Jupyter Documentation

Release 4.1.1 alpha

https://jupyter.org

Mar 02, 2017

Contents

Table of Contents

CHAPTER 1

Jupyter Notebook Quickstart

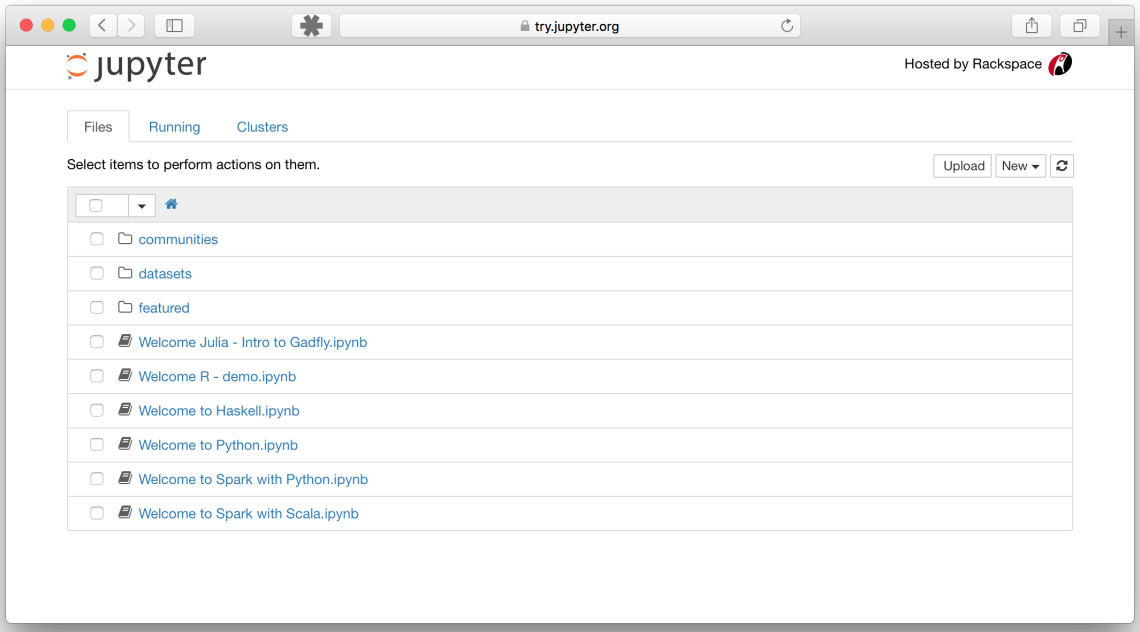
Try Jupyter

Contents

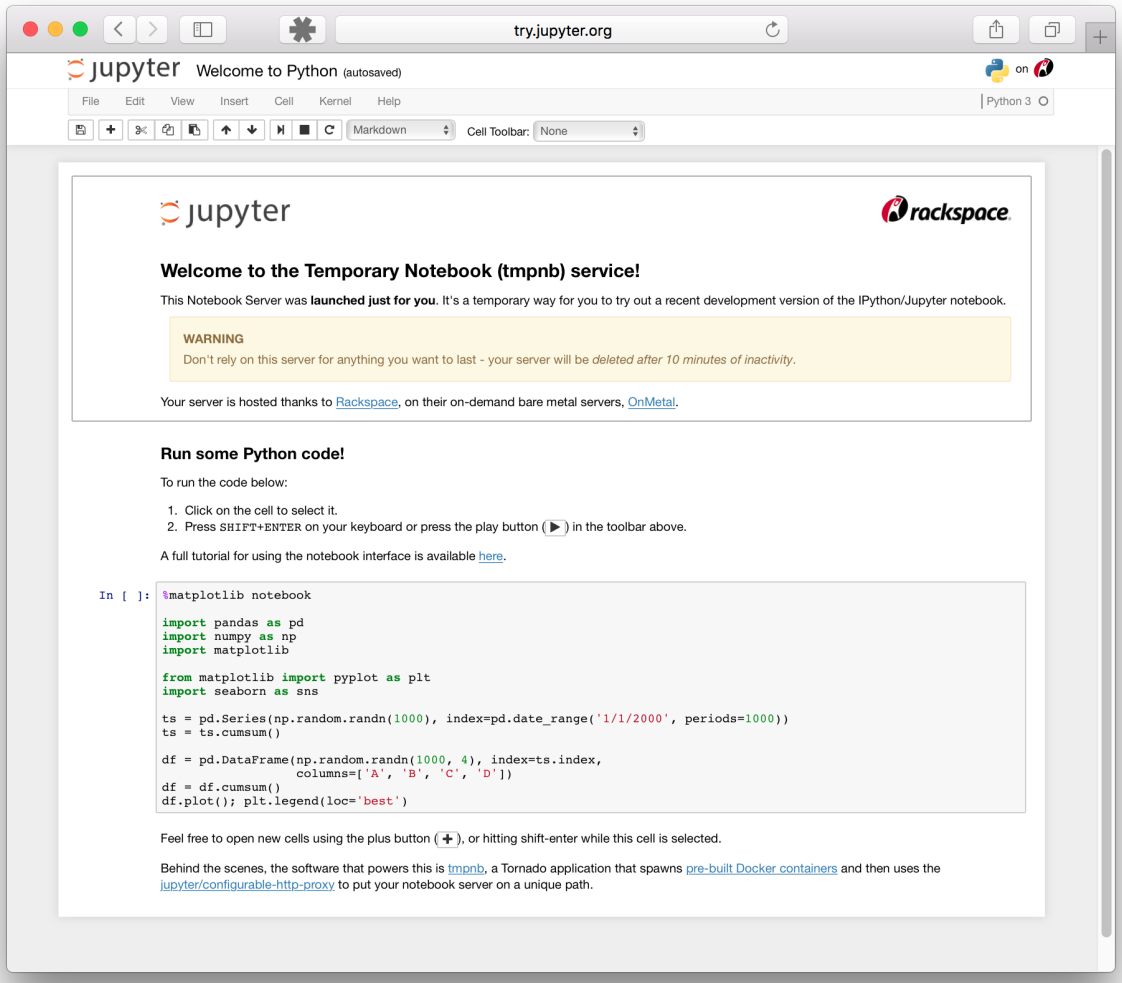
Try in Your Browser. No Installation Needed.

Go to https://try.jupyter.org. No installation is needed.

Notebook Dashboard



Notebook Editor



Are You Ready to Install Jupyter?

If you have tried Jupyter and like it, please use our detailed *Installation Guide* to install Jupyter on your computer.

Installing Jupyter Notebook

Contents

This information explains how to install the Jupyter Notebook and the IPython kernel.

Prerequisite: Python

While Jupyter runs code in many programming languages, **Python** is a requirement (Python 3.3 or greater, or Python 2.7) for installing the Jupyter Notebook.

We recommend using the Anaconda distribution to install Python and Jupyter. We'll go through its installation in the next section.

Installing Jupyter using Anaconda and conda

For new users, we **highly recommend** installing Anaconda. Anaconda conveniently installs Python, the Jupyter Notebook, and other commonly used packages for scientific computing and data science.

Use the following installation steps:

1. Download Anaconda. We recommend downloading Anaconda's latest Python 3 version (currently Python 3.5).
2. Install the version of Anaconda which you downloaded, following the instructions on the download page.
3. Congratulations, you have installed Jupyter Notebook. To run the notebook:

```
jupyter notebook
```

See *Running the Notebook* for more details.

Alternative for experienced Python users: Installing Jupyter with pip

Important: Jupyter installation requires Python 3.3 or greater, or Python 2.7. IPython 1.x, which included the parts that later became Jupyter, was the last version to support Python 3.2 and 2.6.

As an existing Python user, you may wish to install Jupyter using Python's package manager, *pip*, instead of Anaconda. First, ensure that you have the latest pip; older versions may have trouble with some dependencies:

```
pip3 install --upgrade pip
```

Then install the Jupyter Notebook using:

```
pip3 install jupyter
```

(Use `pip` if using legacy Python 2.)

Congratulations. You have installed Jupyter Notebook. See *Running the Notebook* for more details.

Optional: Installing Kernels

Contents

This information gives a high-level view of using Jupyter Notebook with different programming languages (kernels).

Are any languages pre-installed?

Yes, installing the Jupyter Notebook will also install the IPython *kernel*. This allows working on notebooks using the Python programming language.

How do I install Python 2 and Python 3?

To install an additional version of Python, i.e. to have both Python 2 and 3 available, see the IPython docs on installing kernels.

How do I install other languages like R or Julia?

To run notebooks in languages other than Python, such as R or Julia, you will need to install additional kernels. For more information, see the full list of available kernels.

See also:

Jupyter Projects Detailed installation instructions for individual Jupyter or IPython projects.

Kernels Information about additional programming language kernels.

Kernels documentation for Jupyter client Technical information about kernels.

Running the Notebook

Contents

Basic Steps

1. Start the notebook server from the *command line*:

```
jupyter notebook
```

2. You should see the notebook open in your browser.

Starting the Notebook Server

After you have installed the Jupyter Notebook on your computer, you are ready to run the notebook server. You can start the notebook server from the *command line* (using *Terminal* on Mac/Linux, *Command Prompt* on Windows) by running:

```
jupyter notebook
```

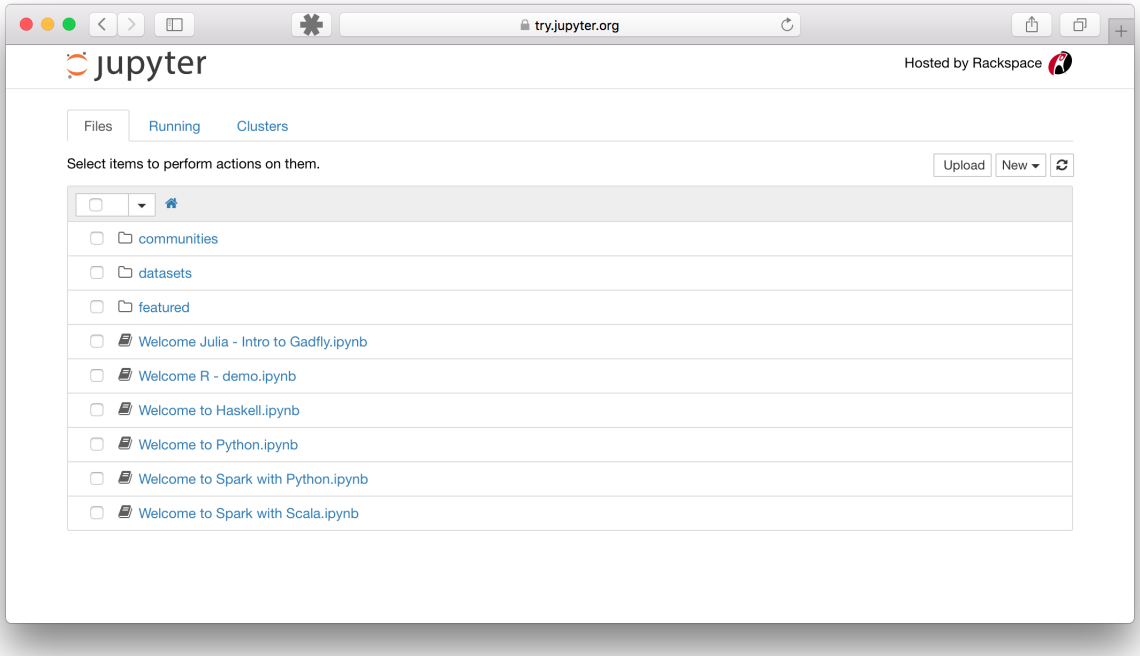
This will print some information about the notebook server in your terminal, including the URL of the web application (by default, `http://localhost:8888`):

```
$ jupyter notebook
[I 08:58:24.417 NotebookApp] Serving notebooks from local directory: /Users/catherine
[I 08:58:24.417 NotebookApp] 0 active kernels
[I 08:58:24.417 NotebookApp] The Jupyter Notebook is running at: http://
↪localhost:8888/
[I 08:58:24.417 NotebookApp] Use Control-C to stop this server and shut down all␣
↪kernels (twice to skip confirmation).
```

It will then open your default web browser to this URL.

When the notebook opens in your browser, you will see the *Notebook Dashboard*, which will show a list of the notebooks, files, and subdirectories in the directory where the notebook server was started. Most of the time, you will wish to start a notebook server in the highest level directory containing notebooks. Often this will be your home directory.

Notebook Dashboard



Introducing the Notebook Server's Command Line Options

How do I start the Notebook using a custom IP or port?

By default, the notebook server starts on port 8888. If port 8888 is unavailable or in use, the notebook server searches the next available port. You may also specify a port manually. In this example, we set the server's port to 9999:

```
jupyter notebook --port 9999
```

How do I start the Notebook server without opening a browser?

Start notebook server without opening a web browser:

```
jupyter notebook --no-browser
```

How do I get help about Notebook server options?

The notebook server provides help messages for other command line arguments using the `--help` flag:

```
jupyter notebook --help
```

See also:

Jupyter Installation, Configuration, and Usage Detailed information about command line arguments, configuration, and usage.

Migrating from IPython Notebook

Contents

Abstract

The Big Split moved IPython's various language-agnostic components under the Jupyter umbrella. Going forward, Jupyter will contain the language-agnostic projects that serve many languages. IPython will continue to focus on Python and its use with Jupyter.

This document describes what has changed, and how you may need to modify your code or configuration when migrating from IPython version 3 to Jupyter.

Understanding the Migration Process

Automatic migration of files

The first time you run any `jupyter` command, it will perform an automatic migration of files. The automatic migration process **copies** files, instead of moving files, leaving the originals in place and the copies in the Jupyter file locations. You can re-run the migration, if needed, by calling `jupyter migrate`. Your custom configuration will be migrated automatically and should work with Jupyter without further editing. When you update or modify your configuration in the future, please keep in mind that the file locations may have changed.

Where have my configuration files gone?

Also known as: "Why isn't my configuration having any effect anymore?"

Jupyter splitting out from IPython means that the locations of some files have moved, and Jupyter projects have not inherited everything from how IPython did it.

When you start your first Jupyter application, the relevant configuration files are automatically copied to their new Jupyter locations. The original configuration files in the IPython locations have no effect on Jupyter's execution. If you accidentally edit your original IPython config file, you may not see the desired effect with Jupyter now. You should check that you are editing Jupyter's configuration file, and you should see the expected effect after restarting the Jupyter server.

Finding the Location of Important Files

This section provides quick reference for common locations of IPython 3 files and the migrated Jupyter files.

Configuration files

Configuration files customize Jupyter to the user's preferences. The migrated files should all be **automatically copied** to their new Jupyter locations. Here are the location changes:

IPython location		Jupyter location
`~/.ipython/profile_default/static/custom`	→	`~/.jupyter/custom`
`~/.ipython/profile_default/ipython_notebook_config.py`	→	`~/.jupyter/jupyter_notebook_config.py`
`~/.ipython/profile_default/ipython_nbconvert_config.py`	→	`~/.jupyter/jupyter_nbconvert_config.py`
`~/.ipython/profile_default/ipython_qtconsole_config.py`	→	`~/.jupyter/jupyter_qtconsole_config.py`
`~/.ipython/profile_default/ipython_console_config.py`	→	`~/.jupyter/jupyter_console_config.py`

To choose a directory location other than the default `~/.jupyter`, set the `JUPYTER_CONFIG_DIR` environment variable. You may need to run `jupyter migrate` after setting the environment variable for files to be copied to the desired directory.

Data files: kernelspecs and notebook extensions

Data files include files, other than configuration files, which are user installed. Examples include kernelspecs and notebook extensions. Like the configuration files, data files are also **automatically migrated** to their new Jupyter locations.

In **IPython 3**, data files lived in `~/.ipython`.

In **Jupyter**, data files use platform-appropriate locations:

- OS X: `~/Library/Jupyter`
- Windows: the location specified in `%APPDATA%` environment variable
- Elsewhere, `$XDG_DATA_HOME` is respected, with the default of `~/.local/share/jupyter`

In all cases, the `JUPYTER_DATA_DIR` environment variable can be used to set a location explicitly.

Data files installed system-wide (e.g. in `/usr/local/share/jupyter`) have not changed. Per-user installation of data files has changed location from `.ipython` to the platform-appropriate Jupyter location.

Since Jupyter does not have profiles, how do I customize it?

While IPython has the concept of *profiles*, **Jupyter does not have profiles**.

In IPython, profiles are collections of configuration and runtime files. Inside the IPython directory (`~/.ipython`), there are directories with names like `profile_default` or `profile_demo`. In each of these are configuration files (`ipython_config.py`, `ipython_notebook_config.py`) and runtime files (`history.sqlite`, `security/kernel-*.json`). Profiles could be used to switch between configurations of IPython.

Previously, people could use commands like `ipython notebook --profile demo` to set the profile for *both* the notebook server and the IPython kernel. This is no longer possible in one go with Jupyter, just like it wasn't possible in IPython 3 for any other kernels.

Changing the Jupyter notebook configuration directory

If you want to change the notebook configuration, you can set the `JUPYTER_CONFIG_DIR`:

```
JUPYTER_CONFIG_DIR=./jupyter_config
jupyter notebook
```

Changing the Jupyter notebook configuration file

If you just want to change the config file, you can do:

```
jupyter notebook --config=/path/to/myconfig.py
```

Changing kernelspecs

If you do want to change the IPython kernel's profile, you can't do this at the server command-line anymore. Kernel arguments must be changed by modifying the kernelspec. You can do this without relaunching the server. Kernelspec changes take effect every time you start a new kernel. However, there isn't a great way to modify the kernelspecs. One approach uses `jupyter kernelspec list` to find the `kernel.json` file and then modifies it, e.g. `kernels/python3/kernel.json`, by hand. Alternatively, a2km is an experimental project that tries to make these things easier.

Understanding Installation Changes

See the *Installing Jupyter Notebook* page for more information about installing Jupyter. Jupyter automatically migrates some things, like Notebook extensions and kernels.

Notebook extensions

Any IPython notebook extensions should be **automatically migrated** as part of the data files migration.

Notebook extensions were installed with:

```
ipython install-nbextension [--user] EXTENSION
```

Now, extensions are installed with:

```
jupyter nbextension install [--user] EXTENSION
```

The notebook extensions will be installed in a system-wide location (e.g. `/usr/local/share/jupyter/nbextensions`). If doing a `--user` install, the notebook extensions will go in the `JUPYTER_DATA_DIR` location. Installation **SHOULD NOT** be done manually by guessing where the files should go.

Kernels

Kernels are installed in much the same way as notebook extensions. They will also be **automatically migrated**.

Kernel specs used to be installed with:

```
ipython kernelspec install [--user] KERNEL
```

They are now installed with:

```
jupyter kernelspec install [--user] KERNEL
```

By default, kernel specs will go in a system-wide location (e.g. `/usr/local/share/jupyter/kernels`). If doing a `--user` install, the kernel specs will go in the `JUPYTER_DATA_DIR` location. Installation **SHOULD NOT** be done manually by guessing where the files should go.

Understanding Changes in imports

IPython 4.0 includes shims to manage dependencies; so, all imports that work on IPython 3 should continue to work on IPython 4. If you find any differences, please let us know.

Some changes include:

IPython 3		Jupyter and IPython 4.0
`IPython.html`	→	`notebook`
`IPython.html.widgets`	→	`ipywidgets`
`IPython.kernel`	→	`jupyter_client`, `ipykernel`
`IPython.parallel`	→	`ipyparallel`
`IPython.qt.console`	→	`qtconsole`
`IPython.utils.traitlets`	→	`traitlets`
`IPython.config`	→	`traitlets.config`

Important: The `IPython.kernel` Split

`IPython.kernel` became two packages:

- `jupyter_client` for the Jupyter client-side APIs.
- `ipykernel` for Jupyter's IPython kernel

CHAPTER 2

Architecture Guides

How IPython and Jupyter Notebook work

Contents

Abstract

This section focuses on IPython and Jupyter notebook and how they interact. When we discuss `IPython`, we talk about two fundamental roles:

- Terminal IPython as the familiar REPL
- The IPython kernel that provides computation and communication with the frontend interfaces, like the notebook

Jupyter Notebook and its flexible interface extends the notebook beyond code to visualization, multimedia, collaboration, and more.

