jupyter_client Documentation

Release 4.1.0.dev

Jupyter Development Team

October 08, 2015

Contents

This package provides the Python API for starting, managing and communicating with Jupyter kernels.

Important: This document contains the authoritative description of the Jupyter messaging protocol. All developers are strongly encouraged to keep it updated as the implementation evolves, so that we have a single common reference for all protocol details.

CHAPTER 1

Messaging in Jupyter

This document explains the basic communications design and messaging specification for how Jupyter frontends and kernels communicate. The ZeroMQ library provides the low-level transport layer over which these messages are sent.

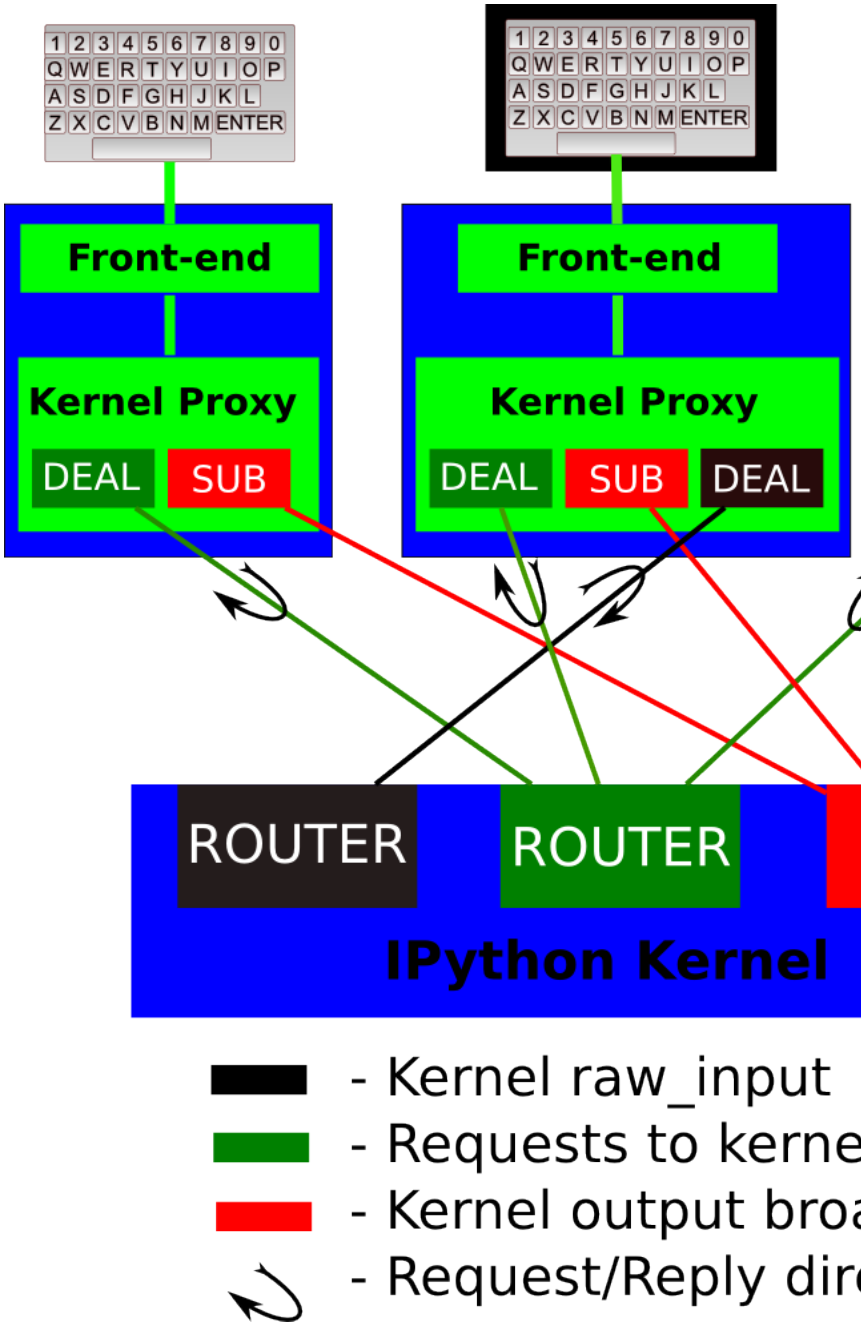
Important: This document contains the authoritative description of the IPython messaging protocol. All developers are strongly encouraged to keep it updated as the implementation evolves, so that we have a single common reference for all protocol details.

1.1 Versioning

The Jupyter message specification is versioned independently of the packages that use it. The current version of the specification is 5.0.

Note: *New in* and *Changed in* messages in this document refer to versions of the **Jupyter message specification**, not versions of `jupyter_client`.

1.2 Introduction



The basic design is explained in the following diagram:
A single kernel can be simultaneously connected to one or more frontends. The kernel has four sockets that serve the following functions:

1. **Shell**: this single ROUTER socket allows multiple incoming connections from frontends, and this is the socket where requests for code execution, object information, prompts, etc. are made to the kernel by any frontend. The communication on this socket is a sequence of request/reply actions from each frontend and the kernel.

2. **IOPub**: this socket is the 'broadcast channel' where the kernel publishes all side effects (stdout, stderr, etc.) as well as the requests coming from any client over the shell socket and its own requests on the stdin socket. There are a number of actions in Python which generate side effects: `print()` writes to `sys.stdout`, errors

generate tracebacks, etc. Additionally, in a multi-client scenario, we want all frontends to be able to know what each other has sent to the kernel (this can be useful in collaborative scenarios, for example). This socket allows both side effects and the information about communications taking place with one client over the shell channel to be made available to all clients in a uniform manner.

3. **stdin**: this ROUTER socket is connected to all frontends, and it allows the kernel to request input from the active frontend when `raw_input()` is called. The frontend that executed the code has a DEALER socket that acts as a 'virtual keyboard' for the kernel while this communication is happening (illustrated in the figure by the black outline around the central keyboard). In practice, frontends may display such kernel requests using a special input widget or otherwise indicating that the user is to type input for the kernel instead of normal commands in the frontend.

   All messages are tagged with enough information (details below) for clients to know which messages come from their own interaction with the kernel and which ones are from other clients, so they can display each type appropriately.

4. **Control**: This channel is identical to Shell, but operates on a separate socket, to allow important messages to avoid queueing behind execution requests (e.g. shutdown or abort).

