Reference Pages Documentation

Release 0.2

Eric Snow

January 15, 2017

Contents

This is my stab at replacing my collections of bookmarks. The goal is to create expository content surrounding the links which gives them meaning. As time allows, I'll be migrating all my bookmarks here under that format.

The reference pages are collected topically:

CHAPTER 1

Understanding Python

Contents:

1.1 Python Imports

status Work In Progress

An overview of the history and functionality of Python's import machinery.

This page is an outgrowth of a talk proposal I made for PyCon 2012. Hopefully it's helpful as a reference and adds a little perspective on imports in Python. If you have any suggestions or corrections just let me know (see the project issue tracker).

1.1.1 A High-level Overview of Python's import

Python's import statement and the concept of self-contained namespaces have been a feature of the language since the very beginning. It's one of the simple yet powerful ways that Python enables you to write code you'll still be able to read in 6 months.

Chances are that you take imports for granted: the simplicity of the syntax; the whirring and intermeshing going on behing the scenes. Imports are, to a large measure, what make Python tick. You use them in every piece of Python code you write.

Why does all this matter? It's because understanding the lower layers empowers you to get the most out of the higher ones [1]. With that tool in your belt you can fix your problems more quickly. To top it all off, Python provides a number of import tools that actually make sense once you wrap your brain around the behind-the-scenes stuff.

So, what makes import tick? Glad you asked...

Pulling It Apart

As implied, Python's import machinery is made up of many pieces at many different levels. Why the complexity? Because imports cover a *lot* of territory and are called on to do some pretty hairy stuff. By the time you're done here, you'll understand.

Since it's the point of this document, we'll be looking at each cog and sprocket, with a high-level view coming immediately. The principal piece is the module object, which we'll discuss next. Later sections will go more in-depth, while the appendices go all the way.

Imports work at several levels. Recognizing them is key to putting together the big picture. While we'll look at each later, here's a quick run-down of those layers, from top to bottom:

1. the import statement
2. builtins.__import__
3. PEP 302 finders
4. PEP 302 path importers

(and woven throughout is the interpreter's import state).

What is a Module?

The central piece to imports is the module. A module is the object that the import machinery spits out. While we may call our files "modules" sometimes, the file is not the module. Instead, the module object gets created during import and the file gets *executed* in the module's namespace. It's a subtle difference, but a crucial one. That's because imports are about much more than just files.

Let's look at how modules fit in.

Note: make sure you understand the difference between running a .py file as a script and importing the corresponding module (see *Modules vs. Scripts*).

What Happens During Import?

With that concept of modules in mind, let's step down, layer by layer, through import process.

At the highest level the compiler maps the various forms of the import statement to a handful of instructions for the interpreter. The end result is that one or more module has been imported and one or more names has been bound (in the local namespace).

At the point that the interpreter actually goes to import the module, it calls the builtins.__import__() function, which does the bulk of the work. The wasn't always the case, but thankfully it is now, because we can take advantage of it to customize the behavior of imports.

When you import a module, you're actually importing the chain of modules defined by the dots in the module name. Each of the names is imported, from left to right, with each imported relative to the previous one (the parent).

If one of these modules has not already been imported, then the process described in PEP 302 is used to find it and load it. This is implemented within builtins.__import__().

First it tries using custom import hooks to find the module. Then the system falls back to looking in special internal modules. Finally, it looks across a variety of filesystem paths. Through the API defined by PEP 302, this part is the last (and deepest) opportunity for customization. Realistically, it's also the last chance for a module to be located during import.

If the module is never located, an ImportError gets raised. Otherwise the module gets loaded and the process continues.

Python's Data and Execution Models

Namespaces

...

In the context of imports, there are two important namespaces: modules and packages. We've already talked about modules and how they are the namespace in which your files are executed. A package is simply a module associated with a directory rather than a file.

(packages should never be put directly on sys.path (even by the sys.path[0] behavior of the __main__ module))

See the “modules” section of the data model documentation.

Execution Blocks

See the execution model documentation.

Scope

See the execution model documentation.

Modules vs. Scripts

The Import Syntax

http://docs.python.org/dev/reference/simple_stmts.html

For more detail see the appendix.

The import statement

- usage
- effect

The as clause

- usage
- effect
- benefits

The from statement

- usage
- effect
- from ... import *
- dangers

Relative Imports

http://docs.python.org/dev/tutorial/modules.html#intra-package-references

PEP 328

- usage
- effect

Parentheses

- usage
- effect

Other Semantics

- implicit relative imports
- files (__init__.py, .py, .pyc, .pyo, etc.)
- builtins.__import__()

Note: As noted at the beginning of this section, the appendix provides a more thorough under-the-hood look at the import syntax.

The Import State

http://docs.python.org/library/sys.html

sys.modules

sys.path

- .pth files

(Also see Appendix B) (Also see I.1 for more on why '', a.k.a. CWD, is added to sys.path)

sys.meta_path

sys.path_hooks

sys.path_importer_cache

site-packages

http://docs.python.org/dev/library/site.html

site-packages user site-packages

(Also see Appendix B)

The site Module

http://docs.python.org/dev/library/site.html

site.py sitecustomize.py usercustomize.py

(Also see Appendix B)

Customizing Import Behavior

We'll take a look at each layer.

The Import Syntax

Overriding builtins.__import__()

Using PEP 302 Finders (and Loaders)

Manipulating the Path Importer Cache

Using PEP 302 Path Importers

Directly Modifying the Import State

Consenting Adults

.pth Files

Other Resources

Appendices

- annotated step-by-step through the CPython source for the import process

  orphan

Appendix: Import Syntax Under the Hood

Appendix: Import-related Files

orphan

sys.path related

http://docs.python.org/library/sys.html#sys.path http://docs.python.org/c-api/init.html#Py_GetPath
http://hg.python.org/cpython/file/default/Modules/getpath.c#l121

http://docs.python.org/dev/library/site.html

1. calculate the 4 site-packages dirs
2. add them to sys.path
3. check for and execute .pth files in those site-packages dirs
4. calculate user site-packages

site.getsitepackages()

Unix/Mac:

<sys.prefix>/lib/python#.#/site-packages <sys.exec_prefix>/lib/python#.#/site-packages <sys.prefix>/lib/site-python <sys.exec_prefix>/lib/site-python

Windows:

<sys.prefix>/ <sys.exec_prefix>/ <sys.prefix>/lib/site-packages <sys.exec_prefix>/lib/site-packages

site related

http://docs.python.org/dev/library/site.html

- module: site
- module: sitecustomize
- module: usercustomize

site.getusersitepackages()

Standard Library

Lib/importlib/*.py Lib/pkgutil.py

Tests

Lib/Test/...