Terminal IPython

When you type `ipython`, you get the original IPython interface, running in the terminal. It does something like this:

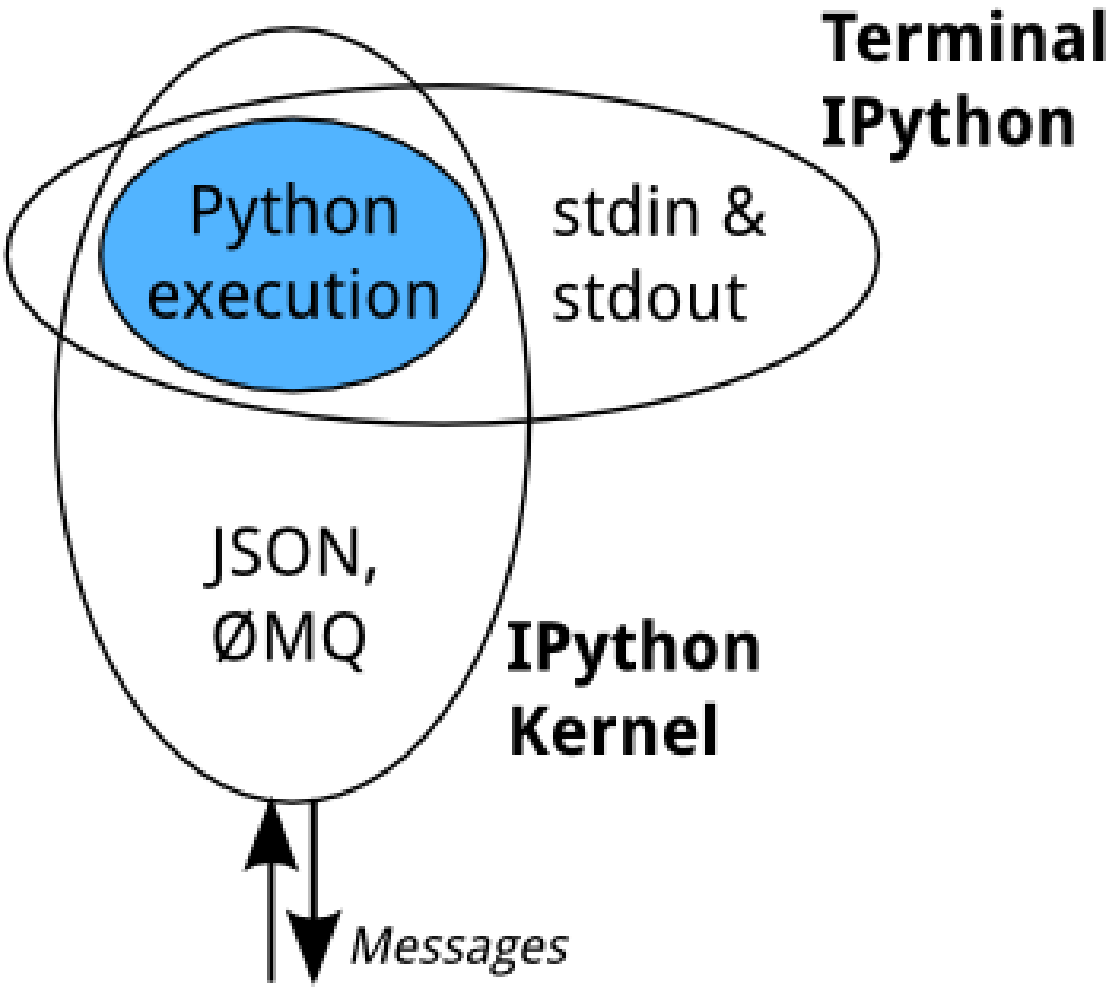
```
while True:
    code = input(">>> ")
    exec(code)
```

Of course, it's much more complex, because it has to deal with multi-line code, tab completion using `readline`, magic commands, and so on. But the model is like code example: prompt the user for some code, and when they've entered it, execute it in the same process. This model is often called a *REPL*, or Read-Eval-Print-Loop.

The IPython Kernel

All the other interfaces —- the Notebook, the Qt console, `ipython console` in the terminal, and third party interfaces —- use the IPython Kernel. The IPython Kernel is a separate process which is responsible for running user code, and things like computing possible completions. Frontends, like the notebook or the Qt console, communicate with the IPython Kernel using JSON messages sent over ZeroMQ sockets; the protocol used between the frontends and the IPython Kernel is described in Messaging in Jupyter.

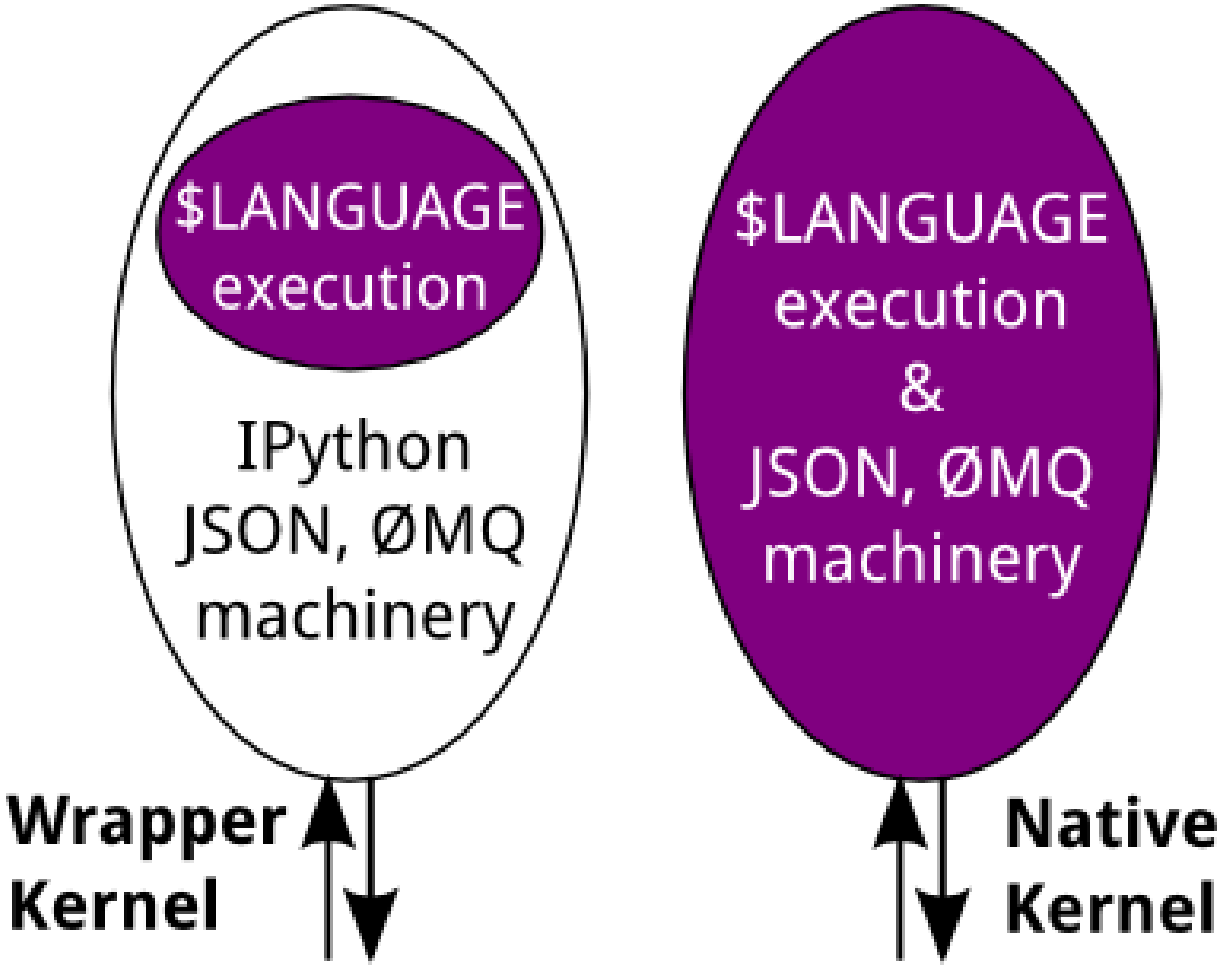
The core execution machinery for the kernel is shared with terminal IPython:



A kernel process can be connected to more than one frontend simultaneously. In this case, the different frontends will have access to the same variables.

This design was intended to allow easy development of different frontends based on the same kernel, but it also made it possible to support new languages in the same frontends, by developing kernels in those languages, and we are refining IPython to make that more practical.

Today, there are two ways to develop a kernel for another language. Wrapper kernels reuse the communications machinery from IPython, and implement only the core execution part. Native kernels implement execution and communications in the target language:



Wrapper kernels are easier to write quickly for languages that have good Python wrappers, like octave_kernel, or languages where it's impractical to implement the communications machinery, like bash_kernel. Native kernels are likely to be better maintained by the community using them, like IJulia or IHaskell.

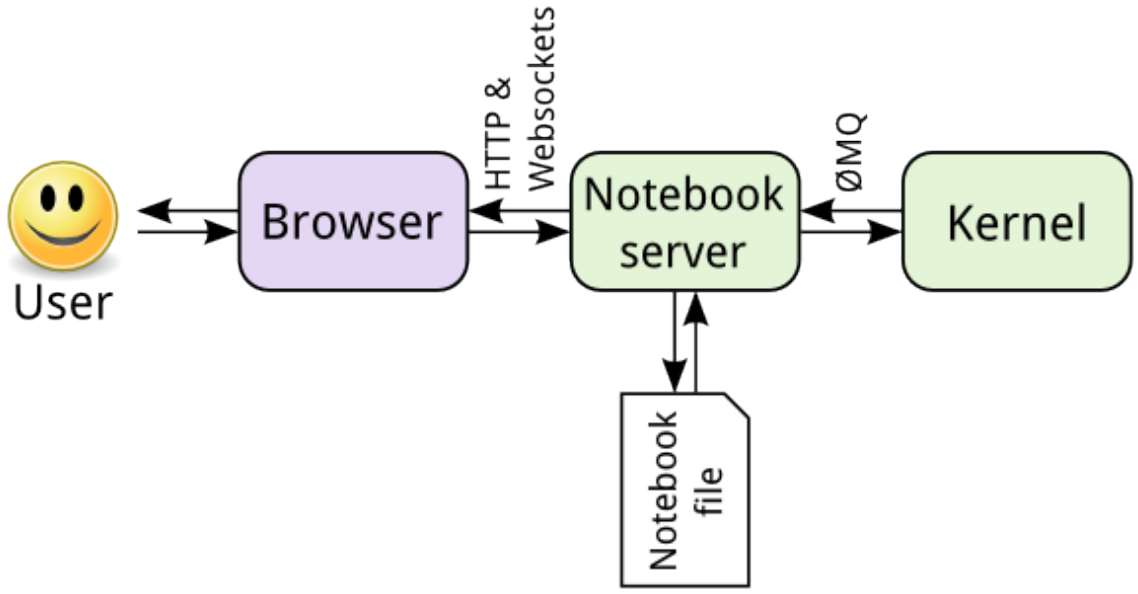
See also:

Making kernels for Jupyter

Kernels

Notebooks

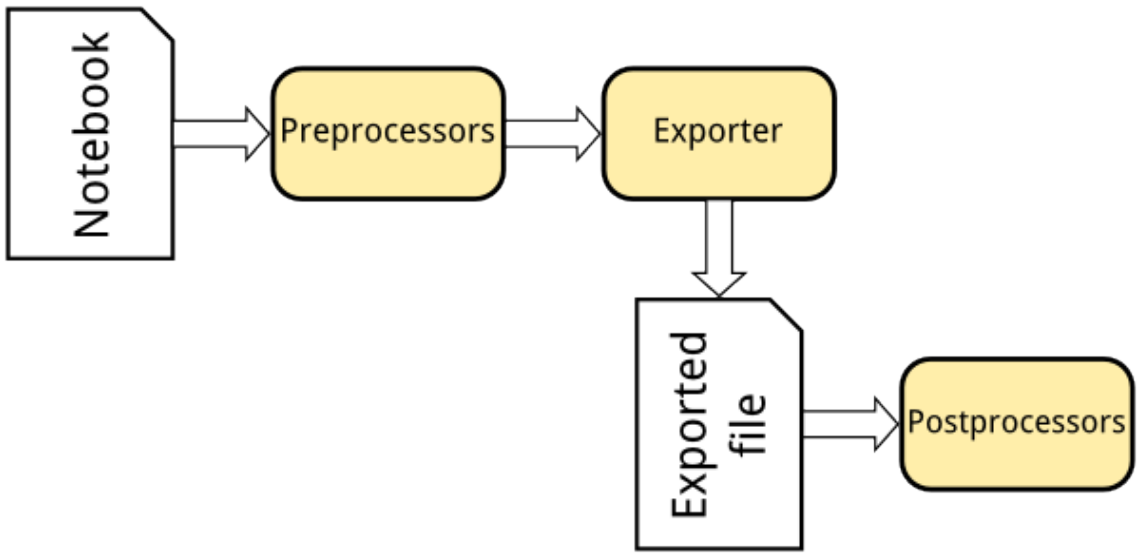
The Notebook frontend does something extra. In addition to running your code, it stores code and output, together with markdown notes, in an editable document called a notebook. When you save it, this is sent from your browser to the notebook server, which saves it on disk as a JSON file with a `.ipynb` extension.



The notebook server, not the kernel, is responsible for saving and loading notebooks, so you can edit notebooks even if you don't have the kernel for that language—you just won't be able to run code. The kernel doesn't know anything about the notebook document: it just gets sent cells of code to execute when the user runs them.

Exporting notebooks to other formats

The Nbconvert tool in Jupyter converts notebook files to other formats, such as HTML, LaTeX, or reStructuredText. This conversion goes through a series of steps:



1. Preprocessors modify the notebook in memory. E.g. ExecutePreprocessor runs the code in the notebook and updates the output.
2. An exporter converts the notebook to another file format. Most of the exporters use templates for this.
3. Postprocessors work on the file produced by exporting.

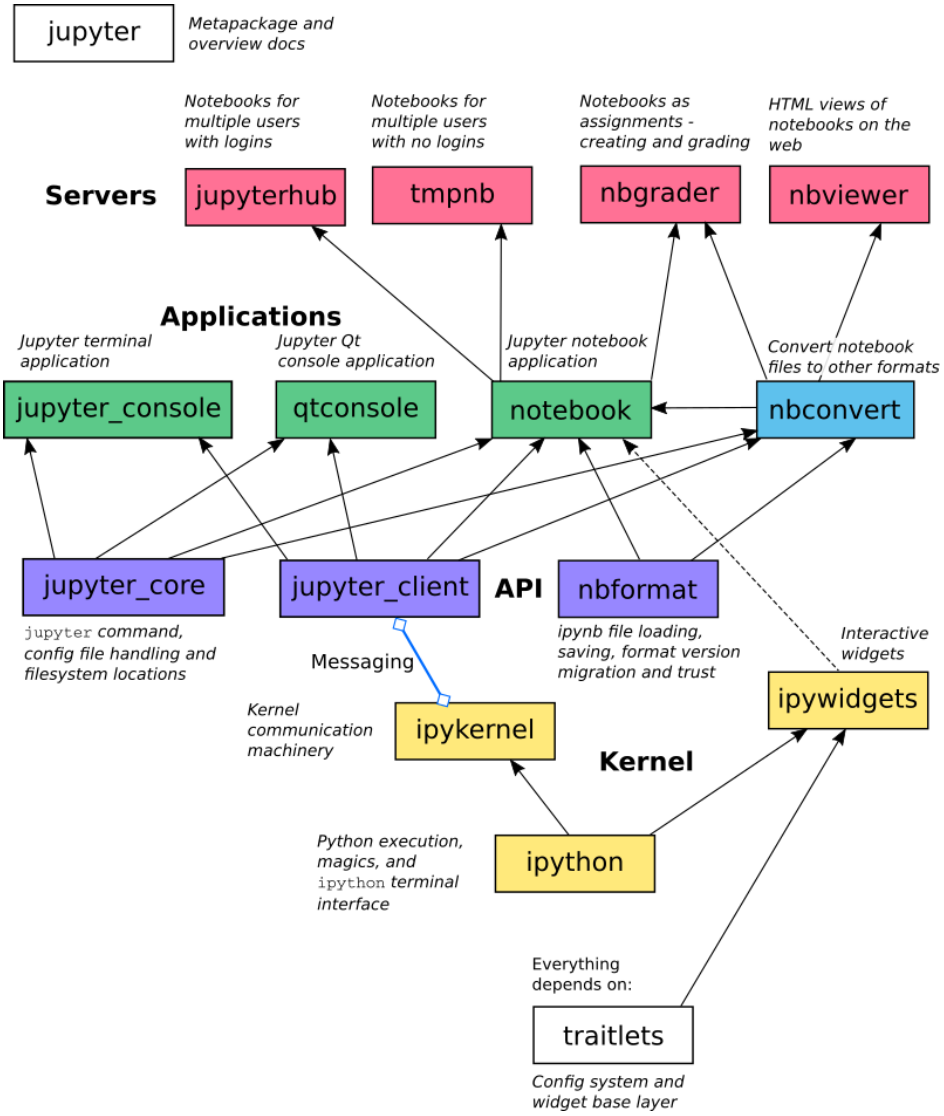
The nbviewer website uses nbconvert with the HTML exporter. When you give it a URL, it fetches the notebook from that URL, converts it to HTML, and serves that HTML to you.

IPython.parallel

IPython also includes a parallel computing framework, IPython.parallel. This allows you to control many individual engines, which are an extended version of the IPython kernel described above.

A Visual Overview of Projects

A high level visual overview of project relationships


jupyter
Metapackage and overview docs
Notebooks for multiple users with logins
Notebooks for multiple users with no logins
Notebooks as assignments - creating and grading
HTML views of notebooks on the web
Servers
jupyterhub
tmpnb
nbgrader
nbviewer
Applications
Jupyter terminal application
Jupyter Qt console application
Jupyter notebook application
Convert notebook files to other formats
jupyter_console
qtconsole
notebook
nbconvert
jupyter_core
jupyter_client
API
nbformat
jupyter command, config file handling and filesystem locations
Messaging
ipynb file loading, saving, format version migration and trust
Interactive widgets
ipywidgets
Kernel communication machinery
ipykernel
Kernel
Python execution, magics, and ipython terminal interface
ipython
Everything depends on:
traitlets
Config system and widget base layer

CHAPTER 3

Narratives and Use Cases

What are Narratives?

Narratives are *collaborative*, *shareable*, *publishable*, and *reproducible*. We believe that Narratives help both yourself and other researchers by sharing your use of Jupyter projects, technical specifics of your deployment, and installation and configuration tips so that others can learn from your experiences.

Notebook Narratives

Contents

Description

The Notebook Narratives explore uses of the Jupyter Notebook in a variety of applications.

Narrative examples

- Using the Notebook for data exploration
- Using extensions and widgets
- Using nbconvert for code execution and workflow simplification
- Using nbconvert for publishing
- Using multiple language kernels

Note: We're actively working on this section of the documentation to improve it for you. Thanks for your patience.

JupyterHub Narratives

Contents

Description

JupyterHub Narratives explore deployment and scaling of the Jupyter Notebook for a group of users. JupyterHub allows flexibility in configuration and deployment which makes JupyterHub a valuable to education, industry research teams, and service providers. In these Narratives, we will look at differences in deployment, deployment advantages, and best practices.

Narrative examples

- A basic JupyterHub deployment
- A reference deployment of JupyterHub using Docker
- Teaching a Course with JupyterHub and nbgrader using a reference deployment on a single server and Ansible scripts to automate set up
- Teaching a Course with JupyterHub, nbgrader, and containers
- JupyterHub deployments using Containers including Docker

Note: We're actively working on this section of the documentation to improve it for you. Thanks for your patience.

Narratives - Building blocks

Contents

Description

This section presents some examples of integrating different projects together. These projects form the foundation of innovative services and provide building blocks for future applications.

Narrative examples

- A Narrative about Creating Dashboards
- A Narrative about Thebe
- A Narrative about Hydrogen

Note: We're actively working on this section of the documentation to improve it for you. Thanks for your patience.

Jupyter for Data Science

The purpose of this page is to highlight kernels and other projects that are central to the usage of Jupyter in data science. This page is not meant to be comprehensive or unbiased, but rather to provide an opinionated view of the usage of Jupyter in data science based on our interactions with users.

The following Jupyter kernels are widely used in data science:

- **python**
  - IPython (GitHub Repo)
- **R**
  - IRkernel (Documentation, GitHub Repo)
  - IRdisplay (GitHub Repo)
  - repr (GitHub Repo)
- **Julia**
  - IJulia Kernel (GitHub Repo)
  - Interactive Widgets (GitHub Repo)
- Bash (GitHub Repo)

Jupyter and Scientific Computing

Note: We're actively working on this section of the documentation to improve it for you. Thanks for your patience.

Jupyter in Education

Note: We're actively working on this section of the documentation to improve it for you. Thanks for your patience.

CHAPTER 4

IPython

Contents

Description

IPython provides a rich architecture for interactive computing with:

- A powerful interactive shell.
- A kernel for Jupyter.
- Support for interactive data visualization and use of GUI toolkits.
- Flexible, embeddable interpreters to load into your own projects.
- Easy to use, high performance tools for parallel computing.