The actual format of the messages allowed on each of these channels is specified below. Messages are dicts of dicts with string keys and values that are reasonably representable in JSON. Our current implementation uses JSON explicitly as its message format, but this shouldn't be considered a permanent feature. As we've discovered that JSON has non-trivial performance issues due to excessive copying, we may in the future move to a pure pickle-based raw message format. However, it should be possible to easily convert from the raw objects to JSON, since we may have non-python clients (e.g. a web frontend). As long as it's easy to make a JSON version of the objects that is a faithful representation of all the data, we can communicate with such clients.

Note: Not all of these have yet been fully fleshed out, but the key ones are, see kernel and frontend files for actual implementation details.

1.3 General Message Format

A message is defined by the following four-dictionary structure:

```
{
  # The message header contains a pair of unique identifiers for the
  # originating session and the actual message id, in addition to the
  # username for the process that generated the message.  This is useful in
  # collaborative settings where multiple users may be interacting with the
  # same kernel simultaneously, so that frontends can label the various
  # messages in a meaningful way.
  'header' : {
                'msg_id' : uuid,
                'username' : str,
                'session' : uuid,
                # ISO 8601 timestamp for when the message is created
                'date': str,
                # All recognized message type strings are listed below.
                'msg_type' : str,
                # the message protocol version
                'version' : '5.0',
    },

  # In a chain of messages, the header from the parent is copied so that
  # clients can track where messages come from.
  'parent_header' : dict,
```

```
  # Any metadata associated with the message.
  'metadata' : dict,

  # The actual content of the message must be a dict, whose structure
  # depends on the message type.
  'content' : dict,
}
```

Changed in version 5.0: `version` key added to the header.

Changed in version 5.1: `date` in the header was accidentally omitted from the spec prior to 5.1, but it has always been in the canonical implementation, so implementers are strongly encouraged to include it. It will be mandatory in 5.1.

1.4 The Wire Protocol

This message format exists at a high level, but does not describe the actual *implementation* at the wire level in zeromq. The canonical implementation of the message spec is our `Session` class.

Note: This section should only be relevant to non-Python consumers of the protocol. Python consumers should simply import and the use implementation of the wire protocol in `jupyter_client.session.Session`.

Every message is serialized to a sequence of at least six blobs of bytes:

```
[
  b'u-u-i-d',          # zmq identity(ies)
  b'<IDS|MSG>',        # delimiter
  b'baddad42',         # HMAC signature
  b'{header}',         # serialized header dict
  b'{parent_header}', # serialized parent header dict
  b'{metadata}',       # serialized metadata dict
  b'{content}',        # serialized content dict
  b'blob',             # extra raw data buffer(s)
  ...
]
```

The front of the message is the ZeroMQ routing prefix, which can be zero or more socket identities. This is every piece of the message prior to the delimiter key `<IDS|MSG>`. In the case of IOPub, there should be just one prefix component, which is the topic for IOPub subscribers, e.g. `execute_result`, `display_data`.

Note: In most cases, the IOPub topics are irrelevant and completely ignored, because frontends just subscribe to all topics. The convention used in the IPython kernel is to use the msg_type as the topic, and possibly extra information about the message, e.g. `execute_result` or `stream.stdout`

After the delimiter is the HMAC signature of the message, used for authentication. If authentication is disabled, this should be an empty string. By default, the hashing function used for computing these signatures is sha256.

Note: To disable authentication and signature checking, set the *key* field of a connection file to an empty string.

The signature is the HMAC hex digest of the concatenation of:

- A shared key (typically the `key` field of a connection file)
- The serialized header dict
- The serialized parent header dict

- The serialized metadata dict
- The serialized content dict

In Python, this is implemented via:

```
# once:
digester = HMAC(key, digestmod=hashlib.sha256)

# for each message
d = digester.copy()
for serialized_dict in (header, parent, metadata, content):
    d.update(serialized_dict)
signature = d.hexdigest()
```

After the signature is the actual message, always in four frames of bytes. The four dictionaries that compose a message are serialized separately, in the order of header, parent header, metadata, and content. These can be serialized by any function that turns a dict into bytes. The default and most common serialization is JSON, but msgpack and pickle are common alternatives.

After the serialized dicts are zero to many raw data buffers, which can be used by message types that support binary data (mainly apply and data_pub).

1.5 Python functional API

As messages are dicts, they map naturally to a `func(**kw)` call form. We should develop, at a few key points, functional forms of all the requests that take arguments in this manner and automatically construct the necessary dict for sending.

In addition, the Python implementation of the message specification extends messages upon deserialization to the following form for convenience:

```
{
  'header' : dict,
  # The msg's unique identifier and type are always stored in the header,
  # but the Python implementation copies them to the top level.
  'msg_id' : uuid,
  'msg_type' : str,
  'parent_header' : dict,
  'content' : dict,
  'metadata' : dict,
}
```

All messages sent to or received by any IPython process should have this extended structure.

1.6 Messages on the shell ROUTER/DEALER sockets

1.6.1 Execute

This message type is used by frontends to ask the kernel to execute code on behalf of the user, in a namespace reserved to the user's variables (and thus separate from the kernel's own internal code and variables).

Message type: `execute_request`:

```
content = {
    # Source code to be executed by the kernel, one or more lines.
'code' : str,

# A boolean flag which, if True, signals the kernel to execute
# this code as quietly as possible.
# silent=True forces store_history to be False,
# and will *not*:
#   - broadcast output on the IOPUB channel
#   - have an execute_result
# The default is False.
'silent' : bool,

# A boolean flag which, if True, signals the kernel to populate history
# The default is True if silent is False.  If silent is True, store_history
# is forced to be False.
'store_history' : bool,

# A dict mapping names to expressions to be evaluated in the
# user's dict. The rich display-data representation of each will be evaluated after execution.
# See the display_data content for the structure of the representation data.
'user_expressions' : dict,

# Some frontends do not support stdin requests.
# If raw_input is called from code executed from such a frontend,
# a StdinNotImplementedError will be raised.
'allow_stdin' : True,

# A boolean flag, which, if True, does not abort the execution queue, if an exception is encountered
# This allows the queued execution of multiple execute_requests, even if they generate exceptions.
'stop_on_error' : False,
}
```

Changed in version 5.0: `user_variables` removed, because it is redundant with user_expressions.

The `code` field contains a single string (possibly multiline) to be executed.

