Jupyter Qt Console Documentation

Release 4.3.0

Jupyter Development Team

March 29, 2017

Contents

Release 4.3.0

Date March 29, 2017

To start the Qt console:

```
$ jupyter qtconsole
```

Installation

The Qt console requires Qt, such as PyQt5, PyQt4, or PySide.

Although pip and conda may be used to install the Qt console, conda is simpler to use since it automatically installs PyQt. Alternatively, qtconsole installation with pip needs additional steps since pip cannot install the Qt requirement.

Install using conda

To install:

```
conda install qtconsole
```

Note: If the Qt console is installed using conda, it will **automatically** install the Qt requirement as well.

Install using pip

To install:

```
pip install qtconsole
```

Important: Make sure that Qt is installed. Unfortunately, Qt cannot be installed using pip. The next section gives instructions on installing Qt.

Installing Qt (if needed)

We recommend installing PyQt with conda:

```
conda install pyqt
```

or with a system package manager. For Windows, PyQt binary packages may be used.

For example with Linux Debian's system package manager, use:

```
sudo apt-get install python3-pyqt5   # PyQt5 on Python 3
sudo apt-get install python3-pyqt4   # PyQt4 on Python 3
sudo apt-get install python-qt4      # PyQt4 on Python 2
```

See also:

Installing Jupyter The Qt console is part of the Jupyter ecosystem.

Configuration options

These options can be set in the configuration file, `~/.jupyter/jupyter_qtconsole_config.py`, or at the command line when you start Qt console.

You may enter `jupyter qtconsole --help-all` to get information about all available configuration options.

Options

ConnectionFileMixin.connection_file [Unicode] Default: `''`

> JSON file in which to store connection info [default: kernel-<pid>.json]
>
> This file will contain the IP, ports, and authentication key needed to connect clients to this kernel. By default, this file will be created in the security dir of the current profile, but can be specified by absolute path.

ConnectionFileMixin.control_port [Int] Default: `0`

> set the control (ROUTER) port [default: random]

ConnectionFileMixin.hb_port [Int] Default: `0`

> set the heartbeat port [default: random]

ConnectionFileMixin.iopub_port [Int] Default: `0`

> set the iopub (PUB) port [default: random]

ConnectionFileMixin.ip [Unicode] Default: `''`

> Set the kernel's IP address [default localhost]. If the IP address is something other than localhost, then Consoles on other machines will be able to connect to the Kernel, so be careful!

ConnectionFileMixin.shell_port [Int] Default: `0`

> set the shell (ROUTER) port [default: random]

ConnectionFileMixin.stdin_port [Int] Default: `0`

> set the stdin (ROUTER) port [default: random]

ConnectionFileMixin.transport ['tcp'|'ipc'] Default: `'tcp'`

>   No description

JupyterConsoleApp.confirm_exit [CBool] Default: `True`

>   Set to display confirmation dialog on exit. You can always use 'exit' or 'quit', to force a direct exit without any confirmation.

JupyterConsoleApp.existing [CUnicode] Default: `''`

>   Connect to an already running kernel

JupyterConsoleApp.kernel_name [Unicode] Default: `'python'`

>   The name of the default kernel to start.

JupyterConsoleApp.sshkey [Unicode] Default: `''`

>   Path to the ssh key to use for logging in to the ssh server.

JupyterConsoleApp.sshserver [Unicode] Default: `''`

>   The SSH server to use to connect to the kernel.

Application.log_datefmt [Unicode] Default: `'%Y-%m-%d %H:%M:%S'`

>   The date format used by logging formatters for %(asctime)s

Application.log_format [Unicode] Default: `'[%(name)s]%(highlevel)s %(message)s'`

>   The Logging format template

Application.log_level [0|10|20|30|40|50|'DEBUG'|'INFO'|'WARN'|'ERROR'|'CRITICAL'] Default: `30`

>   Set the log level by value or name.

JupyterApp.answer_yes [Bool] Default: `False`

>   Answer yes to any prompts.

JupyterApp.config_file [Unicode] Default: `''`

>   Full path of a config file.

JupyterApp.config_file_name [Unicode] Default: `''`

>   Specify a config file to load.

JupyterApp.generate_config [Bool] Default: `False`

>   Generate default config file.

JupyterQtConsoleApp.display_banner [CBool] Default: `True`

>   Whether to display a banner upon starting the QtConsole.

JupyterQtConsoleApp.hide_menubar [CBool] Default: `False`

>   Start the console window with the menu bar hidden.

