1.1 Introduction

The notebook extends the console-based approach to interactive computing in a qualitatively new direction, providing a web-based application suitable for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results. The Jupyter notebook combines two components:

A web application: a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output.

Notebook documents: a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

See also:

See the installation guide on how to install the notebook and its dependencies.

1.1.1 Main features of the web application

- In-browser editing for code, with automatic syntax highlighting, indentation, and tab completion/introspection.
- The ability to execute code from the browser, with the results of computations attached to the code which generated them.
- Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc. For example, publication-quality figures rendered by the matplotlib library, can be included inline.
- In-browser editing for rich text using the Markdown markup language, which can provide commentary for the code, is not limited to plain text.
- The ability to easily include mathematical notation within markdown cells using LaTeX, and rendered natively by MathJax.
1.1.2 Notebook documents

Notebook documents contain the inputs and outputs of an interactive session as well as additional text that accompanies the code but is not meant for execution. In this way, notebook files can serve as a complete computational record of a session, interleaving executable code with explanatory text, mathematics, and rich representations of resulting objects. These documents are internally JSON files and are saved with the .ipynb extension. Since JSON is a plain text format, they can be version-controlled and shared with colleagues.

Notebooks may be exported to a range of static formats, including HTML (for example, for blog posts), reStructuredText, LaTeX, PDF, and slide shows, via the nbconvert command.

Furthermore, any .ipynb notebook document available from a public URL can be shared via the Jupyter Notebook Viewer (nbviewer). This service loads the notebook document from the URL and renders it as a static web page. The results may thus be shared with a colleague, or as a public blog post, without other users needing to install the Jupyter notebook themselves. In effect, nbviewer is simply nbconvert as a web service, so you can do your own static conversions with nbconvert, without relying on nbviewer.

See also:

Details on the notebook JSON file format

1.2 Starting the notebook server

You can start running a notebook server from the command line using the following command:

```
jupyter notebook
```

This will print some information about the notebook server in your console, and open a web browser to the URL of the web application (by default, http://127.0.0.1:8888).

The landing page of the Jupyter notebook web application, the dashboard, shows the notebooks currently available in the notebook directory (by default, the directory from which the notebook server was started).

You can create new notebooks from the dashboard with the New Notebook button, or open existing ones by clicking on their name. You can also drag and drop .ipynb notebooks and standard .py Python source code files into the notebook list area.

When starting a notebook server from the command line, you can also open a particular notebook directly, bypassing the dashboard, with jupyter notebook my_notebook.ipynb. The .ipynb extension is assumed if no extension is given.

When you are inside an open notebook, the File → Open... menu option will open the dashboard in a new browser tab, to allow you to open another notebook from the notebook directory or to create a new notebook.

Note: You can start more than one notebook server at the same time, if you want to work on notebooks in different directories. By default the first notebook server starts on port 8888, and later notebook servers search for ports near that one. You can also manually specify the port with the --port option.

1.2.1 Creating a new notebook document

A new notebook may be created at any time, either from the dashboard, or using the File → New menu option from within an active notebook. The new notebook is created within the same directory and will open in a new browser tab. It will also be reflected as a new entry in the notebook list on the dashboard.
1.2.2 Opening notebooks

An open notebook has **exactly one** interactive session connected to a kernel, which will execute code sent by the user and communicate back results. This kernel remains active if the web browser window is closed, and reopening the same notebook from the dashboard will reconnect the web application to the same kernel. In the dashboard, notebooks with an active kernel have a **Shutdown** button next to them, whereas notebooks without an active kernel have a **Delete** button in its place.

Other clients may connect to the same kernel. When each kernel is started, the notebook server prints to the terminal a message like this:

```
[NotebookApp] Kernel started: 87f7d2c0-13e3-43df-8bb8-1bd37aaf3373
```

This long string is the kernel’s ID which is sufficient for getting the information necessary to connect to the kernel. If the notebook uses the IPython kernel, you can also see this connection data by running the `%connect_info` magic, which will print the same ID information along with other details.

You can then, for example, manually start a Qt console connected to the *same* kernel from the command line, by passing a portion of the ID:

```
$ jupyter qtconsole --existing 87f7d2c0
```

Without an ID, `--existing` will connect to the most recently started kernel.

With the IPython kernel, you can also run the `%qtconsole` magic in the notebook to open a Qt console connected to the same kernel.

**See also:**

- Decoupled two-process model

1.3 Notebook user interface

When you create a new notebook document, you will be presented with the notebook name, a menu bar, a toolbar and an empty code cell.

**Notebook name:** The name of the notebook document is displayed at the top of the page, next to the IP[y]: Notebook logo. This name reflects the name of the .ipynb notebook document file. Clicking on the notebook name brings up a dialog which allows you to rename it. Thus, renaming a notebook from “Untitled0” to “My first notebook” in the browser, renames the Untitled0.ipynb file to My first notebook.ipynb.

**Menu bar:** The menu bar presents different options that may be used to manipulate the way the notebook functions.

**Toolbar:** The tool bar gives a quick way of performing the most-used operations within the notebook, by clicking on an icon.

**Code cell:** The default type of cell, read on for an explanation of cells

---

**Note:** As of notebook version 4.1, the user interface allows for multiple cells to be selected. The quick celltype selector, found in the menubar, will display a dash – when multiple cells are selected to indicate that the type of the cells in the selection might not be unique. The quick selector can still be used to change the type of the selection and will change the type of all the currently selected cells.
1.4 Structure of a notebook document

The notebook consists of a sequence of cells. A cell is a multiline text input field, and its contents can be executed by using Shift-Enter, or by clicking either the “Play” button the toolbar, or Cell I Run in the menu bar. The execution behavior of a cell is determined by the cell’s type. There are three types of cells: code cells, markdown cells, and raw cells. Every cell starts off being a code cell, but its type can be changed by using a drop-down on the toolbar (which will be “Code”, initially), or via keyboard shortcuts.

For more information on the different things you can do in a notebook, see the collection of examples.

1.4.1 Code cells

A code cell allows you to edit and write new code, with full syntax highlighting and tab completion. By default, the language associated to a code cell is Python, but other languages, such as Julia and R, can be handled using cell magic commands.

When a code cell is executed, code that it contains is sent to the kernel associated with the notebook. The results that are returned from this computation are then displayed in the notebook as the cell’s output. The output is not limited to text, with many other possible forms of output are also possible, including matplotlib figures and HTML tables (as used, for example, in the pandas data analysis package). This is known as IPython’s rich display capability.

See also:
Rich Output example notebook

1.4.2 Markdown cells

You can document the computational process in a literate way, alternating descriptive text with code, using rich text. In IPython this is accomplished by marking up text with the Markdown language. The corresponding cells are called Markdown cells. The Markdown language provides a simple way to perform this text markup, that is, to specify which parts of the text should be emphasized (italics), bold, form lists, etc.

If you want to provide structure for your document, you can use markdown headings. Markdown headings consist of 1 to 6 hash # signs # followed by a space and the title of your section. The markdown heading will be converted to a clickable link for a section of the notebook. It is also used as a hint when exporting to other document formats, like PDF.

When a Markdown cell is executed, the Markdown code is converted into the corresponding formatted rich text. Markdown allows arbitrary HTML code for formatting.

Within Markdown cells, you can also include mathematics in a straightforward way, using standard LaTeX notation: $...$ for inline mathematics and $$...$$ for displayed mathematics. When the Markdown cell is executed, the LaTeX portions are automatically rendered in the HTML output as equations with high quality typography. This is made possible by MathJax, which supports a large subset of LaTeX functionality.

Standard mathematics environments defined by LaTeX and AMS-LaTeX (the amsmath package) also work, such as \begin{equation}...\end{equation}, and \begin{align}...\end{align}. New LaTeX macros may be defined using standard methods, such as \newcommand, by placing them anywhere between math delimiters in a Markdown cell. These definitions are then available throughout the rest of the IPython session.

See also:
Working with Markdown Cells example notebook
1.4.3 Raw cells

Raw cells provide a place in which you can write output directly. Raw cells are not evaluated by the notebook. When passed through nbconvert, raw cells arrive in the destination format unmodified. For example, this allows you to type full LaTeX into a raw cell, which will only be rendered by LaTeX after conversion by nbconvert.

1.5 Basic workflow

The normal workflow in a notebook is, then, quite similar to a standard IPython session, with the difference that you can edit cells in-place multiple times until you obtain the desired results, rather than having to rerun separate scripts with the %run magic command.

Typically, you will work on a computational problem in pieces, organizing related ideas into cells and moving forward once previous parts work correctly. This is much more convenient for interactive exploration than breaking up a computation into scripts that must be executed together, as was previously necessary, especially if parts of them take a long time to run.

At certain moments, it may be necessary to interrupt a calculation which is taking too long to complete. This may be done with the Kernel | Interrupt menu option, or the Ctrl-m i keyboard shortcut. Similarly, it may be necessary or desirable to restart the whole computational process, with the Kernel | Restart menu option or Ctrl-m . shortcut.

A notebook may be downloaded in either a .ipynb or .py file from the menu option File | Download as. Choosing the .py option downloads a Python .py script, in which all rich output has been removed and the content of markdown cells have been inserted as comments.

See also:
Running Code in the Jupyter Notebook example notebook
Notebook Basics example notebook

1.5.1 Keyboard shortcuts

All actions in the notebook can be performed with the mouse, but keyboard shortcuts are also available for the most common ones. The essential shortcuts to remember are the following:

- **Shift-Enter: run cell** Execute the current cell, show output (if any), and jump to the next cell below. If Shift-Enter is invoked on the last cell, a new code cell will also be created. Note that in the notebook, typing Enter on its own never forces execution, but rather just inserts a new line in the current cell. Shift-Enter is equivalent to clicking the Cell | Run menu item.

- **Ctrl-Enter: run cell in-place** Execute the current cell as if it were in “terminal mode”, where any output is shown, but the cursor remains in the current cell. The cell’s entire contents are selected after execution, so you can just start typing and only the new input will be in the cell. This is convenient for doing quick experiments in place, or for querying things like filesystem content, without needing to create additional cells that you may not want to be saved in the notebook.

- **Alt-Enter: run cell, insert below** Executes the current cell, shows the output, and inserts a new cell between the current cell and the cell below (if one exists). This is thus a shortcut for the sequence Shift-Enter, Ctrl-m a. (Ctrl-m a adds a new cell above the current one.)

- **Esc and Enter: Command mode and edit mode** In command mode, you can easily navigate around the notebook using keyboard shortcuts. In edit mode, you can edit text in cells.

For the full list of available shortcuts, click Help, Keyboard Shortcuts in the notebook menus.
1.6 Plotting

One major feature of the Jupyter notebook is the ability to display plots that are the output of running code cells. The IPython kernel is designed to work seamlessly with the matplotlib plotting library to provide this functionality. Specific plotting library integration is a feature of the kernel.

1.7 Installing kernels

For information on how to install a Python kernel, refer to the IPython install page. Kernels for other languages can be found in the IPython wiki. They usually come with instruction what to run to make the kernel available in the notebook.

1.8 Signing Notebooks

To prevent untrusted code from executing on users’ behalf when notebooks open, we have added a signature to the notebook, stored in metadata. The notebook server verifies this signature when a notebook is opened. If the signature stored in the notebook metadata does not match, javascript and HTML output will not be displayed on load, and must be regenerated by re-executing the cells.

Any notebook that you have executed yourself in its entirety will be considered trusted, and its HTML and javascript output will be displayed on load.

If you need to see HTML or Javascript output without re-executing, you can explicitly trust notebooks, such as those shared with you, or those that you have written yourself prior to IPython 2.0, at the command-line with:

```bash
$ jupyter trust mynotebook.ipynb [other notebooks.ipynb]
```

This just generates a new signature stored in each notebook.

You can generate a new notebook signing key with:

```bash
$ jupyter trust --reset
```

1.9 Browser Compatibility

The Jupyter Notebook is officially supported by the latest stable versions of the following browsers:

- Chrome
- Safari
- Firefox

The is mainly due to the notebook’s usage of WebSockets and the flexible box model.

The following browsers are unsupported:

- Safari < 5
- Firefox < 6
- Chrome < 13
- Opera (any): CSS issues, but execution might work
- Internet Explorer < 10
- Internet Explorer 10 (same as Opera)

Using Safari with HTTPS and an untrusted certificate is known to not work (websockets will fail).
UI Components

When opening bug reports or sending emails to the Jupyter mailing list, it is useful to know the names of different UI components so that other developers and users have an easier time helping you diagnose your problems. This section will familiarize you with the names of UI elements within the Notebook and the different Notebook modes.