CPython

Python/import.c Python/importdl.c Python/importdl.h Include/import.h Python/sysmodule.c Python/pythonrun.c

PyPy

Jython

IronPython

Appendix: An Extended Timeline of Importing in Python

orphan

Some Context

(origins) http://www.python.org/community/sigs/retired/import-sig/ http://www.python.org/dev/peps/pep-3121/#id11 (1.5) http://www.python.org/doc/essays/packages.html #http://www.python.org/doc/essays/packages/ Modula-3 influence: http://python-history.blogspot.com/2009/02/adding-support-for-user-defined-classes.html http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html http://python-history.blogspot.com/2009/03/dynamically-loaded-modules.html http://docs.python.org/dev/whatsnew/index.html http://python.org/download/releases/src/ http://hg.python.org/cpython-fullhistory/tags

http://hg.python.org/cpython-fullhistory/graph/3cd033e6b530?revcount=800 http://hg.python.org/cpython-fullhistory/log/62bdb1cbe0f5/Python/import.c?revcount=120 initial: http://hg.python.org/cpython-fullhistory/file/fc6fcd7df4f7/Python/import.c 0.9.8: http://hg.python.org/cpython-fullhistory/file/17eff686be30/Python/import.c builtin___import__(), importdl.c: http://hg.python.org/cpython-fullhistory/rev/d7e91437f0a2 PyImport_Import: http://hg.python.org/cpython-fullhistory/rev/292193170da1 highlights of "What's New": http://nedbatchelder.com/blog/201109/whats_in_which_python.html code_swarm: http://vimeo.com/1093745

(ni) introduced (1.3): http://hg.python.org/cpython-fullhistory/rev/ec0b42889243 deprecated (1.5): http://docs.python.org/release/1.5/lib/node40.html still lives: http://docs.python.org/library/imputil.html#examples

(ihooks) introduced (1.3): http://hg.python.org/cpython-fullhistory/rev/ec0b42889243 removed (3.0): http://docs.python.org/release/2.6.2/library/undoc.html#miscellaneous-useful-utilities http://pydoc.org/2.4.1/ihooks.html

—

The versions and dates are derived from a post on Guido's "History of Python" blog. I've correlated the entries in section B.1 to versions by either explicit reference or by matching their commits to a version. Section B.2 also maps commits to versions. In both cases, I did my best to determine that mapping, but some may be off by a version.

The Extended Timeline

Initial Checkin (1990)

- Checks sys.modules
- Loads modules from sys.path or current dir (if sys.path is empty)
- Supports IMPORT_NAME and IMPORT_FROM opcodes
- No support for .pyc files
- No support for packages
- No support for C extension modules?
- No ImportError

Python 0.9.1 (Feb. 1991)

- builtin module support (C extention modules)

Python 1.0 (1994)

- Support for extension modules
- Support for .pyc files

Python 1.2 (1995)

- (Python/bltinmodule.c) __import__() builtin introduced
- (Python/import.c) dynamic module support factored out into importdl.c

Python 1.3 (1995)

- "ni" module introduced

Python 1.4 (1996) http://docs.python.org/release/1.4/ref/ *

Python 1.5 (1998)

- Support for packages
- "site-packages" and "site-python" directories introduced

- "__all__" introduced
- "ni" module deprecated
- (Python/import.c) PyImport_Import() introduced

Python 2.0 (2000)

- PEP 221 – Import As

Python 2.1 (2001)

- PEP 235 – Import on Case-Insensitive Platforms

Python 2.2 (2001)

-

Python 2.3 (2003)

- PEP 273 – Import Modules from Zip Archives
- PEP 302 – New Import Hooks

Python 2.4 (2004)

- PEP 328 – Imports: Multi-Line and Absolute/Relative (multi-line portion)

Python 2.5 (2006)

- PEP 328 (relative imports portion)
- PEP 338 – Executing modules as scripts

Python 2.6/3.0 (2008)

- PEP 366 – Main module explicit relative imports
- PEP 370 – Per user site-packages directory

Python 3.0 (2008)

- reload removed from builtins
- ihooks module removed from stdlib
- imputil module removed from stdlib

Python 3.1 (2009)

- importlib module added

Python 3.2 (2011)

- PEP 3147 – PYC Repository Directories

Python 3.3 (2012)

- see appendix D

Appendix: A Timeline of Import-related Commits

orphan

http://hg.python.org/cpython-fullhistory/log/62bdb1cbe0f5/Python/import.c?revcount=120

Python 0.9.0 (1991)

-

Python 0.9.1 (1991)

-

Python 0.9.2 (1991)

-

Python 0.9.4 (1991)

-

Python 0.9.5 (1992)

-

Python 0.9.6 (1992)

-

Python 0.9.7 (1992)

-

Python 0.9.8 (1993)

-

Python 0.9.9 (1993)

-

Python 1.0.0 (1994)

-

Python 1.0.2 (1994)

-

Python 1.0.3 (1994)

-

Python 1.0.4 (1994)

-

Python 1.1 (1994)

-

Python 1.1.1 (1994)

-

Python 1.2 (1995)

-

Python 1.3 (1995)

-

Python 1.4 (1996)

-

Python 1.5 (1998)

-

Python 1.5.1 (1998)

-

Python 1.5.2 (1999)

-

Python 1.6 (2000)

-

Python 2.0 (2000)

-

Python 2.1 (2001)

-

Python 2.2 (2001)

-

Python 2.3 (2003)

-

Python 2.4 (2004)

-

Python 2.5 (2006)

-

Python 2.6 (2008)

-

Python 3.0 (2008)

-

Python 2.7 (2010)

-

Python 3.1 (2010)

-

Python 3.2 (2011)

-

Python 3.3 (2012)

-

Appendix: Ongoing Core Efforts to Improve Importing

orphan

PEPs

- **PEP 369** – Post import hooks
- **PEP 382** – Namespace Packages
- **PEP 395** – Module Aliasing
- **PEP 402** – Simplified Package Layout and Partitioning
- PEP ??? – import engine

Rejected PEPs:

- **PEP 299** – Special __main__() function in modules
- **PEP 3122** – Delineation of the main module

Projects

- importlib.__import__ as the default builtins.__import__

Currently in Python, "builtin___import__()" in Python/bltinmodule.c makes a call to PyImport_ImportModuleLevelObject. Brett Cannon is working on making importlib.__import__ the default import call[1].