The `user_expressions` field deserves a detailed explanation. In the past, IPython had the notion of a prompt string that allowed arbitrary code to be evaluated, and this was put to good use by many in creating prompts that displayed system status, path information, and even more esoteric uses like remote instrument status acquired over the network. But now that IPython has a clean separation between the kernel and the clients, the kernel has no prompt knowledge; prompts are a frontend feature, and it should be even possible for different frontends to display different prompts while interacting with the same kernel. `user_expressions` can be used to retrieve this information.

Any error in evaluating any expression in `user_expressions` will result in only that key containing a standard error message, of the form:

```
{
    'status' : 'error',
    'ename' : 'NameError',
    'evalue' : 'foo',
    'traceback' : ...
}
```

Note: In order to obtain the current execution counter for the purposes of displaying input prompts, frontends may make an execution request with an empty code string and `silent=True`.

Upon completion of the execution request, the kernel *always* sends a reply, with a status code indicating what happened

and additional data depending on the outcome. See *below* for the possible return codes and associated data.

See also:

Execution semantics in the IPython kernel

Execution counter (prompt number)

The kernel should have a single, monotonically increasing counter of all execution requests that are made with `store_history=True`. This counter is used to populate the `In[n]` and `Out[n]` prompts. The value of this counter will be returned as the `execution_count` field of all `execute_reply` and `execute_input` messages.

Execution results

Message type: `execute_reply`:

```
content = {
  # One of: 'ok' OR 'error' OR 'abort'
  'status' : str,

  # The global kernel counter that increases by one with each request that
  # stores history.  This will typically be used by clients to display
  # prompt numbers to the user.  If the request did not store history, this will
  # be the current value of the counter in the kernel.
  'execution_count' : int,
}
```

When status is 'ok', the following extra fields are present:

```
{
  # 'payload' will be a list of payload dicts, and is optional.
  # payloads are considered deprecated.
  # The only requirement of each payload dict is that it have a 'source' key,
  # which is a string classifying the payload (e.g. 'page').

  'payload' : list(dict),

  # Results for the user_expressions.
  'user_expressions' : dict,
}
```

Changed in version 5.0: `user_variables` is removed, use user_expressions instead.

When status is 'error', the following extra fields are present:

```
{
  'ename' : str,   # Exception name, as a string
  'evalue' : str,  # Exception value, as a string

  # The traceback will contain a list of frames, represented each as a
  # string.  For now we'll stick to the existing design of ultraTB, which
  # controls exception level of detail statefully.  But eventually we'll
  # want to grow into a model where more information is collected and
  # packed into the traceback object, with clients deciding how little or
  # how much of it to unpack.  But for now, let's start with a simple list
  # of strings, since that requires only minimal changes to ultratb as
  # written.
```

```
  'traceback' : list,
}
```

When status is 'abort', there are for now no additional data fields. This happens when the kernel was interrupted by a signal.

Payloads (DEPRECATED)

Execution payloads

Payloads are considered **deprecated**, though their replacement is not yet implemented.

Payloads are a way to trigger frontend actions from the kernel. Current payloads:

page: display data in a pager.

Pager output is used for introspection, or other displayed information that's not considered output. Pager payloads are generally displayed in a separate pane, that can be viewed alongside code, and are not included in notebook documents.

```
{
  "source": "page",
  # mime-bundle of data to display in the pager.
  # Must include text/plain.
  "data": mimebundle,
  # line offset to start from
  "start": int,
}
```

set_next_input: create a new output

used to create new cells in the notebook, or set the next input in a console interface. The main example being `%load`.

```
{
  "source": "set_next_input",
  # the text contents of the cell to create
  "text": "some cell content",
  # If true, replace the current cell in document UIs instead of inserting
  # a cell. Ignored in console UIs.
  "replace": bool,
}
```

edit: open a file for editing.

Triggered by *%edit*. Only the QtConsole currently supports edit payloads.

```
{
  "source": "edit",
  "filename": "/path/to/file.py", # the file to edit
  "line_number": int, # the line number to start with
}
```

ask_exit: instruct the frontend to prompt the user for exit

Allows the kernel to request exit, e.g. via `%exit` in IPython. Only for console frontends.

```
{
  "source": "ask_exit",
  # whether the kernel should be left running, only closing the client
  "keepkernel": bool,
}
```

1.6.2 Introspection

Code can be inspected to show useful information to the user. It is up to the Kernel to decide what information should be displayed, and its formatting.

Message type: `inspect_request`:

```
content = {
    # The code context in which introspection is requested
    # this may be up to an entire multiline cell.
    'code' : str,

    # The cursor position within 'code' (in unicode characters) where inspection is requested
    'cursor_pos' : int,

    # The level of detail desired.  In IPython, the default (0) is equivalent to typing
    # 'x?' at the prompt, 1 is equivalent to 'x??'.
    # The difference is up to kernels, but in IPython level 1 includes the source code
    # if available.
    'detail_level' : 0 or 1,
}
```

Changed in version 5.0: `object_info_request` renamed to `inspect_request`.

Changed in version 5.0: `name` key replaced with `code` and `cursor_pos`, moving the lexing responsibility to the kernel.

The reply is a mime-bundle, like a *display_data* message, which should be a formatted representation of information about the context. In the notebook, this is used to show tooltips over function calls, etc.

Message type: `inspect_reply`:

```
content = {
    # 'ok' if the request succeeded or 'error', with error information as in all other replies.
    'status' : 'ok',

    # found should be true if an object was found, false otherwise
    'found' : bool,

    # data can be empty if nothing is found
    'data' : dict,
    'metadata' : dict,
}
```

Changed in version 5.0: `object_info_reply` renamed to `inspect_reply`.

Changed in version 5.0: Reply is changed from structured data to a mime bundle, allowing formatting decisions to be made by the kernel.

1.6.3 Completion

Message type: `complete_request`:

```
content = {
    # The code context in which completion is requested
    # this may be up to an entire multiline cell, such as
    # 'foo = a.isal'
    'code' : str,
```

```
    # The cursor position within 'code' (in unicode characters) where completion is requested
    'cursor_pos' : int,
}
```

Changed in version 5.0: `line`, `block`, and `text` keys are removed in favor of a single `code` for context. Lexing is up to the kernel.

Message type: `complete_reply`:

```
content = {
# The list of all matches to the completion request, such as
# ['a.isalnum', 'a.isalpha'] for the above example.
'matches' : list,

# The range of text that should be replaced by the above matches when a completion is accepted.
# typically cursor_end is the same as cursor_pos in the request.
'cursor_start' : int,
'cursor_end' : int,

# Information that frontend plugins might use for extra display information about completions.
'metadata' : dict,

# status should be 'ok' unless an exception was raised during the request,
# in which case it should be 'error', along with the usual error message content
# in other messages.
'status' : 'ok'
}
```

Changed in version 5.0: `matched_text` is removed in favor of `cursor_start` and `cursor_end`. `metadata` is added for extended information.