2.1 Notebook Dashboard

When you launch `jupyter notebook` the first page that you encounter is the Notebook Dashboard.
2.2 Notebook Editor

Once you’ve selected a Notebook to edit, the Notebook will open in the Notebook Editor.

2.3 Interactive User Interface Tour of the Notebook

If you would like to learn more about the specific elements within the Notebook Editor, you can go through the User Interface Tour by selecting Help in the menubar then selecting User Interface Tour.

2.3.1 Edit Mode and Notebook Editor

When a cell is in edit mode, the Cell Mode Indicator will change to reflect the cell’s state. This state is indicated by a small pencil icon on the top right of the interface. When the cell is in command mode, there is no icon in that location.
2.4 File Editor

Now let’s say that you’ve chosen to open a Markdown file instead of a Notebook file whilst in the Notebook Dashboard. If so, the file will be opened in the File Editor.
Comms allow custom messages between the frontend and the kernel. They are used, for instance, in ipywidgets to update widget state.

A comm consists of a pair of objects, in the kernel and the frontend, with an automatically assigned unique ID. When one side sends a message, a callback on the other side is triggered with that message data. Either side, the frontend or kernel, can open or close the comm.

See also:

Custom Messages The messaging specification section on comms

3.1 Opening a comm from the kernel

First, the function to accept the comm must be available on the frontend. This can either be specified in a requirejs module, or registered in a registry, for example when an extension is loaded. This example shows a frontend comm target registered in a registry:

```python
Jupyter.notebook.kernel.comm_manager.register_target('my_comm_target',
    function(comm, msg) {
      // comm is the frontend comm instance
      // msg is the comm_open message, which can carry data

      // Register handlers for later messages:
      comm.on_msg(function(msg) {...});
      comm.on_close(function(msg) {...});
      comm.send({'foo': 0});
    });
```

Now that the frontend comm is registered, you can open the comm from the kernel:

```python
from ipykernel.comm import Comm

# Use comm to send a message from the kernel
```
my_comm = Comm(target_name='my_comm_target', data={'foo': 1})
my_comm.send({'foo': 2})

# Add a callback for received messages.
@my_comm.on_msg
def _recv(msg):
    # Use msg['content']['data'] for the data in the message

This example uses the IPython kernel; it’s up to each language kernel what API, if any, it offers for using comms.

### 3.2 Opening a comm from the frontend

This is very similar to above, but in reverse. First, a comm target must be registered in the kernel. For instance, this may be done by code displaying output: it will register a target in the kernel, and then display output containing Javascript to connect to it.

def target_func(comm, msg):
    # comm is the kernel Comm instance
    # msg is the comm_open message

    # Register handler for later messages
    @comm.on_msg
def _recv(msg):
        # Use msg['content']['data'] for the data in the message

        # Send data to the frontend
        comm.send({'foo': 5})

get_ipython().kernel.comm_manager.register_target('my_comm_target', target_func)

This example uses the IPython kernel again; this example will be different in other kernels that support comms. Refer to the specific language kernel’s documentation for comms support.

And then open the comm from the frontend:

comm = Jupyter.notebook.kernel.comm_manager.new_comm('my_comm_target',
    {'foo': 6})

// Send data
comm.send({'foo': 7})

// Register a handler
comm.on_msg(function(msg) {
    console.log(msg.content.data.foo);
});
Beyond the default configuration settings, you can configure a rich array of options to suit your workflow. Here are areas that are commonly configured when using Jupyter Notebook:

- Jupyter’s common configuration system
- Notebook server
- Notebook front-end client
- Notebook extensions

Let’s look at highlights of each area.

### 4.1 Jupyter’s Common Configuration system

Jupyter applications, from the Notebook to JupyterHub to nbgrader, share a common configuration system. The process for creating a configuration file and editing settings is similar for all the Jupyter applications.

- Jupyter’s Common Configuration Approach
- Common Directories and File Locations
- Language kernels
- traitlets provide a low-level architecture for configuration.

### 4.2 Notebook server

The Notebook server runs the language kernel and communicates with the front-end Notebook client (i.e. the familiar notebook interface).

- Configuring the Notebook server
To create a `jupyter_notebook_config.py` file in the `.jupyter` directory, with all the defaults commented out, use the following command:

```bash
$ jupyter notebook --generate-config
```

:ref:`Command line arguments for configuration <config>` settings are documented in the configuration file and the user documentation.

- **Running a Notebook server**
  - Related: Configuring a language kernel to run in the Notebook server enables your server to run other languages, like R or Julia.

### 4.3 Notebook front-end client

- **How front-end configuration works**
  - Example: Changing the notebook’s default indentation setting
  - Example: Restoring the notebook’s default indentation setting

- **Persisting configuration settings**

### 4.4 Notebook extensions

- Distributing Jupyter Extensions as Python Packages
- Extending the Notebook

**Security in Jupyter notebooks:** Since security policies vary from organization to organization, we encourage you to consult with your security team on settings that would be best for your use cases. Our documentation offers some responsible security practices, and we recommend becoming familiar with the practices.
CHAPTER 5

Config file and command line options

The notebook server can be run with a variety of command line arguments. A list of available options can be found below in the options section.

Defaults for these options can also be set by creating a file named `jupyter_notebook_config.py` in your Jupyter folder. The Jupyter folder is in your home directory, `~/.jupyter`.

To create a `jupyter_notebook_config.py` file, with all the defaults commented out, you can use the following command line:

```
$ jupyter notebook --generate-config
```

### 5.1 Options

This list of options can be generated by running the following and hitting enter:

```
$ jupyter notebook --help
```

- **Application.log_datefmt** [Unicode] Default: `%Y-%m-%d %H:%M:%S`
  
  The date format used by logging formatters for `%asctime`

- **Application.log_format** [Unicode] Default: `'[%(name)s]%(levelname)s %(message)s'`
  
  The Logging format template

- **Application.log_level** [0-10-20-30-40-50|'DEBUG'|'INFO'|'WARN'|'ERROR'|'CRITICAL'] Default: 30
  
  Set the log level by value or name.

- **JupyterApp.answer_yes** [Bool] Default: False
  
  Answer yes to any prompts.

- **JupyterApp.config_file** [Unicode] Default: ' '
  
  Full path of a config file.
**JupyterApp.config_file_name** [Unicode] Default: ''
Specify a config file to load.

**JupyterApp.generate_config** [Bool] Default: False
Generate default config file.

**NotebookApp.allow_credentials** [Bool] Default: False
Set the Access-Control-Allow-Credentials: true header

**NotebookApp.allow_origin** [Unicode] Default: ''
Set the Access-Control-Allow-Origin header
Use '*' to allow any origin to access your server.
Takes precedence over allow_origin_pat.

**NotebookApp.allow_origin_pat** [Unicode] Default: ''
Use a regular expression for the Access-Control-Allow-Origin header
Requests from an origin matching the expression will get replies with:

Access-Control-Allow-Origin: origin

where origin is the origin of the request.
Ignored if allow_origin is set.

**NotebookApp.allow_password_change** [Bool] Default: True
Allow password to be changed at login for the notebook server.
While loggin in with a token, the notebook server UI will give the opportunity to the user to enter a new password at the same time that will replace the token login mechanism.
This can be set to false to prevent changing password from the UI/API.

**NotebookApp.allow_root** [Bool] Default: False
Whether to allow the user to run the notebook as root.

**NotebookApp.base_project_url** [Unicode] Default: '/'
DEPRECATED use base_url

**NotebookApp.base_url** [Unicode] Default: '/'
The base URL for the notebook server.
Leading and trailing slashes can be omitted, and will automatically be added.

**NotebookApp.browser** [Unicode] Default: '''
Specify what command to use to invoke a web browser when opening the notebook. If not specified, the default browser will be determined by the webbrowser standard library module, which allows setting of the BROWSER environment variable to override it.

**NotebookApp.certfile** [Unicode] Default: '''
The full path to an SSL/TLS certificate file.

**NotebookApp.client_ca** [Unicode] Default: '''
The full path to a certificate authority certificate for SSL/TLS client authentication.
<table>
<thead>
<tr>
<th><strong>NotebookApp.config_manager_class</strong> [Type]</th>
<th>Default:</th>
<th>'notebook.services.config.manager.ConfigManager'</th>
</tr>
</thead>
<tbody>
<tr>
<td>The config manager class to use</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NotebookApp.contents_manager_class</strong> [Type]</th>
<th>Default:</th>
<th>'notebook.services.contents.largefilemanager.LargeFileManager'</th>
</tr>
</thead>
<tbody>
<tr>
<td>The notebook manager class to use</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NotebookApp.cookie_options</strong> [Dict]</th>
<th>Default:</th>
<th>{}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra keyword arguments to pass to <strong>set_secure_cookie</strong>. See tornado’s <strong>set_secure_cookie</strong> docs for details.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NotebookApp.cookie_secret</strong> [Bytes]</th>
<th>Default:</th>
<th>b''</th>
</tr>
</thead>
<tbody>
<tr>
<td>The random bytes used to secure cookies. By default this is a new random number every time you start the Notebook. Set it to a value in a config file to enable logins to persist across server sessions.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note: Cookie secrets should be kept private, do not share config files with cookie_secret stored in plaintext (you can read the value from a file).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NotebookApp.cookie_secret_file</strong> [Unicode]</th>
<th>Default:</th>
<th>''</th>
</tr>
</thead>
<tbody>
<tr>
<td>The file where the cookie secret is stored.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NotebookApp.default_url</strong> [Unicode]</th>
<th>Default:</th>
<th>'/tree'</th>
</tr>
</thead>
<tbody>
<tr>
<td>The default URL to redirect to from /</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NotebookApp.disable_check_xsrf</strong> [Bool]</th>
<th>Default:</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable cross-site-request-forgery protection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jupyter notebook 4.3.1 introduces protection from cross-site request forgeries, requiring API requests to either:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• originate from pages served by this server (validated with XSRF cookie and token), or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• authenticate with a token</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Some anonymous compute resources still desire the ability to run code, completely without authentication. These services can disable all authentication and security checks, with the full knowledge of what that implies.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NotebookApp.enable_mathjax</strong> [Bool]</th>
<th>Default:</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whether to enable MathJax for typesetting math/TeX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MathJax is the javascript library Jupyter uses to render math/LaTeX. It is very large, so you may want to disable it if you have a slow internet connection, or for offline use of the notebook.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>When disabled, equations etc. will appear as their untransformed TeX source.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NotebookApp.extra_nbextensions_path</strong> [List]</th>
<th>Default:</th>
<th>[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>extra paths to look for Javascript notebook extensions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NotebookApp.extra_static_paths</strong> [List]</th>
<th>Default:</th>
<th>[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra paths to search for serving static files.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>This allows adding javascript/css to be available from the notebook server machine, or overriding individual files in the IPython</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NotebookApp.extra_template_paths</strong> [List]</th>
<th>Default:</th>
<th>[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra paths to search for serving jinja templates.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Can be used to override templates from notebook.templates.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.1. Options
**NotebookApp.file_to_run**  [Unicode] Default: ''

No description

**NotebookApp.ignore_minified_js**  [Bool] Default: False

Deprecated: Use minified JS file or not, mainly use during dev to avoid JS recompilation

**NotebookApp.iopub_data_rate_limit**  [Float] Default: 1000000

(bytes/sec) Maximum rate at which stream output can be sent on iopub before they are limited.

**NotebookApp.iopub_msg_rate_limit**  [Float] Default: 1000

(msgs/sec) Maximum rate at which messages can be sent on iopub before they are limited.

**NotebookApp.ip**  [Unicode] Default: 'localhost'

The IP address the notebook server will listen on.

**NotebookApp.jinja_environment_options**  [Dict] Default: {}

Supply extra arguments that will be passed to Jinja environment.

**NotebookApp.jinja_template_vars**  [Dict] Default: {}

Extra variables to supply to jinja templates when rendering.

**NotebookApp.kernel_manager_class**  [Type] Default: 'notebook.services.kernels.kernelmanager.MappingKernelManager'

The kernel manager class to use.

**NotebookApp.kernel_spec_manager_class**  [Type] Default: 'jupyter_client.kernelspec.KernelSpecManager'

The kernel spec manager class to use. Should be a subclass of jupyter_client.kernelspec.KernelSpecManager.