- the __experimental__ module

http://mail.python.org/pipermail/python-ideas/2010-June/007357.html http://mail.python.org/pipermail/python-ideas/2011-August/011278.html

like the __future__ module, but for less-stable APIs that are likely to go in focus on stdlib (room for experimental syntax too?) (higher exposure testing)

Appendix: Imports in Alternate Python Implementations

PyPy

http://readthedocs.org/search/project/?q=import&selected_facets=project%3Apypy http://codespeak.net/pypy/dist/pypy/doc/clr-module.html http://codespeak.net/pypy/dist/pypy/doc/coding-guide.html https://bugs.pypy.org/issue367

Jython

http://www.google.com/search?sitesearch=www.jython.org&q=import&Search=Search

IronPython

http://ironpython.codeplex.com/wiki/search?tab=Home&SearchText=import

Appendix: Easter Eggs

The Python devs are a playful lot.

import this

http://www.wefearchange.org/2010/06/import-this-and-zen-of-python.html

import antigravity

http://xkcd.com/353/ http://python-history.blogspot.com/2010/06/import-antigravity.html

from __future__ import flufl

http://www.python.org/dev/peps/pep-0401/ http://sayspy.blogspot.com/2009/03/guido-has-retired-as-bdfl.html

from __future__ import braces

import __hello__

Appendix: Import Examples

orphan

How It Works

- Example: Plain Syntax Handler
- Example: From Name Syntax Handler
- Example: From Star Syntax Handler
- Example: builtins.__import__

Uncategorized

- Example: Naively Track Imports
- Example: Import Tracking, Take 2
- Example: Statement Local Namespaces
- Example: Protecting a High-Latency Filesystem
- Example: Customizing Access to a Specific Module Path
- Example: PEPS 382 and 402 as Import Hooks
- Example: Import Engine as an Import Hook
- Example: lazy imports
- Example: “importing” straight from a file

Appendix: Imports in the Python Community

Appendix: Troubleshooting Imports

orphan

Causes of ImportError

- turn into ImportError subclasses, __cause__

Other Exceptions During Import

- SyntaxError
- IOError?

Common Import-related Problems

- don't run non-scripts as scripts; import in a test script
- minimize the amount of code in scripts
- relative imports behave a little differently in scripts
- .pyc for <name> is created only for "import <name>"
- caching was turned off
- file is actually in __pycache__ directory
- python run with -O flag (optimized) so .pyc files created

By default the current working directory is first on sys.path. If this is causing trouble, you can move it to the back of the line:

```
try: sys.path.remove('')
except ValueError: pass
else: sys.path.append('')
```

Alternatively, you could remove it entirely (don't append it back on).

http://bugs.python.org/issue8087

If a directory has a pyc file but no matching py file, the module will be loaded from the pyc file directly. Starting with **PEP 3147** (Python 3.2), orphaned pyc files in the __pycache__ directory are NOT loaded. The behavior use of pyc files otherwise stays the same. Either way, if you don't want the module to be loaded from an orphaned pyc file, delete that file. Also see *this tracker ticket <orphaned_pyc_files>*.

Sometimes it can be helpful to replace sys.modules with a custom dictionary. However, in CPython, this does not affect the underlying dictionary that was originally bound to sys.modules. That is a separate part of the interpreter state. The behavior of the default _builtin___import__(), implemented in Python/import.c, actually uses this underlying dictionary through PyImport_GetModuleDict(), rather than explicitly pulling sys.modules. So your fancy-pantsy sys.modules is never used.

Luckily, importlib does explicitly use sys.modules, so if you switch over to that it should work just fine. This will be an even smaller issue once importlib's __import__ because the default builtin.

(see http://bugs.python.org/issue12633).

If you have a module file in your sys.path, and you try to import it, sometimes the import will succeed but the module will be the wrong one. This can be both mysterious and perplexing.

The first thing to do is to see if you have a package (directory with a __init__.py) by the same name in the same place as that module file. If so, Python will import from the package instead of the module. To verify this, import the module: "import <name>" and then check the module in sys.modules: "import sys; print(sys.modules['<name>'])". You should see it pointing to the __init__.py of the package instead of the module file you were expecting.

Appendix: Import Tips and Tricks

orphan

block imports on the current working directory

By default Python will look for a module in your current working directory before trying the stdlib. The explicit relative import syntax of 2.7 help with this, but only to an extent.

To completely keep Python from trying the CWD, simply run "sys.path.remove('')" and optionally follow that with "sys.path.append('')".

So the question remains, when did the empty string get added to (the front of) sys.path, and why?

...

Appendix: Other Import-related Resources

Online References

orphan

Dr. Brett Cannon gave a talk at *PyCon 2010 <http://python.mirocommunity.org/video/1491>* and PyCon 2008 (can't find video).

importlib extensions: http://packages.python.org/importers/

flowchart: http://svn.python.org/view/sandbox/trunk/import_in_py/docs/

http://mail.python.org/mailman/listinfo/import-sig http://docs.python.org/dev/reference/simple_stmts.html#the-import-statement http://docs.python.org/dev/reference/simple_stmts.html#future-statements http://docs.python.org/dev/reference/toplevel_components.html http://docs.python.org/dev/reference/executionmodel.html#naming-and-binding http://docs.python.org/dev/reference/datamodel.html#the-standard-type-hierarchy (modules)

http://docs.python.org/dev/tutorial/modules.html http://docs.python.org/release/1.5.1p1/tut/modules.html http://docs.python.org/release/1.4/tut/node41.html#SECTION00700000000000000000 http://docs.python.org/dev/library/modules.html http://docs.python.org/dev/faq/programming.html#what-are-the-best-practices-for-using-import-in-a-module http://docs.python.org/dev/faq/programming.html#how-can-i-have-modules-that-mutually-import-each-other http://docs.python.org/dev/faq/programming.html#import-x-y-z-returns-module-x-how-do-i-get-z http://docs.python.org/dev/faq/programming.html#when-i-edit-an-imported-module-and-reimport-it-the-changes-don-t-show-up-why-does-this-happen

http://docs.python.org/dev/library/py_compile.html http://docs.python.org/dev/library/compileall.html

http://www.doughellmann.com/PyMOTW/sys/imports.html

http://lucumr.pocoo.org/2011/9/21/python-import-blackbox/

http://code.google.com/p/backport/ http://mirnazim.org/writings/python-ecosystem-introduction/ http://nedbatchelder.com/blog/201112/duplicitous_django_settings.html

http://lucumr.pocoo.org/2011/12/7/thoughts-on-python3/ http://pythonic.pocoo.org/2009/3/4/imports-in-functions-you-sure-about-that

