() (in module IPython.lib.deepreload), 505
env
  magic command, 290
environment variable
  EDITOR, 364
  HOME, 319
  IPYTHON, 167, 391
  IPYTHON_DIR, 167, 391
  IPYTHONDIR, 167, 319, 347, 391
PATH, 2
  PYTHONSTARTUP, 328, 579
eqrepr() (in module IPython.core.ultratb), 488
error() (in module IPython.utils.warn), 577
error() (IPython.core.magic_arguments.MagicArgumentParser method), 463
escaped_commands() (in module IPython.core.inputtransformer), 441
ev() (IPython.core.interactiveshell.InteractiveShell method), 445
Index 601

**get_pyos_inputhook()**
(IPython.lib.inputhook.InputHookManager method), 515

**get_pyos_inputhook_as_func()**
(IPython.lib.inputhook.InputHookManager method), 515

**get_range()** (IPython.core.history.HistoryAccessor method), 430

**get_range()** (IPython.core.history.HistoryManager method), 432

**get_range_by_str()** (IPython.core.history.HistoryAccessor method), 430

**get_readline_tail()** (IPython.terminal.interactiveshell.ReadlineNoRecord method), 529

**get_root_modules()** (in module IPython.core.completerlib), 413

**get_session_info()** (IPython.core.history.HistoryAccessor method), 431

**get_session_info()** (IPython.core.history.HistoryManager method), 432

**get_stream_enc()** (in module IPython.utils.encoding), 547

**get_sys_info()** (in module IPython.utils.sysinfo), 565

**get_tail()** (IPython.core.history.HistoryAccessor method), 431

**get_terminal_size()** (in module IPython.utils.terminal), 567

**get_text_list()** (in module IPython.utils.text), 574

**get_xdg_cache_dir()** (in module IPython.utils.path), 560

**get_xdg_dir()** (in module IPython.utils.path), 560

**getargs()** (in module IPython.core.ultratb), 488

**getargspec()** (in module IPython.core.oinspect), 468

**getcwd()** (in module IPython.utils.py3compat), 390

**getdefaultencoding()** (in module IPython.utils.encoding), 547

**getdoc()** (in module IPython.core.oinspect), 467

**getfigs()** (in module IPython.core.pylabtools), 479

**getline()** (in module IPython.utils.ulinecache), 576

**getoutput()** (IPython.core.interactiveshell.InteractiveShell method), 447

**getsource()** (in module IPython.core.oinspect), 467

**global_matches()** (IPython.core.completer.Completer method), 409

**GlutInputHook** (class in IPython.lib.inputhook), 517

**gnuclient()** (in module IPython.lib.editorhooks), 512

**grep()** (IPython.utils.text.SList method), 568

**Group** (class in IPython.lib.pretty), 525

**GroupQueue** (class in IPython.lib.pretty), 525

**Gtk3InputHook** (class in IPython.lib.inputhook), 518

**GtkInputHook** (class in IPython.lib.inputhook), 516

**gui**

**getargs()** (in module IPython.core.ultratb), 488

**getargspec()** (in module IPython.core.oinspect), 468

**getcwd()** (in module IPython.utils.py3compat), 390

**getdefaultencoding()** (in module IPython.utils.encoding), 547

**getdoc()** (in module IPython.core.oinspect), 467

**getfigs()** (in module IPython.core.pylabtools), 479

**getline()** (in module IPython.utils.ulinecache), 576

**getoutput()** (IPython.core.interactiveshell.InteractiveShell method), 447

**getsource()** (in module IPython.core.oinspect), 467

**global_matches()** (IPython.core.completer.Completer method), 409

**GlutInputHook** (class in IPython.lib.inputhook), 517
IPython.utils.frame (module), 548
IPython.utils.generics (module), 549
IPython.utils.importstring (module), 549
IPython.utils.io (module), 550
IPython.utils.ipstruct (module), 551
IPython.utils.module_paths (module), 556
IPython.utils.openpy (module), 557
IPython.utils.path (module), 559
IPython.utils.process (module), 561
IPython.utils.py3compat (module), 387
IPython.utils.PyColorize (module), 541
IPython.utils.rlineimpl (module), 562
IPython.utils.sentinel (module), 563
IPython.utils.shimmodule (module), 563
IPython.utils.strdispatch (module), 564
IPython.utils.sysinfo (module), 566
IPython.utils.syspathcontext (module), 565
IPython.utils.tempdir (module), 566
IPython.utils.terminal (module), 567
IPython.utils.text (module), 567
IPython.utils.timing (module), 574
IPython.utils.tokenutil (module), 575
IPython.utils.tz (module), 576
IPython.utils.ulinecache (module), 576
IPython.utils.warn (module), 577
IPython2PythonConverter (class in IPython.testing.ipunittest), 537
IPYTHON_DIR, 167, 391
IPythonConsoleLexer (class in IPython.lib.lexers), 520
IPythonCoreError (class in IPython.core.error), 418
IPythonDemo (class in IPython.lib.demo), 510
IPYTHONDIR, 167, 391
IPythonDisplayFormatter (class in IPython.core.formatters), 428
IPythonInputSplitter (class in IPython.core.inputsplitter), 438
IPythonLineDemo (class in IPython.lib.demo), 510
IPythonPartialTracebackLexer (class in IPython.lib.lexers), 519
IPythonTracebackLexer (class in IPython.lib.lexers), 520
is_alias() (IPython.core.alias.AliasManager method), 403
is_cmd_found() (in module IPython.utils.process), 562
is_event_loop_running_qt4() (in module IPython.lib.guisupport), 513
is_event_loop_running_wx() (in module IPython.lib.guisupport), 513
is_importable() (in module IPython.core.completerlib), 413
is_recursion_error() (in module IPython.core.ultratb), 488
is_shadowed() (in module IPython.core.prefilter), 475
is_simple_callable() (in module IPython.core.oinspect), 467
is_type() (in module IPython.utils.wildcard), 577
isidentifier() (in module IPython.utils.py3compat), 390
iteritems() (in module IPython.utils.py3compat), 387
itervalues() (in module IPython.utils.py3compat), 387

J
javascript
cell magic, 316
Javasclipt (class in IPython.display), 494
JavascriptFormatter (class in IPython.core.formatters), 427
jed() (in module IPython.lib.editorhooks), 512
JPEGFormatter (class in IPython.core.formatters), 427
JSON (class in IPython.display), 494
JSONFormatter (class in IPython.core.formatters), 427
jump() (IPython.lib.demo.Demo method), 509

K
kate() (in module IPython.lib.editorhooks), 512
kill_embedded() (IPython.terminal.ipython.EmbeddedMagics method), 527
killbgscripts
magic command, 292
KillEmbedded (class in IPython.terminal.ipython), 527
komodo() (in module IPython.lib.editorhooks), 511
kpsewhich() (in module IPython.lib.latextools), 519
kwds (class in IPython.core.magic_arguments), 464