The Api of KernelSpecManager is provisional and might change without warning between this version of Jupyter and the next stable one.

**NotebookApp.keyfile**  [Unicode] Default: ''

The full path to a private key file for usage with SSL/TLS.

**NotebookApp.login_handler_class**  [Type] Default: 'notebook.auth.login.LoginHandler'

The login handler class to use.

**NotebookApp.logout_handler_class**  [Type] Default: 'notebook.auth.logout.LogoutHandler'

The logout handler class to use.

**NotebookApp.mathjax_config**  [Unicode] Default: 'TeX-AMS-MML_HTMLorMML-full, Safe'

The MathJax.js configuration file that is to be used.

**NotebookApp.mathjax_url**  [Unicode] Default: ''

A custom url for MathJax.js. Should be in the form of a case-sensitive url to MathJax, for example: /static/components/MathJax/MathJax.js

**NotebookApp.nbserver_extensions**  [Dict] Default: {}

Dict of Python modules to load as notebook server extensions. Entry values can be used to enable and disable the loading of the extensions. The extensions will be loaded in alphabetical order.

**NotebookApp.notebook_dir**  [Unicode] Default: ''

The directory to use for notebooks and kernels.
NotebookApp.open_browser  [Bool] Default: True

Whether to open in a browser after starting. The specific browser used is platform dependent and determined by the python standard library webbrowser module, unless it is overridden using the --browser (NotebookApp.browser) configuration option.

NotebookApp.password  [Unicode] Default: ''

Hashed password to use for web authentication.
To generate, type in a python/IPython shell:

from notebook.auth import passwd; passwd()

The string should be of the form type:salt:hashed-password.

NotebookApp.password_required  [Bool] Default: False

Forces users to use a password for the Notebook server. This is useful in a multi user environment, for instance when everybody in the LAN can access each other’s machine through ssh.
In such a case, server the notebook server on localhost is not secure since any user can connect to the notebook server via ssh.

NotebookApp.port  [Int] Default: 8888

The port the notebook server will listen on.

NotebookApp.port_retries  [Int] Default: 50

The number of additional ports to try if the specified port is not available.

NotebookApp.pylab  [Unicode] Default: 'disabled'

DISABLED: use %pylab or %matplotlib in the notebook to enable matplotlib.

NotebookApp.rate_limit_window  [Float] Default: 3

(sec) Time window used to check the message and data rate limits.


Reraise exceptions encountered loading server extensions?

NotebookApp.server_extensions  [List] Default: []

DEPRECATED use the nbserver_extensions dict instead

NotebookApp.session_manager_class  [Type] Default: 'notebook.services.sessions.sessionmanager.SessionManager'

The session manager class to use.

NotebookApp.shutdown_no_activity_timeout  [Int] Default: 0

Shut down the server after N seconds with no kernels or terminals running and no activity. This can be used together with culling idle kernels (MappingKernelManager.cull_idle_timeout) to shutdown the notebook server when it’s not in use. This is not precisely timed: it may shut down up to a minute later. 0 (the default) disables this automatic shutdown.

NotebookApp.ssl_options  [Dict] Default: {}

Supply SSL options for the tornado HTTPServer. See the tornado docs for details.

NotebookApp.terminado_settings  [Dict] Default: {}

Supply overrides for terminado. Currently only supports “shell_command”.

5.1. Options
NotebookApp.token [Unicode] Default: '<generated>'

Token used for authenticating first-time connections to the server.
When no password is enabled, the default is to generate a new, random token.
Setting to an empty string disables authentication altogether, which is NOT RECOMMENDED.

NotebookApp.tornado_settings [Dict] Default: {}

Supply overrides for the tornado.web.Application that the Jupyter notebook uses.

NotebookApp.trust_xheaders [Bool] Default: False

Whether to trust or not X-Scheme/X-Forwarded-Proto and X-Real-Ip/X-Forwarded-For headers sent by the up-stream reverse proxy. Necessary if the proxy handles SSL.

NotebookApp.webapp_settings [Dict] Default: {}

DEPRECATED, use tornado_settings

NotebookApp.webbrowser_open_new [Int] Default: 2

Specify Where to open the notebook on startup. This is the new argument passed to the standard library method webbrowser.open. The behaviour is not guaranteed, but depends on browser support. Valid values are:

2 opens a new tab, 1 opens a new window, 0 opens in an existing window.

See the webbrowser.open documentation for details.

NotebookApp.websocket_compression_options [Any] Default: None

Set the tornado compression options for websocket connections.
This value will be returned from WebSocketHandler.get_compression_options(). None (default) will disable compression. A dict (even an empty one) will enable compression.
See the tornado docs for WebSocketHandler.get_compression_options for details.

NotebookApp.websocket_url [Unicode] Default: ''

The base URL for websockets, if it differs from the HTTP server (hint: it almost certainly doesn’t).
Should be in the form of an HTTP origin: ws[s]://hostname[:port]

ConnectionFileMixin.connection_file [Unicode] Default: ''

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

ConnectionFileMixin.control_port [Int] Default: 0

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

ConnectionFileMixin hb_port [Int] Default: 0

set the heartbeat port [default: random]

ConnectionFileMixin.iopub_port [Int] Default: 0

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

ConnectionFileMixin.ip [Unicode] Default: ''

Set the kernel’s IP address [default localhost]. If the IP address is something other than localhost, then Consoles on other machines will be able to connect to the Kernel, so be careful!
ConnectionFileMixin.shell_port  [Int] Default: 0
    set the shell (ROUTER) port [default: random]
ConnectionFileMixin.stdin_port  [Int] Default: 0
    set the stdin (ROUTER) port [default: random]
ConnectionFileMixin.transport  ['tcp'|'ipc'] Default: 'tcp'
    No description
KernelManager.autorestart  [Bool] Default: True
    Should we autorestart the kernel if it dies.
KernelManager.kernel_cmd  [List] Default: []
    DEPRECATED: Use kernel_name instead.
    The Popen Command to launch the kernel. Override this if you have a custom kernel. If kernel_cmd is specified in a configuration file, Jupyter does not pass any arguments to the kernel, because it cannot make any assumptions about the arguments that the kernel understands. In particular, this means that the kernel does not receive the option --debug if it given on the Jupyter command line.
KernelManager.shutdown_wait_time  [Float] Default: 5.0
    Time to wait for a kernel to terminate before killing it, in seconds.
Session.buffer_threshold  [Int] Default: 1024
    Threshold (in bytes) beyond which an object’s buffer should be extracted to avoid pickling.
Session.check_pid  [Bool] Default: True
    Whether to check PID to protect against calls after fork.
    This check can be disabled if fork-safety is handled elsewhere.
Session.copy_threshold  [Int] Default: 65536
    Threshold (in bytes) beyond which a buffer should be sent without copying.
Session.debug  [Bool] Default: False
    Debug output in the Session
Session.digest_history_size  [Int] Default: 65536
    The maximum number of digests to remember.
    The digest history will be culled when it exceeds this value.
Session.item_threshold  [Int] Default: 64
    The maximum number of items for a container to be introspected for custom serialization. Containers larger than this are pickled outright.
Session.key  [CBytes] Default: b''
    execution key, for signing messages.
Session.keyfile  [Unicode] Default: ''
    path to file containing execution key.
Session.metadata  [Dict] Default: {} 
    Metadata dictionary, which serves as the default top-level metadata dict for each message.
Session.packer [DottedObjectName] Default: 'json'

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

Session.session [CUnicode] Default: ''

The UUID identifying this session.

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

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

Session.unpacker [DottedObjectName] Default: 'json'

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

Session.username [Unicode] Default: 'username'

Username for the Session. Default is your system username.

MultiKernelManager.default_kernel_name [Unicode] Default: 'python3'

The name of the default kernel to start

MultiKernelManager.kernel_manager_class [DottedObjectName] Default: 'jupyter_client.ioloop.IOLoopKernelManager'

The kernel manager class. This is configurable to allow subclassing of the KernelManager for customized behavior.

MappingKernelManager.buffer_offline_messages [Bool] Default: True

Whether messages from kernels whose frontends have disconnected should be buffered in-memory.

When True (default), messages are buffered and replayed on reconnect, avoiding lost messages due to interrupted connectivity.

Disable if long-running kernels will produce too much output while no frontends are connected.

MappingKernelManager.cull_busy [Bool] Default: False

Whether to consider culling kernels which are busy. Only effective if cull_idle_timeout > 0.

MappingKernelManager.cull_connected [Bool] Default: False

Whether to consider culling kernels which have one or more connections. Only effective if cull_idle_timeout > 0.

MappingKernelManager.cull_idle_timeout [Int] Default: 0

Timeout (in seconds) after which a kernel is considered idle and ready to be culled. Values of 0 or lower disable culling. Very short timeouts may result in kernels being culled for users with poor network connections.

MappingKernelManager.cull_interval [Int] Default: 300

The interval (in seconds) on which to check for idle kernels exceeding the cull timeout value.

MappingKernelManager.root_dir [Unicode] Default: ''

No description

ContentsManager.checkpoints [Instance] Default: None

No description
**ContentsManager.checkpoints_class** [Type] Default: 'notebook.services.contents.checkpoints.Checkpoints'

No description

**ContentsManager.checkpoints_kwargs** [Dict] Default: {}

No description

**ContentsManager.files_handler_class** [Type] Default: 'notebook.files.handlers.FilesHandler'

Handler class to use when serving raw file requests.

Default is a fallback that talks to the ContentsManager API, which may be inefficient, especially for large files. Local files-based ContentsManagers can use a StaticFileHandler subclass, which will be much more efficient. Access to these files should be Authenticated.

**ContentsManager.files_handler_params** [Dict] Default: {}

Extra parameters to pass to files_handler_class.

For example, StaticFileHandlers generally expect a *path* argument specifying the root directory from which to serve files.

**ContentsManager.hide_globs** [List] Default: ['__pycache__', '*.pyc', '*.pyo', '.DS_Store', '*.so', '*.dylib...']

Glob patterns to hide in file and directory listings.

**ContentsManager.pre_save_hook** [Any] Default: None

Python callable or importstring thereof

To be called on a contents model prior to save.

This can be used to process the structure, such as removing notebook outputs or other side effects that should not be saved.

It will be called as (all arguments passed by keyword):

```python
hook(path=path, model=model, contents_manager=self)
```

- model: the model to be saved. Includes file contents. Modifying this dict will affect the file that is stored.
- path: the API path of the save destination
- contents_manager: this ContentsManager instance

**ContentsManager.root_dir** [Unicode] Default: '/'

No description

**ContentsManager.untitled_directory** [Unicode] Default: 'Untitled Folder'

The base name used when creating untitled directories.

**ContentsManager.untitled_file** [Unicode] Default: 'untitled'

The base name used when creating untitled files.

**ContentsManager.untitled_notebook** [Unicode] Default: 'Untitled'

The base name used when creating untitled notebooks.
FileManagerMixin.use_atomic_writing  [Bool] Default: True
By default notebooks are saved on disk on a temporary file and then if successfully written, it replaces the old ones. This procedure, namely 'atomic_writing', causes some bugs on file system without operation order enforcement (like some networked fs). If set to False, the new notebook is written directly on the old one which could fail (eg: full filesystem or quota)

FileManager.delete_to_trash  [Bool] Default: True
If True (default), deleting files will send them to the platform’s trash/recycle bin, where they can be recovered. If False, deleting files really deletes them.

FileManager.post_save_hook  [Any] Default: None
Python callable or importstring thereof
to be called on the path of a file just saved.
This can be used to process the file on disk, such as converting the notebook to a script or HTML via nbconvert.
It will be called as (all arguments passed by keyword):

```
hook(os_path=os_path, model=model, contents_manager=instance)
```

- path: the filesystem path to the file just written
- model: the model representing the file
- contents_manager: this ContentsManager instance

FileManager.root_dir  [Unicode] Default: ''
No description

FileManager.save_script  [Bool] Default: False
DEPRECATED, use post_save_hook. Will be removed in Notebook 5.0

NotebookNotary.algorithm  ['md5'|'sha512'|'sha384'|'sha256'|'sha224'|'sha1'] Default: 'sha256'
The hashing algorithm used to sign notebooks.

NotebookNotary.db_file  [Unicode] Default: ''
The sqlite file in which to store notebook signatures. By default, this will be in your Jupyter data directory. You can set it to ':memory:' to disable sqlite writing to the filesystem.