Open bugs: http://bugs.python.org/issue?%40search_text=import&ignore=file%3Acontent&title=&%40columns=title&id=&%40colum queryname=&%40action=search

Not closed: http://bugs.python.org/issue?%40search_text=&ignore=file%3Acontent&title=&%40columns=title&id=&%40columns=id&
1%2C1%2C3&%40columns=status&%40group=status&resolution=&nosy_count=&message_count=&%40pagesize=50&%40startwit
queryname=&%40action=search

Closed: http://bugs.python.org/issue?%40search_text=&ignore=file%3Acontent&title=&%40columns=title&id=&%40columns=id&sta
queryname=&%40action=search

<stack overflow>

http://stackoverflow.com/questions/279237/python-import-a-module-from-a-folder

<cookbook recipes>

<others>

https://github.com/zacharyvoase/metaspace

http://www.youtube.com/watch?v=DkW5CSZ_VII http://aroberge.blogspot.com/2006/02/python-wish-new-meaning-for-import-as.html

Import Who's Who

orphan

People who have been involved with Python's imports (incomplete):

"experts": Brett Cannon, Nick Coghlan

Brett Cannon (importlib) Just van Rossum (**PEP 302**) Paul Moore (**PEP 302**) Aahz (**PEP 328**) Martin v. Loewis (**PEP 382**) P.J. Eby (**PEP 402**) James C. Ahlstrom (**PEP 273**) Nick Coghlan (**PEP 338**, **PEP 366**, **PEP 395**) Christian Heimes (**PEP 370**) Thomas Wouters (**PEP 221**) Barry Warsaw (**PEP 3147**) Tim Peters (**PEP 225**) Guido van Rossum (pretty much everything else <wink>)

Glossary

orphan

The terminology surrounding imports can get confusing. This glossary should help.

import hook ...

finder An object with a *find()* method that conforms to **PEP 302**. May also refer to the class of such an object.

loader An object with a *load()* method that conforms to **PEP 302**. May also refer to the class of such an object.

importer Mostly synonymous with *path importer*.

path importer An object, class, or other code that may be plugged into the **PEP 302** import machinery. Often this term refers specifically to those that are used with sys.path_hooks.

module The object generated at the highest level of the import process. In the normal import statement, it is the object bound to the name.

module name The value bound to the *__name__* attribute of the corresponding module object. This will be the full qualified name relative to the sys.path value at import time.

package A module corresponding to a directory. The module is populated with the results of evaluating the __init__.py file in the directory. Other .py files and directories in the directory may be imported as submodules of the package.

namespace package A package, possibly without its own module execution, into which subpackages are aggregated according to a single namespace. The "zope" package is a good example.

package portion ...

- genindex
- search
- genindex
- search

The import Statement

The import statement is the syntactic mechanism you use to invoke Python's powerful import machinery. It has two forms: the regular import and from-import. In a moment we'll walk through the ins and outs of both forms.

When you use the import statement in either form, you identify a *module* and its *parent modules* together as a *module name*. By default, each successive parent module is imported from the outside in, followed by the actual module you wanted. After that the appropriate name is bound in the current local namespace. Most imports are going to happen at the module level where the name will be bound in that module's [global] namespace.

A module object is the result of importing. We use the term "module" to refer to this object as well as to the thing that Python used to create the object, usually a file. A package is a special kind of module. Where a normal module corresponds to a file, a package corresponds to a directory.

Changing the Import Behavior

You can override the full import machinery by overriding builtins.__import__().

Prior to Python 2.3 the only way to override the import behavior was by replacing builtins.__import__() with some other function that did what you wanted. This changed with PEP **PEP 302**.

Now you can add special "loader" objects to a couple of different places in the sys module to take control of imports in more targeted ways. A loader translates a module name into a "finder" object, if it can. The finder, in turn, converts the module name into the corresponding module object, which it sticks into sys.modules.

This entire process is explained much more in-depth in a later section and in the appendix.

Import State

All the Python variables related to the default import behavior is stored in the sys module. This includes sys.path, sys.modules, sys.meta_path, and sys.path_hooks.

- http://docs.python.org/library/sys.html
- http://www.doughellmann.com/PyMOTW/sys/imports.html

ImportError

What It Means

When It Happens and When Not

PEP 302

With the release of Python 3.2, a powerful way of customizing imports became available with **PEP 302**.

Finders

Loaders

The Default Import Process

See the appendix.

Implicit Finders

Builtin Modules

Frozen Modules

Zipped Modules

The Python Path Finder

Implementations

The imp Module

- iterative
- Python/import.c

PyImport_GetModuleDict() used to get sys.modules (see J.3.9).

- Python/bltinmodule.c - _builtin___import__()

This is the default handler for the import statement. In 2.7 it is *__builtin__.__import__()*.

The importlib Module

- recursive
- Lib/importlib/

sys.modules used to get sys.modules.

- importlib.__import__()
- importlib.import_module()

Import-related Modules

http://docs.python.org/dev/library/modules.html

- pkgutil
- runpy
- modulefinder and zipimport

.pth Files

See the site module documentation.

More in Appendix B.

A History of Python's import Statement

A Brief History of Python

The Origins of Python's import

http://python-history.blogspot.com/2009/02/adding-support-for-user-defined-classes.html
http://www.python.org/doc/essays/foreword/ http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html

The import statement has been a part of Python since the very beginning, though with more limited behavior.

Like many things in Python, the syntax for the import statement has its roots in Modula-3.

Early Changes

- builtins.__import__()
- ni.py
- ihooks.py

The Intervening Years

- **PEP 302**

Recent Changes

-

1.2 Interfaces in Python

A work-in-progress reference all about interfaces in Python.