JupyterQtConsoleApp.maximize [CBool] Default: `False`

>   Start the console window maximized.

JupyterQtConsoleApp.plain [CBool] Default: `False`

>   Use a plaintext widget instead of rich text (plain can't print/save).

JupyterQtConsoleApp.stylesheet [Unicode] Default: `''`

> path to a custom CSS stylesheet

ConsoleWidget.ansi_codes [Bool] Default: `True`

> Whether to process ANSI escape codes.

ConsoleWidget.buffer_size [Int] Default: `500`

> The maximum number of lines of text before truncation. Specifying a non-positive number disables text truncation (not recommended).

ConsoleWidget.console_height [Int] Default: `25`

> The height of the console at start time in number of characters (will double with `vsplit` paging)

ConsoleWidget.console_width [Int] Default: `81`

> The width of the console at start time in number of characters (will double with `hsplit` paging)

ConsoleWidget.execute_on_complete_input [Bool] Default: `True`

> Whether to automatically execute on syntactically complete input.
>
> If False, Shift-Enter is required to submit each execution. Disabling this is mainly useful for non-Python kernels, where the completion check would be wrong.

ConsoleWidget.font_family [Unicode] Default: `''`

> The font family to use for the console. On OSX this defaults to Monaco, on Windows the default is Consolas with fallback of Courier, and on other platforms the default is Monospace.

ConsoleWidget.font_size [Int] Default: `0`

> The font size. If unconfigured, Qt will be entrusted with the size of the font.

ConsoleWidget.gui_completion ['plain'|'droplist'|'ncurses'] Default: `'ncurses'`

> The type of completer to use. Valid values are:
>
> **'plain'** [Show the available completion as a text list] Below the editing area.
>
> **'droplist': Show the completion in a drop down list navigable** by the arrow keys, and from which you can select completion by pressing Return.
>
> **'ncurses'** [Show the completion as a text list which is navigable by] `tab` and arrow keys.

ConsoleWidget.include_other_output [Bool] Default: `False`

> Whether to include output from clients other than this one sharing the same kernel.
>
> Outputs are not displayed until enter is pressed.

ConsoleWidget.kind ['plain'|'rich'] Default: `'plain'`

> The type of underlying text widget to use. Valid values are 'plain', which specifies a QPlainTextEdit, and 'rich', which specifies a QTextEdit.

ConsoleWidget.paging ['inside'|'hsplit'|'vsplit'|'custom'|'none'] Default: `'inside'`

> The type of paging to use. Valid values are:
>
> **'inside'** The widget pages like a traditional terminal.
>
> **'hsplit'** When paging is requested, the widget is split horizontally. The top pane contains the console, and the bottom pane contains the paged text.
>
> **'vsplit'** Similar to 'hsplit', except that a vertical splitter is used.

'custom' No action is taken by the widget beyond emitting a 'custom_page_requested(str)' signal.

'none' The text is written directly to the console.

HistoryConsoleWidget.history_lock [Bool] Default: `False`

    No description

FrontendWidget.banner [Unicode] Default: `''`

    No description

FrontendWidget.clear_on_kernel_restart [Bool] Default: `True`

    Whether to clear the console when the kernel is restarted

FrontendWidget.confirm_restart [Bool] Default: `True`

    Whether to ask for user confirmation when restarting kernel

FrontendWidget.enable_calltips [Bool] Default: `True`

    Whether to draw information calltips on open-parentheses.

FrontendWidget.is_complete_timeout [Float] Default: `0.25`

    Seconds to wait for is_complete replies from the kernel.

FrontendWidget.lexer_class [DottedObjectName] Default: `traitlets.Undefined`

    The pygments lexer class to use.

JupyterWidget.editor [Unicode] Default: `''`

    A command for invoking a system text editor. If the string contains a {filename} format specifier, it will be used. Otherwise, the filename will be appended to the end the command.

JupyterWidget.editor_line [Unicode] Default: `''`

    The editor command to use when a specific line number is requested. The string should contain two format specifiers: {line} and {filename}. If this parameter is not specified, the line number option to the %edit magic will be ignored.