1.6.4 History

For clients to explicitly request history from a kernel. The kernel has all the actual execution history stored in a single location, so clients can request it from the kernel when needed.

Message type: `history_request`:

```
content = {

  # If True, also return output history in the resulting dict.
  'output' : bool,

  # If True, return the raw input history, else the transformed input.
  'raw' : bool,

  # So far, this can be 'range', 'tail' or 'search'.
  'hist_access_type' : str,

  # If hist_access_type is 'range', get a range of input cells. session can
  # be a positive session number, or a negative number to count back from
  # the current session.
  'session' : int,
  # start and stop are line numbers within that session.
  'start' : int,
  'stop' : int,

  # If hist_access_type is 'tail' or 'search', get the last n cells.
```

```
  'n' : int,

  # If hist_access_type is 'search', get cells matching the specified glob
  # pattern (with * and ? as wildcards).
  'pattern' : str,

  # If hist_access_type is 'search' and unique is true, do not
  # include duplicated history.  Default is false.
  'unique' : bool,

}
```

New in version 4.0: The key `unique` for `history_request`.

Message type: `history_reply`:

```
content = {
  # A list of 3 tuples, either:
  # (session, line_number, input) or
  # (session, line_number, (input, output)),
  # depending on whether output was False or True, respectively.
  'history' : list,
}
```

1.6.5 Code completeness

New in version 5.0.

When the user enters a line in a console style interface, the console must decide whether to immediately execute the current code, or whether to show a continuation prompt for further input. For instance, in Python `a = 5` would be executed immediately, while `for i in range(5):` would expect further input.

There are four possible replies:

- *complete* code is ready to be executed
- *incomplete* code should prompt for another line
- *invalid* code will typically be sent for execution, so that the user sees the error soonest.
- *unknown* - if the kernel is not able to determine this. The frontend should also handle the kernel not replying promptly. It may default to sending the code for execution, or it may implement simple fallback heuristics for whether to execute the code (e.g. execute after a blank line).

Frontends may have ways to override this, forcing the code to be sent for execution or forcing a continuation prompt.

Message type: `is_complete_request`:

```
content = {
    # The code entered so far as a multiline string
    'code' : str,
}
```

Message type: `is_complete_reply`:

```
content = {
    # One of 'complete', 'incomplete', 'invalid', 'unknown'
    'status' : str,

    # If status is 'incomplete', indent should contain the characters to use
```

```
    # to indent the next line. This is only a hint: frontends may ignore it
    # and use their own autoindentation rules. For other statuses, this
    # field does not exist.
    'indent': str,
}
```

1.6.6 Connect

When a client connects to the request/reply socket of the kernel, it can issue a connect request to get basic information about the kernel, such as the ports the other ZeroMQ sockets are listening on. This allows clients to only have to know about a single port (the shell channel) to connect to a kernel.

Message type: `connect_request`:

```
content = {
}
```

Message type: `connect_reply`:

```
content = {
    'shell_port' : int,    # The port the shell ROUTER socket is listening on.
    'iopub_port' : int,    # The port the PUB socket is listening on.
    'stdin_port' : int,    # The port the stdin ROUTER socket is listening on.
    'hb_port' : int,       # The port the heartbeat socket is listening on.
}
```

1.6.7 Comm info

When a client needs the currently open comms in the kernel, it can issue a request for the currently open comms. When the optional `target_name` is specified, the reply only contains the currently open comms for the target.

Message type: `comm_info_request`:

```
content = {
    # Optional, the target name
    'target_name': str,
}
```

Message type: `comm_info_reply`:

```
content = {
    # A dictionary of the comms, indexed by uuids.
    'comms': {
        comm_id: {
            'target_name': str,
        },
    },
}
```

New in version 5.1: `comm_info` is a proposed addition for msgspec v5.1.

1.6.8 Kernel info

If a client needs to know information about the kernel, it can make a request of the kernel's information. This message can be used to fetch core information of the kernel, including language (e.g., Python), language version number and IPython version number, and the IPython message spec version number.

Message type: `kernel_info_request`:

```
content = {
}
```

Message type: `kernel_info_reply`:

```
content = {
    # Version of messaging protocol.
    # The first integer indicates major version.  It is incremented when
    # there is any backward incompatible change.
    # The second integer indicates minor version.  It is incremented when
    # there is any backward compatible change.
    'protocol_version': 'X.Y.Z',

    # The kernel implementation name
    # (e.g. 'ipython' for the IPython kernel)
    'implementation': str,

    # Implementation version number.
    # The version number of the kernel's implementation
    # (e.g. IPython.__version__ for the IPython kernel)
    'implementation_version': 'X.Y.Z',

    # Information about the language of code for the kernel
    'language_info': {
        # Name of the programming language in which kernel is implemented.
        # Kernel included in IPython returns 'python'.
        'name': str,

        # Language version number.
        # It is Python version number (e.g., '2.7.3') for the kernel
        # included in IPython.
        'version': 'X.Y.Z',

        # mimetype for script files in this language
        'mimetype': str,

        # Extension including the dot, e.g. '.py'
        'file_extension': str,

        # Pygments lexer, for highlighting
        # Only needed if it differs from the 'name' field.
        'pygments_lexer': str,

        # Codemirror mode, for for highlighting in the notebook.
        # Only needed if it differs from the 'name' field.
        'codemirror_mode': str or dict,

        # Nbconvert exporter, if notebooks written with this kernel should
        # be exported with something other than the general 'script'
        # exporter.
        'nbconvert_exporter': str,
    },

    # A banner of information about the kernel,
    # which may be desplayed in console environments.
    'banner' : str,
```

```
    # Optional: A list of dictionaries, each with keys 'text' and 'url'.
    # These will be displayed in the help menu in the notebook UI.
    'help_links': [
        {'text': str, 'url': str}
    ],
}
```

Refer to the lists of available Pygments lexers and codemirror modes for those fields.

Changed in version 5.0: Versions changed from lists of integers to strings.

Changed in version 5.0: `ipython_version` is removed.

Changed in version 5.0: `language_info`, `implementation`, `implementation_version`, `banner` and `help_links` keys are added.