Feel free to give me feedback on the project page

My PyCon talk on interfaces: *slides <https://docs.google.com/present/...> on the PyCon site <https://us.pycon.org/2012/schedule/presentation/126/>*

`understanding_software/interfaces`

1.2.1 Abstract API Model Approaches

Python

Informally specified protocols, facilitated by duck-typing, have been the mainstay of Python since the beginning. Since 2.6/3.0 abstract base classes have been available as a formal means of specifying interfaces. Other proposals have come and gone.

Python also makes it pretty easy to build-your-own interface system, as evidenced by the variety of solutions out there.

1.2.2 Data Types

Python

Python is a strongly-typed, dynamically-typed, interpreted language. Let's take a look at the data model a bit more.

Class-based Approaches

1.2.3 Protocols

Python

This is the bread and butter of Python's interfaces.

1.2.4 Abstract Base Classes

Python

Abstract base classes have been a part of Python since the Py3k efforts led to **PEP 3119** in 2007.

Adaptation

http://www.python.org/dev/peps/pep-0246/
http://pythonnotes.blogspot.com/2004/11/what-is-adaptation.html

Traits

http://en.wikipedia.org/wiki/Trait_(computer_programming)
http://scg.unibe.ch/research/traits/
http://pypi.python.org/pypi/strait/0.5.1
http://code.enthought.com/projects/traits/

Interfaces in Python

http://dirtsimple.org/2004/12/python-interfaces-are-not-java.html
http://nedbatchelder.com/text/interfaces.html
http://nedbatchelder.com/text/pythonic-interfaces.html
http://stackoverflow.com/questions/372042/difference-between-abstract-class-and-interface-in-python
http://www.rexx.com/~dkuhlman/python_comments.html#interfaces
http://pydanny.blogspot.com/2008/08/thoughts-on-python-interfaces.html

http://www.eecho.info/Echo/python/interfaces-python/

http://www.google.com/search?hl=en&rlz=1C1SNNT_enUS421US421&biw=1920&bih=989&q=+site:mail.python.org+python+interf

http://mail.python.org/pipermail/tutor/2006-June/047508.html
http://mail.python.org/pipermail/tutor/2006-June/047648.html
http://www.velocityreviews.com/forums/t570039-python-interfaces.html
(http://mail.python.org/pipermail/python-list/2008-January/523116.html)

History

- the Great Adaptation Debate of 2005

Third-party Libraries

- Zope
- twisted
- peak

http://peak.telecommunity.com/PyProtocols.html

http://twistedmatrix.com/documents/11.0.0/api/twisted.python.components.html
http://twistedmatrix.com/documents/current/core/upgrades/2.0/components.html

http://wiki.zope.org/Interfaces/FrontPage
http://apidoc.zope.org/++apidoc++/
http://docs.zope.org/zope2/zdgbook/ComponentsAndInterfaces.html
http://pypi.python.org/pypi/zope.interface

http://twistedmatrix.com/documents/current/core/howto/components.html

Python's Data Model

http://docs.python.org/dev/reference/datamodel.html:

```
Objects are Python.s abstraction for data. All data in a Python
program is represented by objects or by relations between objects.
```

So, everything is an object in Python, including modules, classes, and literals. Every object is an instance of the base object type or of a subclass thereof:

```
class X:
    pass

isinstance(object(), object) == True
isinstance(object, object) == True
isinstance("abc", object) == True
isinstance(1, object) == True
isinstance(X, object) == True
isinstance(X(), object) == True
isinstance(type, object) == True
```

Every object has a type and every type is an instance of the base type:

```
class MetaY(type):
    pass
class Y(metaclass=MetaY)

type(object()) == object
type(object) == type
isinstance(object, type) == True

type("abc") == str
isinstance(str, type) == True

type(1) == int
isinstance(int, type) == True

type(X()) == X
type(X) == type
isinstance(X, type) == True

type(Y) == MetaY
isinstance(Y, type) == True

type(type) == type
isinstance(type, type) == True
```

Be sure to notice the special-cased nature of the base object and base type:

```
type(type) == type
isinstance(type, object) == True

type(object) == type
isinstance(object, object) == True
```

1.2.5 Python's Dynamic Typing

Names don't have type declarations, like they do in statically-typed languages. You could also look at it like all names have the same implicit type declaration: object. Either way, any object can be bound to any valid name (including as a function argument).

Objects are bound to names. Names are *not* bound to objects. As a consequence, objects do not "know" the names to which they are bound.

Duck-typing

"Polymorphism without inheritance"

http://en.wikipedia.org/wiki/Duck_typing

Duck-typing is polymorphism by capability, as opposed to polymorphism by type.

- *"signature-based" polymorphism <http://zephyrfalcon.org/labs/beginners_mistakes.html>*
- *Requiring a specific interface instead of a specific type. <>*
- *Determining an object's type by inspection of its method / attribute signature rather than by explicit relationship to some type object. <>*
- *Even without formal interface declarations, good practice mostly depends on conformant interfaces rather than subclassing to determine an object's type. <>*

Python has always been about what an object can do, rather than its type. This has changed somewhat with the advent of abstract base classes (see **PEP 3119**), where isinstance checks lessen the performance hit of LBYL (see below).

http://dobesland.wordpress.com/2007/10/07/python-isinstance-considered-useful/
http://stackoverflow.com/questions/1549801/differences-between-isinstance-and-type-in-python
http://www.shindich.com/sources/patterns/implied.html
http://www.canonical.org/~kragen/isinstance/
http://www.voidspace.org.uk/python/articles/duck_typing.shtml
http://www.themacaque.com/?p=155

1.2.6 Key Concept: LBYL vs. EAFP

LBYL: look before you leap EAFP: easier to ask forgiveness than get permission

EAFP is more pythonic.

http://docs.python.org/glossary.html#term-eafp
http://mail.python.org/pipermail/python-list/2003-May/203039.html
http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#eafp-vs-lbyl
http://sayspy.blogspot.com/2008/12/why-explicit-type-checking-is-mostly.html
http://mail.python.org/pipermail/python-3000/2006-May/001909.html
http://stackoverflow.com/questions/5589532/try-catch-or-validation-for-speed
http://mail.python.org/pipermail/python-ideas/2012-January/013510.html

1.2.7 Examples

Duck-typing is all about the attributes an object has and what that object can do. For example:

```
# LBYL
if hasattr(obj, "quack"):
    obj.quack()

#EAFP
try:
    quack = obj.quack
except Exception:
    ...
quack()
```

```
#EAFP - just try it
obj.quack()
```

This is not duck-typing (though perfectly valid):

```
# LBYL
if isinstance(obj, Duck):
    obj.quack()

# LBYL
if implements(obj, Duck):
    obj.quack()
```

1.2.8 Python's Protocols

While duck-typing is an integral part of writing Python, the application of it in the language itself is a key part of understanding how Python works under the hood.