JupyterWidget.in_prompt [Unicode] Default: `'In [<span class="in-prompt-number">%i</span>]: '`

    No description

JupyterWidget.input_sep [Unicode] Default: `'\\n'`

    No description

JupyterWidget.out_prompt [Unicode] Default: `'Out[<span class="out-prompt-number">%i</span>]: '`

    No description

JupyterWidget.output_sep [Unicode] Default: `''`

    No description

JupyterWidget.output_sep2 [Unicode] Default: `''`

    No description

JupyterWidget.style_sheet [Unicode] Default: `''`

    A CSS stylesheet. The stylesheet can contain classes for:

        1. Qt: QPlainTextEdit, QFrame, QWidget, etc

2. Pygments: .c, .k, .o, etc. (see PygmentsHighlighter)

3. QtConsole: .error, .in-prompt, .out-prompt, etc

JupyterWidget.syntax_style [Unicode] Default: `''`

If not empty, use this Pygments style for syntax highlighting. Otherwise, the style sheet is queried for Pygments style information.

KernelManager.autorestart [Bool] Default: `True`

Should we autorestart the kernel if it dies.

KernelManager.kernel_cmd [List] Default: `[]`

DEPRECATED: Use kernel_name instead.

The Popen Command to launch the kernel. Override this if you have a custom kernel. If kernel_cmd is specified in a configuration file, Jupyter does not pass any arguments to the kernel, because it cannot make any assumptions about the arguments that the kernel understands. In particular, this means that the kernel does not receive the option –debug if it given on the Jupyter command line.

KernelManager.shutdown_wait_time [Float] Default: `5.0`

Time to wait for a kernel to terminate before killing it, in seconds.

KernelRestarter.debug [Bool] Default: `False`

Whether to include every poll event in debugging output.

Has to be set explicitly, because there will be *a lot* of output.

KernelRestarter.restart_limit [Int] Default: `5`

The number of consecutive autorestarts before the kernel is presumed dead.

KernelRestarter.time_to_dead [Float] Default: `3.0`

Kernel heartbeat interval in seconds.

Session.buffer_threshold [Int] Default: `1024`

Threshold (in bytes) beyond which an object's buffer should be extracted to avoid pickling.

Session.check_pid [Bool] Default: `True`

Whether to check PID to protect against calls after fork.

This check can be disabled if fork-safety is handled elsewhere.

Session.copy_threshold [Int] Default: `65536`

Threshold (in bytes) beyond which a buffer should be sent without copying.

Session.debug [Bool] Default: `False`

Debug output in the Session

Session.digest_history_size [Int] Default: `65536`

The maximum number of digests to remember.

The digest history will be culled when it exceeds this value.

Session.item_threshold [Int] Default: `64`

The maximum number of items for a container to be introspected for custom serialization. Containers larger than this are pickled outright.

Session.key [CBytes] Default: `b''`

>   execution key, for signing messages.

Session.keyfile [Unicode] Default: `''`

>   path to file containing execution key.

Session.metadata [Dict] Default: `{}`

>   Metadata dictionary, which serves as the default top-level metadata dict for each message.

Session.packer [DottedObjectName] Default: `'json'`

>   The name of the packer for serializing messages. Should be one of 'json', 'pickle', or an import name for a custom callable serializer.

Session.session [CUnicode] Default: `''`

>   The UUID identifying this session.

Session.signature_scheme [Unicode] Default: `'hmac-sha256'`

>   The digest scheme used to construct the message signatures. Must have the form 'hmac-HASH'.

Session.unpacker [DottedObjectName] Default: `'json'`

>   The name of the unpacker for unserializing messages. Only used with custom functions for `packer`.

Session.username [Unicode] Default: `'username'`

>   Username for the Session. Default is your system username.

CHAPTER 3

Changes in Jupyter Qt console

4.3

4.3 on GitHub

- Rename `ConsoleWidget.width/height` traits to `console_width/console_height` to avoid a name clash with the `QWidget` properties. Note: the name change could be, in rare cases if a name collision exists, a code-breaking change.

Additions

- Add `Shift-Tab` shortcut to unindent text
- Add `Control-R` shortcut to rename the current tab
- Add `Alt-R` shortcut to set the main window title
- Add `Command-Alt-Left` and `Command-Alt-Right` shortcut to switch tabs on macOS
- Add support for PySide2
- Add support for Python 3.5
- Add support for 24 bit ANSI color codes
- Add option to create new tab connected to the existing kernel

Changes

- Change `Tab` key behavior to always indent to the next increment of 4 spaces
- Change `Home` key behavior to alternate cursor between the beginning of text (ignoring leading spaces) and beginning of the line
- Improve documentation of various options and clarified the docs in some places