NotebookNotary.secret  [Bytes] Default: b''
The secret key with which notebooks are signed.

NotebookNotary.secret_file  [Unicode] Default: ''
The file where the secret key is stored.

NotebookNotary.store_factory  [Callable] Default: traitlets.Undefined
A callable returning the storage backend for notebook signatures. The default uses an SQLite database.

KernelSpecManager.ensure_native_kernel  [Bool] Default: True
If there is no Python kernelspec registered and the IPython kernel is available, ensure it is added to the spec list.

KernelSpecManager.kernel_spec_class  [Type] Default: 'jupyter_client.kernelspec.KernelSpec'
The kernel spec class. This is configurable to allow subclassing of the KernelSpecManager for customized behavior.
**KernelSpecManager.whitelist** [Set] Default: `set()`

Whitelist of allowed kernel names.

By default, all installed kernels are allowed.
Running a notebook server

The Jupyter notebook web application is based on a server-client structure. The notebook server uses a two-process kernel architecture based on ZeroMQ, as well as Tornado for serving HTTP requests.

Note: By default, a notebook server runs locally at 127.0.0.1:8888 and is accessible only from localhost. You may access the notebook server from the browser using http://127.0.0.1:8888.

This document describes how you can secure a notebook server and how to run it on a public interface.

Important: This is not the multi-user server you are looking for. This document describes how you can run a public server with a single user. This should only be done by someone who wants remote access to their personal machine. Even so, doing this requires a thorough understanding of the set-up’s limitations and security implications. If you allow multiple users to access a notebook server as it is described in this document, their commands may collide, clobber and overwrite each other.

If you want a multi-user server, the official solution is JupyterHub. To use JupyterHub, you need a Unix server (typically Linux) running somewhere that is accessible to your users on a network. This may run over the public internet, but doing so introduces additional security concerns.

6.1 Securing a notebook server

You can protect your notebook server with a simple single password by configuring the NotebookApp.password setting in jupyter_notebook_config.py.

6.1.1 Prerequisite: A notebook configuration file

Check to see if you have a notebook configuration file, jupyter_notebook_config.py. The default location for this file is your Jupyter folder in your home directory, ~/.jupyter.
If you don’t already have one, create a config file for the notebook using the following command:

```
$ jupyter notebook --generate-config
```

### 6.1.2 Automatic Password setup

As of notebook 5.3, the first time you log-in using a token, the notebook server should give you the opportunity to setup a password from the user interface.

You will be presented with a form asking for the current `_token_`, as well as your `_new_ `_password_`; enter both and click on LogIn and setup new password.

Next time you need to log in you’ll be able to use the new password instead of the login token, otherwise follow the procedure to set a password from the command line.

The ability to change the password at first login time may be disabled by integrations by setting the `--NotebookApp.allow_password_change=False`

Starting at notebook version 5.0, you can enter and store a password for your notebook server with a single command. `jupyter notebook password` will prompt you for your password and record the hashed password in your `jupyter_notebook_config.json`.

```
$ jupyter notebook password
Enter password: ****
Verify password: ****
[NotebookPasswordApp] Wrote hashed password to /Users/you/.jupyter/jupyter_notebook_config.json
```

This can be used to reset a lost password; or if you believe your credentials have been leaked and desire to change your password. Changing your password will invalidate all logged-in sessions after a server restart.

### 6.1.3 Preparing a hashed password

You can prepare a hashed password manually, using the function `notebook.auth.security.passwd()`:

```
In [1]: from notebook.auth import passwd
In [2]: passwd()
Enter password:
Verify password:
Out[2]: 'sha1:67c9e60bb8b6:9ffede0825894254b2e042ea597d771089e11aed'
```

**Caution:** `passwd()` when called with no arguments will prompt you to enter and verify your password such as in the above code snippet. Although the function can also be passed a string as an argument such as `passwd('mypassword')`, please **do not** pass a string as an argument inside an IPython session, as it will be saved in your input history.

### 6.1.4 Adding hashed password to your notebook configuration file

You can then add the hashed password to your `jupyter_notebook_config.py`. The default location for this file `jupyter_notebook_config.py` is in your Jupyter folder in your home directory, `~/.jupyter`, e.g.: ```
c.NotebookApp.password = u'sha1:67c9e60bb8b6:9ffede0825894254b2e042ea597d771089e11aed'
```
Automatic password setup will store the hash in `jupyter_notebook_config.json` while this method store in `jupyter_notebook_config.py`. The `.json` configuration options take precedence over the `.py` one, thus the manual password may not take effect if the Json file as a password set.

### 6.1.5 Using SSL for encrypted communication

When using a password, it is a good idea to also use SSL with a web certificate, so that your hashed password is not sent unencrypted by your browser.

**Important:** Web security is rapidly changing and evolving. We provide this document as a convenience to the user, and recommend that the user keep current on changes that may impact security, such as new releases of OpenSSL. The Open Web Application Security Project (OWASP) website is a good resource on general security issues and web practices.

You can start the notebook to communicate via a secure protocol mode by setting the `certfile` option to your self-signed certificate, i.e. `mycert.pem`, with the command:

```
$ jupyter notebook --certfile=mycert.pem --keyfile mykey.key
```

**Tip:** A self-signed certificate can be generated with `openssl`. For example, the following command will create a certificate valid for 365 days with both the key and certificate data written to the same file:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout mykey.key -out mycert.pem
```

When starting the notebook server, your browser may warn that your self-signed certificate is insecure or unrecognized. If you wish to have a fully compliant self-signed certificate that will not raise warnings, it is possible (but rather involved) to create one, as explained in detail in this tutorial. Alternatively, you may use Let’s Encrypt to acquire a free SSL certificate and follow the steps in *Using Let’s Encrypt* to set up a public server.

### 6.2 Running a public notebook server

If you want to access your notebook server remotely via a web browser, you can do so by running a public notebook server. For optimal security when running a public notebook server, you should first secure the server with a password and SSL/HTTPS as described in *Securing a notebook server*.

Start by creating a certificate file and a hashed password, as explained in *Securing a notebook server*.

If you don’t already have one, create a config file for the notebook using the following command line:

```
$ jupyter notebook --generate-config
```

In the `~/.jupyter` directory, edit the notebook config file, `jupyter_notebook_config.py`. By default, the notebook config file has all fields commented out. The minimum set of configuration options that you should uncomment and edit in `jupyter_notebook_config.py` is the following:

```python
# Set options for certfile, ip, password, and toggle off
# browser auto-opening
c.NotebookApp.certfile = u'/absolute/path/to/your/certificate/mycert.pem'
c.NotebookApp.keyfile = u'/absolute/path/to/your/certificate/mykey.key'
# Set ip to '*' to bind on all interfaces (ips) for the public server
```
You can then start the notebook using the `jupyter notebook` command.

### 6.2.1 Using Let’s Encrypt

Let’s Encrypt provides free SSL/TLS certificates. You can also set up a public server using a Let’s Encrypt certificate.

Running a public notebook server will be similar when using a Let’s Encrypt certificate with a few configuration changes. Here are the steps:

1. Create a Let’s Encrypt certificate.
2. Use `Preparing a hashed password` to create one.
3. If you don’t already have config file for the notebook, create one using the following command:

   ```bash
   $ jupyter notebook --generate-config
   ```

4. In the `~/.jupyter` directory, edit the notebook config file, `jupyter_notebook_config.py`. By default, the notebook config file has all fields commented out. The minimum set of configuration options that you should uncomment and edit in `jupyter_notebook_config.py` is the following:

   ```python
   # Set options for certfile, ip, password, and toggle off
   # browser auto-opening
   c.NotebookApp.certfile = u'/absolute/path/to/your/certificate/fullchain.pem'
   c.NotebookApp.keyfile = u'/absolute/path/to/your/certificate/privkey.pem'
   # Set ip to '*' to bind on all interfaces (ips) for the public server
   c.NotebookApp.ip = '*'
   c.NotebookApp.password = u'sha1:bcd259ccf...<your hashed password here>'
   c.NotebookApp.open_browser = False
   # It is a good idea to set a known, fixed port for server access
   c.NotebookApp.port = 9999
   ```

You can then start the notebook using the `jupyter notebook` command.

**Important**: Use ‘https’. Keep in mind that when you enable SSL support, you must access the notebook server over `https://`, not over plain `http://`. The startup message from the server prints a reminder in the console, but it is easy to overlook this detail and think the server is for some reason non-responsive.

When using SSL, always access the notebook server with ‘https://’.

You may now access the public server by pointing your browser to `https://your.host.com:9999` where `your.host.com` is your public server’s domain.

### 6.2.2 Firewall Setup

To function correctly, the firewall on the computer running the jupyter notebook server must be configured to allow connections from client machines on the access port `c.NotebookApp.port` set in
jupyter_notebook_config.py to allow connections to the web interface. The firewall must also allow connections from 127.0.0.1 (localhost) on ports from 49152 to 65535. These ports are used by the server to communicate with the notebook kernels. The kernel communication ports are chosen randomly by ZeroMQ, and may require multiple connections per kernel, so a large range of ports must be accessible.

### 6.3 Running the notebook with a customized URL prefix

The notebook dashboard, which is the landing page with an overview of the notebooks in your working directory, is typically found and accessed at the default URL http://localhost:8888/.

If you prefer to customize the URL prefix for the notebook dashboard, you can do so through modifying jupyter_notebook_config.py. For example, if you prefer that the notebook dashboard be located with a subdirectory that contains other ipython files, e.g. http://localhost:8888/ipython/, you can do so with configuration options like the following (see above for instructions about modifying jupyter_notebook_config.py):

```python
c.NotebookApp.base_url = '/ipython/'
```

### 6.4 Embedding the notebook in another website

Sometimes you may want to embed the notebook somewhere on your website, e.g. in an IFrame. To do this, you may need to override the Content-Security-Policy to allow embedding. Assuming your website is at https://mywebsite.example.com, you can embed the notebook on your website with the following configuration setting in jupyter_notebook_config.py:

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

When embedding the notebook in a website using an iframe, consider putting the notebook in single-tab mode. Since the notebook opens some links in new tabs by default, single-tab mode keeps the notebook from opening additional tabs. Adding the following to ~/.jupyter/custom/custom.js will enable single-tab mode:

```javascript
define(['base/js/namespace'], function(Jupyter){
    Jupyter._target = '_self';
});
```

### 6.5 Known issues

#### 6.5.1 Proxies

When behind a proxy, especially if your system or browser is set to autodetect the proxy, the notebook web application might fail to connect to the server’s websockets, and present you with a warning at startup. In this case, you need to configure your system not to use the proxy for the server’s address.

For example, in Firefox, go to the Preferences panel, Advanced section, Network tab, click ‘Settings…’, and add the address of the notebook server to the ‘No proxy for’ field.
6.5.2 Docker CMD

Using `jupyter notebook` as a Docker CMD results in kernels repeatedly crashing, likely due to a lack of PID reaping. To avoid this, use the `tini init` as your Dockerfile `ENTRYPOINT`:

```bash
# Add Tini. Tini operates as a process subreaper for jupyter. This prevents
# kernel crashes.
ENV TINI_VERSION v0.6.0
ADD https://github.com/krallin/tini/releases/download/${TINI_VERSION}/tini /usr/bin/
    \-tini
RUN chmod +x /usr/bin/tini
ENTRYPOINT ["/usr/bin/tini", "--"]

EXPOSE 8888
CMD ["jupyter", "notebook", "--port=8888", "--no-browser", "--ip=0.0.0.0"]
```
CHAPTER 7

Security in the Jupyter notebook server

Since access to the Jupyter notebook server means access to running arbitrary code, it is important to restrict access to the notebook server. For this reason, notebook 4.3 introduces token-based authentication that is **on by default**.

**Note:** If you enable a password for your notebook server, token authentication is not enabled by default, and the behavior of the notebook server is unchanged from versions earlier than 4.3.

When token authentication is enabled, the notebook uses a token to authenticate requests. This token can be provided to login to the notebook server in three ways:

- **in the Authorization header, e.g.:**
  ```
  Authorization: token abcdef...
  ```

- **In a URL parameter, e.g.:**
  ```
  https://my-notebook/tree/?token=abcdef...
  ```

- **In the password field of the login form that will be shown to you if you are not logged in.**

When you start a notebook server with token authentication enabled (default), a token is generated to use for authentication. This token is logged to the terminal, so that you can copy/paste the URL into your browser:

```
[I 11:59:16.597 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/?token=c8de56fa4deed24899803e93c227592aef6538f93025fe01
```

If the notebook server is going to open your browser automatically (the default, unless --no-browser has been passed), an **additional** token is generated for launching the browser. This additional token can be used only once, and is used to set a cookie for your browser once it connects. After your browser has made its first request with this one-time-token, the token is discarded and a cookie is set in your browser.