L
language (MyKernel attribute), 382
language_info (MyKernel attribute), 382
language_version (MyKernel attribute), 382
magic command, 295
magic command
alias, 282
alias_magic, 283
autocall, 283
automagic, 284
bookmark, 284
cd, 285
colors, 285
config, 286
debug, 287
dhist, 287
dirs, 287
doctest_mode, 287
dict, 288
eval, 289
history, 291
install_default_config, 292
install_ext, 292
install_profiles, 292
killbgscripts, 292
load, 292
load_ext, 293
loadpy, 293
logoff, 293
logon, 293
logstart, 293
logstate, 294
logstop, 294
lsmagic, 294
macro, 294
magic, 295
matplotlib, 295
notebook, 296
page, 297
pastebin, 297
pdb, 297
pdef, 297
pdoc, 298
pfile, 298
pinfo, 298
pinfo2, 298
popd, 298
pprint, 298
precision, 298
profile, 299
prun, 299
psearch, 301
psource, 302
pushd, 302
pwd, 302
pycat, 302
pylab, 302
quickref, 303
recall, 303
rehashx, 304
reload_ext, 304
rerun, 304
reset, 304
reset_selective, 305
run, 306
save, 308
csc, 309
set_env, 310
sx, 310
system, 311
tb, 311
time, 312
timeit, 312
unalias, 313
unload_ext, 314
who, 314
who_ls, 314
whos, 315
xdel, 315
xmode, 315
magic() (IPython.core.interactive Shell.InteractiveShell method), 448
magic_arguments (class in IPython.core.magic_arguments), 464
magic_matches() (IPython.core.completer.IPCompleter.method), 410
magic_run_completer() (in module IPython.core.completerlib), 413
MagicAlias (class in IPython.core.magic), 461
MagicArgumentParser (class in IPython.core.magic_arguments), 463
MagicHandler (class in IPython.core.prefilter), 475
Magics (class in IPython.core.magic), 460
magics_class() (in module IPython.core.magic), 461
MagicsManager (class in IPython.core.magic), 458
main() (in module IPython.utils.PyColorize), 542
mainloop() (IPython.terminal.embed.InteractiveShellEmbed method), 528
ostream (IPython.core.ultratb.TBTools attribute), 484
osx_clipboard_get() (in module IPython.lib.clipboard), 504
outputs (IPython.utils.capture.CapturedIO attribute), 543