11

- Move documentation to ReadTheDocs

Fixes

- Fix automatic indentation of new lines that are inserted in the middle of a cell
- Fix regression where prompt would never be shown for `--existing` consoles
- Fix `python.exe -m qtconsole` on Windows
- Fix showing error messages when running a script using `%run`
- Fix `invalid cursor position` error and subsequent freezing of user input
- Fix syntax coloring when attaching to non-IPython kernels
- Fix printing when using QT5
- Fix `Control-K` shortcut (delete until end of line) on macOS
- Fix history browsing (`Up`/`Down` keys) when lines are longer than the terminal width
- Fix saving HTML with inline PNG for Python 3
- Various internal bugfixes

4.2

4.2 on GitHub

- various latex display fixes
- improvements for embedding in Qt applications (use existing Qt API if one is already loaded)

4.1

4.1.1

4.1.1 on GitHub

- Set AppUserModelID for taskbar icon on Windows 7 and later

4.1.0

4.1 on GitHub

- fix regressions in copy/paste, completion
- fix issues with inprocess IPython kernel
- fix `jupyter qtconsole --generate-config`

4.0

4.0.1

- fix installation issues, including setuptools entrypoints for Windows
- Qt5 fixes

4.0.0

4.0 on GitHub

First release of the Qt console as a standalone package.

Overview

The Qt console is a very lightweight application that largely feels like a terminal, but provides a number of enhancements only possible in a GUI, such as inline figures, proper multi-line editing with syntax highlighting, graphical calltips, and much more. The Qt console can use any Jupyter kernel.

The Qt console frontend has hand-coded emacs-style bindings for text navigation. This is not yet configurable.