At any later time, you can see the tokens and URLs for all of your running servers with **jupyter notebook list**:
For servers with token-authentication enabled, the URL in the above listing will include the token, so you can copy and paste that URL into your browser to login. If a server has no token (e.g. it has a password or has authentication disabled), the URL will not include the token argument. Once you have visited this URL, a cookie will be set in your browser and you won’t need to use the token again, unless you switch browsers, clear your cookies, or start a notebook server on a new port.

7.1 Alternatives to token authentication

If a generated token doesn’t work well for you, you can set a password for your notebook. `jupyter notebook password` will prompt you for a password, and store the hashed password in your `jupyter_notebook_config.json`.

New in version 5.0: `jupyter notebook password` command is added.

It is possible disable authentication altogether by setting the token and password to empty strings, but this is NOT RECOMMENDED, unless authentication or access restrictions are handled at a different layer in your web application:

```python
c.NotebookApp.token = ''
c.NotebookApp.password = ''
```
As Jupyter notebooks become more popular for sharing and collaboration, the potential for malicious people to attempt to exploit the notebook for their nefarious purposes increases. IPython 2.0 introduced a security model to prevent execution of untrusted code without explicit user input.

8.1 The problem

The whole point of Jupyter is arbitrary code execution. We have no desire to limit what can be done with a notebook, which would negatively impact its utility.

Unlike other programs, a Jupyter notebook document includes output. Unlike other documents, that output exists in a context that can execute code (via Javascript).

The security problem we need to solve is that no code should execute just because a user has opened a notebook that they did not write. Like any other program, once a user decides to execute code in a notebook, it is considered trusted, and should be allowed to do anything.

8.2 Our security model

- Untrusted HTML is always sanitized
- Untrusted Javascript is never executed
- HTML and Javascript in Markdown cells are never trusted
- Outputs generated by the user are trusted
- Any other HTML or Javascript (in Markdown cells, output generated by others) is never trusted
- The central question of trust is “Did the current user do this?”
8.3 The details of trust

When a notebook is executed and saved, a signature is computed from a digest of the notebook’s contents plus a secret key. This is stored in a database, writable only by the current user. By default, this is located at:

<table>
<thead>
<tr>
<th>Path</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>~/.local/share/jupyter/nbsignatures.db</td>
<td>Linux</td>
</tr>
<tr>
<td>~/Library/Jupyter/nbsignatures.db</td>
<td>OS X</td>
</tr>
<tr>
<td>%APPDATA%/jupyter/nbsignatures.db</td>
<td>Windows</td>
</tr>
</tbody>
</table>

Each signature represents a series of outputs which were produced by code the current user executed, and are therefore trusted.

When you open a notebook, the server computes its signature, and checks if it’s in the database. If a match is found, HTML and Javascript output in the notebook will be trusted at load, otherwise it will be untrusted.

Any output generated during an interactive session is trusted.

8.3.1 Updating trust

A notebook’s trust is updated when the notebook is saved. If there are any untrusted outputs still in the notebook, the notebook will not be trusted, and no signature will be stored. If all untrusted outputs have been removed (either via Clear Output or re-execution), then the notebook will become trusted.

While trust is updated per output, this is only for the duration of a single session. A newly loaded notebook file is either trusted or not in its entirety.

8.3.2 Explicit trust

Sometimes re-executing a notebook to generate trusted output is not an option, either because dependencies are unavailable, or it would take a long time. Users can explicitly trust a notebook in two ways:

- At the command-line, with:

  jupyter trust /path/to/notebook.ipynb

- After loading the untrusted notebook, with File / Trust Notebook

These two methods simply load the notebook, compute a new signature, and add that signature to the user’s database.

8.4 Reporting security issues

If you find a security vulnerability in Jupyter, either a failure of the code to properly implement the model described here, or a failure of the model itself, please report it to security@ipython.org.

If you prefer to encrypt your security reports, you can use this PGP public key.

8.5 Affected use cases

Some use cases that work in Jupyter 1.0 became less convenient in 2.0 as a result of the security changes. We do our best to minimize these annoyances, but security is always at odds with convenience.
8.5.1 Javascript and CSS in Markdown cells

While never officially supported, it had become common practice to put hidden Javascript or CSS styling in Markdown cells, so that they would not be visible on the page. Since Markdown cells are now sanitized (by Google Caja), all Javascript (including click event handlers, etc.) and CSS will be stripped.

We plan to provide a mechanism for notebook themes, but in the meantime styling the notebook can only be done via either custom.css or CSS in HTML output. The latter only have an effect if the notebook is trusted, because otherwise the output will be sanitized just like Markdown.

8.5.2 Collaboration

When collaborating on a notebook, people probably want to see the outputs produced by their colleagues’ most recent executions. Since each collaborator’s key will differ, this will result in each share starting in an untrusted state. There are three basic approaches to this:

- re-run notebooks when you get them (not always viable)
- explicitly trust notebooks via jupyter trust or the notebook menu (annoying, but easy)
- share a notebook signatures database, and use configuration dedicated to the collaboration while working on the project.

To share a signatures database among users, you can configure:

```
c.NotebookNotary.data_dir = "path/to/signature_dir"
```

to specify a non-default path to the SQLite database (of notebook hashes, essentially). We are aware that SQLite doesn’t work well on NFS and we are working out better ways to do this.
Chapter 9

Configuring the notebook frontend

Note: The ability to configure the notebook frontend UI and preferences is still a work in progress.

This document is a rough explanation on how you can persist some configuration options for the notebook JavaScript. There is no exhaustive list of all the configuration options as most options are passed down to other libraries, which means that non valid configuration can be ignored without any error messages.

9.1 How front end configuration works

The frontend configuration system works as follows:

- get a handle of a configurable JavaScript object.
- access its configuration attribute.
- update its configuration attribute with a JSON patch.

9.2 Example - Changing the notebook’s default indentation

This example explains how to change the default setting `indentUnit` for CodeMirror Code Cells:

```javascript
var cell = Jupyter.notebook.get_selected_cell();
var config = cell.config;
var patch = {
    CodeCell: {
        cm_config: {indentUnit: 2}
    }
}
config.update(patch)
```
You can enter the previous snippet in your browser’s JavaScript console once. Then reload the notebook page in your browser. Now, the preferred indent unit should be equal to two spaces. The custom setting persists and you do not need to reissue the patch on new notebooks.

`indentUnit`, used in this example, is one of the many CodeMirror options which are available for configuration.

### 9.3 Example - Restoring the notebook’s default indentation

If you want to restore a notebook frontend preference to its default value, you will enter a JSON patch with a null value for the preference setting.

For example, let’s restore the indent setting `indentUnit` to its default of four spaces. Enter the following code snippet in your JavaScript console:

```javascript
var cell = Jupyter.notebook.get_selected_cell();
var config = cell.config;
var patch = {
    CodeCell: {
        cm_config: {indentUnit: null} // only change here.
    }
}
config.update(patch)
```

Reload the notebook in your browser and the default indent should again be two spaces.

### 9.4 Persisting configuration settings

Under the hood, Jupyter will persist the preferred configuration settings in `~/.jupyter/nbconfig/<section>.json`, with `<section>` taking various value depending on the page where the configuration is issued. `<section>` can take various values like `notebook`, `tree`, and `editor`. A common section contains configuration settings shared by all pages.
10.1 Overview

10.1.1 How can the notebook be extended?

The Jupyter Notebook client and server application are both deeply customizable. Their behavior can be extended by creating, respectively:

- nbextension: a notebook extension
  - a single JS file, or directory of JavaScript, Cascading StyleSheets, etc. that contain at minimum a JavaScript module packaged as an AMD modules that exports a function \texttt{load_ipython_extension}

- server extension: an importable Python module
  - that implements \texttt{load_jupyter_server_extension}

- bundler extension: an importable Python module with generated File -> Download as / Deploy as menu item trigger
  - that implements \texttt{bundle}

10.1.2 Why create a Python package for Jupyter extensions?

Since it is rare to have a server extension that does not have any frontend components (an nbextension), for convenience and consistency, all these client and server extensions with their assets can be packaged and versioned together as a Python package with a few simple commands. This makes installing the package of extensions easier and less error-prone for the user.
10.2 Installation of Jupyter Extensions

10.2.1 Install a Python package containing Jupyter Extensions

There are several ways that you may get a Python package containing Jupyter Extensions. Commonly, you will use a package manager for your system:

```
pip install helpful_package
# or
conda install helpful_package
# or
apt-get install helpful_package
```

where 'helpful_package' is a Python package containing one or more Jupyter Extensions.

10.2.2 Enable a Server Extension

The simplest case would be to enable a server extension which has no frontend components.

A pip user that wants their configuration stored in their home directory would type the following command:

```
jupyter serverextension enable --py helpful_package
```

Alternatively, a virtualenv or conda user can pass --sys-prefix which keeps their environment isolated and reproducible. For example:

```
# Make sure that your virtualenv or conda environment is activated
[source] activate my-environment

jupyter serverextension enable --py helpful_package --sys-prefix
```

10.2.3 Install the nbextension assets

If a package also has an nbextension with frontend assets that must be available (but not necessarily enabled by default), install these assets with the following command:

```
jupyter nbextension install --py helpful_package # or --sys-prefix if using virtualenv or conda
```

10.2.4 Enable nbextension assets

If a package has assets that should be loaded every time a Jupyter app (e.g. lab, notebook, dashboard, terminal) is loaded in the browser, the following command can be used to enable the nbextension:

```
jupyter nbextension enable --py helpful_package # or --sys-prefix if using virtualenv or conda
```
10.3 Did it work? Check by listing Jupyter Extensions.

After running one or more extension installation steps, you can list what is presently known about nbextensions, server extensions, or bundler extensions. The following commands will list which extensions are available, whether they are enabled, and other extension details:

```
jupyter nbextension list
jupyter serverextension list
jupyter bundlerextension list
```

10.4 Additional resources on creating and distributing packages

Of course, in addition to the files listed, there are number of other files one needs to build a proper package. Here are some good resources:
- The Hitchhiker’s Guide to Packaging
- Repository Structure and Python by Kenneth Reitz
- Packaging and Distributing Projects
- conda: Building packages

Here are some tools to get you started:
- generator-nbextension

10.5 Example - Server extension

10.5.1 Creating a Python package with a server extension

Here is an example of a python module which contains a server extension directly on itself. It has this directory structure:

```
- setup.py
- MANIFEST.in
- my_module/
  - __init__.py
```

10.5.2 Defining the server extension

This example shows that the server extension and its `load_jupyter_server_extension` function are defined in the `__init__.py` file.

```
my_module/__init__.py

def _jupyter_server_extension_paths():
    return [{'
        "module": "my_module"
    }]

def load_jupyter_server_extension(nbapp):
    nbapp.log.info("my module enabled!")
```
10.5.3 Install and enable the server extension

Which a user can install with:

```
jupyter serverextension enable --py my_module [--sys-prefix]
```

10.6 Example - Server extension and nbextension

10.6.1 Creating a Python package with a server extension and nbextension

Here is another server extension, with a front-end module. It assumes this directory structure:

```
- setup.py
- MANIFEST.in
- my_fancy_module/
  - __init__.py
  - static/
    - index.js
```

10.6.2 Defining the server extension and nbextension

This example again shows that the server extension and its `load_jupyter_server_extension` function are defined in the `__init__.py` file. This time, there is also a function `jupyter_nbextension_path` for the nbextension.

```
my_fancy_module/__init__.py

def __jupyter_server_extension_paths():
    return [{
        "module": "my_fancy_module"
    }]

# Jupyter Extension points

def __jupyter_nbextension_paths():
    return [dict(
        section="notebook",
        # the path is relative to the 'my_fancy_module' directory
        src="static",
        # directory in the 'nbextension/' namespace
        dest="my_fancy_module",
        # _also_ in the 'nbextension/' namespace
        require="my_fancy_module/index")]

def load_jupyter_server_extension(nbapp):
    nbapp.log.info("my module enabled!")
```

10.6.3 Install and enable the server extension and nbextension

The user can install and enable the extensions with the following set of commands:
10.7 Example - Bundler extension