Changed in version 5.0: `language_version` moved to `language_info.version`

Changed in version 5.0: `language` moved to `language_info.name`

1.6.9 Kernel shutdown

The clients can request the kernel to shut itself down; this is used in multiple cases:

- when the user chooses to close the client application via a menu or window control.
- when the user types 'exit' or 'quit' (or their uppercase magic equivalents).
- when the user chooses a GUI method (like the 'Ctrl-C' shortcut in the IPythonQt client) to force a kernel restart to get a clean kernel without losing client-side state like history or inlined figures.

The client sends a shutdown request to the kernel, and once it receives the reply message (which is otherwise empty), it can assume that the kernel has completed shutdown safely.

Upon their own shutdown, client applications will typically execute a last minute sanity check and forcefully terminate any kernel that is still alive, to avoid leaving stray processes in the user's machine.

Message type: `shutdown_request`:

```
content = {
    'restart' : bool # whether the shutdown is final, or precedes a restart
}
```

Message type: `shutdown_reply`:

```
content = {
    'restart' : bool # whether the shutdown is final, or precedes a restart
}
```

Note: When the clients detect a dead kernel thanks to inactivity on the heartbeat socket, they simply send a forceful process termination signal, since a dead process is unlikely to respond in any useful way to messages.

1.7 Messages on the PUB/SUB socket

1.7.1 Streams (stdout, stderr, etc)

Message type: `stream`:

```
content = {
    # The name of the stream is one of 'stdout', 'stderr'
    'name' : str,

    # The text is an arbitrary string to be written to that stream
    'text' : str,
}
```

Changed in version 5.0: 'data' key renamed to 'text' for consistency with the notebook format.

1.7.2 Display Data

This type of message is used to bring back data that should be displayed (text, html, svg, etc.) in the frontends. This data is published to all frontends. Each message can have multiple representations of the data; it is up to the frontend to decide which to use and how. A single message should contain all possible representations of the same information. Each representation should be a JSON'able data structure, and should be a valid MIME type.

Some questions remain about this design:

- Do we use this message type for execute_result/displayhook? Probably not, because the displayhook also has to handle the Out prompt display. On the other hand we could put that information into the metadata section.

Message type: `display_data`:

```
content = {

    # Who create the data
    'source' : str,

    # The data dict contains key/value pairs, where the keys are MIME
    # types and the values are the raw data of the representation in that
    # format.
    'data' : dict,

    # Any metadata that describes the data
    'metadata' : dict
}
```

The `metadata` contains any metadata that describes the output. Global keys are assumed to apply to the output as a whole. The `metadata` dict can also contain mime-type keys, which will be sub-dictionaries, which are interpreted as applying only to output of that type. Third parties should put any data they write into a single dict with a reasonably unique name to avoid conflicts.

The only metadata keys currently defined in IPython are the width and height of images:

```
metadata = {
  'image/png' : {
    'width': 640,
    'height': 480
  }
}
```

Changed in version 5.0: *application/json* data should be unpacked JSON data, not double-serialized as a JSON string.

1.7.3 Raw Data Publication

`display_data` lets you publish *representations* of data, such as images and html. This `data_pub` message lets you publish *actual raw data*, sent via message buffers.

data_pub messages are constructed via the `IPython.lib.datapub.publish_data()` function:

```
from IPython.kernel.zmq.datapub import publish_data
ns = dict(x=my_array)
publish_data(ns)
```

Message type: `data_pub`:

```
content = {
    # the keys of the data dict, after it has been unserialized
    'keys' : ['a', 'b']
}
# the namespace dict will be serialized in the message buffers,
# which will have a length of at least one
buffers = [b'pdict', ...]
```

The interpretation of a sequence of data_pub messages for a given parent request should be to update a single namespace with subsequent results.

Note: No frontends directly handle data_pub messages at this time. It is currently only used by the client/engines in `IPython.parallel`, where engines may publish *data* to the Client, of which the Client can then publish *representations* via `display_data` to various frontends.

1.7.4 Code inputs

To let all frontends know what code is being executed at any given time, these messages contain a re-broadcast of the `code` portion of an *execute_request*, along with the *execution_count*.

Message type: `execute_input`:

```
content = {
    'code' : str,  # Source code to be executed, one or more lines

    # The counter for this execution is also provided so that clients can
    # display it, since IPython automatically creates variables called _iN
    # (for input prompt In[N]).
    'execution_count' : int
}
```

Changed in version 5.0: `pyin` is renamed to `execute_input`.

1.7.5 Execution results

Results of an execution are published as an `execute_result`. These are identical to *display_data* messages, with the addition of an `execution_count` key.

Results can have multiple simultaneous formats depending on its configuration. A plain text representation should always be provided in the `text/plain` mime-type. Frontends are free to display any or all of these according to its capabilities. Frontends should ignore mime-types they do not understand. The data itself is any JSON object and depends on the format. It is often, but not always a string.

Message type: `execute_result`:

```
content = {

    # The counter for this execution is also provided so that clients can
    # display it, since IPython automatically creates variables called _N
    # (for prompt N).
    'execution_count' : int,

    # data and metadata are identical to a display_data message.
    # the object being displayed is that passed to the display hook,
    # i.e. the *result* of the execution.
    'data' : dict,
    'metadata' : dict,
}
```

1.7.6 Execution errors

When an error occurs during code execution

Message type: `error`:

```
content = {
   # Similar content to the execute_reply messages for the 'error' case,
   # except the 'status' field is omitted.
}
```

Changed in version 5.0: `pyerr` renamed to `error`

1.7.7 Kernel status

This message type is used by frontends to monitor the status of the kernel.

Message type: `status`:

```
content = {
    # When the kernel starts to handle a message, it will enter the 'busy'
    # state and when it finishes, it will enter the 'idle' state.
    # The kernel will publish state 'starting' exactly once at process startup.
    execution_state : ('busy', 'idle', 'starting')
}
```

Changed in version 5.0: Busy and idle messages should be sent before/after handling every message, not just execution.

Note: Extra status messages are added between the notebook webserver and websocket clients that are not sent by the kernel. These are:

- restarting (kernel has died, but will be automatically restarted)
- dead (kernel has died, restarting has failed)

1.7.8 Clear output

This message type is used to clear the output that is visible on the frontend.

Message type: `clear_output`:

```
content = {

    # Wait to clear the output until new output is available.  Clears the
    # existing output immediately before the new output is displayed.
    # Useful for creating simple animations with minimal flickering.
    'wait' : bool,
}
```

Changed in version 4.1: `stdout`, `stderr`, and `display` boolean keys for selective clearing are removed, and `wait` is added. The selective clearing keys are ignored in v4 and the default behavior remains the same, so v4 clear_output messages will be safely handled by a v4.1 frontend.