...

Abstract Base Classes in Python

http://docs.python.org/dev/library/abc.html http://stackoverflow.com/questions/3570796/why-use-abstract-base-classes-in-python http://www.doughellmann.com/PyMOTW/abc/ http://mail.python.org/pipermail/python-ideas/2011-October/012075.html http://sayspy.blogspot.com/2009/12/how-to-handle-multiple-inheritance-of.html http://www.python.org/dev/peps/pep-3119/#abcs-vs-alternatives http://docs.python.org/whatsnew/2.6.html#pep-3119-abstract-base-classes

Duck-typing is focused on protocols. ABC/Interface is focused on formal interfaces.

ABC/Interface vs. duck-typing – means LBYL vs. EAFP?

- Test for IX w/o using it. Tests for all of it vs.
- Test for IX by using a part of it, when you need it. Only use what you need.

http://stackoverflow.com/questions/5589532/try-catch-or-validation-for-speed/5591737#5591737

1.2.9 collections.abc

1.2.10 Writing Your Own ABC

1.2.11 Controversies

---

history

existing_protocols

existing_abstract_base_classes

libs

/understanding_software/interfaces/languages

/understanding_software/interfaces/use_cases

`/understanding_software/concurrency`

- genindex
- search

1.3 Python's Type System

status Work In Progress

An overview of the history and functionality of Python's type system.

This page is an outgrowth of a work I've been doing for adding a C version of collections.OrderedDict to CPython. It also relates to the SimpleNamespace type I added for sys.implementation.

1.3.1 A High-level Overview of Python's Type System

<TBD>

Other Resources

`PyTypeObject`

<TBD>

- genindex
- search

Indices and tables

- genindex
- modindex
- search

CHAPTER 2

Understanding Software

This is a big part of my life, so it covers a lot of my attention...

Contents:

2.1 Interfaces

An interface is just an abstraction of expected interactions with some bit of code, sometimes formalized syntactically. Class/function/module APIs are the general case. Before anything else, let's look at how you might use interfaces:

`use_cases`

2.1.1 What are Interfaces?

An interface is simply a description of how a programmer should expect to be able to interact with block of code. Generally this focuses on functions. However, in an object-oriented context, object state may also be considered a part of an interface.

As an alternatve, interfaces could also be called "Abstract API Models". All programming languages have at least one means of modeling APIs.

The most common form of interface aims at describing how functions should be used. This includes type declarations for function parameters and return values, a staple of many languages. However, such declarations only capture one aspect of a function's interface. In most languages, regardless of type system, each function is essentially an API wrapping a block of code.

Functions don't have the interface market cornered, either. Most languages have a means of encapsulating program state, usually as part of an object-oriented type system. When the topic of interfaces comes up, it tends to refer to formal definitions of what functions and attributes an object (and abstractly, it's type) should provide.

However, in all cases, interfaces are still just specifications of how one should interact with a block of code.

More references:

http://en.wikipedia.org/wiki/Interface_(computing)
http://en.wikipedia.org/wiki/Interface_(object-oriented_programming)

Abstract API Model Approaches

To the extent that interfaces are a means of definining expected interactions, they can be approached in a variety of ways. This bears out across the many programming languages in the wild today.

Below is a list of several ways interfaces may be done. Each of these is treated more thoroughly in following sections.

- Classes
- Interfaces/Protocols
- Abstract Base Classes
- Adaptation
- Multiple-dispatch/Generics
- Roles
- Traits
- Design by Contract

When are Interfaces Appropriate?

class vs. interface

ABC vs. interface

mixins vs. multiple inheritance

inheritance vs. delegation

inheritance vs. composition

2.1.2 Appendices

adaptation transforming wrappers

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Advice For PyCon Speakers

This page is a consequence of my inexperience at speaking at technical conferences (read: none). It's a combination of content provided by others, small bit of editorial discretion, and various resources from ye ol' internet. Finally, it's a work-in-progress and I will gladly take feedback. Ultimately, my intention is to get a form of this page up on the official PyCon site, if people find it helpful.

Mainly, a bunch of folks in the Python community were very generous with their time and shared some great advice. Most of the content on this page is just a compilation of what they had to say. The value here is that their perspective is collectively quite appropriate to the Python community. You might go as far as to call it pythonic. :)

The whole point it to focus on the practical advice of people that know what speaking at PyCon is all about. PyCon is it's own creature and who would understand it better?

Some of the contributers provided links to things they have written on their blogs. These, along with other more general public speaking links, can be found below in the *Resources* section.

May this be as helpful to you as it has been to me!

3.1 Talk Proposals

It's a little late for this one...

3.2 Preparing

To start off, here's a great bit of general advice from one of the contributers, inspired by Dale Carnegie:

> Talk about something you know well; speak passionately about it; speak to the audience as you would your friends (they want to hear what you have to say).

3.2.1 Know Your Stuff

- Make sure you're genuinely interested in your topic and just have fun with it.
- Know your material. Don't give a talk on something you only vaguely know about. Don't think that the few months you have between submission time and the talk will be enough for you to learn about the topic.

3.2.2 Building Your Talk

- One approach: structure the talk as you would a story, where each section leads naturally into the next, building in complexity.
- Another approach: write out every word you plan to say, and time yourself saying. Then turn it into an outline and throw away the text before you actually speak. This way you'll have the precise words somewhere in your head, but you won't be reading from a paper. Your outline can help guide you.
- An overarching mindset: "Entertain, Educate, Practice". (See this blog post.) Remember, play to your strengths.
- Treat your "PyCon presentation [as] be a trailer for your expertise." (See this blog post.)
- Often you should try to avoid introducing concepts if you will have to say things like "and I will explain that more in a few slides." Making "forward references" like that breaks the linear flow and makes it more difficult for the audience to concentrate on what you are saying now, since they start worrying about what you will be saying in a minute. It isn't always possible to be completely linear, but consider starting with that as a goal.
- A lot of "intro to X" talks start by doing a live demo and installing the tool. Seriously avoid that. No one cares about how easy it is to install a tool until they have seen why they care about using it, at which point they can look at the instructions on the project web site. Instead, jump in and get right to something interesting to grab the audience's attention, or they are going to go back to checking email.
- "Tell em what you are going to tell em, tell em, and then tell em that you told em."
- Keep the talks short and focused. Explain to the audience why they should also be passionate about the subject. GET THEM HUNGRY.
- If you are funny, use it. Good, geeky tech jokes == good.
- Gender references and sexual/racy refs are right out. Don't. Just don't.
- If in doubt, err on the side of not talking long enough. It's better to have the audience thinking "That talk left me wanting more. I need to go talk to the presenter/download the package/go to the BoF," than "That talk stretched 15 minutes of material into an hour. What a waste of time."
- Don't waste time on introductory material, e.g. explaining Python's syntax, explaining XML for fifteen slides.
- Have your important research done before you start trying to prepare the talk. If you run out of preparation time, don't skimp on rehearsal; instead, cut scope from the talk, or make do with a simple but readable visual design.
- Don't think of talk length as an indicator of value. There is a reason why the most popular sessions of the entire conference are the lightning talks. Less is More.