P
page
magic command, 297
page() (in module IPython.core.page), 469
page() (in module IPython.core.payloadpage), 471
page_dumb() (in module IPython.core.page), 469
page_file() (in module IPython.core.page), 469
page_guiref() (in module IPython.core.usage), 489
page_more() (in module IPython.core.page), 470
pager_page() (in module IPython.core.page), 469
parse_argstring() (in module IPython.core.magic_arguments), 464
parse_argstring() (IPython.core.magic_args.MagicArguments method), 463
parse_command_line()
(IPython.terminal.ipapp.TerminalIPythonApp method), 532
parse_options()
(IPython.core.magic.Magic class method), 460
parse_test_output() (in module IPython.core.formatters), 539
Parser (class in IPython.utils.PyColorize), 542
passwd() (in module IPython.lib.security), 525
passwd_check() (in module IPython.lib.security), 526
paste() (IPython.terminal.interactiveshell.TerminalMagics method), 530
pastebin
magic command, 297
PATH, 2
PayloadManager (class in IPython.core.payload), 470
pdb
magic command, 297
Pdb (class in IPython.core.debugger), 416
pdef
magic command, 297
pdef() (IPython.core.oinspect.Inspector method), 466
PDFFormatter (class in IPython.core.formatters), 428
pdoc
magic command, 298
pdoc() (IPython.core.oinspect.Inspector method), 466
penalize_magics_key() (in module IPython.core.completer), 412
perl
cell magic, 316
pfile
magic command, 298
pfile() (IPython.core.oinspect.Inspector method), 466
pinfo
magic command, 298
pinfo() (IPython.core.oinspect.Inspector method), 466
pinfo2
magic command, 298
pkg_commit_hash() (in module IPython.utils.sysinfo), 564
pkg_info() (in module IPython.utils.sysinfo), 565
PlainTextFormatter (class in IPython.core.formatters), 426
PNGFormatter (class in IPython.core.formatters), 427
pop() (IPython.core.formatters.BaseFormatter method), 425
popd
magic command, 298
post_cmd() (IPython.lib.demo.Demo method), 509
post_execute() (in module IPython.core.events), 420
post_run_cell() (in module IPython.core.events), 420
pprint
magic command, 298
pprint() (in module IPython.lib.pretty), 525
pre_cmd() (IPython.lib.demo.ClearMixin method), 511
pre_execute() (in module IPython.core.events), 420
pre_prompt_hook() (in module IPython.core.hooks), 436
pre_readline() (IPython.terminal.interactiveshell.TerminalInteractiveShell method), 531
pre_run_cell() (in module IPython.core.events), 420
pre_run_code_hook() (in module IPython.core.hooks), 436
precision
magic command, 298
reset() (IPython.core.inputtransformer.CoroutineInputTransformer method), 452
reset() (IPython.core.inputtransformer.InputTransformer method), 440
reset() (IPython.core.inputtransformer.StatelessInputTransformer method), 440
reset() (IPython.core.interactiveshell.InteractiveShell method), 450
reset() (IPython.lib.demo.Demo method), 509
reset_compiler_flags()
  (IPython.core.compilerop.CachingCompiler method), 407
reset_completer() (in module IPython.core.completerlib), 413
reset_selective
  magic command, 305
reset_selective() (IPython.core.interactiveshell.InteractiveShell method), 450
restore_sys_module_state()
  (IPython.core.interactiveshell.InteractiveShell method), 450
result() (IPython.lib.backgroundjobs.BackgroundJobManager method), 502
retina_figure() (in module IPython.core.pylabtools), 479
retrieve_alias() (IPython.core.alias.AliasManager method), 403
RichOutput (class in IPython.display), 495
ruby
  cell magic, 317
run
  magic command, 306
run_ast_nodes() (IPython.core.interactiveshell.InteractiveShell method), 451
run_cell() (IPython.core.interactiveshell.InteractiveShell method), 451
run_cell() (IPython.lib.demo.Demo method), 509