Tip: Since the Qt console tries hard to behave like a terminal, by default it immediately executes single lines of input that are complete. If you want to force multi-line input, hit `Ctrl-Enter` at the end of the first line instead of `Enter`, and it will open a new line for input. At any point in a multi-line block, you can force its execution (without having to go to the bottom) with `Shift-Enter`.
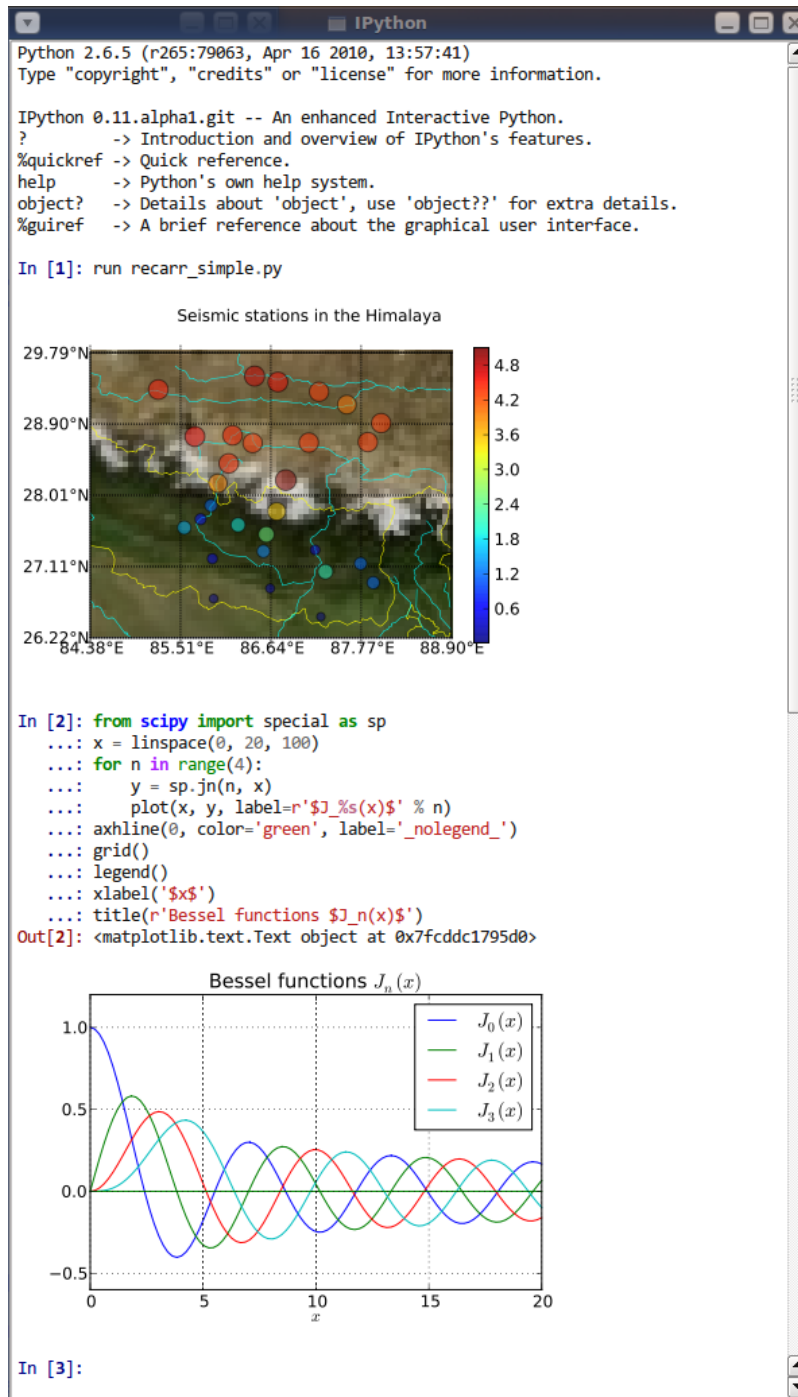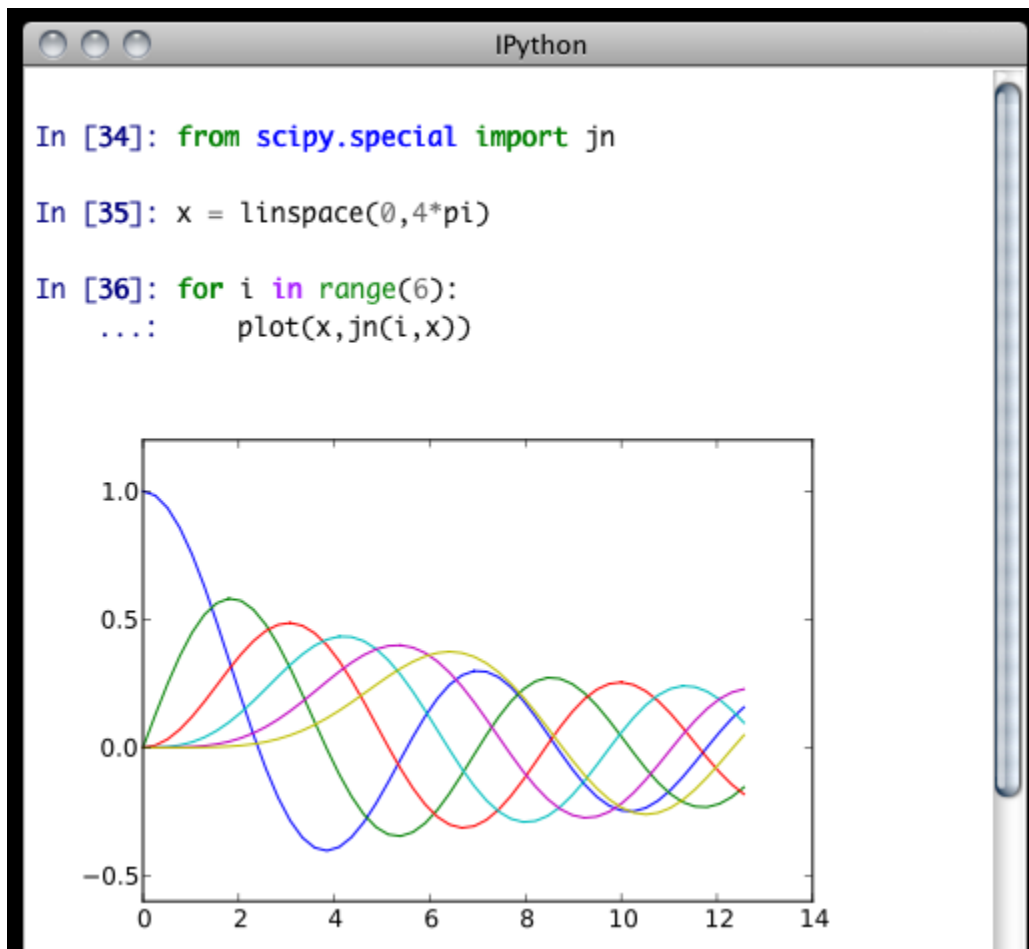

Fig. 4.1: The Qt console with IPython, using inline matplotlib plots.

Inline graphics

One of the most exciting features of the Qt Console is embedded figures. You can plot with matplotlib in IPython, or the plotting library of choice in your kernel.

Saving and Printing

The Qt Console has the ability to save your current session, as either HTML or XHTML. Your inline figures will be PNG in HTML, or inlined as SVG in XHTML. PNG images have the option to be either in an external folder, as in many browsers' "Webpage, Complete" option, or inlined as well, for a larger, but more portable file.