Background

IPython is a growing project, with increasingly language-agnostic components. IPython 3.x was the last monolithic release of IPython, containing the notebook server, qtconsole, etc. As of IPython 4.0, the language-agnostic parts of the project: the notebook format, message protocol, qtconsole, notebook web application, etc. have moved to new

projects under the name Jupyter. IPython itself is focused on interactive Python, part of which is providing a Python kernel for Jupyter.

Resources

CHAPTER 5

Installation, Configuration, and Usage

Installation

Upgrading Jupyter Notebook

Contents

Upgrading Jupyter Notebook using Anaconda

If using **Anaconda**, update Jupyter using `conda`:

```
conda update jupyter
```

See *Run the Notebook* for running the Jupyter Notebook.

Upgrading IPython Notebook to Jupyter Notebook

The Jupyter Notebook used to be called the IPython Notebook. If you are running an older version of the IPython Notebook (version 3 or earlier) you can use the following to upgrade to the latest version of the Jupyter Notebook.

If using **Anaconda**, update Jupyter using `conda`:

```
conda update jupyter
```

or

If using *pip*:

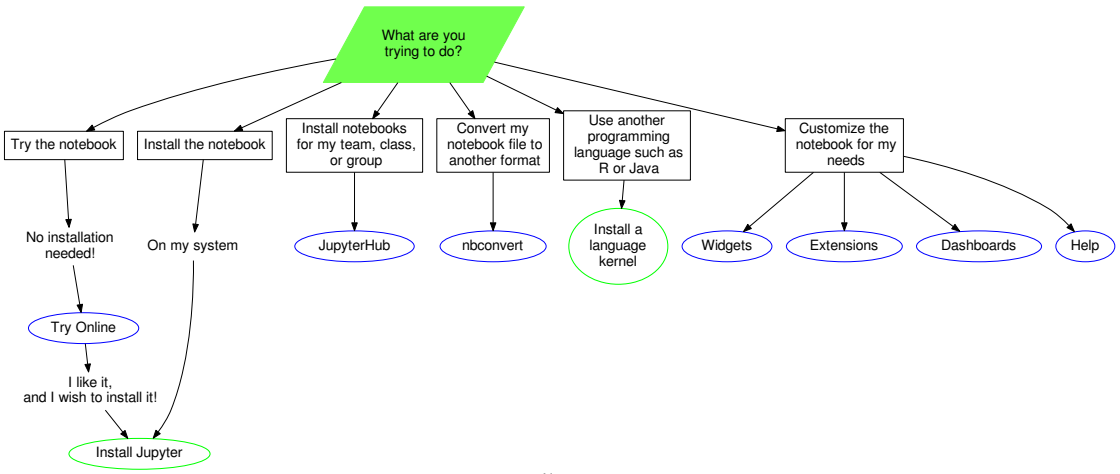
```
pip install -U jupyter
```

See *Run the Notebook* for running the Jupyter Notebook.

See also:

The *migrating* document has additional information about migrating from IPython 3 to Jupyter.

How do I decide which packages I need?



Configuration

The `jupyter` Command

Synopsis

```
jupyter <subcommand> [options]
```

Description

Commands like `jupyter notebook` start Jupyter applications. The `jupyter` command is primarily a namespace for subcommands. A command like `jupyter-foo` found on your `PATH` will be available as a subcommand `jupyter foo`.

The **`jupyter`** command can also be used to do actions other than starting a Jupyter application.

Command options

-h, --help
Show help information, including available subcommands.

--config-dir
Show the location of the config directory.

--data-dir
Show the location of the data directory.

--runtime-dir
Show the location of the data directory.

--paths
Show all Jupyter directories and search paths.

--json
Print directories and search paths in machine-readable JSON format.

Common Directories and File Locations

Contents

Jupyter stores different files (i.e. configuration, data, runtime) in a number of different locations. Environment variables may be set to customize for the location of each file type.

Configuration files

Config files are stored by default in the `~/.jupyter` directory.

JUPYTER_CONFIG_DIR
Set this environment variable to use a particular directory, other than the default, for Jupyter config files.

Besides the user config directory mentioned above, Jupyter has a search path of additional locations from which a config file will be loaded. Here's a table of the locations to be searched, in order of preference:

Unix	Windows
JUPYTER_CONFIG_DIR	
{sys.prefix}/etc/jupyter/	
/usr/local/etc/jupyter/ /etc/jupyter/	%PROGRAMDATA%\jupyter\

To list the config directories currrently being used you can run the below command from the *command line*:

```
jupyter --paths
```

Data files

Jupyter uses a search path to find installable data files, such as kernelspecs and notebook extensions. When searching for a resource, the code will search the search path starting at the first directory until it finds where the resource is contained.

Each category of file is in a subdirectory of each directory of the search path. For example, kernel specs are in `kernels` subdirectories.

JUPYTER_PATH
: Set this environment variable to provide extra directories for the data search path. *JUPYTER_PATH* should contain a series of directories, separated by `os.pathsep` (`;` on Windows, `:` on Unix). Directories given in *JUPYTER_PATH* are searched before other locations.

Linux (& other free desktops)	Mac	Windows
JUPYTER_PATH		
`~/.local/share/jupyter/` (respects `$XDG_DATA_HOME`)	`~/Library/Jupyter`	`%APPDATA%\jupyter`
`{sys.prefix}/share/jupyter/`		
`/usr/local/share/jupyter` `/usr/share/jupyter`		`%PROGRAMDATA%\jupyter`

Runtime files

Things like connection files, which are only useful for the lifetime of a particular process, have a runtime directory.

On Linux and other free desktop platforms, these runtime files are stored in `$XDG_RUNTIME_DIR/jupyter` by default. On other platforms, it's a `runtime/` subdirectory of the user's data directory (second row of the table above).

An environment variable may also be used to set the runtime directory.

JUPYTER_RUNTIME_DIR
: Set this to override where Jupyter stores runtime files.

Summary

JUPYTER_CONFIG_DIR for config file location

JUPYTER_PATH for datafile directory locations

JUPYTER_RUNTIME_DIR for runtime file location

See also:

jupyter_core.paths The Python API to locate these directories.

The jupyter Command Locate these directories from the command line.

Jupyter's Common Configuration Approach

Contents

Summary

Common Jupyter configuration system The Jupyter applications have a common config system, and a common *config directory*. By default, this directory is `~/.jupyter`.

Kernel configuration directories If kernels use config files, these will normally be organised in separate directories for each kernel. For instance, the IPython kernel looks for files in the IPython directory instead of the default Jupyter directory `~/.jupyter`.

The Python config file

To create a default config file, run:

```
jupyter {application} --generate-config
```

The generated file will be named `jupyter_application_config.py`.

By editing the `jupyter_application_config.py` file, you can configure class attributes like this:

```
c.NotebookApp.port = 8754
```

Be careful with spelling. Incorrect names will simply be ignored, with no error message.

To add to a collection which may have already been defined elsewhere, you can use methods like those found on lists, dicts and sets: `append`, `extend`, `prepend()` (like extend, but at the front), `add`, and `update` (which works both for dicts and sets):

```
c.TemplateExporter.template_path.append('./templates')
```

Command line options for configuration

Every configurable value can also be set from the command line and passed as an argument, using this syntax:

```
jupyter notebook --NotebookApp.port=8754
```

Frequently used options will also have short aliases and flags, such as `--port 8754` or `--no-browser`.

To see the abbreviated options, pass `--help` or `--help-all` as follows:

```
jupyter {application} --help       # Just the short options
jupyter {application} --help-all   # Includes options without short names
```

Command line options **will override** options set within a configuration file.

See also:

`traitlets.config` The low-level architecture of this config system.

Usage and Projects

Jupyter Projects

Contents

Project Jupyter is developed as a set of subprojects. This section describes the subprojects with links to their documentation or GitHub repositories.

Jupyter User Interfaces

The Jupyter user interfaces offer a foundation of interactive computing environments where scientific computing, data science, and analytics can be performed using a wide range of programming languages.

Jupyter Notebook Web-based application for authoring documents that combine live-code with narrative text, equations and visualizations. Documentation | Repo

Jupyter Console Terminal based console for interactive computing. Documentation | Repo

Jupyter QtConsole Qt application for interactive computing with rich output. Documentation | Repo

Kernels

Kernels are *programming language specific* processes that run independently and interact with the Jupyter Applications and their user interfaces. IPython is the reference Jupyter kernel, providing a powerful environment for interactive computing in Python.

IPython interactive computing in Python. Documentation | Repo

ipywidgets interactive widgets for Python in the Jupyter Notebook. Documentation | Repo

ipyparallel lightweight parallel computing in Python offering seamless notebook integration. Documentation | Repo

See also:

Jupyter kernels

A full list of kernels available for other languages. Many of these kernels are developed by third parties and may or may not be stable.

Formatting and Conversion

Notebooks are rich interactive documents that combine live code, narrative text (using markdown), visualizations, and other rich media. The following utility subprojects allow programmatic format conversion and manipulation of notebook documents.

nbconvert Convert dynamic notebooks to static formats such as HTML, Markdown, LaTeX/PDF, and reStrucured-Text. Documentation | Repo

nbformat Work with notebook documents programmatically. Documentation | Repo

Education

Jupyter Notebooks offer exciting and creative possibilities in education. The following subprojects are focused on supporting the use of Jupyter Notebook in a variety of educational settings.

nbgrader tools for managing, grading, and reporting of notebook based assignments. Documentation | Repo

Deployment

To serve a variety of users and use cases, these subprojects are being developed to support notebook deployment in various contexts, including multiuser capabilities and secure, scalable cloud deployments.

jupyterhub Multi-user notebook for organizations with pluggable authentication and scalability. Documentation | Repo

jupyter-drive Store notebooks on Google Drive. Documentation | Repo

nbviewer Share notebooks as static HTML on the web. Documentation | Repo

tmpnb Create temporary, transient notebooks in the cloud. Documentation | Repo

tmpnb-deploy Deployment tools for tmpnb. Documentation | Repo

dockerspawner Deploy notebooks for 'jupyterhub' inside Docker containers. Documentation | Repo

docker-stacks Stacks of Jupyter applications and kernels as Docker containers. Documentation | Repo

Architecture

The Jupyter architecture relies on these projects' specifications and implementation.

jupyter_client The specification of the Jupyter message protocol and a client library in Python. Documentation | Repo

jupyter_core Core functionality and miscellaneous utilities. Documentation | Repo

IPython Projects

Contents

Description

IPython provides a rich architecture for interactive computing with:

- A powerful interactive shell.
- A kernel for Jupyter.
- Support for interactive data visualization and use of GUI toolkits.
- Flexible, embeddable interpreters to load into your own projects.

- Easy to use, high performance tools for parallel computing.

Background

IPython is a growing project, with increasingly language-agnostic components. IPython 3.x was the last monolithic release of IPython, containing the notebook server, qtconsole, etc. As of IPython 4.0, the language-agnostic parts of the project: the notebook format, message protocol, qtconsole, notebook web application, etc. have moved to new projects under the name Jupyter. IPython itself is focused on interactive Python, part of which is providing a Python kernel for Jupyter.

Resources

The projects with repos in the IPython organization on GitHub include:

- IPython ipykernel interactive computing in Python.
- ipyparallel lightweight parallel computing in Python offering seamless notebook integration
- ipywidgets interactive widgets for Python in the Jupyter Notebook

Incubator Projects

Contents

The Jupyter incubator gives emerging projects a place to evolve.

Descriptions

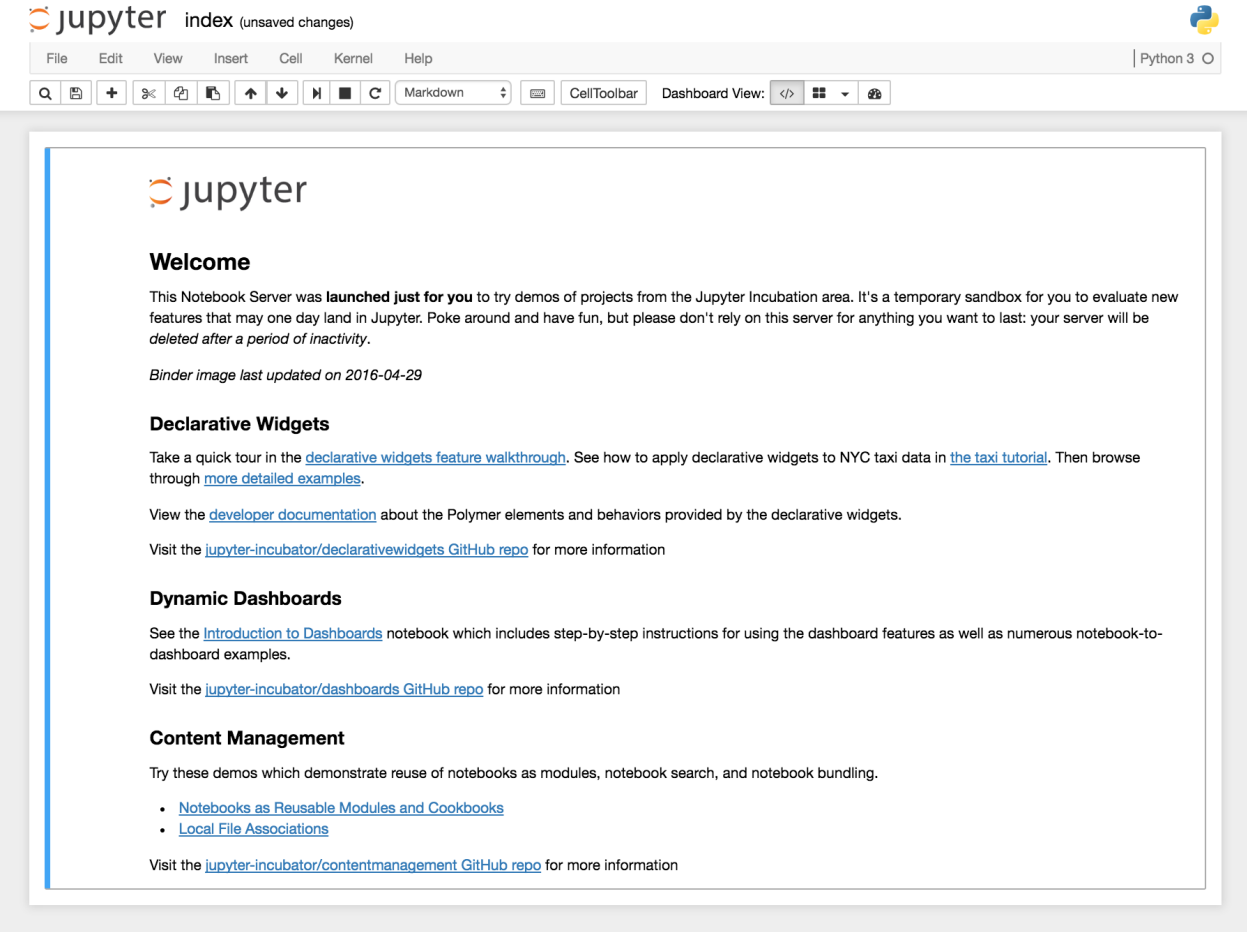
Interesting projects include:

- content management extensions - Jupyter Content Management Extensions
- dashboards - Jupyter Dynamic Dashboards from Notebooks
- declarative widgets - Jupyter Declarative Widgets Extension
- kernel gateway bundlers - Converts a notebook to a kernel gateway microservice bundle for download
- showcase - A spot to try demos of one or more incubating Jupyter projects in Binder
- sparkmagic - Jupyter magics and kernels for working with remote Spark clusters
- traittypes - Traitlets types for NumPy, SciPy and friends

There's also a repository with proposals of projects wishing to enter the incubator.

Try the Incubator Projects

The showcase application allows you to try demos of one or more incubating Jupyter projects in Binder. Just head over to the showcase repo and press the *Launch Binder* button badge to launch the online trial. You should see an interactive notebook similar to this one:



Kernels (Programming Languages)

The Jupyter team maintains the IPython kernel since the Jupyter notebook server depends on the IPython *kernel* functionality. Many other languages, in addition to Python, may be used in the notebook.

The community maintains many other language kernels, and new kernels become available often. Please see the list of available kernels for additional languages and kernel installation instructions to begin using these language kernels.

Project Documentation

Contents

Links to **information on usage, configuration and development** hosted on Read The Docs or in the GitHub project repo.

Jupyter User Interfaces

- Jupyter Notebook
- jupyter_console
- qtconsole

JupyterHub

- JupyterHub
- configurable-http-proxy
- dockerspawner
- ldapauthenticator
- oauthenticator
- sudospawner

Education

- nbgrader

Notebook Conversion and Formatting

- nbconvert
- nbformat

Kernels

- IPython
- IRkernel
- IJulia
- List of community maintained language kernels

IPython

- IPython
- ipykernel
- ipyparallel

Deployment

- docker-stacks
- ipywidgets
- jupyter-drive
- jupyter-sphinx-theme
- kernel_gateway
- nbviewer
- tmpnb
- traitlets

JupyterLab

- jupyter-js-notebook
- jupyter-js-phosphide
- jupyter-js-plugins
- jupyter-js-services
- jupyter-js-ui
- jupyter-js-utils

Architecture

- jupyter_client
- jupyter_core

CHAPTER 6

Community Guides

Welcome to the Community Guides for Jupyter. These guides are intended to provide information about the Jupyter community such as background, events, and communication channels. As our community is highly dynamic, information may change, and we will do our best to keep it up to date.

Weekly Dev meeting

The core developers have weekly meetings to discuss and demo what they have been working on, discuss future plans, and bootstrap conversation. These meetings are public. The direct link to attend the meeting will be posted to the Hackpad before the meeting. Afterwards, the recording of the meeting will be posted to the IPython channel on YouTube.

Jupyter/IPython meetings:

- Tuesdays at 9am PST

JupyterLab meetings:

- Fridays at 9am PST

Collaborative notes are taken before/during the meeting on Hackpad. Hackpads are organized by month.

We will ping the dev-meeting-attendance Gitter channel 1-2 days before each meeting to know who is likely going to attend.

Jupyter communications

- Blog https://blog.jupyter.org/
- Newsletter https://newsletter.jupyter.org/
- Website https://jupyter.org
- Twitter https://twitter.com/ProjectJupyter

- Mailing lists (Jupyter, Jupyter in Education) https://jupyter.org/community.html

Governance

- Steering council: Information about the steering council and its members can be found on the Jupyter website.
- Jupyter Enhancement Proposal (JEP) process: Details about the process can be found in the jupyter/enhancement-proposals GitHub repo.

Code of conduct

Information can be found in the Jupyter Governance repo on GitHub.

CHAPTER 7

Contributor Guides

Developer Guide

Contents

Developer Guide

Contents

A Note on Contributing to Open Source

Contributing to open source can be a nerve-wrecking process, but don't worry everyone on the Jupyter team is dedicated to making sure that your open source experience is as fun as possible. At any time during the process described below, you can reach out to the Jupyter team on Gitter or the mailing list for assistance. If you are nervous about asking questions in public, you can also reach out to one of the Jupyter developers in private. You can use the public Gitter to find someone who has the best knowledge about the code you are working with and interact with the in a personal chat.

As you begin your open source journey, remember that it's OK if you don't understand something, it's OK to make mistakes, and it's OK to only contribute a small amount of the code necessary to fix the issue you are tackling. Any and all help is welcome and any and all people are encouraged to contribute.

How can I help?

Individuals are welcome, and encouraged, to submit pull requests and contribute to the Jupyter source. If you are a first-time contributor looking to get involved with Jupyter, you can use the following query in a GitHub search to find beginner-friendly issues to tackle across the Jupyter codebase. This query is particularly useful because the Jupyter codebase is scattered across several repositories within the jupyter organization, as opposed to a single repository. You can click the link below to find sprint-friendly issues.

is:issue is:open is:sprint-friendly user:jupyter

Once you've found an issue that you are eager to solve, you can use the guide below to get started. If you experience any problems while working on the issue, leave a comment on the issue page in GitHub and someone on the core team will be able to lend you assistance.

Please keep in mind that what follows are guidelines. If you work through the steps and have questions or run into time constraints, please submit what you already have worked on as a pull request and ask questions on it. Your effort, including partial or in-progress work, is appreciated.