10.7.1 Creating a Python package with a bundlerextension

Here is a bundler extension that adds a Download as -> Notebook Tarball (tar.gz) option to the notebook File menu. It assumes this directory structure:

- setup.py
- MANIFEST.in
- my_tarball_bundler/
  - __init__.py

10.7.2 Defining the bundler extension

This example shows that the bundler extension and its bundle function are defined in the __init__.py file.

my_tarball_bundler/__init__.py

```python
import tarfile
import io
import os
import nbformat

def __jupyter_bundlerextension_paths():
    
    return [{
        "name": "tarball_bundler",
        "module_name": "my_tarball_bundler",
        "label": "Notebook Tarball (tar.gz)",
        "group": "download",
    }]

def bundle(handler, model):
    
    notebook_filename = model['name']
```

10.7. Example - Bundler extension

```
jupyter nbextension install --py my_fancy_module [--sys-prefix|--user]
jupyter nbextension enable --py my_fancy_module [--sys-prefix|--system]
jupyter serverextension enable --py my_fancy_module [--sys-prefix|--system]
```
notebook_content = nbformat.writes(model['content']).encode('utf-8')
notebook_name = os.path.splitext(notebook_filename)[0]
tar_filename = '{}.tar.gz'.format(notebook_name)

info = tarfile.TarInfo(notebook_filename)
info.size = len(notebook_content)

with io.BytesIO() as tar_buffer:
    with tarfile.open(tar_filename, "w:gz", fileobj=tar_buffer) as tar:
        tar.addfile(info, io.BytesIO(notebook_content))

    # Set headers to trigger browser download
    handler.set_header('Content-Disposition', 'attachment; filename="{}"'.format(tar_filename))
    handler.set_header('Content-Type', 'application/gzip')

    # Return the buffer value as the response
    handler.finish(tar_buffer.getvalue())

See *Extending the Notebook* for more documentation about writing nbextensions, server extensions, and bundler extensions.
CHAPTER 11

Extending the Notebook

Certain subsystems of the notebook server are designed to be extended or overridden by users. These documents explain these systems, and show how to override the notebook’s defaults with your own custom behavior.

11.1 Contents API

The Jupyter Notebook web application provides a graphical interface for creating, opening, renaming, and deleting files in a virtual filesystem.

The ContentsManager class defines an abstract API for translating these interactions into operations on a particular storage medium. The default implementation, FileContentsManager, uses the local filesystem of the server for storage and straightforwardly serializes notebooks into JSON. Users can override these behaviors by supplying custom subclasses of ContentsManager.

This section describes the interface implemented by ContentsManager subclasses. We refer to this interface as the Contents API.

11.1.1 Data Model

Filesystem Entities

ContentsManager methods represent virtual filesystem entities as dictionaries, which we refer to as models. Models may contain the following entries:
Certain model fields vary in structure depending on the type field of the model. There are three model types: notebook, file, and directory.

- **notebook models**
  - The format field is always "json".
  - The mimetype field is always None.
  - The content field contains a `nbformat.notebooknode.NotebookNode` representing the .ipynb file represented by the model. See the NBFormat documentation for a full description.

- **file models**
  - The format field is either "text" or "base64".
  - The mimetype field is text/plain for text-format models and application/octet-stream for base64-format models.
  - The content field is always of type unicode. For text-format file models, content simply contains the file’s bytes after decoding as UTF-8. Non-text (base64) files are read as bytes, base64 encoded, and then decoded as UTF-8.

- **directory models**
  - The format field is always "json".
  - The mimetype field is always None.
  - The content field contains a list of content-free models representing the entities in the directory.

**Note:** In certain circumstances, we don’t need the full content of an entity to complete a Contents API request. In such cases, we omit the mimetype, content, and format keys from the model. This most commonly occurs when listing a directory, in which circumstance we represent files within the directory as content-less models to avoid having to recursively traverse and serialize the entire filesystem.

## Sample Models

```python
# Notebook Model with Content
{
    'content': {
        'metadata': {},
        'nbformat': 4,
        'nbformat_minor': 0,
        'cells': [
            {
                'cell_type': 'markdown',
                'metadata': {},
                'source': 'Some **Markdown**'
            }
        ]
    }
}```
API Paths

ContentsManager methods represent the locations of filesystem resources as API-style paths. Such paths are interpreted as relative to the root directory of the notebook server. For compatibility across systems, the following guarantees are made:

- Paths are always unicode, not bytes.
- Paths are not URL-escaped.
- Paths are always forward-slash (/) delimited, even on Windows.
- Leading and trailing slashes are stripped. For example, /foo/bar/buzz/ becomes foo/bar/buzz.
- The empty string ("") represents the root directory.

11.1.2 Writing a Custom ContentsManager

The default ContentsManager is designed for users running the notebook as an application on a personal computer. It stores notebooks as .ipynb files on the local filesystem, and it maps files and directories in the Notebook UI to files and directories on disk. It is possible to override how notebooks are stored by implementing your own custom subclass of ContentsManager. For example, if you deploy the notebook in a context where you don’t trust or don’t have access to the filesystem of the notebook server, it’s possible to write your own ContentsManager that stores notebooks and files in a database.

Required Methods

A minimal complete implementation of a custom ContentsManager must implement the following methods:
### 11.1.3 Customizing Checkpoints

TODO:

### 11.1.4 Testing

`notebook.services.contents.tests` includes several test suites written against the abstract Contents API. This means that an excellent way to test a new `ContentsManager` subclass is to subclass our tests to make them use your `ContentsManager`.

**Note:** `PGContents` is an example of a complete implementation of a custom `ContentsManager`. It stores notebooks and files in PostgreSQL and encodes directories as SQL relations. `PGContents` also provides an example of how to re-use the notebook’s tests.

### 11.2 File save hooks

You can configure functions that are run whenever a file is saved. There are two hooks available:

- `ContentsManager.pre_save_hook` runs on the API path and model with content. This can be used for things like stripping output that people don’t like adding to VCS noise.

- `FileContentsManager.post_save_hook` runs on the filesystem path and model without content. This could be used to commit changes after every save, for instance.

They are both called with keyword arguments:

```python
pre_save_hook(model=model, path=path, contents_manager=cm)
post_save_hook(model=model, os_path=os_path, contents_manager=cm)
```

#### 11.2.1 Examples

These can both be added to `jupyter_notebook_config.py`.

A pre-save hook for stripping output:

```python
def scrub_output_pre_save(model, **kwargs):
    """scrub output before saving notebooks""
    # only run on notebooks
    if model['type'] != 'notebook':
        return
```
# only run on nbformat v4
if model['content']['nbformat'] != 4:
    return

for cell in model['content']['cells']:
    if cell['cell_type'] != 'code':
        continue
    cell['outputs'] = []
    cell['execution_count'] = None

c.FileContentsManager.pre_save_hook = scrub_output_pre_save

A post-save hook to make a script equivalent whenever the notebook is saved (replacing the --script option in older versions of the notebook):

```python
import io
import os
from notebook.utils import to_api_path

__script_exporter = None

def script_post_save(model, os_path, contents_manager, **kwargs):
    
    """convert notebooks to Python script after save with nbconvert
    replaces `ipython notebook --script`
    ""
    from nbconvert.exporters.script import ScriptExporter

    if model['type'] != 'notebook':
        return

    global __script_exporter

    if __script_exporter is None:
        __script_exporter = ScriptExporter(parent=contents_manager)

    log = contents_manager.log

    base, ext = os.path.splitext(os_path)
    py_fname = base + '.py'
    script, resources = __script_exporter.from_filename(os_path)
    script_fname = base + resources.get('output_extension', '.txt')
    log.info("Saving script /%s", to_api_path(script_fname, contents_manager.root_dir))

    with io.open(script_fname, 'w', encoding='utf-8') as f:
        f.write(script)

c.FileContentsManager.post_save_hook = script_post_save
```

This could be a simple call to `jupyter nbconvert --to script`, but spawning the subprocess every time is quite slow.

11.2. File save hooks 53
11.3 Custom request handlers

The notebook webserver can be interacted with using a well defined RESTful API. You can define custom RESTful API handlers in addition to the ones provided by the notebook. As described below, to define a custom handler you need to first write a notebook server extension. Then, in the extension, you can register the custom handler.

11.3.1 Writing a notebook server extension

The notebook webserver is written in Python, hence your server extension should be written in Python too. Server extensions, like IPython extensions, are Python modules that define a specially named load function, `load_jupyter_server_extension`. This function is called when the extension is loaded.

```python
def load_jupyter_server_extension(nb_server_app):
    """
    Called when the extension is loaded.
    
    Args:
        nb_server_app (NotebookWebApplication): handle to the Notebook webserver instance.
    """
    pass
```

To get the notebook server to load your custom extension, you’ll need to add it to the list of extensions to be loaded. You can do this using the config system. `NotebookApp.nbserver_extensions` is a config variable which is a dictionary of strings, each a Python module to be imported, mapping to `True` to enable or `False` to disable each extension. Because this variable is notebook config, you can set it two different ways, using config files or via the command line.

For example, to get your extension to load via the command line add a double dash before the variable name, and put the Python dictionary in double quotes. If your package is “mypackage” and module is “mymodule”, this would look like `jupyter notebook --NotebookApp.nbserver_extensions="{'mypackage.mymodule':True}"`. Basically the string should be Python importable.

Alternatively, you can have your extension loaded regardless of the command line args by setting the variable in the Jupyter config file. The default location of the Jupyter config file is `~/.jupyter/jupyter_notebook_config.py` (see Configuration Overview). Inside the config file, you can use Python to set the variable. For example, the following config does the same as the previous command line example.

```python
c = get_config()
c.NotebookApp.nbserver_extensions = {
    'mypackage.mymodule': True,
}
```

Before continuing, it’s a good idea to verify that your extension is being loaded. Use a print statement to print something unique. Launch the notebook server and you should see your statement printed to the console.

11.3.2 Registering custom handlers

Once you’ve defined a server extension, you can register custom handlers because you have a handle to the Notebook server app instance (`nb_server_app` above). However, you first need to define your custom handler. To declare a custom handler, inherit from `notebook.base.handlers.IPythonHandler`. The example below[1] is a Hello World handler:
from notebook.base.handlers import IPythonHandler
class HelloWorldHandler(IPythonHandler):
    def get(self):
        self.finish('Hello, world!')

The Jupyter Notebook server uses Tornado as its web framework. For more information on how to implement request handlers, refer to the Tornado documentation on the matter.

After defining the handler, you need to register the handler with the Notebook server. See the following example:

```
web_app = nb_server_app.web_app
host_pattern = '.*$
route_pattern = url_path_join(web_app.settings['base_url'], '/hello')
web_app.add_handlers(host_pattern, [(route_pattern, HelloWorldHandler)])
```

Putting this together with the extension code, the example looks like the following:

```
from notebook.utils import url_path_join
from notebook.base.handlers import IPythonHandler

class HelloWorldHandler(IPythonHandler):
    def get(self):
        self.finish('Hello, world!')

def load_jupyter_server_extension(nb_server_app):
    """Called when the extension is loaded."
    Args:
        nb_server_app (NotebookWebApplication): handle to the Notebook webserver instance.
    """
    web_app = nb_server_app.web_app
    host_pattern = '.*$
    route_pattern = url_path_join(web_app.settings['base_url'], '/hello')
    web_app.add_handlers(host_pattern, [(route_pattern, HelloWorldHandler)])
```

References:
1. Peter Parente’s Mindtrove

### 11.4 Custom front-end extensions

This describes the basic steps to write a JavaScript extension for the Jupyter notebook front-end. This allows you to customize the behaviour of the various pages like the dashboard, the notebook, or the text editor.

#### 11.4.1 The structure of a front-end extension

**Note:** The notebook front-end and Javascript API are not stable, and are subject to a lot of changes. Any extension written for the current notebook is almost guaranteed to break in the next release.
A front-end extension is a JavaScript file that defines an **AMD module** which exposes at least a function called 'load_ipython_extension', which takes no arguments. We will not get into the details of what each of these terms consists of yet, but here is the minimal code needed for a working extension:

```javascript
// file my_extension/main.js
define(function(){
    function load_ipython_extension(){
        console.info('this is my first extension');
    }

    return {
        load_ipython_extension: load_ipython_extension
    };
});
```

**Note:** Although for historical reasons the function is called 'load_ipython_extension', it does apply to the Jupyter notebook in general, and will work regardless of the kernel in use.

If you are familiar with JavaScript, you can use this template to require any Jupyter module and modify its configuration, or do anything else in client-side Javascript. Your extension will be loaded at the right time during the notebook page initialisation for you to set up a listener for the various events that the page can trigger.

You might want access to the current instances of the various Jupyter notebook components on the page, as opposed to the classes defined in the modules. The current instances are exposed by a module named 'base/js/namespace'. If you plan on accessing instances on the page, you should **require** this module rather than accessing the global variable 'Jupyter', which will be removed in future. The following example demonstrates how to access the current notebook instance:

```javascript
// file my_extension/main.js
define(['base/js/namespace'], function(Jupyter) {
    function load_ipython_extension() {
        console.log('This is the current notebook application instance:', Jupyter.notebook);
    }

    return {
        load_ipython_extension: load_ipython_extension
    };
});
```

### 11.4.2 Modifying key bindings

One of the abilities of extensions is to modify key bindings, although once again this is an API which is not guaranteed to be stable. However, custom key bindings are frequently requested, and are helpful to increase accessibility, so in the following we show how to access them.
Here is an example of an extension that will unbind the shortcut `0,0` in command mode, which normally restarts the kernel, and bind `0,0,0` in its place:

```javascript
// file my_extension/main.js
define([
  'base/js/namespace'
], function (Jupyter)

  function load_ipython_extension() {
    Jupyter.keyboard_manager.command_shortcuts.remove_shortcut('0,0');
    Jupyter.keyboard_manager.command_shortcuts.add_shortcut('0,0,0', 'jupyter-notebook:restart-kernel');
  }

  return {
    load_ipython_extension: load_ipython_extension
 );
});
```

**Note:** The standard keybindings might not work correctly on non-US keyboards. Unfortunately, this is a limitation of browser implementations and the status of keyboard event handling on the web in general. We appreciate your feedback if you have issues binding keys, or have any ideas to help improve the situation.

You can see that I have used the **action name** `jupyter-notebook:restart-kernel` to bind the new shortcut. There is no API yet to access the list of all available **actions**, though the following in the JavaScript console of your browser on a notebook page should give you an idea of what is available:

```javascript
Object.keys(require('base/js/namespace').actions._actions);
```

In this example, we changed a keyboard shortcut in **command mode**; you can also customize keyboard shortcuts in **edit mode**. However, most of the keyboard shortcuts in edit mode are handled by CodeMirror, which supports custom key bindings via a completely different API.

### 11.4.3 Defining and registering your own actions

As part of your front-end extension, you may wish to define actions, which can be attached to toolbar buttons, or called from the command palette. Here is an example of an extension that defines an (not very useful!) action to show an alert, and adds a toolbar button using the full action name:

```javascript
// file my_extension/main.js
define([
  'base/js/namespace'
], function (Jupyter)

  function load_ipython_extension() {
    var handler = function () {
      alert('this is an alert from my_extension!');
    };

    // Define the action
    function show_alert() {
      handler();
    }

    // Add toolbar button
    Jupyter.toolbar.add_button('Show Alert', show_alert);
  }

  return {
    load_ipython_extension: load_ipython_extension
 );
});
```
Every action needs a name, which, when joined with its prefix to make the full action name, should be unique. Built-in actions, like the `jupyter-notebook:restart-kernel` we bound in the earlier Modifying key bindings example, use the prefix `jupyter-notebook`. For actions defined in an extension, it makes sense to use the extension name as the prefix. For the action name, the following guidelines should be considered:

- First pick a noun and a verb for the action. For example, if the action is “restart kernel,” the verb is “restart” and the noun is “kernel”.
- Omit terms like “selected” and “active” by default, so “delete-cell”, rather than “delete-selected-cell”. Only provide a scope like “-all-” if it is other than the default “selected” or “active” scope.
- If an action has a secondary action, separate the secondary action with “-and-“, so “restart-kernel-and-clear-output”.
- Use above/below or previous/next to indicate spatial and sequential relationships.
- Don’t ever use before/after as they have a temporal connotation that is confusing when used in a spatial context.
- For dialogs, use a verb that indicates what the dialog will accomplish, such as “confirm-restart-kernel”.

### 11.4.4 Installing and enabling extensions

You can install your nbextension with the command:

```
jupyter nbextension install path/to/my_extension/ [--user|--sys-prefix]
```

The default installation is system-wide. You can use --user to do a per-user installation, or --sys-prefix to install to Python’s prefix (e.g. in a virtual or conda environment). Where my_extension is the directory containing the Javascript files. This will copy it to a Jupyter data directory (the exact location is platform dependent - see Data files).

For development, you can use the --symlink flag to symlink your extension rather than copying it, so there’s no need to reinstall after changes.

To use your extension, you’ll also need to **enable** it, which tells the notebook interface to load it. You can do that with another command:

```
jupyter nbextension enable my_extension/main [--sys-prefix]
```
The argument refers to the Javascript module containing your `load_ipython_extension` function, which is `my_extension/main.js` in this example. There is a corresponding `disable` command to stop using an extension without uninstalling it.

Changed in version 4.2: Added `--sys-prefix` argument

### 11.4.5 Kernel Specific extensions

**Warning:** This feature serves as a stopgap for kernel developers who need specific JavaScript injected onto the page. The availability and API are subject to change at anytime.

It is possible to load some JavaScript on the page on a per kernel basis. Be aware that doing so will make the browser page reload without warning as soon as the user switches the kernel without notice.

If you, a kernel developer, need a particular piece of JavaScript to be loaded on a “per kernel” basis, such as:
- if you are developing a CodeMirror mode for your language
- if you need to enable some specific debugging options

Your kernelspecs are allowed to contain a `kernel.js` file that defines an AMD module. The AMD module should define an `onload` function that will be called when the kernelspec loads, such as:
- when you load a notebook that uses your kernelspec
- change the active kernelspec of a notebook to your kernelspec.

Note that adding a `kernel.js` to your kernelspec will add an unexpected side effect to changing a kernel in the notebook. As it is impossible to “unload” JavaScript, any attempt to change the kernelspec again will save the current notebook and reload the page without confirmations.

Here is an example of `kernel.js`:

```javascript
define(function(){
  return {onload: function(){
    console.info('Kernel specific javascript loaded');

    // do more things here, like define a codemirror mode
  }}
});
```

### 11.5 Customize keymaps

**Note:** Declarative Custom Keymaps is a provisional feature with unstable API which is not guaranteed to be kept in future versions of the notebook, and can be removed or changed without warnings.

The notebook shortcuts that are defined by jupyter both in edit mode and command mode are configurable in the frontend configuration file `~/.jupyter/nbconfig/notebook.json`. The modification of Keyboard shortcut suffer of several limitations, mainly that your Browser and OS might prevent certain shortcut to work correctly. If this is the case, there are unfortunately not much than can be done. The second issue can arise with keyboard that have a layout different than US English. Again even if we are aware of the issue, there is not much we can do about that.
Shortcut are also limited by the underlying library that handle code and text edition: CodeMirror. If some Keyboard shortcuts are conflicting, the method describe below might not work to create new keyboard shortcuts, especially in the edit mode of the notebook.

The 4 sections of interest in ~/.jupyter/nbconfig/notebook.json are the following:

- keys.command.unbind
- keys.edit.unbind
- keys.command.bind
- keys.edit.bind

The first two section describe which default keyboard shortcut not to register at notebook startup time. These are mostly useful if you need to unbind a default keyboard shortcut before binding it to a new command.

These two first sections apply respectively to the command and edit mode of the notebook. They take a list of shortcut to unbind.

For example, to unbind the shortcut to split a cell at the position of the cursor (Ctrl-Shift-Minus) use the following:

```json
// file ~/.jupyter/nbconfig/notebook.json
{
  "keys": {
    "edit": {
      "unbind": [
        "Ctrl-Shift-Minus"
      ]
    },
  },
}
```

The last two section describe which new keyboard shortcut to register at notebook startup time, and which actions they trigger.

These two last sections apply respectively to the command and edit mode of the notebook. They take a dictionary with shortcuts as keys and commands name as value.

For example, to bind the shortcut G, G, G (Press G three time in a row) in command mode, to the command that restart the kernel and run all cells, use the following:

```json
// file ~/.jupyter/nbconfig/notebook.json
{
  "keys": {
    "command": {
      "bind": {
        "G,G,G":"jupyter-notebook:restart-kernel-and-run-all-cells"
      }
    },
  },
}
```

The name of the available commands can be find by hovering the right end of a row in the command palette.
11.6 Custom bundler extensions

The notebook server supports the writing of bundler extensions that transform, package, and download/deploy notebook files. As a developer, you need only write a single Python function to implement a bundler. The notebook server automatically generates a File -> Download as or File -> Deploy as menu item in the notebook front-end to trigger your bundler.

Here are some examples of what you can implement using bundler extensions:

- Convert a notebook file to a HTML document and publish it as a post on a blog site
- Create a snapshot of the current notebook environment and bundle that definition plus notebook into a zip download
- Deploy a notebook as a standalone, interactive dashboard

To implement a bundler extension, you must do all of the following:

- Declare bundler extension metadata in your Python package
- Write a bundle function that responds to bundle requests
- Instruct your users on how to enable/disable your bundler extension

The following sections describe these steps in detail.

11.6.1 Declaring bundler metadata

You must provide information about the bundler extension(s) your package provides by implementing a _jupyter_bundlerextensions_paths function. This function can reside anywhere in your package so long as it can be imported when enabling the bundler extension. (See Enabling/disabling bundler extensions.)

```python
# in mypackage.hello_bundler

def _jupyter_bundlerextension_paths():
    """Example "hello world" bundler extension""
    return [{
        'name': 'hello_bundler',  # unique bundler name
        'label': 'Hello Bundler',  # human-readable menu item label
        'module_name': 'mypackage.hello_bundler',  # module containing bundle()
        'group': 'deploy'  # group under 'deploy' or 'download' menu
    ]
```

Note that the return value is a list. By returning multiple dictionaries in the list, you allow users to enable/disable sets of bundlers all at once.

11.6.2 Writing the bundle function

At runtime, a menu item with the given label appears either in the File -> Deploy as or File -> Download as menu depending on the group value in your metadata. When a user clicks the menu item, a new browser tab opens and notebook server invokes a bundle function in the module_name specified in the metadata.

You must implement a bundle function that matches the signature of the following example:

```python
# in mypackage.hello_bundler

def bundle(handler, model):
```

11.6. Custom bundler extensions 61
"""Transform, convert, bundle, etc. the notebook referenced by the given model.

Then issue a Tornado web response using the 'handler' to redirect
the user's browser, download a file, show a HTML page, etc. This function
must finish the handler response before returning either explicitly or by
raising an exception.

Parameters
----------
handler : tornado.web.RequestHandler
    Handler that serviced the bundle request
model : dict
    Notebook model from the configured ContentManager
"""
handler.finish('I bundled {}!'.format(model['path']))

Your bundle function is free to do whatever it wants with the request and respond in any manner. For example, it may
read additional query parameters from the request, issue a redirect to another site, run a local process (e.g., nbconvert),
make a HTTP request to another service, etc.

The caller of the bundle function is @tornado.gen.coroutine decorated and wraps its call with tornado.gen.maybe_future. This behavior means you may handle the web request synchronously, as in the
example above, or asynchronously using @tornado.gen.coroutine and yield, as in the example below.

```python
from tornado import gen

@gen.coroutine
def bundle(handler, model):
    # simulate a long running IO op (e.g., deploying to a remote host)
    yield gen.sleep(10)

    # now respond
    handler.finish('I spent 10 seconds bundling {}!'.format(model['path']))
```

You should prefer the second, asynchronous approach when your bundle operation is long-running and would other-
wise block the notebook server main loop if handled synchronously.

For more details about the data flow from menu item click to bundle function invocation, see Bundler invocation
details.

### 11.6.3 Enabling/disabling bundler extensions

The notebook server includes a command line interface (CLI) for enabling and disabling bundler extensions.

You should document the basic commands for enabling and disabling your bundler. One possible command for
enabling the hello_bundler example is the following:

```
jupyter bundlerextension enable --py mypackage.hello_bundler --sys-prefix
```

The above updates the notebook configuration file in the current conda/virtualenv environment (--sys-prefix) with the
metadata returned by the mypackage.hello_bundler._jupyter_bundlerextension_paths function.

The corresponding command to later disable the bundler extension is the following:

```
jupyter bundlerextension disable --py mypackage.hello_bundler --sys-prefix
```
For more help using the `bundlerextension` subcommand, run the following.