3.2.3 All About Slides

- The slides are primarily to support your talk.
- Slides should not be too "busy". Keep them short, effectively as "reminders what to say".
- If a bullet point gets up to 15 words, consider breaking it up.
- Group related points.
- Only hit the most important points on the slides. Expand as you speak if there's audience interest.
- Presentations look best when the slide size is the same as the projector's native resolution. For the conference this year, that is ????x???.
- The most important thing about slides is that the audience needs to be able to read them. That seems obvious, but all too often the slides are hard to read.

- Many people have trouble reading light text on a dark background. It may work for you on your laptop screen, but projected in front of the audience in a dark room is a completely different story. Use a white or other light background color with high contrast dark text in a large font.
- Background colors that look great on a laptop or monitor screen often lose something in the transition to a projector. You can't predict what the venue will give you in regards to quality/brand of projector, so why take unnecessary risks?
- Strongly consider using the default font of the slide software. Maybe its not fancy or artistic, but your message won't be obfuscated by forcing people to squint to see slides reinforcing what you're saying.
- Use more slides with less code per slide in order to increase your font size. Wrapping lines to make them less than 80 columns helps with the size, too.
- If all of the slides show code and output, there probably isn't enough visual reinforcement of whatever framing story you are using to tie everything together. Use pictures to reinforce concepts, without simply throwing keywords up on a bullet list. Use diagrams to explain the architecture of the thing you are describing.
- Try to finish your slides way before the conference. It's tempting to put them off, but the more you go through them, the more secure you'll be with your timing and your content.
- One way to make the slides and the talk work together is to ramble through your talk a few times, recording it, then organize your slides off of that.
- Aim for big text, clear images, good contrast. Stand about five feet from your laptop screen – can you see the text from that distance?
- PyCon does not have a published volume of proceedings, but the slides and other materials for talks are often made available on-line. Therefore, be sure your presentation can be turned into a format suitable for online viewing. While PDF is permitted, HTML is better. Keep graphics reasonably sized for web access.
- Hopefully your slides are finished up in advance of the conference. Consider uploading the presentation to the conference talk proposal system or to a page linked from your talk's page on the PyCon site. This gives the audience more information in choosing which talks to attend, and people can refer to the slides if they miss something during your talk.
- Don't try to squeeze more than 10 lines of code onto the slide; if the font gets too small, the code will just be a meaningless set of squiggles to people in the back of the auditorium.
- If possible, view your slides on a projector and see if they're readable. Are the font sizes large enough? Is there enough contrast between the text and the background?
- Plan on spending absolutely no more than 60 seconds on any slide.
- Conversely, only a few seconds for a slide may be too little.
- Above all, try to be consistent about how long you spend on each slide. The audience will respond well to consistency.

3.2.4 Demos

- As noted above, don't do a demo of how to install a tool.
- Be hesitant to rely on live demos. Fumbling around on stage changing between a code editor and a terminal where the code is running takes time that could be spent telling the audience something else interesting. They believe you can type and they believe you can run programs. Just show them the meaty bits.
- Phrased another way, don't do live demos.

3.2.5 Practice, Practice, Practice

- (Try to find all the references to practice that you've already read.)
- If it is you first time around, it may be worth going to a local interest group or somesuch to practice your talk in front of a small audience.
- Everyone you'll see at Pycon giving "good" talks has also given their fair share of bad talks. It just takes practice. The best way to practice is just to give talks.
- Video yourself (even just for part of your talk) and see yourself "in action" as others see you.
- Giving a talk is not a writing problem or a design problem. It is a performance problem. If you are a new speaker, you should probably spend more time practicing your presentation than you spend writing and designing the slides.
- Practice! Go through your talk at least twice just to yourself. You'll find yourself much more confident if you know the talk well enough not to worry about forgetting it or what you will say next. It just flows better and you'll feel much more relaxed.

3.2.6 What to Bring

- Bring your own dongle, and your presentation on a thumb drive, in several formats.

3.2.7 Somewhat More Officially

For a 30-minute slot, you have 25 minutes to talk plus 5 minutes for questions. 45-minute slots mean you have 40 minutes to talk and 5 minutes for questions. Time your talk accordingly.

3.3 At the Talk

3.3.1 Before You Get Started

- Make sure you got a good night sleep (yeah right).
- In fact, be rested, fed, and sober (not somber) for your talk. Skip the late night party and get a good night's rest. The day of the talk eat food that makes you feel physically better.
- The backdrops are generally black so don't wear dark clothes. On video it can look like you it is just your head bobbing around by itself. Steve Jobs can get away with it because he has a professional lighting crew, you don't.
- Remove your conference lanyard. It can distract you, you will play with it, or it will get caught in your wireless microphone and cause problems.
- Turn off or silence your own mobile phone and in general remove any large objects from your pockets which make it look like you are hiding your next bottle of beer in there.
- If you don't need wifi for your talk, disconnect yourself from the network, shutdown all applications besides the presentation software. Temporarily turn off any notifications, or sources of notifications as the popups can sometimes cause presentation software such as KeyNote to drop out of presenter mode. The audience also doesn't want to hear all the tones as people mention you on live Twitter streams saying how cool or lame your talk is.
- Try to verify ahead of time that your computer works with the AV system. If you are going to rely on speaker's notes, consider printing them out ahead of time in case you can't use your laptop screen for some reason. Remember Murphy.