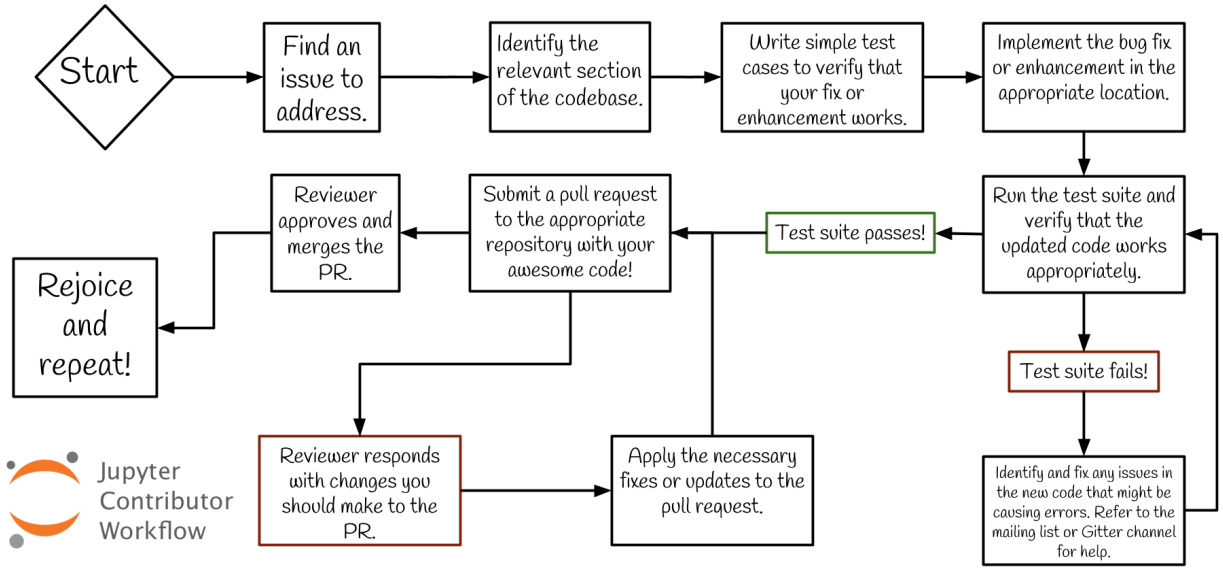
1. Fork the repository associated with the issue you are addressing and clone it to a local directory on your machine.
2. `cd` into the directory and create a new branch using `git checkout -b insert-branch-name-here`. Pick a branch name that gives some insight into what the issue you are fixing is. For example, if you are updating the text that is logged out by the program when a certain error happens you might name your branch *update-error-text*.
3. Refer to the repository's README and documentation for details on configuring your system for development.
4. Identify the module or class where the code change you will make will reside and leave a comment in the file describing what issue you are trying to address.
5. Open a pull request to the repository with `[WIP]` appended to the front so that the core team is aware that you are actively pursuing the issue. When creating a pull request, make sure that the title clearly and concisely described what your code does. For example, we might use the title "Updated error message on ExampleException". In the body of the pull request, make sure that you include the phrase "Closes #issue-number-here", where the issue number is the issue number of the issue that you are addressing in this PR.

   Feel free to open a PR as early as possible. Getting early feedback on your approach will save you time and prevent the need for an extensive refactor later.
6. Run the test suite locally in order to ensure that everything is properly configured on your system. Refer to the repository's README for information on how to run the test suite. This will typically require that you run the `nosetests` command on the commandline. Alternatively, you may submit a pull request. Our Continuous Integration system will test your code and report test results.
7. Find the test file associated with the module that you will be changing. In the test file, add some tests that outline what you expect the behavior of the change should be. If we continue with our example of updating the text that is logged on error, we might write test cases that check to see if the exception raised when you induce the error contains the appropriate string. When writing test cases, make sure that you test for the following things.
   - What is the simplest test case I can write for this issue?
   - What will happen if your code is given messy inputs?
   - What will happen if your code is given no inputs?
   - What will happen if your code is given too few inputs?
   - What will happen if your code is given too many inputs?

   If you need assistance writing test cases, you can place a comment on the pull request that was opened earlier and one of the core team members will be able to help you.
8. Go back to the file that you are updating and begin adding the code for your pull request.

9. Run the test suite again to see if your changes have caused any of the test cases to pass. If any of the test cases have failed, go back to your code and make the updates necessary to have them pass.
10. Once all of your test cases have passed, commit both the test cases and the updated module and push the updates to the branch on your forked repository.
11. Once you are ready for your pull request to be reviewed, remove the [WIP] tag from the front of issue, a project reviewer will review your code for quality. You can expect the reviewer to check for the documentation provided in the changes you made, how thorough the test cases you provided are, and how efficient your code is. Your reviewer will provide feedback on your code and you will have the chance to edit your code and apply fixes.
12. Once your PR is ready to become a part of the code base, it will be merged by a member of the core team.

Contribution Workflow



Core Developer Workflow

To help you understand our review process by core developers after you submit a pull request, here's a guide that outlines the general process (specifics may vary a bit across our repositories). Here is an example for Jupyter notebook 4.x:

In general, Pull Requests are against `master` unless they only affect a backport branch. If a PR affects master and should be backported, the general flow is:

0. mark the PR with milestone for the next backport release (4.3)
1. merge into master
2. backport to 4.x
3. push updated 4.x branch

Backports can be done in a variety of ways, but we have a script for automating the common process to:

1. download the patch ' e.g. <https://patch-diff.githubusercontent.com/raw/jupyter/notebook/pull/1645.patch>'
2. checkout the 4.x branch
3. apply the patch

4. make a commit

which works for simple cases, at least.

In this case, it would be:

    python /path/to/ipython-repo/tools/backport_pr.py jupyter/notebook 4.x 1645

Submitting a Bug

While using the Notebook, you might experience a bug that manifests itself in unexpected behavior. If so, we encourage you to open issues on GitHub. To make the navigating issues easier for both developers and users, we ask that you take the following steps before submitting an issue.

1. Search through StackOverflow and existing GitHub issues to ensure that the issue has not already been reported by another user. If so, provide your input on the existing issue if you think it would be valuable.
2. Prepare a small, self-contained snippet of code that will allow others to reproduce the issue that you are experiencing.
3. Prepare information about the environment that you are executing the code in, in order to aid in the debugging of the issue. You will need to provide information about the Python version, Jupyter version, operating system, and browser that you are using when submitting bugs. You can also use `pip list` or `conda list` and `grep` in order to identify the versions of the libraries that are relevant to the issue that you are submitting.
4. Prepare a simple test that outlines the expected behavior of the code or a description of the what the expected behavior should be.
5. Prepare an explanation of why the current behavior is not desired and what it should be.

Jupyter Enhancement Proposals

Submitting an Enhancement Proposal

While using the Notebook, you might discover opportunities for growth and ideas for useful new features. If so, feel free to submit an enhancement proposal. The process for submitting enhancements is as follows:

1. Identify the scope of the enhancement. Is it a change that affects only on part of the codebase? Is the enhancement, to the best of your knowledge, fairly trivial to implement? If the scope of the enhancement is small, it should be be submitted as an issue in the project's repository. If the scope of your enhancement is large, it should be submitted to the official Jupyter Enhancement Proposals repository.
2. Prepare a brief write-up of the problem that your enhancement will address.
3. Prepare a brief write-up of the proposed enhancement itself.
4. If the scope of your enhancement (as defined in step 1) is large, then prepare a detailed write-up of how your enhancement can be potentially implemented.
5. Identify a brief list of the pros and cons associated with implementing the enhancement that you propose.
6. Identify the individuals who might be interested in implementing the enhancement.
7. Depending on the scope of your enhancement, submit it either as an issue to the appropriate repository or as a Jupyter Enhancement Proposal.

Basic template for releasing a Jupyter project

Jupyter consists of a bunch of small projects, and a few larger ones. This lays out the basic process of releasing a smaller project, which should also apply to larger projects, though they may have some added steps.

Milestones

Most Jupyter projects use a GitHub milestone system for marking issues and pull requests in releases. Each release should have a milestone associated with it. The first step in preparing for a release is to make sure that every issue and pull request has the right milestone.

1. Go through any **open** Issues and Pull Requests marked with the current milestone. If there are any, they need to be resolved or bumped to the next milestone. It's fine to bump issues - they are typically marked with the earliest feasible milestone, but many such optimistically marked tasks aren't complete when it's time to release. There's always next time!
2. Check **closed** Issues and Pull Requests, using the milestone filter "Issues with no milestone". There should never be any closed issues or pull requests without a milestone. If you find any, go through and mark them with the current milestone or "no action" as appropriate.

A release may be ready to go when it has zero open issues or pull requests.

Release notes

Once all of the issues and pull requests are dealt with, it's time to make release notes. The smaller projects generally have a `changelog.rst` in the docs directory, where you can add a section for the new release. Look through the pull requests merged for the current milestone (this is why we use milestones), and write a short summary of the highlights of the changes in this release. There should generally be a link to the milestone itself for more details.

Make a pull requests with these notes. It's a good idea to cc @willingc for review of this PR. Make sure to mark this PR with your release's milestone!

Making the release

Now that your changelog is merged, we can actually build and publish the release. We'll assume that `V` has been declared as a shell variable containing the release version:

```
export V=5.1.2
```

Start by making sure you have a clean checkout of master, with no extra files:

```
git pull
git clean -xfd
```

First, update the version of the package, often in the file `<pkg>/_version.py` or similar.

Commit that change:

```
git commit -am "release $V"
```

Note: At this point, I like to run the tests just to be sure that setting the version didn't confuse anything.

Build the distributions:

```
python setup.py sdist --formats=gztar
python setup.py bdist_wheel
```

Tag the commit:

```
git tag -am "release $V" $V
```

And finally, publish everything, to github and PyPI using twine:

```
twine upload dist/*
git push origin
git push origin --tags
```

We have a release! You can now bump the version to the next '.dev' version, by editing `<pkg>/_version.py` (or similar) again, and commit:

```
git commit -am "back to dev"
git push origin
```

Note: The pushes assume that *origin* points to the main jupyter/ipython repo. Depending how you use git, this could be *upstream* or something else.

IPython Development Guide (source: old IPython wiki)

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

IPython does all of its development using GitHub. All of our development information is hosted on this GitHub wiki. This page indexes all of that information. Developers interested in getting involved with IPython development should start here.

IPython on GitHub

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

Notes on working with GitHub

Milestones

- 100% of confirmed issues should have a milestone.
- An issue should never be closed without a milestone.
- All pull requests should have a milestone.
- All issues closed should be marked with the next release milestone, next backport milestone, or **no action**.

- Open issues should only lack a milestone if:
  - more clarification is required (label: `needs-info`)
- In general, there will be four milestones with open issues:
  - **next minor release**. This milestone contains issues that should be backported for the next minor release. See *below* for information on backporting.
  - **next major release**. This should be the default milestone of all issues. As the release approaches, issues can be explicitly bumped to next release + 1.
  - **next major release + 1**. Only issues that we are confident will *not* be included in the next release go here. This milestone should be mostly empty until relatively close to release.
  - **wishlist**. This is the milestone for issues that we have no immediate plans to address.
- The remaining milestone is **no action** for open or closed issues that require no action:
  - Not actually an issue (e.g. questions, discussion, etc.)
  - Duplicate of an existing Issue
  - Closed because we won't fix it
  - When an issue is closed with **no action**, it means that the issue will not be fixed, or it was not an issue at all.
- When closing an issue, it should always have one of these milestones:
  - **next minor release** because the issue has been addressed
  - **next major release** because the issue has been addressed
  - **no action** because the issue *will not* be addressed, or it is a duplicate/non-issue.

In general: When in doubt, mark with **next release**. We can always push back when we get closer to a given release.

Labels and issues

Issues should always be labeled once they are confirmed (not necessary for issues that are still being clarified, or may be duplicates or not issues at all).

Some significant labels:

- `needs-info`: issue needs more information from submitter before progress can be made
- `bug`: errors are raised, or unintended behavior occurs
- `enhancement`: improvements that are not bugs
- backport-X.Y.Z: Any fix for this issue should be backported to the maintenance branch. backports are expressed with milestones, starting with 2.1.
- `prio-foo`: a priority level for ranking issues - nonessential, but prio-high/low are useful for explicitly promoting/demoting issues, particularly when nearing release.
- `ClosedPR`: This issue is a reminder for a PR that was closed for going stale.
- `sprint-friendly`: Obvious or easy fixes, where

All confirmed issues should at least have a `bug` or `enhancement` label, and be marked with any affected components (e.g `parallel`, `qtconsole`, `htmlnotebook`), or particular sources of error (e.g. `py3k` or `unicode`) if we have appropriate labels.

The `sprint-friendly` label is probably the best place for new contributors to start.

Pull Requests

- All work is submitted via Pull Requests.
- Pull Requests can be submitted as soon as there is code worth discussing. Pull Requests track the branch, so you can continue to work after the PR is submitted. Review and discussion can begin well before the work is complete, and the more discussion the better. The worst case is that the PR is closed.
- Pull Requests that have stalled should be closed (see [[our policy on closing PRs|Dev: Closing Pull Requests]])
- Pull Requests should always be made against master (backporting PRs is described below).
- Pull Requests should be tested, if feasible:
  - bugfixes should include regression tests
  - new behavior should at least get minimal exercise

Travis does a pretty good job testing IPython and Pull Requests, but it may make sense to manually perform tests (possibly with our `test_pr` script), particularly for PRs that affect `IPython.parallel` or Windows.

Opening an Issue

When opening a new issue, please take the following steps:

1. Search GitHub and/or Google for your issue to avoid duplicate reports. Keyword searches for your error messages are most helpful.
2. If possible, try updating to master and reproducing your issue, because we may have already fixed it.
3. Try to include a minimal reproducible test case
4. Include relevant system information. Start with the output of:

```
python -c "import IPython; print(IPython.sys_info())"
```

And include any relevant package versions, depending on the issue, such as matplotlib, numpy, Qt, Qt bindings (PyQt/PySide), tornado, web browser, etc.

Backporting

- We should keep an `A.x` maintenance branch for backporting fixes from master.
- That branch shall be called `A.x`, e.g. `2.x`, not `2.1`. This way, there is only one maintenance branch per release series.
- When an Issue is determined to be appropriate for backporting, it should be marked with the `A.B` milestone.
- Any Pull Request which addresses a backport issue should also receive the same milestone.
- Patches are backported to the maintenance branch by applying the pull request patch to the maintenance branch (currently with the backport_pr script).

The Perfect Pull Request

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

A brief guide to making and reviewing pull requests.

1. It works

The code does what it's supposed to!

2. It works on all of the platforms that IPython officially supports

IPython has to work on:

- Linux of various kinds, Windows & Mac
- Python 2 & 3

3. Handles unicode issues properly

Much of our code base deals with strings and unicode. This needs to be done in a manner that is unicode aware and works on Python 2 and 3. [This article] (http://www.joelonsoftware.com/articles/Unicode.html) is a good intro to unicode.

4. Adheres to our coding style

Coding style refers to how source code is formatted and how variables, functions, methods and classes are named. Your code should follow our coding style, which is described [[here|Dev: Coding style]].

5. Clean & commented

The code should be well organized, and have inline comments where appropriate. When we look at the code, it should be clear what it's doing and why. It should not break abstractions that we have established in the project.

6. Tested

If it fixes a bug, the pull request should ideally add an automated test that fails without the fix, and passes with it. Normally it should be sufficient to copy an existing test and tweak it. New functionality should come with its own tests as well. Details about testing IPython can be found [[here|Dev: Testing]].

7. Well documented

Don't forget to update docstrings, and any relevant parts of the official documentation. New features or significant changes warrant an entry in the *What's New* section too. Details about documenting IPython can be found [[here|Dev: Documenting IPython]].

Coding Style

> **Attention:** This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

This document describes our coding style. Coding style refers to the following:

- How source code is formatted (indentation, spacing, etc.)
- How things are named (variables, functions, classes, modules, etc.)

General coding conventions

In general, we follow the standard Python style conventions as described in Python's PEP 8, the official Python Style Guide.

Other general comments:

- In a large file, top level classes and functions should be separated by 2 lines to make it easier to separate them visually.
- Use 4 spaces for indentation, **never** use hard tabs.
- Keep the ordering of methods the same in classes that have the same methods. This is particularly true for classes that implement similar interfaces and for interfaces that are similar.

Naming conventions

For naming conventions, we also follow the guidelines of PEP 8. Some of the existing code doesn't honor this perfectly, but for all new and refactored IPython code, we'll use:

- All `lowercase` module names. Long module names can have words separated by underscores (`really_long_module_name.py`), but this is not required. Try to use the convention of nearby files.
- `CamelCase` for class names.
- `lowercase_with_underscores` for methods, functions, variables and attributes.
- Implementation-specific *private* methods will use `_single_underscore_prefix`. Names with a leading double underscore will *only* be used in special cases, as they makes subclassing difficult (such names are not easily seen by child classes).
- Occasionally some run-in lowercase names are used, but mostly for very short names or where we are implementing methods very similar to existing ones in a base class (like `runlines()` where `runsource()` and `runcode()` had established precedent).
- The old IPython codebase has a big mix of classes and modules prefixed with an explicit `IP` of `ip`. This is not necessary and all new code should not use this prefix. The only case where this approach is justified is for classes or functions which are expected to be imported into external namespaces and a very generic name (like Shell) that is likely to clash with something else. However, if a prefix seems absolutely necessary the more specific `IPY` or `ipy` are preferred.
- All JavaScript code should follow these naming conventions as well.

Attribute declarations for objects

In general, objects should declare, in their *class*, all attributes the object is meant to hold throughout its life. While Python allows you to add an attribute to an instance at any point in time, this makes the code harder to read and requires methods to constantly use checks with hasattr() or try/except calls. By declaring all attributes of the object in the class header, there is a single place one can refer to for understanding the object's data interface, where comments can explain the role of each variable and when possible, sensible deafaults can be assigned.

If an attribute is meant to contain a mutable object, it should be set to `None` in the class and its mutable value should be set in the object's constructor. Since class attributes are shared by all instances, failure to do this can lead to difficult to track bugs. But you should still set it in the class declaration so the interface specification is complete and documented in one place.

A simple example:

```
class Foo(object):
    # X does..., sensible default given:
    x = 1
    # y does..., default will be set by constructor
    y = None
    # z starts as an empty list, must be set in constructor
    z = None

    def __init__(self, y):
        self.y = y
        self.z = []
```

New files

When starting a new Python file for IPython, you can use the following template as a starting point that has a few common things pre-written for you.

Documenting IPython

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

When contributing code to IPython, you should strive for clarity and consistency, without falling prey to a style straitjacket. Basically, 'document everything, try to be consistent, do what makes sense.'

By and large we follow existing Python practices in major projects like Python itself or NumPy, this document provides some additional detail for IPython.

Standalone documentation

All standalone documentation should be written in plain text (`.txt`) files using reStructuredText [reStructuredText]_ for markup and formatting. All such documentation should be placed in the directory docs/source of the IPython source tree. Or, when appropriate, a suitably named subdirectory should be used. The documentation in this location will serve as the main source for IPython documentation.

The actual HTML and PDF docs are built using the Sphinx [Sphinx]_ documentation generation tool. Once you have Sphinx installed, you can build the html docs yourself by doing:

```
$ cd ipython-mybranch/docs
$ make html
```

Our usage of Sphinx follows that of matplotlib [Matplotlib]_ closely. We are using a number of Sphinx tools and extensions written by the matplotlib team and will mostly follow their conventions, which are nicely spelled out in their documentation guide [MatplotlibDocGuide]_. What follows is thus a abridged version of the matplotlib documentation guide, taken with permission from the matplotlib team.

If you are reading this in a web browser, you can click on the "Show Source" link to see the original reStructuredText for the following examples.

A bit of Python code:

```
for i in range(10):
    print i,
print "A big number:",2**34
```

An interactive Python session:

```
>>> from IPython.utils.path import get_ipython_dir
>>> get_ipython_dir()
'/home/fperez/.config/ipython'
```

An IPython session:

```
In [7]: import IPython

In [8]: print "This IPython is version:",IPython.__version__
This IPython is version: 0.9.1

In [9]: 2+4
Out[9]: 6
```

A bit of shell code:

```
cd /tmp
echo "My home directory is: $HOME"
ls
```

Docstring format

Good docstrings are very important. Unfortunately, Python itself only provides a rather loose standard for docstrings [PEP257]_, and there is no universally accepted convention for all the different parts of a complete docstring. However, the NumPy project has established a very reasonable standard, and has developed some tools to support the smooth inclusion of such docstrings in Sphinx-generated manuals. Rather than inventing yet another pseudo-standard, IPython will be henceforth documented using the NumPy conventions; we carry copies of some of the NumPy support tools to remain self-contained, but share back upstream with NumPy any improvements or fixes we may make to the tools.

The NumPy documentation guidelines [NumPyDocGuide]_ contain detailed information on this standard, and for a quick overview, the NumPy example docstring [NumPyExampleDocstring]_ is a useful read.