Note: Export to SVG+XHTML requires that you are using SVG figures, which is *not* the default. To switch the inline figure format in IPython to use SVG, do:

```
In [10]: %config InlineBackend.figure_format = 'svg'
```

Or, you can add the same line (c.Inline... instead of %config Inline...) to your config files.

This will only affect figures plotted after making this call

The widget also exposes the ability to print directly, via the default print shortcut or context menu.

See these examples of png/html and svg/xhtml output. Note that syntax highlighting does not survive export. This is a known issue, and is being investigated.

Colors and Highlighting

Terminal IPython has always had some coloring, but never syntax highlighting. There are a few simple color choices, specified by the `colors` flag or `%colors` magic:

- LightBG for light backgrounds
- Linux for dark backgrounds
- NoColor for a simple colorless terminal

The Qt widget, however, has full syntax highlighting as you type, handled by the pygments library. The `style` argument exposes access to any style by name that can be found by pygments, and there are several already installed.

Screenshot of `jupyter qtconsole --style monokai`, which uses the 'monokai' theme:

Note: Calling `jupyter qtconsole -h` will show all the style names that pygments can find on your system.

You can also pass the filename of a custom CSS stylesheet, if you want to do your own coloring, via the `stylesheet` argument. The default LightBG stylesheet:

```css
QPlainTextEdit, QTextEdit { background-color: white;
        color: black ;
        selection-background-color: #ccc}
.error { color: red; }
.in-prompt { color: navy; }
.in-prompt-number { font-weight: bold; }
.out-prompt { color: darkred; }
.out-prompt-number { font-weight: bold; }
/* .inverted is used to highlight selected completion */
.inverted { background-color: black ; color: white; }
```

Fonts

The Qt console is configurable via the ConsoleWidget. To change these, set the `font_family` or `font_size` traits of the ConsoleWidget. For instance, to use 9pt Anonymous Pro:

```
$> jupyter qtconsole --ConsoleWidget.font_family="Anonymous Pro" --ConsoleWidget.font_
↪size=9
```

Process Management

With the two-process ZMQ model, the frontend does not block input during execution. This means that actions can be taken by the frontend while the Kernel is executing, or even after it crashes. The most basic such command is via 'Ctrl-.', which restarts the kernel. This can be done in the middle of a blocking execution. The frontend can also know, via a heartbeat mechanism, that the kernel has died. This means that the frontend can safely restart the kernel.

Multiple Consoles

Since the Kernel listens on the network, multiple frontends can connect to it. These do not have to all be qt frontends - any Jupyter frontend can connect and run code.

Other frontends can connect to your kernel, and share in the execution. This is great for collaboration. The `--existing` flag means connect to a kernel that already exists. Starting other consoles with that flag will not try to start their own kernel, but rather connect to yours. `kernel-12345.json` is a small JSON file with the ip, port, and authentication information necessary to connect to your kernel. By default, this file will be in your Jupyter runtime directory. If it is somewhere else, you will need to use the full path of the connection file, rather than just its filename.

If you need to find the connection info to send, and don't know where your connection file lives, there are a couple of ways to get it. If you are already running a console connected to an IPython kernel, you can use the `%connect_info` magic to display the information necessary to connect another frontend to the kernel.

```
In [2]: %connect_info
{
  "stdin_port":50255,
  "ip":"127.0.0.1",
  "hb_port":50256,
  "key":"70be6f0f-1564-4218-8cda-31be40a4d6aa",
  "shell_port":50253,
  "iopub_port":50254
}

Paste the above JSON into a file, and connect with:
```

```
    $> ipython <app> --existing <file>
or, if you are local, you can connect with just:
    $> ipython <app> --existing kernel-12345.json
or even just:
    $> ipython <app> --existing
if this is the most recent kernel you have started.
```

Otherwise, you can find a connection file by name (and optionally profile) with `jupyter_client.find_connection_file()`:

```
$> python -c "from jupyter_client import find_connection_file;\
print(find_connection_file('kernel-12345.json'))"
/home/you/Library/Jupyter/runtime/kernel-12345.json
```

Security

> **Warning:** Since the ZMQ code currently has no encryption, listening on an external-facing IP is dangerous. You are giving any computer that can see you on the network the ability to connect to your kernel, and view your traffic. Read the rest of this section before listening on external ports or running a kernel on a shared machine.