- Be in the room a few minutes early if you can, and chat to people already in the room as you prepare.
- Introduce yourself to your session chairperson no later than the break before your talk. Once the presentations start the chair will be focused on managing the session.

3.3.2 DOs and DON'Ts During

- *DON'T* give a talk with any kind of pen in your hand. You might just end up with ink all over your shirt.
- For that matter, be conscientious that having anything in your hands could be a distraction, to the audience or to you.
- *DON'T* move around. Stand still. See this blog post.
- *DO* take the podium... then move to the side. Make sure the audience can see you. This is a good thing. We like seeing the whole person. Once you're out there, stand still. Don't sway. Try not to lean. Keep your hand movements to, maybe, one every five minutes. This works really well if you're actually calm and well-rested.
- If you're tired, stay behind the podium and grab it. This isn't the best thing in the world, as it weakens your visual presence, but it also won't be distracting your audience.
- *DO* speak loudly! This naturally makes you slow down and enunciate your words more clearly. It also makes you seem and *feel* more confident. It's very hard to listen to a talk, even from a very knowledgeable person, who is talking too quietly and mumbling words. It's amazing what effect it has on your confidence too.
- *DON'T* just read the slides. People came to hear what you have to say.
- What you should be doing is using the slides to remind yourself of your next point. Think of them as notes for your speech, not the speech itself.
- *DO* remember about the microphone, whether it's attached to your lapel or is on the podium in front of you. Some speakers will turn to point at the display and talk away from the microphone; be sure to point and then turn back.
- *Never*, *ever* do a live demo, or depend on the wireless.

3.3.3 About the Audience

- Unlike some academic conferences, PyCon is not an adversarial environment–you're not going to be attacked afterwards.
- Just flat-out realize you will be presenting to hundreds of people (even at worst case of 10% of the conference, that's 150 people). But presenting to a lot of people is actually easier than a small group at a users group. Why? Smaller venue means more attentive attendees.
- When you present at PyCon you have to realize a huge portion of people will be on their laptops, staring at their screens. This doesn't mean they are not listening, but it can be disconcerting as you won't be able to use the audience to easily judge how engaging you are being.
- Said one contributer: "I have presented and thought I sucked and then later have tell people they loved my presentation, even with essentially no one laughing at my jokes."
- Look around at your audience and pay attention to their body language.
- Check that the audience is hearing you ("Can you hear me at the back?") and understanding you ("Does that make sense to everyone?"; "Are there any questions about that?").
- It takes people about 10 seconds to realize you have asked a question, so if you ask if people are understanding you need to wait that long for it to be effective, else just always assume that someone will speak up if you are being confusing.

- Encourage the audience to fill all available seats, rather than standing/sitting in the aisles or by the door.
- Open Space, BoF, and Followup. Don't forget to invite your audiance to a BoF or Open Space followup! The part of your audience which is passionate (or has become passionate due to your presentation) are encouraged to continue the conversation, and you the presenter are a key part of that.

3.3.4 Question Time

- There may be that one smart aleck who tries to point out some bad design decision or mistake or something that is really not important or your fault. Feel free to answer them succinctly to get them off the microphone.
- Someone will ask you a tough question that you can't answer on the spot, so just ask them to catch you after the talk.
- If someone asks on the mic a very specific question that is really only helpful to them, ask them to talk to you after so you can get to more questions that are helpful to the whole audience.
- Have a prepped response for when you just don't know an answer. It's okay to say 'I haven't run into that' or 'I'm not familiar with that'. It's *not* okay to bumble and fake it.
- During the Q&A portion of the talk, always repeat any questions that were asked without a microphone - otherwise many people in the audience won't hear the question.
- Consider finishing your talk early for extra question time. Then prepare some bonus material in case people run out of questions. See this blog comment.
- Be nice to people who come up to you after a talk. You never know who is that new person who comes up to you, and you might regret it later. Be nice to them and you'll find out. Try to find time to talk to everyone, even if for just a minute each.

3.3.5 Handling Nervousness

- Remember, they're more scared of you than you are of them!
- You shouldn't get all worried about "being remembered for a bad talk". The honest truth of the matter is that almost nobody is going to remember much about the actual presentation of your talk. So, don't sweat it.
- If you're nervous, thinking that if you screw up that you'll forever ruin your reputation in the community due to fidgeting a tiny bit too much? Chill out. We're all still working on our talks.
- Take a deep breath and relax. One contributer said, "I've yet to see a talk where someone was booed off the stage, and I've seen some horrific talks."
- If you are nervous, there's nothing wrong with admitting that. The information you present is your talk's primary value. PyCon audiences are very forgiving.
- Take time to yourself before you speak. Deep breathing is always good preparation. Your nervousness will be less apparent than you suppose.
- And again, the best remedy for nervousness is to practice, practice, practice.

3.4 Resources

http://therealkatie.net/blog/2011/sep/19/tip-speakers/
http://nedbatchelder.com/text/presentationtips.html
http://nedbatchelder.com/blog/201002/25_minutes_is_a_bitch.html
http://nedbatchelder.com/blog/201102/pycon_presentations_hollywood_style.html

http://web.archive.org/web/20100212084133/http://us.pycon.org/TX2007/HelpForSpeakers
http://pydanny.blogspot.com/2011/02/my-tips-for-speaking.html

http://dalecarnegielesson.blogspot.com/2011/06/12-ways-to-minimize-fear-and-anxiety.html
http://perl.plover.com/yak/presentation/
http://web.archive.org/web/20060628122618/http://www.sage.org/presentation/
http://pages.cs.wisc.edu/~markhill/conference-talk.html
http://solarsail.hcs.harvard.edu/~krstic/08-2005-giving-talks.pdf
http://web.archive.org/web/20110204154428/http://ite.org/meetcon/speech.asp
http://shop.oreilly.com/product/9780596802004.do

3.5 Acknowledgements

Finally, a big thank-you to the folks that have contributed (in no particular order): * Raymond Hettinger, * Katie Cunningham, * David Beazley, * Brett Cannon, * Doug Hellmann, * C. Titus Brown, * Michael Foord, * Ned Batchelder, * Danny Greedfield, * Graham Dumpleton (from comments), * Doug Napoleone (from comments). * Jacob Kaplan-Moss (from comments).

Each of these is a stand-alone document.

Ultimately I'd like to have a tool that makes adding to the reference pages as easy as adding a bookmark is now. All in good time...

Indices and tables

- genindex
- modindex
- search