1.8 Messages on the stdin ROUTER/DEALER sockets

This is a socket where the request/reply pattern goes in the opposite direction: from the kernel to a *single* frontend, and its purpose is to allow `raw_input` and similar operations that read from `sys.stdin` on the kernel to be fulfilled by the client. The request should be made to the frontend that made the execution request that prompted `raw_input` to be called. For now we will keep these messages as simple as possible, since they only mean to convey the `raw_input(prompt)` call.

Message type: `input_request`:

```
content = {
    # the text to show at the prompt
    'prompt' : str,
    # Is the request for a password?
    # If so, the frontend shouldn't echo input.
    'password' : bool
}
```

Message type: `input_reply`:

```
content = { 'value' : str }
```

When `password` is True, the frontend should not echo the input as it is entered.

Changed in version 5.0: `password` key added.

Note: The stdin socket of the client is required to have the same zmq IDENTITY as the client's shell socket. Because of this, the `input_request` must be sent with the same IDENTITY routing prefix as the `execute_reply` in order for the frontend to receive the message.

Note: We do not explicitly try to forward the raw `sys.stdin` object, because in practice the kernel should behave like an interactive program. When a program is opened on the console, the keyboard effectively takes over the `stdin` file descriptor, and it can't be used for raw reading anymore. Since the IPython kernel effectively behaves like a console program (albeit one whose "keyboard" is actually living in a separate process and transported over the zmq connection), raw `stdin` isn't expected to be available.

1.9 Heartbeat for kernels

Clients send ping messages on a REQ socket, which are echoed right back from the Kernel's REP socket. These are simple bytestrings, not full JSON messages described above.

1.10 Custom Messages

New in version 4.1.

Message spec 4.1 (IPython 2.0) added a messaging system for developers to add their own objects with Frontend and Kernel-side components, and allow them to communicate with each other. To do this, IPython adds a notion of a `Comm`, which exists on both sides, and can communicate in either direction.

These messages are fully symmetrical - both the Kernel and the Frontend can send each message, and no messages expect a reply. The Kernel listens for these messages on the Shell channel, and the Frontend listens for them on the IOPub channel.

1.10.1 Opening a Comm

Opening a Comm produces a `comm_open` message, to be sent to the other side:

```
{
  'comm_id' : 'u-u-i-d',
  'target_name' : 'my_comm',
  'data' : {}
}
```

Every Comm has an ID and a target name. The code handling the message on the receiving side is responsible for maintaining a mapping of target_name keys to constructors. After a `comm_open` message has been sent, there should be a corresponding Comm instance on both sides. The `data` key is always a dict and can be any extra JSON information used in initialization of the comm.

If the `target_name` key is not found on the receiving side, then it should immediately reply with a `comm_close` message to avoid an inconsistent state.

1.10.2 Comm Messages

Comm messages are one-way communications to update comm state, used for synchronizing widget state, or simply requesting actions of a comm's counterpart.

Essentially, each comm pair defines their own message specification implemented inside the `data` dict.

There are no expected replies (of course, one side can send another `comm_msg` in reply).

Message type: `comm_msg`:

```
{
  'comm_id' : 'u-u-i-d',
  'data' : {}
}
```

1.10.3 Tearing Down Comms

Since comms live on both sides, when a comm is destroyed the other side must be notified. This is done with a `comm_close` message.

Message type: `comm_close`:

```
{
  'comm_id' : 'u-u-i-d',
  'data' : {}
}
```

1.10.4 Output Side Effects

Since comm messages can execute arbitrary user code, handlers should set the parent header and publish status busy / idle, just like an execute request.

1.11 To Do

Missing things include:

- Important: finish thinking through the payload concept and API.

CHAPTER 2

Making kernels for Jupyter

A 'kernel' is a program that runs and introspects the user's code. IPython includes a kernel for Python code, and people have written kernels for several other languages.

When Jupyter starts a kernel, it passes it a connection file. This specifies how to set up communications with the frontend.

There are two options for writing a kernel:

1. You can reuse the IPython kernel machinery to handle the communications, and just describe how to execute your code. This is much simpler if the target language can be driven from Python. See Making simple Python wrapper kernels for details.
2. You can implement the kernel machinery in your target language. This is more work initially, but the people using your kernel might be more likely to contribute to it if it's in the language they know.

2.1 Connection files

Your kernel will be given the path to a connection file when it starts (see *Kernel specs* for how to specify the command line arguments for your kernel). This file, which is accessible only to the current user, will contain a JSON dictionary looking something like this:

```
{
  "control_port": 50160,
  "shell_port": 57503,
  "transport": "tcp",
  "signature_scheme": "hmac-sha256",
  "stdin_port": 52597,
  "hb_port": 42540,
  "ip": "127.0.0.1",
  "iopub_port": 40885,
  "key": "a0436f6c-1916-498b-8eb9-e81ab9368e84"
}
```

The `transport`, `ip` and five `_port` fields specify five ports which the kernel should bind to using ZeroMQ. For instance, the address of the shell socket in the example above would be:

```
tcp://127.0.0.1:57503
```

New ports are chosen at random for each kernel started.

`signature_scheme` and `key` are used to cryptographically sign messages, so that other users on the system can't send code to run in this kernel. See *The Wire Protocol* for the details of how this signature is calculated.

2.2 Handling messages

After reading the connection file and binding to the necessary sockets, the kernel should go into an event loop, listening on the hb (heartbeat), control and shell sockets.

Heartbeat messages should be echoed back immediately on the same socket - the frontend uses this to check that the kernel is still alive.

Messages on the control and shell sockets should be parsed, and their signature validated. See *The Wire Protocol* for how to do this.

The kernel will send messages on the iopub socket to display output, and on the stdin socket to prompt the user for textual input.

See also:

Messaging in Jupyter Details of the different sockets and the messages that come over them

Creating Language Kernels for IPython A blog post by the author of IHaskell, a Haskell kernel

simple_kernel A simple example implementation of the kernel machinery in Python

2.3 Kernel specs

A kernel identifies itself to IPython by creating a directory, the name of which is used as an identifier for the kernel. These may be created in a number of locations:

	Unix	Windows
System	`/usr/share/jupyter/kernels` `/usr/local/share/jupyter/kernels`	`%PROGRAMDATA%\jupyter\kernels`
Env	`{sys.prefix}/share/jupyter/kernels`	
User	`~/.local/share/jupyter/kernels` (Linux) `~/Library/Jupyter/kernels` (Mac)	`%APPDATA%\jupyter\kernels`

The user location takes priority over the system locations, and the case of the names is ignored, so selecting kernels works the same way whether or not the filesystem is case sensitive.