By default (for security reasons), the kernel only listens on localhost, so you can only connect multiple frontends to the kernel from your local machine. You can specify to listen on an external interface by specifying the `ip` argument:

```
$> jupyter qtconsole --ip=192.168.1.123
```

If you specify the ip as 0.0.0.0 or '*', that means all interfaces, so any computer that can see yours on the network can connect to the kernel.

Messages are not encrypted, so users with access to the ports your kernel is using will be able to see any output of the kernel. They will **NOT** be able to issue shell commands as you due to message signatures.

> **Warning:** If you disable message signatures, then any user with access to the ports your kernel is listening on can issue arbitrary code as you. **DO NOT** disable message signatures unless you have a lot of trust in your environment.

The one security feature Jupyter does provide is protection from unauthorized execution. Jupyter's messaging system will sign messages with HMAC digests using a shared-key. The key is never sent over the network, it is only used to generate a unique hash for each message, based on its content. When the kernel receives a message, it will check that the digest matches, and discard the message. You can use any file that only you have access to to generate this key, but the default is just to generate a new UUID.

SSH Tunnels

Sometimes you want to connect to machines across the internet, or just across a LAN that either doesn't permit open ports or you don't trust the other machines on the network. To do this, you can use SSH tunnels. SSH tunnels are a way to securely forward ports on your local machine to ports on another machine, to which you have SSH access.

In simple cases, Jupyter's tools can forward ports over ssh by simply adding the `--ssh=remote` argument to the usual `--existing...` set of flags for connecting to a running kernel, after copying the JSON connection file (or its contents) to the second computer.

> **Warning:** Using SSH tunnels does *not* increase localhost security. In fact, when tunneling from one machine to
> another *both* machines have open ports on localhost available for connections to the kernel.

There are two primary models for using SSH tunnels with Jupyter. The first is to have the Kernel listen only on
localhost, and connect to it from another machine on the same LAN.

First, let's start a kernel on machine **worker**, listening only on loopback:

```
user@worker $> ipython kernel
[IPKernelApp] To connect another client to this kernel, use:
[IPKernelApp] --existing kernel-12345.json
```

In this case, the IP that you would connect to would still be 127.0.0.1, but you want to specify the additional `--ssh`
argument with the hostname of the kernel (in this example, it's 'worker'):

```
user@client $> jupyter qtconsole  --ssh=worker --existing /path/to/kernel-12345.json
```

Which will write a new connection file with the forwarded ports, so you can reuse them:

```
[JupyterQtConsoleApp] To connect another client via this tunnel, use:
[JupyterQtConsoleApp] --existing kernel-12345-ssh.json
```

Note again that this opens ports on the *client* machine that point to your kernel.

> **Note:** the ssh argument is simply passed to openssh, so it can be fully specified `user@host:port` but it will also
> respect your aliases, etc. in `.ssh/config` if you have any.

The second pattern is for connecting to a machine behind a firewall across the internet (or otherwise wide network).
This time, we have a machine **login** that you have ssh access to, which can see **kernel**, but **client** is on another network.
The important difference now is that **client** can see **login**, but *not* **worker**. So we need to forward ports from client to
worker *via* login. This means that the kernel must be started listening on external interfaces, so that its ports are visible
to `login`:

```
user@worker $> ipython kernel --ip=0.0.0.0
[IPKernelApp] To connect another client to this kernel, use:
[IPKernelApp] --existing kernel-12345.json
```

Which we can connect to from the client with:

```
user@client $> jupyter qtconsole --ssh=login --ip=192.168.1.123 --existing /path/to/
→kernel-12345.json
```

> **Note:** The IP here is the address of worker as seen from *login*, and need only be specified if the kernel used the
> ambiguous 0.0.0.0 (all interfaces) address. If it had used 192.168.1.123 to start with, it would not be needed.

Manual SSH tunnels

It's possible that Jupyter's ssh helper functions won't work for you, for various reasons. You can still connect to remote
machines, as long as you set up the tunnels yourself. The basic format of forwarding a local port to a remote one is:

```
[client] $> ssh <server> <localport>:<remoteip>:<remoteport> -f -N
```

This will forward local connections to **localport** on client to **remoteip:remoteport** *via* **server**. Note that remoteip is interpreted relative to *server*, not the client. So if you have direct ssh access to the machine to which you want to forward connections, then the server *is* the remote machine, and remoteip should be server's IP as seen from the server itself, i.e. 127.0.0.1. Thus, to forward local port 12345 to remote port 54321 on a machine you can see, do:

```
[client] $> ssh machine 12345:127.0.0.1:54321 -f -N
```

But if your target is actually on a LAN at 192.168.1.123, behind another machine called **login**, then you would do:

```
[client] $> ssh login 12345:192.168.1.16:54321 -f -N
```

The `-f -N` on the end are flags that tell ssh to run in the background, and don't actually run any commands beyond creating the tunnel.