For user-facing APIs, we try to be fairly strict about following the above standards (even though they mean more verbose and detailed docstrings). Wherever you can reasonably expect people to do introspection with:

```
In [1]: some_function?
```

the docstring should follow the NumPy style and be fairly detailed.

For purely internal methods that are only likely to be read by others extending IPython itself we are a bit more relaxed, especially for small/short methods and functions whose intent is reasonably obvious. We still expect docstrings to be written, but they can be simpler. For very short functions with a single-line docstring you can use something like:

```
def add(a, b):
    """The sum of two numbers.
```

```
    """
    code
```

and for longer multiline strings:

```
def add(a, b):
    """The sum of two numbers.

    Here is the rest of the docs.
    """
    code
```

Here are two additional PEPs of interest regarding documentation of code. While both of these were rejected, the ideas therein form much of the basis of docutils (the machinery to process reStructuredText):

- Docstring Processing System Framework
- Docutils Design Specification

  note

  In the past IPython used epydoc so currently many docstrings still use epydoc conventions. We will update them as we go, but all new code should be documented using the NumPy standard.

Building and uploading

The built docs are stored in a separate repository. Through some github magic, they're automatically exposed as a website. It works like this:

- You will need to have sphinx and latex installed. In Ubuntu, install `texlive-latex-recommended texlive-latex-extra texlive-fonts-recommended`. Install the latest version of sphinx from PyPI (`pip install sphinx`).
- Ensure that the development version of IPython is the first in your system path. You can either use a virtualenv, or modify your PYTHONPATH.
- Switch into the docs directory, and run `make gh-pages`. This will build your updated docs as html and pdf, then automatically check out the latest version of the docs repository, copy the built docs into it, and commit your changes.
- Open the built docs in a web browser, and check that they're as expected.
- (When building the docs for a new tagged release, you will have to add its link to index.rst, then run `python build_index.py` to update index.html. Commit the change.)
- Upload the docs with `git push`. This only works if you have write access to the docs repository.
- If you are building a version that is not the current dev branch, nor a tagged release, then you must run gh-pages.py directly with `python gh-pages.py <version>`, and *not* with `make gh-pages`.

Lab Meetings on Air

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

Academic labs have long had the tradition of the weekly lab meeting, where all topics of interest to the lab are discussed. IPython has strong roots in academia, but it is also an open source project that needs to engage an active international community. So while our goal with IPython is not to publish the next paper, we've been thinking about the value these regular discussions bring to how teams work on sustained efforts involving difficult problems, and wanted to bring that bit of academic practice into the open source workflow. So we have decided to conduct weekly "lab meetings" for IPython, that will be held publicly using a Google Hangout on Air.

Logistics

We are trying to keep things simple and with a minimum of new moving parts:

- Meetings happen on Tuesdays at 10am California time (i.e. UTC-8 or -7 depending on the time of year).
- We broadcast the meeting as it happens via G+ and leave the public YouTube link afterwards.
- During the meeting, all chat that needs to happen by typing can be done on our Gitter chat room.
- We keep a running document with minutes of the meeting on HackPad where we summarize main points. (2015 part 1)

We welcome and encourage help from others in updating these minutes during the meeting, but we'll make no major effort in ensuring that they are a detailed and accurate record of the whole discussion. We simply don't have the time for that.

Prior meetings

You can find a list of the videos on the ipythondev YouTube user page.

Policy on Closing Pull Requests

> **Attention:** This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

IPython has the following policy on closing pull requests. The goal of this policy is to keep our pull request queue small and allow us to focus on code that is being actively developed and has a strong chance of being merged in master soon.

A pull request will be closed when:

- It has been reviewed, but has sat for a month or more waiting for the submitter to commit more code to address the comments.
- The review process has uncovered larger design or technical issues that extend beyond the details of the specific pull request.
  - In particular, we do not accept whole large "cleanup" changes which do not address any specific bug. This includes trailing whitespace, PEP8, etc. One of the reasons is that such massive cleanup provide plenty of opportunities to introduce new and subtle bugs.

In general we will not close pull requests because of a lack of review. If a pull request has sat for a month or more without review, we need to kick ourselves and get to reviewing it.

When a pull request is closed we will do the following:

- Post a github message to the pull request to confirm that everyone is fine with closing the pull request. This message should cite this policy.
- Open an issue to track the pull request. This issue should describe what would be needed in order to reopen the pull request.
- Post a github message to the pull request encouraging the submitter to continue with the work and detail what issues need to be addressed in order for the pull request to be reopened.

This policy was discussed in the following thread:

https://mail.scipy.org/pipermail/ipython-dev/2012-August/010025.html

Example Message:

```
Hi,

This PR has been inactive for 1 month now, so we are going to close it and open an
issue to reference it. We try to keep our pull request queue small and focused on
active work.  We encourage you to reopen the pull request if and when you
continue to work on this. Please contact us if you have any questions.

Thanks for contributing.

see https://github.com/ipython/ipython/wiki/Dev%3A-Closing-pull-requests/ for
our policies on closing pull request.
```

Testing IPython for users and developers

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

Overview

It is extremely important that all code contributed to IPython has tests. Tests should be written as unittests, doctests or other entities that the IPython test system can detect. See below for more details on this.

Each subpackage in IPython should have its own `tests` directory that contains all of the tests for that subpackage. All of the files in the `tests` directory should have the word "tests" in them to enable the testing framework to find them.

In docstrings, examples (either using IPython prompts like `In [1]:` or 'classic' python >>> ones) can and should be included. The testing system will detect them as doctests and will run them; it offers control to skip parts or all of a specific doctest if the example is meant to be informative but shows non-reproducible information (like filesystem data).

If a subpackage has any dependencies beyond the Python standard library, the tests for that subpackage should be skipped if the dependencies are not found. This is very important so users don't get tests failing simply because they don't have dependencies.

The testing system we use is an extension of the nose test runner. In particular we've developed a nose plugin that allows us to paste verbatim IPython sessions and test them as doctests, which is extremely important for us.

Running the test suite

You can run IPython from the source download directory without even installing it system-wide or having configure anything, by typing at the terminal:

```
python2 -c "import IPython; IPython.start_ipython();"
```

To start the webbased notebook you can use:

```
python2 -c "import IPython; IPython.start_ipython(['notebook']);"
```

In order to run the test suite, you must at least be able to import IPython, even if you haven't fully installed the user-facing scripts yet (common in a development environment). You can then run the tests with:

```
python -c "import IPython; IPython.test()"
```

Once you have installed IPython either via a full install or using:

```
python setup.py develop
```

you will have available a system-wide script called `iptest` that runs the full test suite. You can then run the suite with:

```
iptest  [args]
```

By default, this excludes the relatively slow tests for `IPython.parallel`. To run these, use `iptest --all`.

Please note that the iptest tool will run tests against the code imported by the Python interpreter. If the command `python setup.py symlink` has been previously run then this will always be the test code in the local directory via a symlink. However, if this command has not been run for the version of Python being tested, there is the possibility that iptest will run the tests against an installed version of IPython.

Regardless of how you run things, you should eventually see something like:

```
**********************************************************************
Test suite completed for system with the following information:
{'commit_hash': '144fdae',
 'commit_source': 'repository',
 'ipython_path': '/home/fperez/usr/lib/python2.6/site-packages/IPython',
 'ipython_version': '0.11.dev',
 'os_name': 'posix',
 'platform': 'Linux-2.6.35-22-generic-i686-with-Ubuntu-10.10-maverick',
 'sys_executable': '/usr/bin/python',
 'sys_platform': 'linux2',
 'sys_version': '2.6.6 (r266:84292, Sep 15 2010, 15:52:39) \n[GCC 4.4.5]'}

Tools and libraries available at test time:
   curses matplotlib pymongo qt sqlite3 tornado wx wx.aui zmq

Ran 9 test groups in 67.213s

Status:
OK
```

If not, there will be a message indicating which test group failed and how to rerun that group individually. For example, this tests the `IPython.utils` subpackage, the `-v` option shows progress indicators:

```
$ iptest IPython.utils -- -v
.........................SS..SSS.............................S.S...
.....................................................
----------------------------------------------------------------------
Ran 125 tests in 0.119s

OK (SKIP=7)
```

Because the IPython test machinery is based on nose, you can use all nose syntax. Options after `--` are passed to nose. For example, this lets you run the specific test `test_rehashx` inside the `test_magic` module:

```
$ iptest IPython.core.tests.test_magic:test_rehashx -- -vv
IPython.core.tests.test_magic.test_rehashx(True,) ... ok
IPython.core.tests.test_magic.test_rehashx(True,) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.100s

OK
```

When developing, the `--pdb` and `--pdb-failures` of nose are particularly useful, these drop you into an interactive pdb session at the point of the error or failure respectively: `iptest mymodule -- --pdb`.

The system information summary printed above is accessible from the top level package. If you encounter a problem with IPython, it's useful to include this information when reporting on the mailing list; use:

```
.. code:: python
```

from IPython import sys_info print sys_info()

and include the resulting information in your query.

Testing pull requests

We have a script that fetches a pull request from Github, merges it with master, and runs the test suite on different versions of Python. This uses a separate copy of the repository, so you can keep working on the code while it runs. To run it:

```
python tools/test_pr.py -p 1234
```

The number is the pull request number from Github; the `-p` flag makes it post the results to a comment on the pull request. Any further arguments are passed to `iptest`.

This requires the requests and keyring packages.

For developers: writing tests

By now IPython has a reasonable test suite, so the best way to see what's available is to look at the `tests` directory in most subpackages. But here are a few pointers to make the process easier.

Main tools: `IPython.testing`

The `IPython.testing` package is where all of the machinery to test IPython (rather than the tests for its various parts) lives. In particular, the `iptest` module in there has all the smarts to control the test process. In there, the

`make_exclude` function is used to build a blacklist of exclusions, these are modules that do not get even imported for tests. This is important so that things that would fail to even import because of missing dependencies don't give errors to end users, as we stated above.

The `decorators` module contains a lot of useful decorators, especially useful to mark individual tests that should be skipped under certain conditions (rather than blacklisting the package altogether because of a missing major dependency).

Our nose plugin for doctests

The `plugin` subpackage in testing contains a nose plugin called `ipdoctest` that teaches nose about IPython syntax, so you can write doctests with IPython prompts. You can also mark doctest output with `# random` for the output corresponding to a single input to be ignored (stronger than using ellipsis and useful to keep it as an example). If you want the entire docstring to be executed but none of the output from any input to be checked, you can use the `# all-random` marker. The `IPython.testing.plugin.dtexample` module contains examples of how to use these; for reference here is how to use `# random`:

```
def ranfunc():
"""A function with some random output.

   Normal examples are verified as usual:
   >>> 1+3
   4

   But if you put '# random' in the output, it is ignored:
   >>> 1+3
   junk goes here...   # random

   >>> 1+2
   again,  anything goes #random
   if multiline, the random mark is only needed once.

   >>> 1+2
   You can also put the random marker at the end:
   # random

   >>> 1+2
   # random
   .. or at the beginning.

   More correct input is properly verified:
   >>> ranfunc()
   'ranfunc'
"""
return 'ranfunc'
```

and an example of `# all-random`:

```
def random_all():
"""A function where we ignore the output of ALL examples.

Examples:

  # all-random

  This mark tells the testing machinery that all subsequent examples
  should be treated as random (ignoring their output).  They are still
```

```
  executed, so if a they raise an error, it will be detected as such,
  but their output is completely ignored.

  >>> 1+3
  junk goes here...

  >>> 1+3
  klasdfj;

In [8]: print 'hello'
world  # random

In [9]: iprand()
Out[9]: 'iprand'
"""
return 'iprand'
```

When writing docstrings, you can use the `@skip_doctest` decorator to indicate that a docstring should *not* be treated as a doctest at all. The difference between `# all-random` and `@skip_doctest` is that the former executes the example but ignores output, while the latter doesn't execute any code. `@skip_doctest` should be used for docstrings whose examples are purely informational.

If a given docstring fails under certain conditions but otherwise is a good doctest, you can use code like the following, that relies on the 'null' decorator to leave the docstring intact where it works as a test:

```
# The docstring for full_path doctests differently on win32 (different path
# separator) so just skip the doctest there, and use a null decorator
# elsewhere:

doctest_deco = dec.skip_doctest if sys.platform == 'win32' else dec.null_deco

@doctest_deco
def full_path(startPath,files):
    """Make full paths for all the listed files, based on startPath..."""

    # function body follows...
```

With our nose plugin that understands IPython syntax, an extremely effective way to write tests is to simply copy and paste an interactive session into a docstring. You can writing this type of test, where your docstring is meant *only* as a test, by prefixing the function name with `doctest_` and leaving its body *absolutely empty* other than the docstring. In `IPython.core.tests.test_magic` you can find several examples of this, but for completeness sake, your code should look like this (a simple case):

```
def doctest_time():
"""
In [10]: %time None
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00 s
"""
```

This function is only analyzed for its docstring but it is not considered a separate test, which is why its body should be empty.

JavaScript Tests

We currently use casperjs for testing the notebook javascript user interface.

To run the JS test suite by itself, you can either use `iptest js`, which will start up a new notebook server and test against it, or you can open up a notebook server yourself, and then:

```
cd IPython/html/tests/casperjs;
casperjs test --includes=util.js test_cases
```

If your testing notebook server uses something other than the default port (8888), you will have to pass that as a parameter to the test suite as well.

```
casperjs test --includes=util.js --port=8889 test_cases
```

Running individual tests

To speed up development, you usually are working on getting one test passing at a time. To do this, just pass the filename directly to the `casperjs test` command like so:

```
casperjs test --includes=util.js  test_cases/execute_code_cell.js
```

Wrapping your head around the javascript within javascript:

CasperJS is a browser that's written in javascript, so we write javascript code to drive it. The Casper browser itself also has a javascript implementation (like the ones that come with Firefox and Chrome), and in the test suite we get access to those using `this.evaluate`, and it's cousins (`this.theEvaluate`, etc). Additionally, because of the asynchronous / callback nature of everything, there are plenty of `this.then` calls which define steps in test suite. Part of the reason for this is that each step has a timeout (default of 5 or 10 seconds). Additionally, there are already convenience functions in `util.js` to help you wait for output in a given cell, etc. In our javascript tests, if you see functions which `look_like_pep8_naming_convention`, those are probably coming from `util.js`, whereas functions that come with casper `haveCamelCaseNamingConvention`

Each file in `test_cases` looks something like this (this is `test_cases/check_interrupt.js`):

```
casper.notebook_test(function () {
    this.evaluate(function () {
        var cell = IPython.notebook.get_cell(0);
        cell.set_text('import time\nfor x in range(3):\n    time.sleep(1)');
        cell.execute();
    });


    // interrupt using menu item (Kernel -> Interrupt)
    this.thenClick('li#int_kernel');

    this.wait_for_output(0);

    this.then(function () {
        var result = this.get_output_cell(0);
        this.test.assertEquals(result.ename, 'KeyboardInterrupt', 'keyboard interrupt␣
↪(mouseclick)');
    });

    // run cell 0 again, now interrupting using keyboard shortcut
    this.thenEvaluate(function () {
        cell.clear_output();
        cell.execute();
```

```
    });

    // interrupt using Ctrl-M I keyboard shortcut
    this.thenEvaluate( function() {
        IPython.utils.press_ghetto(IPython.utils.keycodes.I)
    });

    this.wait_for_output(0);

    this.then(function () {
        var result = this.get_output_cell(0);
        this.test.assertEquals(result.ename, 'KeyboardInterrupt', 'keyboard interrupt␣
↪(shortcut)');
    });
});
```

For an example of how to pass parameters to the client-side javascript from casper test suite, see the `casper.wait_for_output` implementation in `IPython/html/tests/casperjs/util.js`

Testing system design notes

This section is a set of notes on the key points of the IPython testing needs, that were used when writing the system and should be kept for reference as it eveolves.

Testing IPython in full requires modifications to the default behavior of nose and doctest, because the IPython prompt is not recognized to determine Python input, and because IPython admits user input that is not valid Python (things like `%magics` and `!system commands`.

We basically need to be able to test the following types of code:

- 1. Pure Python files containing normal tests. These are not a problem, since Nose will pick them up as long as they conform to the (flexible) conventions used by nose to recognize tests.
- 2. Python files containing doctests. Here, we have two possibilities:
- The prompts are the usual >>> and the input is pure Python.
- The prompts are of the form `In [1]:` and the input can contain extended IPython expressions.

In the first case, Nose will recognize the doctests as long as it is called with the `--with-doctest` flag. But the second case will likely require modifications or the writing of a new doctest plugin for Nose that is IPython-aware.

- 3. ReStructuredText files that contain code blocks. For this type of file, we have three distinct possibilities for the code blocks:
- They use >>> prompts.
- They use `In [1]:` prompts.
- They are standalone blocks of pure Python code without any prompts.

The first two cases are similar to the situation #2 above, except that in this case the doctests must be extracted from input code blocks using docutils instead of from the Python docstrings.

In the third case, we must have a convention for distinguishing code blocks that are meant for execution from others that may be snippets of shell code or other examples not meant to be run. One possibility is to assume that all indented code blocks are meant for execution, but to have a special docutils directive for input that should not be executed.

For those code blocks that we will execute, the convention used will simply be that they get called and are considered successful if they run to completion without raising errors. This is similar to what Nose does for standalone test

functions, and by putting asserts or other forms of exception-raising statements it becomes possible to have literate examples that double as lightweight tests.

- 4. Extension modules with doctests in function and method docstrings. Currently Nose simply can't find these docstrings correctly, because the underlying doctest DocTestFinder object fails there. Similarly to #2 above, the docstrings could have either pure python or IPython prompts.

Of these, only 3-c (reST with standalone code blocks) is not implemented at this point.

How to Compile .less Files

> **Attention:** This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

For testing your development work in CSS, you'll need to compile the .less files to CSS. Make sure you have dependencies that LESS requires, including fabric, node, and lessc. Follow the below steps to compile the .less files:

```
python setup.py css
```

Alternatively, you can:

```
$ cd ipython/IPython/html
$ fab css
[localhost] local: components/less.js/bin/lessc -x  style/style.less style/style.min.
↪css
[localhost] local: components/less.js/bin/lessc -x  style/ipython.less style/ipython.
↪min.css

Done
```

Steps for Releasing IPython

> **Attention:** This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

This document contains notes about the process that is used to release IPython. Our release process is currently not very formal and could be improved.

Most of the release process is automated by the `release` script in the `tools` directory of our main repository. This document is just a handy reminder for the release manager.

0. Environment variables

You can set some env variables to note previous release tag and current release milestone, version, and git tag:

```
PREV_RELEASE=rel-1.0.0
MILESTONE=1.1
VERSION=1.1.0
TAG="rel-$VERSION"
BRANCH=master
```

These will be used later if you want to copy/paste, or you can just type the appropriate command when the time comes. These variables are not used by scripts (hence no `export`).

1. Finish release notes

- If a major release:
- merge any pull request notes into what's new:

```
python tools/update_whatsnew.py
```

- update `docs/source/whatsnew/development.rst`, to ensure it covers the major points.
- move the contents of `development.rst` to `versionX.rst`
- generate summary of GitHub contributions, which can be done with:

```
python tools/github_stats.py --milestone $MILESTONE > stats.rst
```

which may need some manual cleanup. Add the cleaned up result and add it to `docs/source/whatsnew/github-stats-X.rst` (make a new file, or add it to the top, depending on whether it is a major release). You can use:

```
git log --format="%aN <%aE>" $PREV_RELEASE... | sort -u -f
```

to find duplicates and update `.mailmap`. Before generating the GitHub stats, verify that all closed issues and pull requests *have appropriate milestones*. This search should return no results.

2. Run the `tools/build_release` script

This does all the file checking and building that the real release script will do. This will let you do test installations, check that the build procedure runs OK, etc. You may want to also do a test build of the docs.

3. Create and push the new tag

Edit `IPython/core/release.py` to have the current version.

Commit the changes to release.py and jsversion:

```
git commit -am "release $VERSION"
git push origin $BRANCH
```

Create and push the tag:

```
git tag -am "release $VERSION" "$TAG"
git push origin --tags
```

Update release.py back to `x.y-dev` or `x.y-maint`, and push:

```
git commit -am "back to development"
git push origin $BRANCH
```

4. Get a fresh clone of the tag for building the release:

```
cd /tmp
git clone --depth 1 https://github.com/ipython/ipython.git -b "$TAG"
```

5. Run the `release` script

```
cd tools && ./release
```

This makes the tarballs, zipfiles, and wheels. It posts them to archive.ipython.org and registers the release with PyPI.

This will require that you have current wheel, Python 3.4 and Python 2.7.

7. Update the IPython website

- release announcement (news, announcements)
- update current version and download links
- (If major release) update links on the documentation page

8. Drafting a short release announcement

This should include i) highlights and ii) a link to the html version of the *What's new* section of the documentation.