run_cell() (IPython.lib.demo.IPythonDemo method), 510
run_cell_magic() (IPython.core.interactiveshell.InteractiveShell method), 451
run_code() (IPython.core.interactiveshell.InteractiveShell method), 451
run_line_magic() (IPython.core.interactiveshell.InteractiveShell method), 454
runcode() (IPython.core.interactiveshell.InteractiveShell method), 452
safe_execfile() (IPython.core.interactiveshell.InteractiveShell method), 452
safe_execfile_ipy() (IPython.core.interactiveshell.InteractiveShell method), 453
safe_hasattr() (in module IPython.utils.dir2), 547
safe_run_module() (IPython.core.interactiveshell.InteractiveShell method), 453
safe_unicode() (in module IPython.utils.py3compat), 390
save
  magic command, 308
save_sys_module_state()
  (IPython.core.interactiveshell.InteractiveShell method), 453
sc
  magic command, 309
scite() (in module IPython.lib.editorhooks), 512
ScribdDocument (class in IPython.display), 495
sager
  cell magic, 317
search() (IPython.core.history.HistoryAccessor method), 431
seek() (IPython.lib.demo.Demo method), 509
select_figure_formats() (in module IPython.core.pylabtools), 480
Sentinel (class in IPython.utils.sentinel), 563
SeparateUnicode (class in IPython.core.interactiveshell), 442
set_active_scheme()
  (IPython.utils.coloransi.ColorSchemeTable method), 545
setindent() (IPython.core.interactiveshell.InteractiveShell method), 453
set_colors() (IPython.core.debugger.Pdb method), 417
set_colors() (IPython.core.ultratb.TBTools method), 484
set_completer_frame()
set_custom_completer()
  (IPython.core.interactiveshell.InteractiveShell method), 453
set_custom_exc() (IPython.core.interactiveshell.InteractiveShell method), 454
set_custom_exc() (IPython.core.interactiveshell.InteractiveShell method), 454
set_active_scheme() (IPython.utils.coloransi.ColorSchemeTable method), 545
set_custom_exc() (IPython.core.interactiveshell.InteractiveShell method), 454
set_custom_exc() (IPython.core.interactiveshell.InteractiveShell method), 454
test() (in module IPython.testing), 533
Text (class in IPython.lib.pretty), 524
text() (IPython.core.ultratb.TBTools method), 484
text() (IPython.lib.pretty.PrettyPrinter method), 524
text_repr() (in module IPython.core.ultratb), 488
TextDisplayObject (class in IPython.display), 495
time
  magic command, 312
timeit
  magic command, 312
timing() (in module IPython.utils.timing), 575
timings() (in module IPython.utils.timing), 575
timings_out() (in module IPython.utils.timing), 574
TkInputHook (class in IPython.lib.inputhook), 517
tkinter_clipboard_get() (in module IPython.lib.clipboard), 504
toggle_set_term_title() (in module IPython.utils.terminal), 567
token_at_cursor() (in module IPython.utils.tokenutil), 575
TokenInputTransformer (class in IPython.core.inputtransformer), 440
Tracer (class in IPython.core.debugger), 415
transform() (IPython.core.prefilter.PrefilterTransformer
  method), 473
transform_ast() (IPython.core.interactiveshell.InteractiveShell
  method), 455
transform_cell() (IPython.core.inputsplitter.IPythonInputSplitter
  method), 438
transform_line() (IPython.core.prefilter.PrefilterManager
  method), 473
transformers (IPython.core.prefilter.PrefilterManager
  attribute), 473
transforms (IPython.core.inputsplitter.IPythonInputSplitter
  attribute), 438
transforms_in_use (IPython.core.inputsplitter.IPythonInputSplitter
  attribute), 438
trigger() (IPython.core.events.EventManager
  method), 419
try_import() (in module IPython.core.completerlib), 413
TryNext (class in IPython.core.error), 418
tzUTC (class in IPython.utils.tz), 576
U
u_format() (in module IPython.utils.py3compat), 390
unalias