See also:

A short discussion of ssh tunnels: http://www.revsys.com/writings/quicktips/ssh-tunnel.html

Stopping Kernels and Consoles

Since there can be many consoles per kernel, the shutdown mechanism and dialog are probably more complicated than you are used to. Since you don't always want to shutdown a kernel when you close a window, you are given the option to just close the console window or also close the Kernel and *all other windows*. Note that this only refers to all other *local* windows, as remote Consoles are not allowed to shutdown the kernel, and shutdowns do not close Remote consoles (to allow for saving, etc.).

Rules:

- Restarting the kernel automatically clears all *local* Consoles, and prompts remote Consoles about the reset.

- Shutdown closes all *local* Consoles, and notifies remotes that the Kernel has been shutdown.

- Remote Consoles may not restart or shutdown the kernel.

Qt and the REPL

Note: This section is relevant regardless of the frontend you use to write Qt Code. This section is mostly there as it is easy to get confused and assume that writing Qt code in the QtConsole should change from usual Qt code. It should not. If you get confused, take a step back, and try writing your code using the pure terminal based `jupyter console` that does not involve Qt.

An important part of working with the REPL – QtConsole, Jupyter notebook, IPython terminal – when you are writing your own Qt code is to remember that user code (in the kernel) is *not* in the same process as the frontend. This means that there is not necessarily any Qt code running in the kernel, and under most normal circumstances there isn't. This is true even if you are running the QtConsole.

Warning: When executing code from the qtconsole prompt, it is **not possible** to access the QtApplication instance of the QtConsole itself.

A common problem listed in the PyQt4 Gotchas is the fact that Python's garbage collection will destroy Qt objects (Windows, etc.) once there is no longer a Python reference to them, so you have to hold on to them. For instance, in:

```python
from PyQt4 import QtGui

def make_window():
    win = QtGui.QMainWindow()

def make_and_return_window():
    win = QtGui.QMainWindow()
    return win
```

`make_window()` will never draw a window, because garbage collection will destroy it before it is drawn, whereas `make_and_return_window()` lets the caller decide when the window object should be destroyed. If, as a developer, you know that you always want your objects to last as long as the process, you can attach them to the `QApplication` instance itself:

```
from PyQt4 import QtGui, QtCore

# do this just once:
app = QtCore.QCoreApplication.instance()
if not app:
    # we are in the kernel in most of the case there is NO qt code running.
    # we need to create a Gui APP.
    app = QtGui.QApplication([])
app.references = set()
# then when you create Windows, add them to the set
def make_window():
    win = QtGui.QMainWindow()
    app.references.add(win)
```

Now the `QApplication` itself holds a reference to `win`, so it will never be garbage collected until the application itself is destroyed.

Embedding the QtConsole in a Qt application

In order to make the QtConsole available to an external Qt GUI application (just as `IPython.embed()` enables one to embed a terminal session of IPython in a command-line application), there are a few options:

- First start IPython, and then start the external Qt application from IPython, as described above. Effectively, this embeds your application in IPython rather than the other way round.

- Use `qtconsole.rich_jupyter_widget.RichJupyterWidget` in your Qt application. This will embed the console widget in your GUI and start the kernel in a separate process, so code typed into the console cannot access objects in your application.

- Start a standard IPython kernel in the process of the external Qt application. See `examples/Embedding/ipkernel_qtapp.py` for an example. Due to IPython's two-process model, the QtConsole itself will live in another process with its own QApplication, and thus cannot be embedded in the main GUI.

- Start a special IPython kernel, the `IPython.kernel.inprocess.ipkernel.InProcessKernel`, that allows a QtConsole in the same process. See `examples/Embedding/inprocess_qtconsole.py` for an example. While the QtConsole can now be embedded in the main GUI, one cannot connect to the kernel from other consoles as there are no real ZMQ sockets anymore.

Regressions

There are some features, where the qt console lags behind the Terminal frontend:

- !cmd input: Due to our use of pexpect, we cannot pass input to subprocesses launched using the '!' escape, so you should never call a command that requires interactive input. For such cases, use the terminal IPython. This will not be fixed, as abandoning pexpect would significantly degrade the console experience.