Post to mailing list, and link from Twitter.

9. Update milestones on GitHub

- close the milestone you just released
- open new milestone for (x, y+1), if it doesn't exist already

10. Celebrate!

IPython Sphinx Directive

> **Attention:** This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

The ipython directive is a stateful ipython shell for embedding in sphinx documents. It knows about standard ipython prompts, and extracts the input and output lines. These prompts will be renumbered starting at `1`. The inputs will be fed to an embedded ipython interpreter and the outputs from that interpreter will be inserted as well. For example, code blocks like the following:

```
.. code:: python3

   In [136]: x = 2
```

```
   In [137]: x**3
   Out[137]: 8
```

will be rendered as

```
In [136]: x = 2

In [137]: x**3
Out[137]: 8
```

Note: This tutorial should be read side-by-side with the Sphinx source for this document because otherwise you will see only the rendered output and not the code that generated it. Excepting the example above, we will not in general be showing the literal ReST in this document that generates the rendered output.

The state from previous sessions is stored, and standard error is trapped. At doc build time, ipython's output and std err will be inserted, and prompts will be renumbered. So the prompt below should be renumbered in the rendered docs, and pick up where the block above left off.

```
In [138]: z = x*3   # x is recalled from previous block

In [139]: z
Out[139]: 6

In [140]: print z
--------> print(z)
6

In [141]: q = z[]   # this is a syntax error -- we trap ipy exceptions
------------------------------------------------------------
   File "<ipython console>", line 1
     q = z[]   # this is a syntax error -- we trap ipy exceptions
     ^
SyntaxError: invalid syntax
```

The embedded interpreter supports some limited markup. For example, you can put comments in your ipython sessions, which are reported verbatim. There are some handy "pseudo-decorators" that let you doctest the output. The inputs are fed to an embedded ipython session and the outputs from the ipython session are inserted into your doc. If the output in your doc and in the ipython session don't match on a doctest assertion, an error will be

```
In [1]: x = 'hello world'

# this will raise an error if the ipython output is different
@doctest
In [2]: x.upper()
Out[2]: 'HELLO WORLD'

# some readline features cannot be supported, so we allow
# "verbatim" blocks, which are dumped in verbatim except prompts
# are continuously numbered
@verbatim
In [3]: x.st<TAB>
x.startswith  x.strip
```

Multi-line input is supported.

```
In [130]: url = 'http://ichart.finance.yahoo.com/table.csv?s=CROX\
   .....: &d=9&e=22&f=2009&g=d&a=1&br=8&c=2006&ignore=.csv'

In [131]: print url.split('&')
--------> print(url.split('&'))
['http://ichart.finance.yahoo.com/table.csv?s=CROX', 'd=9', 'e=22',
```

You can do doctesting on multi-line output as well. Just be careful when using non-deterministic inputs like random numbers in the ipython directive, because your inputs are ruin through a live interpreter, so if you are doctesting random output you will get an error. Here we "seed" the random number generator for deterministic output, and we suppress the seed line so it doesn't show up in the rendered output

```
In [133]: import numpy.random

@suppress
In [134]: numpy.random.seed(2358)

@doctest
In [135]: numpy.random.rand(10,2)
Out[135]:
array([[ 0.64524308,  0.59943846],
 [ 0.47102322,  0.8715456 ],
 [ 0.29370834,  0.74776844],
 [ 0.99539577,  0.1313423 ],
 [ 0.16250302,  0.21103583],
 [ 0.81626524,  0.1312433 ],
 [ 0.67338089,  0.72302393],
 [ 0.7566368 ,  0.07033696],
 [ 0.22591016,  0.77731835],
 [ 0.0072729 ,  0.34273127]])
```

Another demonstration of multi-line input and output

```
In [106]: print x
--------> print(x)
jdh

In [109]: for i in range(10):
   .....:     print i
   .....:
   .....:
0
1
2
3
4
5
6
7
8
9
```

Most of the "pseudo-decorators" can be used an options to ipython mode. For example, to setup matplotlib pylab but suppress the output, you can do. When using the matplotlib `use` directive, it should occur before any import of pylab. This will not show up in the rendered docs, but the commands will be executed in the embedded interpreter and subsequent line numbers will be incremented to reflect the inputs:

```
.. code:: python3

   In [144]: from pylab import *

   In [145]: ion()
```

```
In [144]: from pylab import *

In [145]: ion()
```

Likewise, you can set `:doctest:` or `:verbatim:` to apply these settings to the entire block. For example,

```
In [9]: cd mpl/examples/
/home/jdhunter/mpl/examples

In [10]: pwd
Out[10]: '/home/jdhunter/mpl/examples'


In [14]: cd mpl/examples/<TAB>
mpl/examples/animation/         mpl/examples/misc/
mpl/examples/api/               mpl/examples/mplot3d/
mpl/examples/axes_grid/         mpl/examples/pylab_examples/
mpl/examples/event_handling/    mpl/examples/widgets

In [14]: cd mpl/examples/widgets/
/home/msierig/mpl/examples/widgets

In [15]: !wc *
    2    12    77 README.txt
   40    97   884 buttons.py
   26    90   712 check_buttons.py
   19    52   416 cursor.py
  180   404  4882 menu.py
   16    45   337 multicursor.py
   36   106   916 radio_buttons.py
   48   226  2082 rectangle_selector.py
   43   118  1063 slider_demo.py
   40   124  1088 span_selector.py
  450  1274 12457 total
```

You can create one or more pyplot plots and insert them with the `@savefig` decorator.

```
@savefig plot_simple.png width=4in
In [151]: plot([1,2,3]);

# use a semicolon to suppress the output
@savefig hist_simple.png width=4in
In [151]: hist(np.random.randn(10000), 100);
```

In a subsequent session, we can update the current figure with some text, and then resave

```
In [151]: ylabel('number')

In [152]: title('normal distribution')

@savefig hist_with_text.png width=4in
In [153]: grid(True)
```

You can also have function definitions included in the source.

```
In [3]: def square(x):
   ...:     """
   ...:     An overcomplicated square function as an example.
   ...:     """
   ...:     if x < 0:
   ...:         x = abs(x)
   ...:     y = x * x
   ...:     return y
   ...:
```

Then call it from a subsequent section.

```
In [4]: square(3)
Out [4]: 9

In [5]: square(-2)
Out [5]: 4
```

Writing Pure Python Code

Pure python code is supported by the optional argument *python*. In this pure python syntax you do not include the output from the python interpreter. The following markup:

```
.. code:: python

   foo = 'bar'
   print foo
   foo = 2
   foo**2
```

Renders as

```
foo = 'bar'
print foo
foo = 2
foo**2
```

We can even plot from python, using the savefig decorator, as well as, suppress output with a semicolon

```
@savefig plot_simple_python.png width=4in
plot([1,2,3]);
```

Similarly, std err is inserted

```
foo = 'bar'
foo[]
```

Comments are handled and state is preserved

```
# comments are handled
print foo
```

If you don't see the next code block then the options work.

```
ioff()
ion()
```

Multi-line input is handled.

```
line = 'Multi\
        line &\
        support &\
        works'
print line.split('&')
```

Functions definitions are correctly parsed

```
def square(x):
    """
    An overcomplicated square function as an example.
    """
    if x < 0:
        x = abs(x)
    y = x * x
    return y
```

And persist across sessions

```
print square(3)
print square(-2)
```

Pretty much anything you can do with the ipython code, you can do with with a simple python script. Obviously, though it doesn't make sense to use the doctest option.

Pseudo-Decorators

Here are the supported decorators, and any optional arguments they take. Some of the decorators can be used as options to the entire block (eg `verbatim` and `suppress`), and some only apply to the line just below them (eg `savefig`).

@suppress

> execute the ipython input block, but suppress the input and output block from the rendered output. Also, can be applied to the entire `..ipython` block as a directive option with `:suppress:`.

@verbatim

> insert the input and output block in verbatim, but auto-increment the line numbers. Internally, the interpreter will be fed an empty string, so it is a no-op that keeps line numbering consistent. Also, can be applied to the entire `..ipython` block as a directive option with `:verbatim:`.

@savefig OUTFILE [IMAGE_OPTIONS]

> save the figure to the static directory and insert it into the document, possibly binding it into a minipage and/or putting code/figure label/references to associate the code and the figure. Takes args to pass to the image directive (*scale*, *width*, etc can be kwargs); see image options for details.

@doctest

> Compare the pasted in output in the ipython block with the output generated at doc build time, and raise errors if they donâ€™t match. Also, can be applied to the entire `..ipython` block as a directive option with `:doctest:`.

Configuration Options

ipython_savefig_dir

The directory in which to save the figures. This is relative to the Sphinx source directory. The default is *html_static_path*.

ipython_rgxin

The compiled regular expression to denote the start of IPython input lines. The default is re.compile('In [(d+)]:s?(.*)s*'). You shouldn't need to change this.

ipython_rgxout

The compiled regular expression to denote the start of IPython output lines. The default is re.compile('Out[(d+)]:s?(.*)s*'). You shouldn't need to change this.

ipython_promptin

The string to represent the IPython input prompt in the generated ReST. The default is 'In [%d]:'. This expects that the line numbers are used in the prompt.

ipython_promptout

The string to represent the IPython prompt in the generated ReST. The default is 'Out [%d]:'. This expects that the line numbers are used in the prompt.

Python 3 Compatibility Module

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

The `IPython.utils.py3compat` module provides a number of functions to make it easier to write code for Python 2 and 3. We also use 2to3 in the setup process to change syntax, and the `io.open()` function, which is essentially the built in open function from Python 3.

The names provided are:

- **PY3**: True in Python 3, False in Python 2.

Unicode related

- **decode**, **encode**: Shortcuts to decode or encode strings, using `sys.stdin.encoding` by default, and using replacement characters on errors.
- **str_to_unicode**, **unicode_to_str**, **str_to_bytes**, **bytes_to_str**: Convert to/from the platform's standard `str` type (bytes in Python 2, unicode in Python 3). Each function is a no-op on one of the two platforms.
- **cast_unicode**, **cast_bytes**: Accept unknown unicode or byte strings, and convert them accordingly.
- **cast_bytes_py2**: Casts unicode to byte strings on Python 2, but doesn't do anything on Python 3.

Miscellaneous

- **input**: Refers to `raw_input` on Python 2, `input` on Python 3 (needed because 2to3 only converts calls to raw_input, not assignments to other names).

- **builtin_mod_name**: The string name you import to get the builtins (`__builtin__` -> `builtins`).
- **isidentifier**: Checks if a string is a valid Python identifier.
- **open**: Simple wrapper for Python 3 unicode-enabled open. Similar to `codecs.open`, but allows universal newlines. The current implementation only supports the very simplest use.
- **MethodType**: `types.MethodType` from Python 3. Takes only two arguments: function, instance. The class argument for Python 2 is filled automatically.
- **doctest_refactor_print**: Can be called on a string or a function (or used as a decorator). In Python 3, it converts print statements in doctests to print() calls. 2to3 does this for real doctests, but we need it in several other places. It simply uses a regex, which is good enough for the current cases.
- **u_format**: Where tests use the repr() of a unicode string, it should be written `'{u}"thestring"'`, and fed to this function, which will produce `'u"thestring"'` for Python 2, and `'"thestring"'` for Python 3. Can also be used as a decorator, to work on a docstring.
- **execfile**: Makes a return on Python 3 (where it's no longer a builtin), and upgraded to handle Unicode filenames on Python 2.

Architecture of IPython notebook's Dashboard

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

The tables below show the current RESTful web service architecture implemented in IPython notebook. The listed URL's use the HTTP verbs to return representations of the desired resource.

We are in the process of creating a new dashboard architecture for the IPython notebook, which will allow the user to navigate through multiple directory files to find desired notebooks.

Current Architecture

Miscellaneous

HTTP verb	URL	Action
`GET`	/.*/	Strips trailing slashes.
`GET`	/api	Returns api version information.
`*`	/api/notebooks	Deprecated: redirect to /api/contents
`GET`	/api/nbconvert	

Notebook contents API.

HTTP verb	URL	Action
GET	/api/contents	Return a model for the base directory. See /api/contents/<path>/<file>.
GET	/api/contents/ <file>	Return a model for the given file in the base directory. See /api/contents/<path>/<file>.
GET	/api/contents/ <path>/<file>	Return a model for a file or directory. A directory model contains a list of models (without content) of the files and directories it contains.
PUT	/api/contents/ <path>/<file>	Saves the file in the location specified by name and path. PUT is very similar to POST, but the requester specifies the name, where as with POST, the server picks the name. PUT /api/contents/path/Name.ipynb Save notebook at `path/Name.ipynb`. Notebook structure is specified in `content` key of JSON request body. If content is not specified, create a new empty notebook. PUT /api/contents/path/Name.ipynb with JSON body {"copy_from" : "[path/to/] OtherNotebook.ipynb"} Copy OtherNotebook to Name
PATCH	/api/contents/ <path>/<file>	Renames a notebook without re-uploading content.
POST	/api/contents/ <path>/<file>	Creates a new file or directory in the specified path. POST creates new files or directories. The server always decides on the name. POST /api/contents/path New untitled notebook in path. If content specified, upload a notebook, otherwise start empty. POST /api/contents/path with body {"copy_from":"OtherNotebook.ipynb"} New copy of OtherNotebook in path
DELETE	/api/contents/ <path>/<file>	delete a file in the given path.
GET	/api/contents/ <path>/<file> /checkpoints	get lists checkpoint for a file.
POST	/api/contents/ <path>/<file> /checkpoints	post creates a new checkpoint.
POST	/api/contents/ <path>/<file> /checkpoints/ <checkpoint_ id>	post restores a file from a checkpoint.
DELETE	/api/contents/ <path>/<file> /checkpoints/ <checkpoint_ id>	delete clears a checkpoint for a given file.

Kernel API

HTTP verb	URI	Action
GET	/api/kernels	Return a model of all kernels.
GET	/api/kernels /<kernel_id>	Return a model of kernel with given kernel id.
POST	/api/kernels	Start a new kernel with default or given name.
DELETE	/api/kernels /<kernel_id>	Shutdown the given kernel.
POST	/api/kernels /<kernel_id> /<action>	Perform action on kernel with given kernel id. Actions can be "interrupt" or "restart".
WS	/api/kernels /<kernel_id> /channels	Websocket stream
GET	/api/kernel specs	Return a spec model of all available kernels.
GET	/api/kernel specs/ <kernel_name>	Return a spec model of all available kernels with a given kernel name.

Sessions API

HTTP verb	URL	Action
GET	/api/sesions	Return model of active sessions.
POST	/api/sessions	If session does not already exist, create a new session with given notebook name and path and given kernel name. Return active sesssion.
GET	/api/sessions /<session_id>	Return model of active session with given session id.
PATCH	/api/sessions /<session_id>	Return model of active session with notebook name or path of session with given session id.
DELETE	/api/sessions /<session_id>	Delete model of active session with given session id.

Clusters API

HTTP verb	URL	Action
GET	/api/clusters	Return model of clusters.
GET	/api/clusters <cluster_id>	Return model of given cluster.
POST	/api/clusters <cluster_id> <action>	Perform action with given clusters. Valid actions are “start” and “stop”

Old Architecture

This chart shows the web-services in the single directory IPython notebook.

HTTP verb	URL	Action
GET	/notebooks	return list of dicts with each notebook’s info
POST	/notebooks	if sending a body, saving that body as a new notebook; if no body, create a a new notebook.
GET	/notebooks /<notebook_id>	get JSON data for notebook
PUT	/notebooks /<notebook_id>	saves an existing notebook with body data
DELETE	/notebooks /<notebook_id>	deletes the notebook with the given ID

This chart shows the architecture for the IPython notebook website.

HTTP verb	URL	Action
GET	/	navigates user to dashboard of notebooks and clusters.
GET	/<notebook_id>	go to wepage for that notebook
GET	/new	creates a new notebook with profile (or default, if no profile exists) setings
GET	/<notebook_id> /copy	opens a duplicate copy or the notebook with the given ID in a a new tab
GET	/<notebook_id> /print	prints the notebook with the given ID; if notebook doesn’t exist, displays error message
GET	/login	navigates to login page; if no user profile is defined, it navigates user to dashboard
GET	/logout	logs out of current profile, and navigates user to login page

This chart shows the Web services that act on the kernels and clusters.

HTTP verb	URL	Action
GET	/kernels	return the list of kernel IDs currently running
GET	/kernels /<kernel_id>	—
GET	/kernels /<kernel_id> <action>	performs action (restart/kill) kernel with given ID
GET	/kernels /<kernel_id> /iopub	—
GET	/kernels /<kernel_id> /shell	—
GET	/rstservice/ render	—
GET	/files/(.*)	—
GET	/clusters	returns a list of dicts with each cluster's information
POST	/clusters /<profile_id> /<cluster_ action>	performs action (start/stop) on cluster with given profile ID
GET	/clusters /<profile_id>	returns the JSON data for cluster with given profile ID

JavaScript Events

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

(Note: this page is not currently consistent with IPython master)

Javascript events are used to notify unrelated parts of the notebook interface when something happens. For example, if the kernel is busy executing code, it may send an event announcing as such, which can then be picked up by other services, like the notification area. For details on how the events themselves work, see the JQuery documentation.

This page documents the core set of events, and explains when and why they are triggered.

Cell-related events

- *command_mode.Cell*
- *create.Cell*
- *delete.Cell*
- *edit_mode.Cell*
- *select.Cell*
- *output_appended.OutputArea*

CellToolbar-related events

- *preset_activated.CellToolbar*
- *preset_added.CellToolbar*

Dashboard-related events

- *app_initialized.DashboardApp*
- *sessions_loaded.Dashboard*

app_initialized.DashboardApp

When the iPython Notebook browser window opens for the first time and initializes the Dashboard App. The Dashboard App lists the files and notebooks in the current directory. Additionally, it lets you create and open new iPython Notebooks.

Kernel-related events

- *execution_request.Kernel*
- *input_reply.Kernel*
- *kernel_autorestarting.Kernel*
- *kernel_busy.Kernel*
- *kernel_connected.Kernel*
- *kernel_connection_failed.Kernel*
- *kernel_created.Kernel*
- *kernel_created.Session*
- *kernel_dead.Kernel*
- *kernel_dead.Session*
- *kernel_disconnected.Kernel*
- *kernel_idle.Kernel*
- *kernel_interrupting.Kernel*
- *kernel_killed.Kernel*
- *kernel_killed.Session*
- *kernel_ready.Kernel*
- *kernel_reconnecting.Kernel*
- *kernel_restarting.Kernel*
- *kernel_starting.Kernel*
- *send_input_reply.Kernel*
- *shell_reply.Kernel*
- *spec_changed.Kernel*

kernel_created.Kernel

The kernel has been successfully created or re-created through `/api/kernels`, but a connection to it has not necessarily been established yet.

kernel_created.Session

The kernel has been successfully created or re-created through `/api/sessions`, but a connection to it has not necessarily been established yet.

kernel_reconnecting.Kernel

An attempt is being made to reconnect (via websockets) to the kernel after having been disconnected.

kernel_connected.Kernel

A connection has been established to the kernel. This is triggered as soon as all websockets (e.g. to the shell, iopub, and stdin channels) have been opened. This does not necessarily mean that the kernel is ready to do anything yet, though.

kernel_starting.Kernel

The kernel is starting. This is triggered once when the kernel process is starting up, and can be sent as a message by the kernel, or may be triggered by the frontend if it knows the kernel is starting (e.g., it created the kernel and is connected to it, but hasn't been able to communicate with it yet).

kernel_ready.Kernel

Like kernel_idle.Kernel, but triggered after the kernel has fully started up.

kernel_restarting.Kernel

The kernel is restarting. This is triggered at the beginning of an restart call to `/api/kernels`.

kernel_autorestarting.Kernel

The kernel is restarting on its own, which probably also means that something happened to cause the kernel to die. For example, running the following code in the notebook would cause the kernel to autorestart:

```
import os
os._exit(1)
```

kernel_interrupting.Kernel

The kernel is being interrupted. This is triggered at the beginning of a interrupt call to `/api/kernels`.

kernel_disconnected.Kernel

The connection to the kernel has been lost.

kernel_connection_failed.Kernel

Not only was the connection lost, but it was lost due to an error (i.e., we did not tell the websockets to close).

kernel_idle.Kernel

The kernel's execution state is 'idle'.

kernel_busy.Kernel

The kernel's execution state is 'busy'.

kernel_killed.Kernel

The kernel has been manually killed through `/api/kernels`.

kernel_killed.Session

The kernel has been manually killed through `/api/sessions`.

kernel_dead.Kernel

This is triggered if the kernel dies, and the kernel manager attempts to restart it, but is unable to. For example, the following code run in the notebook will cause the kernel to die and for the kernel manager to be unable to restart it:

```
import os
from IPython.kernel.connect import get_connection_file
with open(get_connection_file(), 'w') as f:
    f.write("garbage")
os._exit(1)
```

kernel_dead.Session

The kernel could not be started through `/api/sessions`. This might be because the requested kernel type isn't installed. Another reason for this message is that the kernel died or was killed, but the session wasn't.