magic command, 313
undoc() (in module IPython.utils.decorators), 547
unescape_glob() (in module IPython.utils.path), 560
unicode_name_matches()
  (IPython.core.completer.IPCompleter
   method), 411
unicode_std_stream() (in module IPython.utils.io), 551
unicode_to_str() (in module IPython.utils.py3compat), 388
unicode_type (in module IPython.utils.py3compat), 389
uniq_stable() (in module IPython.utils.data), 546
unload_ext
  magic command, 314
unload_extension() (IPython.core.extensions.ExtensionManager
  method), 422
unquote_filename() (in module IPython.utils.path), 559
unregister() (IPython.core.events.EventManager
  method), 419
unregister_checker()
unregister_handler()
unregister_transformer()
update_prompt() (IPython.core.prompts.PromptManager
  method), 478
UsageError (class in IPython.core.error), 418
validate() (IPython.core.alias.Alias method), 403
validate_type() (in module IPython.core.magic), 462
var_expand() (IPython.core.interactiveshell.InteractiveShell
  method), 456
VerboseTB (class in IPython.core.ultratb), 485
VimeoVideo (class in IPython.display), 495

V
validate() (IPython.core.alias.Alias method), 403
validate_type() (in module IPython.core.magic), 462
var_expand() (IPython.core.interactiveshell.InteractiveShell
  method), 456
VerboseTB (class in IPython.core.ultratb), 485
VimeoVideo (class in IPython.display), 495
W
warn() (in module IPython.utils.warn), 577
who
    magic command, 314
who_ls
    magic command, 314
whos
    magic command, 315
win32_clipboard_get() (in module IPython.lib.clipboard), 504
with_metaclass() (in module IPython.utils.py3compat), 389
with_patch_inspect() (in module IPython.core.ultratb), 488
wrap() (IPython.core.inputtransformer.InputTransformer class method), 440
wrap_paragraphs() (in module IPython.utils.text), 571
write() (IPython.core.interactiveshell.InteractiveShell method), 456
write() (IPython.utils.io.Tee method), 550
write_err() (IPython.core.interactiveshell.InteractiveShell method), 456
write_payload() (IPython.core.payload.PayloadManager method), 470
writefile
    cell magic, 318
writeout_cache() (IPython.core.history.HistoryAccessor method), 431
writeout_cache() (IPython.core.history.HistoryManager method), 433
WxInputHook (class in IPython.lib.inputhook), 515
X
xdel
    magic command, 315
xmode
    magic command, 315
xrange (in module IPython.utils.py3compat), 387
xsys() (in module IPython.testing.globalipapp), 536
Y
YouTubeVideo (class in IPython.display), 495
Z
ZMQExitAutocall (class in IPython.core.autocall), 405