Other locations may also be searched if the `JUPYTER_PATH` environment variable is set.

Inside the directory, the most important file is *kernel.json*. This should be a JSON serialised dictionary containing the following keys and values:

- **argv**: A list of command line arguments used to start the kernel. The text `{connection_file}` in any argument will be replaced with the path to the connection file.
- **display_name**: The kernel's name as it should be displayed in the UI. Unlike the kernel name used in the API, this can contain arbitrary unicode characters.
- **language**: The name of the language of the kernel. When loading notebooks, if no matching kernelspec key (may differ across machines) is found, a kernel with a matching *language* will be used. This allows a notebook written on any Python or Julia kernel to be properly associated with the user's Python or Julia kernel, even if they aren't listed under the same name as the author's.
- **env** (optional): A dictionary of environment variables to set for the kernel. These will be added to the current environment variables before the kernel is started.

For example, the kernel.json file for IPython looks like this:

```
{
 "argv": ["python3", "-m", "IPython.kernel",
          "-f", "{connection_file}"],
 "display_name": "Python 3",
 "language": "python"
}
```

To see the available kernel specs, run:

```
jupyter kernelspec list
```

To start the terminal console or the Qt console with a specific kernel:

```
jupyter console --kernel bash
jupyter qtconsole --kernel bash
```

The notebook offers you the available kernels in a dropdown menu from the 'New' button.

CHAPTER 3

Making simple Python wrapper kernels

You can re-use IPython's kernel machinery 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

3.1 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 in Jupyter for details of the different message types.

To launch your kernel, add this at the end of your module:

```
if __name__ == '__main__':
    from ipyernel.kernelapp import IPKernelApp
    IPKernelApp.launch_instance(kernel_class=MyKernel)
```

Now create a JSON kernel spec file and install it using `jupyter kernelspec install </path/to/kernel>`. Place your kernel module anywhere Python can import it (try current directory for testing). Finally, you can run your kernel using `jupyter console --kernel <mykernelname>`. Note that `<mykernelname>` in the below example is `echo`.

3.2 Example

`echokernel.py` will simply echo any input it's given to stdout:

```
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:

```
{"argv":["python","-m","echokernel", "-f", "{connection_file}"],
 "display_name":"Echo"
}
```

3.3 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

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

CHAPTER 4

jupyter_client API

4.1 kernelspec - discovering kernels

See also:

Kernel specs

class jupyter_client.kernelspec.**KernelSpec**

argv
The list of arguments to start this kernel.

env
A dictionary of extra environment variables to declare, in addition to the current environment variables, when launching this kernel.

display_name
The name to display for this kernel in UI.

language
The name of the language the kernel implements, to help with picking appropriate kernels when loading notebooks.

resource_dir
The path to the directory with this kernel's resources, such as icons.

to_json()
Serialise this kernelspec to a JSON object.

Returns a string.

class jupyter_client.kernelspec.**KernelSpecManager**

find_kernel_specs()
Returns a dict mapping kernel names to resource directories.

get_kernel_spec(*kernel_name*)
Returns a *KernelSpec* instance for the given kernel_name.

Raises *NoSuchKernel* if the given kernel name is not found.

install_kernel_spec(*source_dir*, *kernel_name=None*, *user=False*, *replace=None*, *prefix=None*)
Install a kernel spec by copying its directory.

If kernel_name is not given, the basename of source_dir will be used.

If `user` is False, it will attempt to install into the systemwide kernel registry. If the process does not have appropriate permissions, an `OSError` will be raised.

If `prefix` is given, the kernelspec will be installed to PREFIX/share/jupyter/kernels/KERNEL_NAME. This can be sys.prefix for installation inside virtual or conda envs.

exception `jupyter_client.kernelspec.`**`NoSuchKernel`**

name
The name of the kernel which was requested.

`jupyter_client.kernelspec.`**`find_kernel_specs`**`()`
`jupyter_client.kernelspec.`**`get_kernel_spec`**`(`*kernel_name*`)`
`jupyter_client.kernelspec.`**`install_kernel_spec`**`(`*source_dir*, *kernel_name=None*, *user=False*, *replace=False*`)`
These methods from *KernelSpecManager* are exposed as functions on the module as well; they will use all the default settings.

4.2 manager - starting, stopping, signalling

class `jupyter_client.`**`KernelManager`**`(`***kwargs*`)`
Manages a single kernel in a subprocess on this host.

This version starts kernels with Popen.

kernel_name
The name of the kernel to launch (see *Kernel specs*).

start_kernel`(`***kw*`)`
Starts a kernel on this host in a separate process.

If random ports (port=0) are being used, this method must be called before the channels are created.

Parameters ****kw** (*optional*) – keyword arguments that are passed down to build the kernel_cmd and launching the kernel (e.g. Popen kwargs).

kernel
Once the kernel has been started, this is the `subprocess.Popen` class for the kernel process.

is_alive`()`
Is the kernel process still running?

interrupt_kernel`()`
Interrupts the kernel by sending it a signal.

Unlike `signal_kernel`, this operation is well supported on all platforms.

signal_kernel`(`*signum*`)`
Sends a signal to the process group of the kernel (this usually includes the kernel and any subprocesses spawned by the kernel).

Note that since only SIGTERM is supported on Windows, this function is only useful on Unix systems.

client`(`***kwargs*`)`
Create a client configured to connect to our kernel

For the client API, see `jupyter_client.client`.

blocking_client`()`
Make a blocking client connected to my kernel

shutdown_kernel(*now=False*, *restart=False*)
Attempts to the stop the kernel process cleanly.

This attempts to shutdown the kernels cleanly by:

1. Sending it a shutdown message over the shell channel.
2. If that fails, the kernel is shutdown forcibly by sending it a signal.

Parameters

- **now** (*bool*) – Should the kernel be forcible killed *now*. This skips the first, nice shutdown attempt.
- **restart** (*bool*) – Will this kernel be restarted after it is shutdown. When this is True, connection files will not be cleaned up.

restart_kernel(*now=False*, ***kw*)
Restarts a kernel with the arguments that were used to launch it.

If the old kernel was launched with random ports, the same ports will be used for the new kernel. The same connection file is used again.

Parameters

- **now** (*bool, optional*) – If True, the kernel is forcefully restarted *immediately*, without having a chance to do any cleanup action. Otherwise the kernel is given 1s to clean up before a forceful restart is issued.

  In all cases the kernel is restarted, the only difference is whether it is given a chance to perform a clean shutdown or not.
- ****kw** (*optional*) – Any options specified here will overwrite those used to launch the kernel.

4.2.1 multikernelmanager - controlling multiple kernels

class jupyter_client.**MultiKernelManager**(***kwargs*)
A class for managing multiple kernels.

This exposes the same methods as `KernelManager`, but their first parameter is a kernel ID, a string identifying the kernel instance. Typically these are UUIDs picked by *start_kernel()*

start_kernel(*kernel_name=None*, ***kwargs*)
Start a new kernel.

The caller can pick a kernel_id by passing one in as a keyword arg, otherwise one will be picked using a uuid.

The kernel ID for the newly started kernel is returned.

list_kernel_ids()
Return a list of the kernel ids of the active kernels.

get_kernel(*kernel_id*)
Get the single KernelManager object for a kernel by its uuid.

Parameters **kernel_id** (*uuid*) – The id of the kernel.

remove_kernel(*kernel_id*)
remove a kernel from our mapping.

Mainly so that a kernel can be removed if it is already dead, without having to call shutdown_kernel.

The kernel object is returned.

shutdown_all(*now=False*)
Shutdown all kernels.

4.2.2 Utility functions

jupyter_client.**run_kernel**(**args*, ***kwds*)
Context manager to create a kernel in a subprocess.

The kernel is shut down when the context exits.

Returns **kernel_client**

Return type connected KernelClient instance

4.3 client - communicating with kernels

See also:

Messaging in Jupyter The Jupyter messaging specification

class jupyter_client.**KernelClient**(***kwargs*)
Communicates with a single kernel on any host via zmq channels.

There are four channels associated with each kernel:

•shell: for request/reply calls to the kernel.

•iopub: for the kernel to publish results to frontends.

•hb: for monitoring the kernel's heartbeat.

•stdin: for frontends to reply to raw_input calls in the kernel.

The messages that can be sent on these channels are exposed as methods of the client (KernelClient.execute, complete, history, etc.). These methods only send the message, they don't wait for a reply. To get results, use e.g. *get_shell_msg()* to fetch messages from the shell channel.

execute(*code*, *silent=False*, *store_history=True*, *user_expressions=None*, *allow_stdin=None*, *stop_on_error=True*)
Execute code in the kernel.

Parameters

- **code** (*str*) – A string of code in the kernel's language.
- **silent** (*bool, optional (default False)*) – If set, the kernel will execute the code as quietly possible, and will force store_history to be False.
- **store_history** (*bool, optional (default True)*) – If set, the kernel will store command history. This is forced to be False if silent is True.
- **user_expressions** (*dict, optional*) – A dict mapping names to expressions to be evaluated in the user's dict. The expression values are returned as strings formatted using `repr()`.
- **allow_stdin** (*bool, optional (default self.allow_stdin)*) – Flag for whether the kernel can send stdin requests to frontends.

  Some frontends (e.g. the Notebook) do not support stdin requests. If raw_input is called from code executed from such a frontend, a StdinNotImplementedError will be raised.

- **stop_on_error** (*bool, optional (default True)*) – Flag whether to abort the execution queue, if an exception is encountered.

Returns

Return type The msg_id of the message sent.

complete(*code*, *cursor_pos=None*)

Tab complete text in the kernel's namespace.

Parameters

- **code** (*str*) – The context in which completion is requested. Can be anything between a variable name and an entire cell.
- **cursor_pos** (*int, optional*) – The position of the cursor in the block of code where the completion was requested. Default: `len(code)`

Returns

Return type The msg_id of the message sent.

inspect(*code*, *cursor_pos=None*, *detail_level=0*)

Get metadata information about an object in the kernel's namespace.

It is up to the kernel to determine the appropriate object to inspect.

Parameters

- **code** (*str*) – The context in which info is requested. Can be anything between a variable name and an entire cell.
- **cursor_pos** (*int, optional*) – The position of the cursor in the block of code where the info was requested. Default: `len(code)`
- **detail_level** (*int, optional*) – The level of detail for the introspection (0-2)

Returns

Return type The msg_id of the message sent.

history(*raw=True*, *output=False*, *hist_access_type='range'*, ***kwargs*)

Get entries from the kernel's history list.

Parameters

- **raw** (*bool*) – If True, return the raw input.
- **output** (*bool*) – If True, then return the output as well.
- **hist_access_type** (*str*) – 'range' (fill in session, start and stop params), 'tail' (fill in n) or 'search' (fill in pattern param).
- **session** (*int*) – For a range request, the session from which to get lines. Session numbers are positive integers; negative ones count back from the current session.
- **start** (*int*) – The first line number of a history range.
- **stop** (*int*) – The final (excluded) line number of a history range.
- **n** (*int*) – The number of lines of history to get for a tail request.
- **pattern** (*str*) – The glob-syntax pattern for a search request.

Returns

Return type The ID of the message sent.

comm_info(*target_name=None*)
Request comm info.

is_complete(*code*)
Ask the kernel whether some code is complete and ready to execute.

input(*string*)
Send a string of raw input to the kernel.

This should only be called in response to the kernel sending an `input_request` message on the stdin channel.

shutdown(*restart=False*)
Request an immediate kernel shutdown.

Upon receipt of the (empty) reply, client code can safely assume that the kernel has shut down and it's safe to forcefully terminate it if it's still alive.

The kernel will send the reply via a function registered with Python's atexit module, ensuring it's truly done as the kernel is done with all normal operation.

get_shell_msg(**args*, ***kwargs*)
Get a message from the shell channel

get_iopub_msg(**args*, ***kwargs*)
Get a message from the iopub channel

get_stdin_msg(**args*, ***kwargs*)
Get a message from the stdin channel

CHAPTER 5

Changes in Jupyter Client

5.1 4.1

5.1.1 4.1.0

4.1.0 on GitHub

Highlights:

- Setuptools fixes for *jupyter kernelspec*
- add `KernelManager.blocking_client()`
- provisional implementation of `comm_info` requests from upcoming 5.1 release of the protocol.

5.2 4.0

The first release of Jupyter Client as its own package.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

j

L

M

N

R

S

T