Notebook-related events

- *app_initialized.NotebookApp*
- *autosave_disabled.Notebook*
- *autosave_enabled.Notebook*
- *checkpoint_created.Notebook*
- *checkpoint_delete_failed.Notebook*
- *checkpoint_deleted.Notebook*
- *checkpoint_failed.Notebook*
- *checkpoint_restore_failed.Notebook*
- *checkpoint_restored.Notebook*

- *checkpoints_listed.Notebook*
- *command_mode.Notebook*
- *edit_mode.Notebook*
- *list_checkpoints_failed.Notebook*
- *notebook_load_failed.Notebook*
- *notebook_loaded.Notebook*
- *notebook_loading.Notebook*
- *notebook_rename_failed.Notebook*
- *notebook_renamed.Notebook*
- *notebook_restoring.Notebook*
- *notebook_save_failed.Notebook*
- *notebook_saved.Notebook*
- *notebook_saving.Notebook*
- *rename_notebook.Notebook*
- *selected_cell_type_changed.Notebook*
- *set_dirty.Notebook*
- *set_next_input.Notebook*
- *trust_changed.Notebook*

Other

- *open_with_text.Pager*
- *rebuild.QuickHelp*

Setup IPython development environment using boot2docker

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

The following are instructions on how to get an IPython development environment up and running without having to install anything on your host machine, other than `` `boot2docker <https://github.com/boot2docker/boot2docker>`__ `` and `` `docker <https://www.docker.com/>`__ ``.

Install `boot2docker`

Install boot2docker. There are multiple ways to install, depending on your environment. See the boot2docker docs.

Mac OS X

On a Mac OS X host with Homebrew installed:

```
$ brew install boot2docker docker
```

Initialize `boot2docker` VM

```
$ boot2docker init
```

Start VM

```
$ boot2docker up
```

The `boot2docker` CLI communicates with the docker daemon on the `boot2docker` VM. To do this, we must set some environment variables, e.g. `DOCKER_HOST`,

```
$ $(boot2docker shellinit)
```

To view the IP address of the VM:

```
$ boot2docker ip
192.168.59.103
```

Install `ipython` from Development Branch

```
$ git clone --recursive https://github.com/ipython/ipython.git
```

Build Docker Image

Use the Dockerfile in the cloned `ipython` directory to build a Docker image.

```
$ cd ipython
$ docker build --rm -t ipython .
```

Run Docker Container

Run a container using the new image. We mount the entire `ipython` source tree on the host into the container at `/srv/ipython` to enable changes we make to the source on the host immediately reflected in the container.

```
# change to the root of the git clone
$ cd ipython
$ docker run -it --rm -p 8888:8888 --workdir /srv/ipython --name ipython-dev -v␣
↪`pwd`:/srv/ipython ipython /bin/bash
```

To list the running container from another shell on the host:

```
$ $(boot2docker shellinit)
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
↪STATUS              PORTS                    NAMES
f6065f206519        ipython             "/bin/bash"         1 minutes ago       Up 1␣
↪minutes        0.0.0.0:8888->8888/tcp   ipython-dev
```

Install IPython in Editable Mode

Once in the container, you'll need to uninstall the `ipython` package and re-install in editable mode to enable your dev changes to be reflected in your environment.

```
container $ pip uninstall ipython

# pip install ipython in editable mode
container $ cd /srv
container $ ls
ipython
container $ pip install -e ipython
```

Run Notebook Server

```
container $ ipython notebook --no-browser --ip=*
```

Visit Notebook Server

On your host, run the following command to get the IP of the boot2docker VM if you forgot:

```
# on host
$ boot2docker ip
192.168.59.103
```

Then visit it in your browser:

```
# browser
http://192.168.59.103:8888
```

As a shortcut on a Mac, you can run the following in a terminal window (or make it a bash alias):

```
$ open http://$(boot2docker ip 2>/dev/null):8888
```

Testing Kernels

Attention: This is copied verbatim from the old IPython wiki and is currently under development. Much of the information in this part of the development guide is out of date.

IPython makes it very easy to create wrapper kernels using its kernel framework. It requires extending the Kernel class and implementing a set of methods for the core functions like execute, history etc. Its also possible to write a full blown kernel in a language of your choice implementing listeners for all the zmq ports.

The key problem for any kernel implemented by these methods is to ensure that it meets the message specification. The kerneltest command is a means to test the installed kernel against the message spec and validate the results.

The kerneltest tool

The kerneltest tool is part of IPython.testing and is also included in the scripts similar to iptest. This takes 2 parameters - the name of the kernel to test and the test script file. The test script file should be in json format as described in the next section.

```
kerneltest python test_script.json
```

You can also pass in an optional message spec version to the command. At the moment only the version 5 is supported, but as newer versions are released this can be used to test the kernel against a specific version of the kernel.

```
kerneltest python test_script.json 5
```

The kernel to be tested needs to be installed and the kernelspec available in the user IPython directory. The tool will instantiate the kernel and send the commands over ZMQ. For each command executed on the kernel, the tool will validate the reply to ensure that it matches the message specification. In some cases the output is also checked, but the reply is always returned and printed out on the console. This can be used to validate that apart from meeting the message spec the kernel also produced the correct output.

The test script file

The test script file is a simple json file that specifies the command to execute and the test code to execute for the command.

```
{
    "command":{
            "test_code":<code>
    }
}
```

For some commands in the message specification like kernel_info there is no need to specify the test_code parameter. The tool validates if it has all the inputs needed to execute the command and will print out an error to the console if it finds a missing parameter. Since the validation is built in, and only required parameters are passed, it is possible to add additional fields in the json file for test documentation.

```
{
    "command":{
            "test_name":"sample test",
            "test_description":"sample test to show how the test script file is created
↪",
            "test_code":<code>
    }
}
```

A sample test script for the redis kernel will look like this

```
{
  "execute":{
    "test_code":"get a",
    "comments":"test basic code execution"
  },
  "complete":{
    "test_code":"get",
    "comments":"test getting command auto complete"
  },
  "kernel_info":{
    "comments":"simple kernel info check"
  },
  "single_payload":{
    "test_code":"get a",
    "comments":"test one payload"
  },
  "history_tail":{
    "test_code":"get a",
    "comments":"test tail history"
  },
  "history_range":{
    "test_code":"get a",
    "comments":"test range history"
  },
  "history_search":{
    "test_code":"get a",
    "comments":"test search history"
  }
}
```

A template for new Python files in IPython: template.py

Whether you are a new contributor or a seasoned developer, we're pleased that you are working on Jupyter. We hope you find the Developer Guide is useful. Please suggest changes or ask questions about the contents. Thanks!

If you are interested in installing a specific project from source, each project has documentation on ReadTheDocs. For example, IPython documentation can be found on ReadTheDocs. Most of our packages can be installed from the source directory like any other Python package, by running:

```
pip install .
```

The Jupyter notebook needs some extra pieces to build Javascript components; the information about that is in the notebook contributor documentation.

Documentation Guide

Contents

Getting started

Contents

Preparing for your first contribution

1. Our documentation uses reStructured Text as well as Jupyter notebooks.
2. We use Sphinx extensively to build documentation.
3. We host our documentation on Read the Docs.

Developing your contribution

Jupyter's documentation is split across several projects, listed on the Jupyter documentation home page. These instructions apply to all Jupyter projects, though some projects have further contribution guidelines.

Clone the repository

1. Fork the appropriate project repository on GitHub, depending on which project's documentation you want to contribute to.
2. Clone the repository to your system.

Edit the documentation source file

Source files for projects are typically found in the project's `docs/source` directory. The reStructured text filenames end with `.rst`, and Jupyter notebook files end with `.ipynb`.

1. In your favorite text editor, make desired changes to the `.rst` file when working with a reStructured text source file.
2. If a notebook file requires editing, you will need to install Jupyter notebook according to the *Installation* document. Then, run the Jupyter notebook and edit the desired file. Before saving the Jupyter `.ipynb` file, please clear the output cells. Save the file and test your change.

Testing changes

Sphinx should be installed to test your documentation changes. For best results, we recommend that you install the stable development version Sphinx (`pip install git+https://github.com/sphinx-doc/sphinx@stable`) or the current released version of Sphinx (`pip install sphinx`).

In addition, you may need the following packages: sphinxcontrib-spelling, sphinx_rtd_theme, nbsphinx, pyenchant, recommonmark and jupyter_sphinx_theme, which can be installed via `pip install sphinxcontrib-spelling sphinx_rtd_theme nbsphinx pyenchant recommonmark jupyter_sphinx_theme`.

If you are on Linux, you may also need to install the Enchant C library by running `sudo apt-get install enchant`.

Once everything is installed, the following commands should be executed using the Terminal/command line from the `docs` directory:

- `make html` builds a local html version of the documentation. The output message will either display errors or provide the location of the html documents. For example, the location provided may be `build/html` and to view these documents in your browser enter `open build/html/index.html`.
- `make linkcheck` will check whether the external links in the documentation are valid or if they are not longer current (i.e. cause a 500 not found error).

Note: We recommend using Python 3.4+ for building the documentation. If you are editing the documentation, you can use Python 2.7.9+ or the Github editor.

Creating a pull request

Once you are satisfied with your changes, submit a GitHub pull request, per the instructions above. If the documentation change is related to an open GitHub issue, please mention the issue number in the pull request message.

A project reviewer will look over your changes and provide feedback or merge your changes into the documentation.

Asking questions

Feel free to ask questions in the Google Group for Jupyter or on an open issue on GitHub.

Understanding our workflow

High level documentation workflow

1. Identify a documentation change.
    - *Typos:* please go ahead and fix it (or report as a bug).
    - *Open issues:* leave a note in the issue comments that you are working on the issue.
    - *New documentation:* open an issue with your idea or suggestion. We'll review the issue and work with you to identify next steps.
2. Update the source file.
3. Commit the change.
4. Test changes locally.
5. Open a pull request.
6. Check response of automated tests.
    - If tests pass: Nice job. Wait for reviewer feedback and/ or your pull request to be merged.
    - If tests show an error: Revise and resubmit your pull request. You do not need to open a new pull request. If needed, please ask for assistance.
7. Celebrate your documentation contribution.
8. Repeat. If you would like suggestions for a new documentation issue to work on, please ask.

Thanks for contributing!

Tools for documentation

Contents

Packages

For user documentation, contributor guides, and communications content, we use:

- Sphinx

For developer API documentation (especially for JupyterLab js repos), we use:

- swagger

Source file formats

We use the following input source file formats when developing Sphinx documentation:

- reStructuredText (`.rst`)
- Markdown (`.md`)
- Notebook (`.ipynb`)

A modern code editor should be used. Many are available including Atom, SublimeText, gedit, vim, emacs. Atom is a good choice for new contributors.

Sphinx themes

Our projects use the following themes:

- sphinx_rtd_theme (currently used by Jupyter projects)
- jupyter_sphinx_theme (used by ipywidgets)

Git and Github Resources

If this is your first time working with Github or git, you can leverage the following resources to learn about the tools.

- Try Git
- Github Guides
- Git Real
- Git Documentation
- Git Rebase

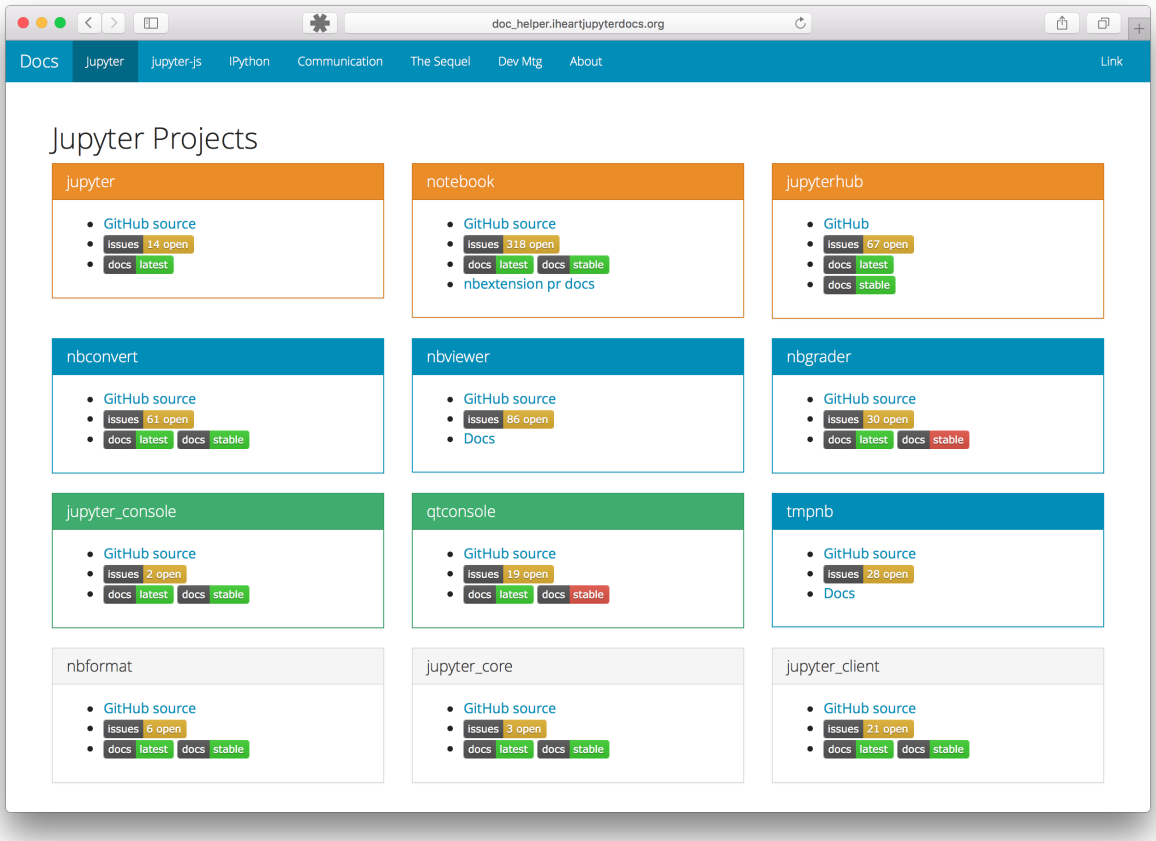
Style guidelines

Correcting bugs

Building the docs

Tracking and metrics

We have created a site to monitor the build status of our documentation. See http://doc_helper.iheartjupyterdocs.org.



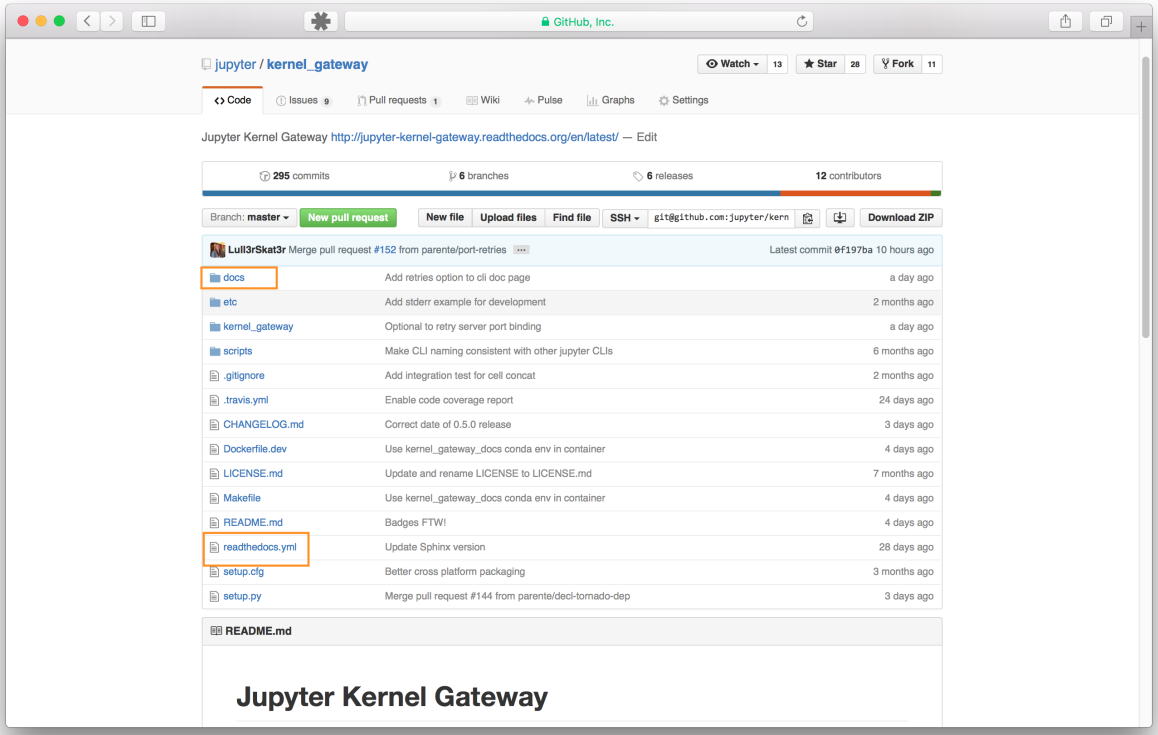
Setting up a project's documentation infrastructure

Contents:

Structuring a repo for docs

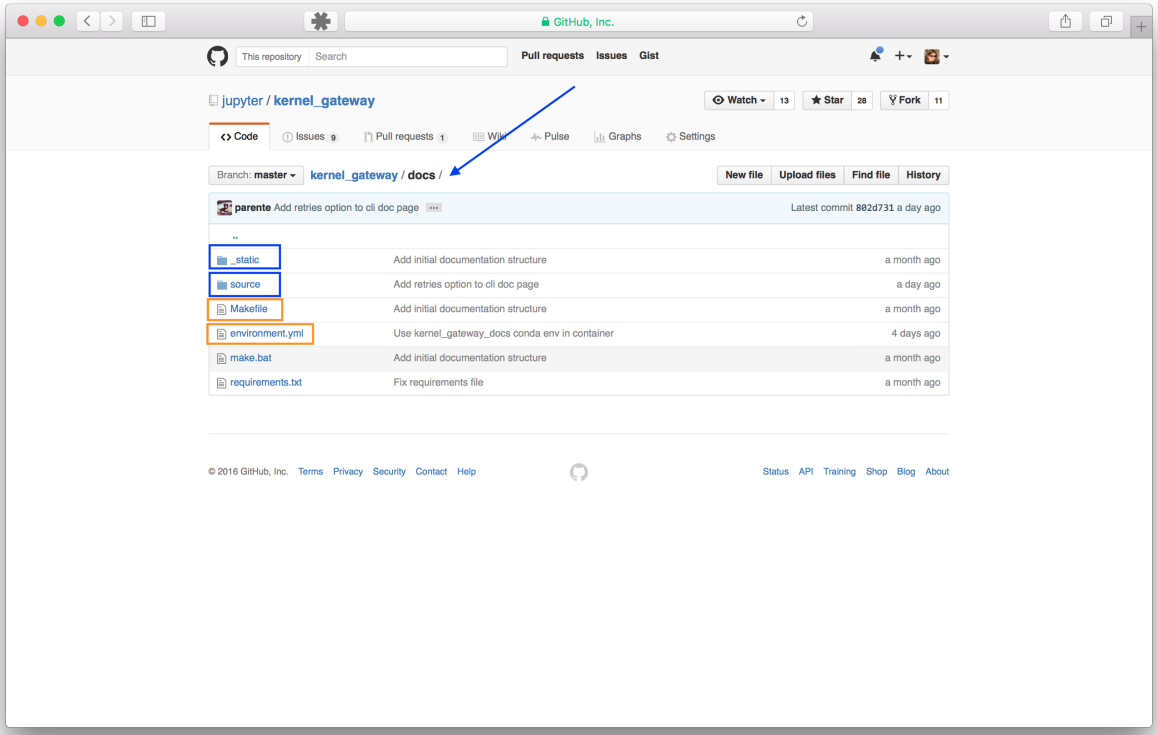
Root level of the repo

- `docs` directory : All source files for documentation go here.
- `readthedocs.yml` : configuration file for readthedocs to build using conda



Inside the docs directory

- `source` directory : contains all content source files in `.rst`, `.md`, or `.ipynb`
- `makefile` : used by Sphinx to build the docs
- `environment.yml` : conda build instructions



Sphinx

- `conf.py` : Sphinx configuration file
- `index.rst` of `contents.rst` : Sphinx master table of contents file
- `_static` directory : contains images, drawings, icons
- `_templates` directory: overrides theme templates and layouts
- `build` directory : html files generated by Sphinx (do not check this directory into GitHub)

Setting up a README

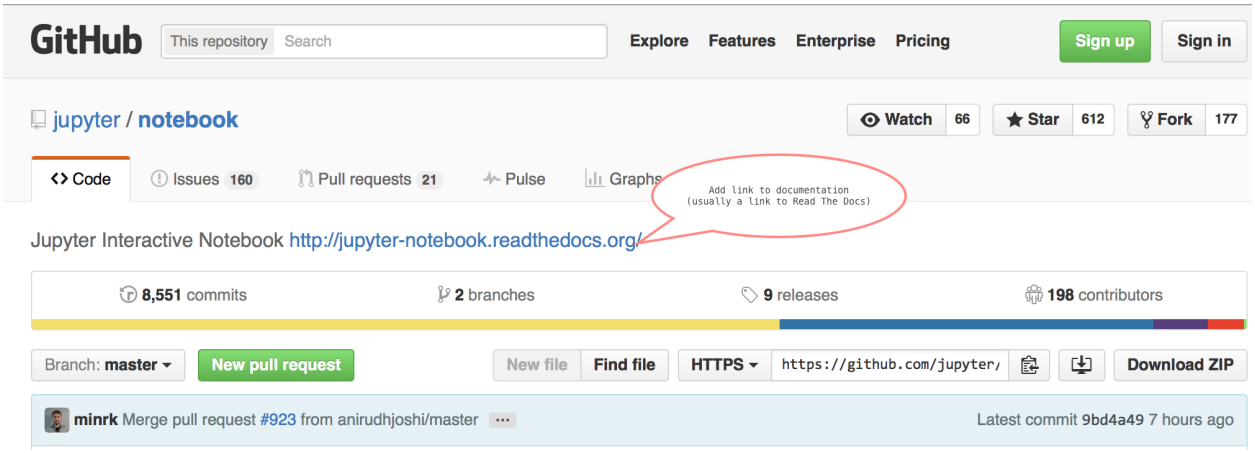
Providing users and developers consistency across repos is a valuable time saver and improves user productivity.

On a larger scope, having the Jupyter name appear prominently in a repo's `README.md` file improves the project's name awareness.

Recommended elements in Jupyter project repos

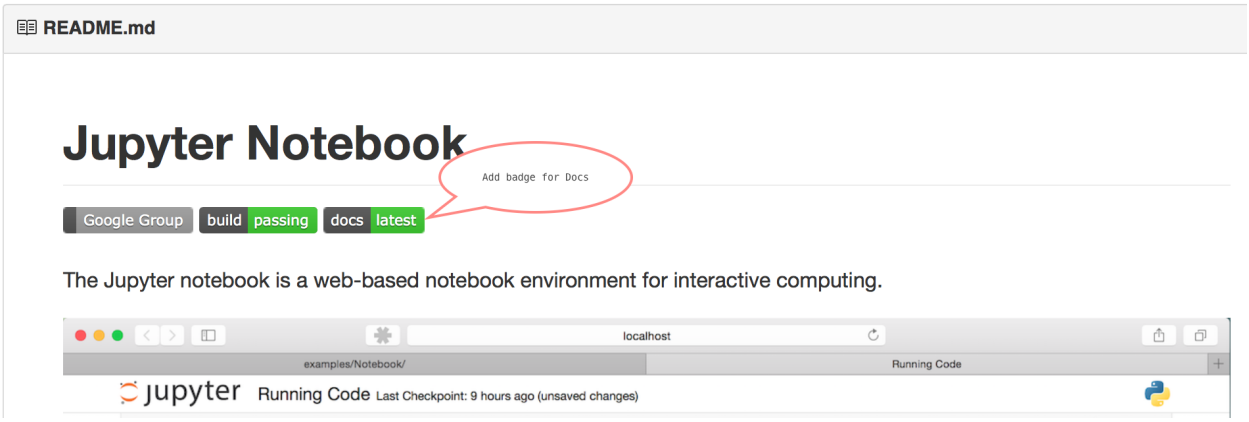
Link in repo description

Please include a link to the documentation in the repo's description.



Badges in README

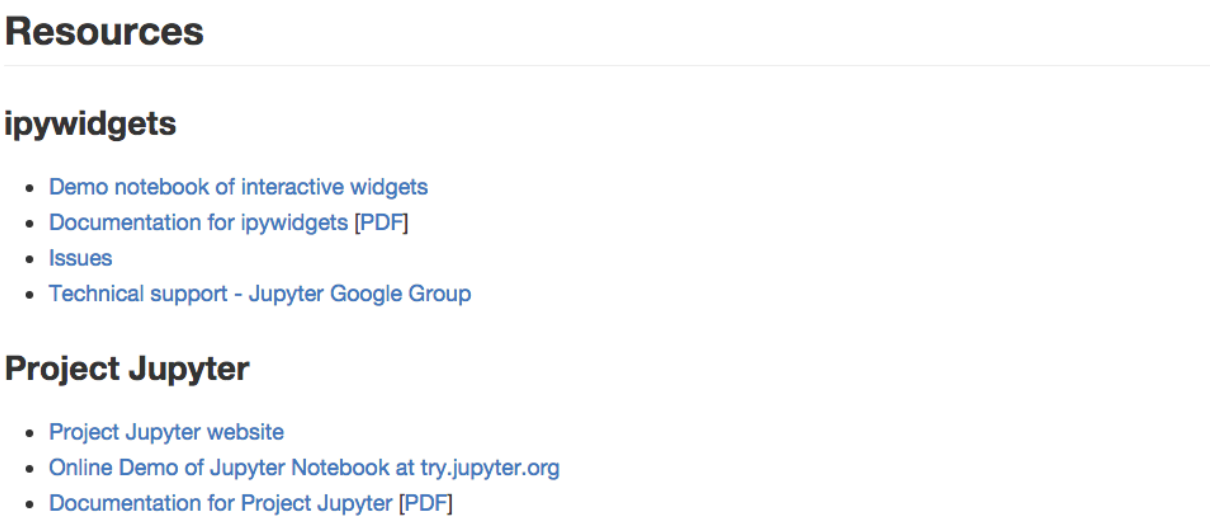
One common way that individuals find documentation is to look for and click on the doc badge that commonly is found right after the title. Another benefit is an easy visual indication if the docs are not rendering properly.



Resources section in README

A *Resources* section at the end of the `README.md` gives useful links and information to users about the individual project and the larger Project Jupyter organization. Make sure to include any links to the individual project's demo notebooks, if available.

The *Resources* section includes:



Resources

ipywidgets

- Demo notebook of interactive widgets
- Documentation for ipywidgets [PDF]
- Issues
- Technical support - Jupyter Google Group

Project Jupyter

- Project Jupyter website
- Online Demo of Jupyter Notebook at try.jupyter.org
- Documentation for Project Jupyter [PDF]

Checklist adding docs to a new or existing GitHub Repo

- [ ] Add a link to documentation in repo description (requires GitHub repo privileges)
- [ ] Add badges to README (Edit `README.md` and submit pull request)
- [ ] Add resources section to README (Edit `README.md` and submit pull request)

Dated: 1-4-2016 Revised: 1-7-2016

Building automatically on ReadTheDocs

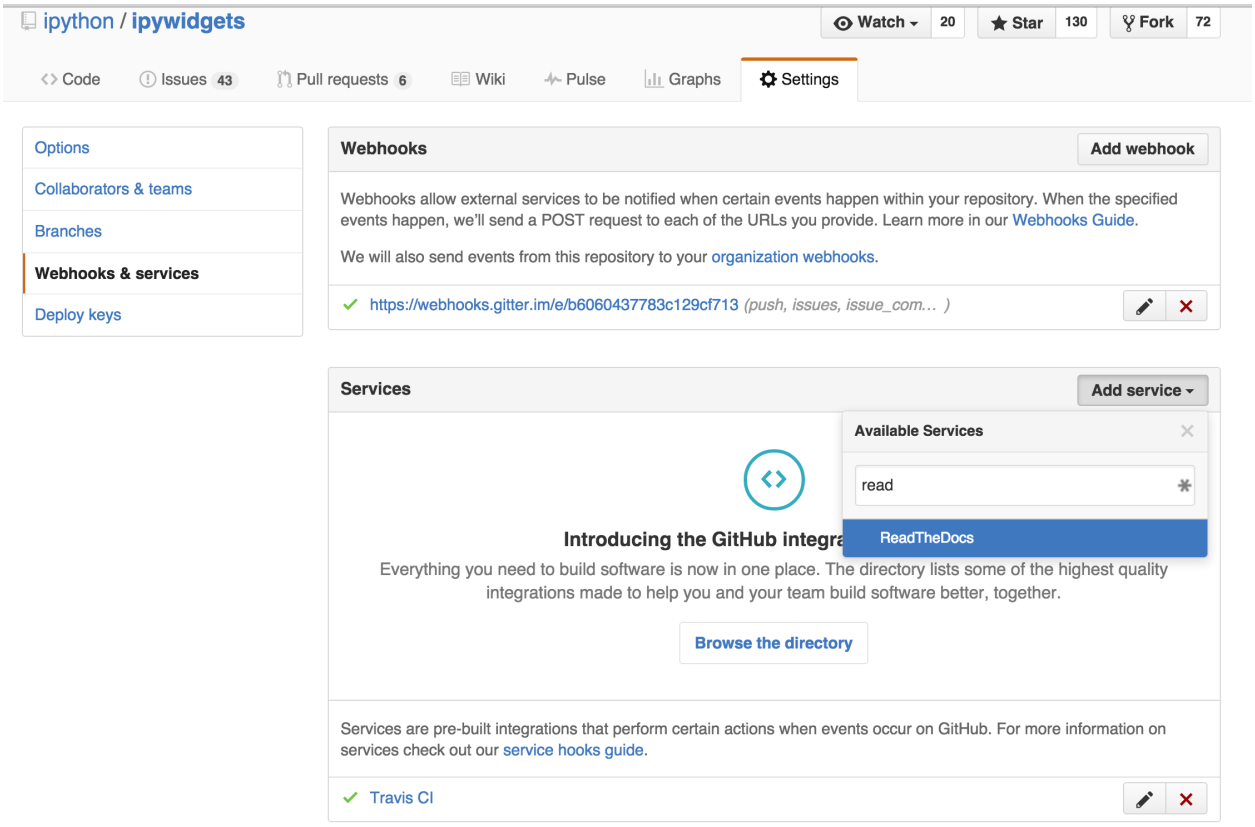
This explains how to automatically rebuild documentation on ReadtheDocs every time a pull request is merged into its corresponding GitHub repo.

Using the ReadTheDocs service

Webhooks and services can be enabled in GitHub repo settings to allow third party services such as ReadTheDocs. The ReadTheDocs service rebuilds the project documentation whenever a pull request is merged into the GitHub repo.

Navigate to Settings

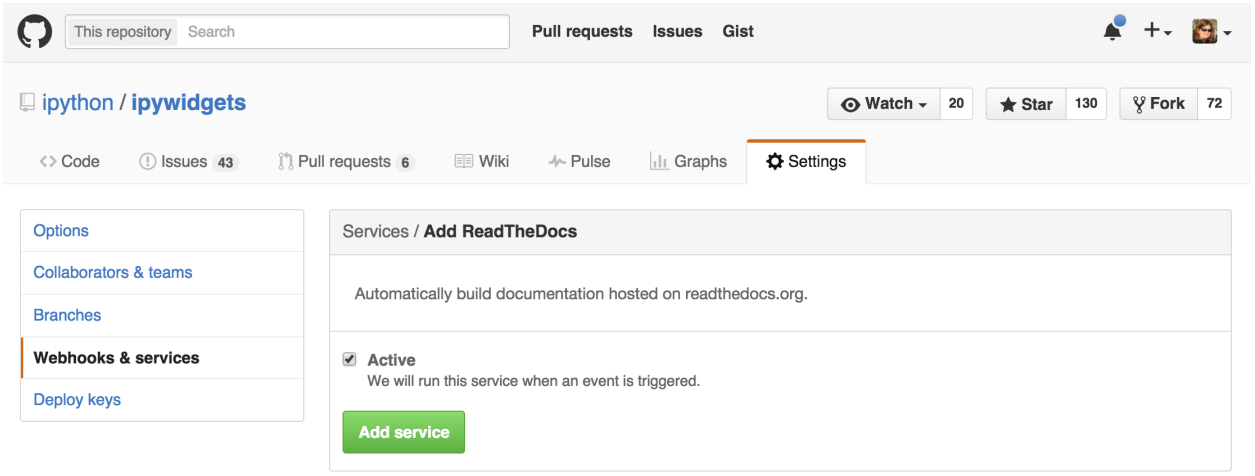
Each GitHub repo has a Settings tab at the far right of the repo menubar. Navigate to Settings and then the **Webhooks & services** submenu tab.



Add the ReadTheDocs service

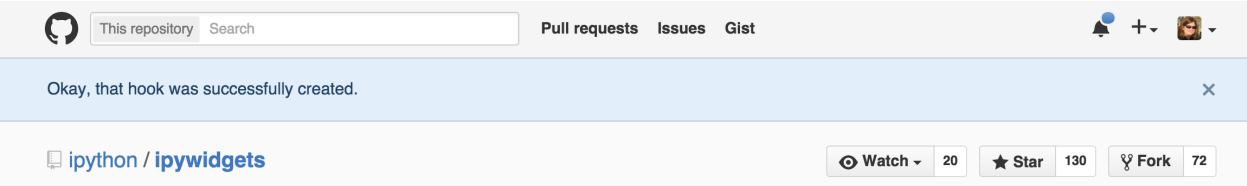
Select **Add service** and enter *ReadTheDocs* in the **Available Services** input box.

The Services/Add ReadTheDocs window will open. Press the green **Add service** button to activate the ReadTheDocs service.



Success

The ReadTheDocs service is added successfully. The service will take effect on the next merged pull request to the project repo.



Created: 01-07-2016

This section helps a contributor set up the documentation infrastructure for a new project or an existing project without Sphinx documentation.

Documentation helps guide new users, fosters communication between developers, and shares tips and best practices with other community members. That's why the Jupyter project is focused on documenting new features and to keeping the documentation up-to-date.

Communications Guide

Contents

Blog

We publish our blog at https://blog.jupyter.org. We welcome ideas for posts or guest posts to the Jupyter blog. If you have a suggestion for a future post, please feel free to share your idea with us. We would like to discuss the idea with you.

Do you enjoy writing? Please contact us about becoming a guest blogger. We can help guide you through the process of creating a post.

Technical overview

Jupyter's blog uses the Ghost blog platform for its contributor flexibility and ease of use. Jupyter's blog is deployed at https://blog.jupyter.org.

Basic workflow from blog idea to published post

There are several major steps in the workflow from blog idea to a published post including:

- Be inspired to write a post
- Send us a message on the Jupyter mailing list and ask us for an author account on our blog
- Creating a draft
- Draft Review
- Editorial acceptance
- Publishing the post

We'll cover each of these as well as how to update a post once it has been published.

Creating a draft

Title and metadata

Alway check in the metadata fields that a blog post has a title and a canonical URL. It is possible to put the date in the canonical URL, in particular for events like jupyter-day, that can occur several times. The date of the event can differ from the date of the blog post.

Once a post is published, **never** change the post's title or the url. These changes will break links of tweets and RSS feeds that have already referenced the existing, published URL. Keep in mind that when publishing some platforms cache the url immediately; as a result changing the title will direct people to a 404 page.

Title and metadata can always be refined after the actual content of the blog is written, but should not be changed after publication. As a guest you do not have to worry about metadata, the editor or admins will take care of that.

Working with images

Try not to link to external images. If you want to put an image in the post, insert `![]()` in the editor view and drag and drop an image from your desktop into the newly created field in in the preview. External images can change, and can break the blog post if they are taken down. This cannot append if you drag and drop images. Moreover, these images will be served from the same CDN (Content Delivery Network) as the blog, which will insure the best overall experience for our readers.

The featured image you see at the top of a blog posts is set from within the metadata field, not using the **. The featured image is treated differently than inlined images by many feedreaders (especially on mobile) and allows a user on a slow connection to read the content of the blog earlier, which is a much better experience for the user than waiting for the featured image to render.

Links

Do not use minified links when possible. The multiple redirects of minified links degrades the mobile browsing experience. If you need analytics of the number of page views, this information is tracked by Google Analytics.

Draft review

Ask for a review

Once you think you are done, ask someone else to reread your post, and check the various parameters that you might have forgotten before publishing. You are not on your own, this is teamwork, we are here to help you. If we do things in a hurry you will probably spend more time fixing mistakes that actually doing things right in a first place.

Editorial acceptance

Publishing the post

Usually an editor or admin will take care of publishing the post. The task of the Editor/Admin is to check all metadata are correctly set, that no external images are used, as well as all other quality check describe before.

It is then just a matter of making th post visible to everyone.

Changing an existing post

Posts Updates

Blog subscribers may receive notification at every update. So use updates and fixes parsimoniously. It is OK to wait a few hours to fix a typo.

If some substantial updates have to be made, like change of location, time etc, please insert an *[Update]* section at top (or bottom of the blog post depending on importance) with the Date/Time of the update. If the information in the body of the blog is wrong, try not to replace it, and just use strike-through to mark it as obsolete. This would help reader determine which information is correct when dealing with multiple source giving different informations.

Newsletter

Documentation in progress.

Website

Documentation in progress.

Whether you are a new, returning, or current contributor to Project Jupyter's subprojects or IPython, **we welcome you**.

Project Jupyter has seen steady growth over the past several years, and it is wonderful to see the many ways people are using these projects. As a result of this rapid expansion, our project maintainers are balancing many requirements, needs, and resources. We ask contributors to take some time to become familiar with our contribution guides and spend some time learning about our project communication and workflow.

The Contributor Guides and individual project documentation offer guidance. If you have a question, please ask us. Community Resources provides information on our commonly used communication methods.

We are very pleased to have you as a contributor, and we hope you will find valuable your impact on the projects. **Thank you** for sharing your interests, ideas, and skills with us.

CHAPTER 8

Release Notes

Each project's documentation and GitHub repository has information about releases and changes from the prior release.

Note: Coming Soon

We're actively working on a graphic that displays each project, their current release, and a link to the changelog. Thanks for your patience.

CHAPTER 9

Reference

Custom mimetypes (MIME types)

What's a mimetype?

When an internet request and response occurs, a `Content-Type` header is passed. A mimetype, also referred to as MIME type, identifies how the content that is being returned should be handled or used, based on type, by the application and browser. A MIME type is made up of a MIME *group* (i.e. application, image, audio, etc.) and a MIME *subtype*. For example, a MIME type is `image/png` where MIME *group* is `image` and *subtype* is `png`.

As types may contain vendor specific items, a custom vendor specific MIME type, `vnd`, can be used. A vendor specific MIME type will contain `vnd` such as `application/vnd.jupyter.cells`.

Custom mimetypes used in Jupyter and IPython projects

- `application/vnd.jupyter`
- `application/vnd.jupyter.cells`
- `application/vnd.jupyter.dragindex` used by nbdime
- `application/x-ipynb+json` for notebooks

Listing of custom mimetypes used for display

- `application/vnd.geo+json` - GeoJSON spec `application/vnd.geo+json` is now deprecated and replaced by `application/geo+json`
- `application/geo+json` - preferred GeoJSON spec
- `application/vnd.plotly.v1+json` - Plotly JSON Schema

Glossary

command line The terminal or console window where you type commands.

Command Prompt On Windows, this is the application where commands are typed into a window for execution.

conda The package manager for Anaconda.

config Refers to the configuration files and process.

kernel A kernel provides programming language support in Jupyter. IPython is the default kernel. Additional kernels include R, Julia, and many more.

Notebook Dashboard The notebook user interface which shows a list of the notebooks, files, and subdirectories in the directory where the notebook server is started.

pip Python package manager.

profiles Not available in Jupyter. In IPython 3, profiles are collections of configuration and runtime files.

REPL read-eval-print-loop.

terminal A window used to type in commands to be executed (Linux and OS X).

widget A user interface component, similar to a plugin, that allows customized input, such as a slider.

Resources

- genindex
- search

CHAPTER 10

Indices and tables

- genindex
- *Glossary*
- search

Resources:

Site	Description
Jupyter website	Keep up to date on Jupyter
IPython website	Learn more about IPython
jupyter/help repo	Start here for help and support questions
Jupyter mailing list	General discussion of Jupyter's use
Jupyter in Education group	Discussion of Jupyter's use in education
NumFocus	Promotes world-class, innovative, open source scientific software
Donate to Project Jupyter	Please contribute to open science collaboration and sustainability

Index

Symbols

C

D

E

I

J

K

N

P

R

T

W