```
jupyter bundlerextension --help
```

The output describes options for listing enabled bundlers, configuring bundlers for single users, configuring bundlers system-wide, etc.

### 11.6.4 Example: IPython Notebook bundle (.zip)

The `hello_bundler` example in this documentation is simplistic in the name of brevity. For more meaningful examples, see `notebook/bundler/zip_bundler.py` and `notebook/bundler/tarball_bundler.py`. You can enable them to try them like so:

```
jupyter bundlerextension enable --py notebook.bundler.zip_bundler --sys-prefix
jupyter bundlerextension enable --py notebook.bundler.tarball_bundler --sys-prefix
```

### 11.6.5 Bundler invocation details

Support for bundler extensions comes from Python modules in `notebook/bundler` and JavaScript in `notebook/static/notebook/js/menubar.js`. The flow of data between the various components proceeds roughly as follows:

1. User opens a notebook document
2. Notebook front-end JavaScript loads notebook configuration
3. Bundler front-end JS creates menu items for all bundler extensions in the config
4. User clicks a bundler menu item
5. JS click handler opens a new browser window/tab to `<notebook base_url>/bundle/<path/to/notebook>?bundler=<name>` (i.e., a HTTP GET request)
6. Bundle handler validates the notebook path and bundler name
7. Bundle handler delegates the request to the `bundle` function in the bundler's `module_name`
8. `bundle` function finishes the HTTP request
CHAPTER 12

Contributing to the Jupyter Notebook

If you’re reading this section, you’re probably interested in contributing to Jupyter. Welcome and thanks for your interest in contributing!

Please take a look at the Contributor documentation, familiarize yourself with using the Jupyter Notebook, and introduce yourself on the mailing list and share what area of the project you are interested in working on.

12.1 General Guidelines

For general documentation about contributing to Jupyter projects, see the Project Jupyter Contributor Documentation.

12.2 Setting Up a Development Environment

12.2.1 Installing Node.js and npm

Building the Notebook from its GitHub source code requires some tools to create and minify JavaScript components and the CSS, specifically Node.js and Node’s package manager, npm. It should be node version 6.0.

If you use conda, you can get them with:

```
conda install -c conda-forge nodejs
```

If you use Homebrew on Mac OS X:

```
brew install node
```

Installation on Linux may vary, but be aware that the nodejs or npm packages included in the system package repository may be too old to work properly.

You can also use the installer from the Node.js website.
12.2.2 Installing the Jupyter Notebook

Once you have installed the dependencies mentioned above, use the following steps:

```
pip install --upgrade setuptools pip
git clone https://github.com/jupyter/notebook
cd notebook
pip install -e .
```

If you are using a system-wide Python installation and you only want to install the notebook for you, you can add `--user` to the install commands.

Once you have done this, you can launch the master branch of Jupyter notebook from any directory in your system with:

```
jupyter notebook
```

12.2.3 Rebuilding JavaScript and CSS

There is a build step for the JavaScript and CSS in the notebook. To make sure that you are working with up-to-date code, you will need to run this command whenever there are changes to JavaScript or LESS sources:

```
npm run build
```

**IMPORTANT:** Don’t forget to run `npm run build` after switching branches. When switching between branches of different versions (e.g. 4.x and master), run `pip install -e .`. If you have tried the above and still find that the notebook is not reflecting the current source code, try cleaning the repo with `git clean -xfd` and reinstalling with `pip install -e .`.

**Development Tip**

When doing development, you can use this command to automatically rebuild JavaScript and LESS sources as they are modified:

```
npm run build:watch
```

**Git Hooks**

If you want to automatically update dependencies and recompile JavaScript and CSS after checking out a new commit, you can install post-checkout and post-merge hooks which will do it for you:

```
git-hooks/install-hooks.sh
```

See `git-hooks/README.md` for more details.

12.3 Running Tests

12.3.1 Python Tests

Install dependencies:
To run the Python tests, use:

```
 nosetests
```

If you want coverage statistics as well, you can run:

```
 nosetests --with-coverage --cover-package=notebook notebook
```

### 12.3.2 JavaScript Tests

To run the JavaScript tests, you will need to have PhantomJS and CasperJS installed:

```
 npm install -g casperjs phantomjs-prebuilt
```

Then, to run the JavaScript tests:

```
 python -m notebook.jstest [group]
```

where `[group]` is an optional argument that is a path relative to `notebook/tests/`. For example, to run all tests in `notebook/tests/notebook`:

```
 python -m notebook.jstest notebook
```

or to run just `notebook/tests/notebook/deletecell.js`:

```
 python -m notebook.jstest notebook/deletecell.js
```

### 12.4 Building the Documentation

To build the documentation you’ll need Sphinx, pandoc and a few other packages.

To install (and activate) a conda environment named `notebook_docs` containing all the necessary packages (except pandoc), use:

```
 conda env create -f docs/environment.yml
 source activate notebook_docs  # Linux and OS X
 activate notebook_docs        # Windows
```

If you want to install the necessary packages with pip instead:

```
 pip install -r docs/doc-requirements.txt
```

Once you have installed the required packages, you can build the docs with:

```
 cd docs
 make html
```

After that, the generated HTML files will be available at `build/html/index.html`. You may view the docs in your browser.

You can automatically check if all hyperlinks are still valid:
Windows users can find `make.bat` in the `docs` folder.

You should also have a look at the Project Jupyter Documentation Guide.
This document guides a contributor through creating a release of the Jupyter notebook.

13.1 Check installed tools

Review CONTRIBUTING.rst. Make sure all the tools needed to generate the minified JavaScript and CSS files are properly installed.

13.2 Clean the repository

You can remove all non-tracked files with:

```
git clean -xfdi
```

This would ask you for confirmation before removing all untracked files. Make sure the dist/ folder is clean and avoid stale build from previous attempts.

13.3 Create the release

1. Update version number in notebook/_version.py.
2. Run this command:

```
python setup.py jsversion
```

It will modify (at least) notebook/static/base/js/namespace.js which makes the notebook version available from within JavaScript.
3. Commit and tag the release with the current version number:
4. You are now ready to build the sdist and wheel:

```bash
python setup.py sdist
python setup.py bdist_wheel
```

5. You can now test the wheel and the sdist locally before uploading to PyPI. Make sure to use `twine` to upload the archives over SSL.

```bash
twine upload dist/*
```

6. If all went well, change the `notebook/_version.py` back adding the `.dev` suffix.

7. Push directly on master, not forgetting to push `--tags` too.
1. How do I install a prerelease version such as a beta or release candidate?

```bash
python -m pip install notebook --pre --upgrade
```

View the original notebooks on nbviewer
The following notebooks have been rendered for your convenience.

15.1 What is the Jupyter Notebook?

15.1.1 Introduction

The Jupyter Notebook is an interactive computing environment that enables users to author notebook documents that include: - Live code - Interactive widgets - Plots - Narrative text - Equations - Images - Video

These documents provide a complete and self-contained record of a computation that can be converted to various formats and shared with others using email, Dropbox, version control systems (like git/GitHub) or nbviewer.jupyter.org.

Components

The Jupyter Notebook combines three components:

- **The notebook web application**: An interactive web application for writing and running code interactively and authoring notebook documents.
- **Kernels**: Separate processes started by the notebook web application that runs users’ code in a given language and returns output back to the notebook web application. The kernel also handles things like computations for interactive widgets, tab completion and introspection.
- **Notebook documents**: Self-contained documents that contain a representation of all content visible in the notebook web application, including inputs and outputs of the computations, narrative text, equations, images, and rich media representations of objects. Each notebook document has its own kernel.

15.1.2 Notebook web application

The notebook web application enables users to:
• **Edit code in the browser**, with automatic syntax highlighting, indentation, and tab completion/introspection.

• **Run code from the browser**, with the results of computations attached to the code which generated them.

• See the results of computations with **rich media representations**, such as HTML, LaTeX, PNG, SVG, PDF, etc.

• Create and use **interactive JavaScript widgets**, which bind interactive user interface controls and visualizations to reactive kernel side computations.

• Author **narrative text** using the Markdown markup language.

• Include mathematical equations using **LaTeX syntax in Markdown**, which are rendered in-browser by MathJax.

### 15.1.3 Kernels

Through Jupyter's kernel and messaging architecture, the Notebook allows code to be run in a range of different programming languages. For each notebook document that a user opens, the web application starts a kernel that runs the code for that notebook. Each kernel is capable of running code in a single programming language and there are kernels available in the following languages:

- Python ([https://github.com/ipython/ipython](https://github.com/ipython/ipython))
- R ([https://github.com/IRkernel/IRkernel](https://github.com/IRkernel/IRkernel))
- Ruby ([https://github.com/minrk/iruby](https://github.com/minrk/iruby))
- Haskell ([https://github.com/gibiansky/IHaskell](https://github.com/gibiansky/IHaskell))
- Scala ([https://github.com/Bridgewater/scala-notebook](https://github.com/Bridgewater/scala-notebook))
- node.js ([https://gist.github.com/Carreau/4279371](https://gist.github.com/Carreau/4279371))
- Go ([https://github.com/takluyver/igo](https://github.com/takluyver/igo))

The default kernel runs Python code. The notebook provides a simple way for users to pick which of these kernels is used for a given notebook.

Each of these kernels communicate with the notebook web application and web browser using a JSON over ZeroMQ/WebSockets message protocol that is described [here](https://jupyter.readthedocs.io/en/latest/messaging.html). Most users don’t need to know about these details, but it helps to understand that “kernels run code.”

### 15.1.4 Notebook documents

Notebook documents contain the **inputs and outputs** of an interactive session as well as **narrative text** that accompanies the code but is not meant for execution. Rich output generated by running code, including HTML, images, video, and plots, is embedded in the notebook, which makes it a complete and self-contained record of a computation.

When you run the notebook web application on your computer, notebook documents are just **files on your local filesystem with a “.ipynb” extension**. This allows you to use familiar workflows for organizing your notebooks into folders and sharing them with others.

Notebooks consist of a **linear sequence of cells**. There are four basic cell types:

- **Code cells**: Input and output of live code that is run in the kernel
- **Markdown cells**: Narrative text with embedded LaTeX equations
- **Heading cells**: 6 levels of hierarchical organization and formatting
• **Raw cells**: Unformatted text that is included, without modification, when notebooks are converted to different formats using nbconvert

Internally, notebook documents are ‘JSON <https://en.wikipedia.org/wiki/JSON>‘ data with binary values ‘base64 <http://en.wikipedia.org/wiki/Base64>‘ encoded. This allows them to be read and manipulated programmatically by any programming language. Because JSON is a text format, notebook documents are version control friendly.

**Notebooks can be exported** to different static formats including HTML, reStructuredText, LaTeX, PDF, and slide shows (reveal.js) using Jupyter’s nbconvert utility.

Furthermore, any notebook document available from a public URL on or GitHub can be shared via nbviewer. This service loads the notebook document from the URL and renders it as a static web page. The resulting web page may thus be shared with others without their needing to install the Jupyter Notebook.

### 15.2 Notebook Basics

#### 15.2.1 The Notebook dashboard

When you first start the notebook server, your browser will open to the notebook dashboard. The dashboard serves as a home page for the notebook. Its main purpose is to display the notebooks and files in the current directory. For example, here is a screenshot of the dashboard page for the examples directory in the Jupyter repository:

![Fig. 15.1: Jupyter dashboard showing files tab](image)

The top of the notebook list displays clickable breadcrumbs of the current directory. By clicking on these breadcrumbs or on sub-directories in the notebook list, you can navigate your file system.
To create a new notebook, click on the “New” button at the top of the list and select a kernel from the dropdown (as seen below). Which kernels are listed depend on what’s installed on the server. Some of the kernels in the screenshot below may not exist as an option to you.

![Fig. 15.2: Jupyter “New” menu](image)

Notebooks and files can be uploaded to the current directory by dragging a notebook file onto the notebook list or by the “click here” text above the list.

The notebook list shows green “Running” text and a green notebook icon next to running notebooks (as seen below). Notebooks remain running until you explicitly shut them down; closing the notebook’s page is not sufficient.

To shutdown, delete, duplicate, or rename a notebook check the checkbox next to it and an array of controls will appear at the top of the notebook list (as seen below). You can also use the same operations on directories and files when applicable.

To see all of your running notebooks along with their directories, click on the “Running” tab:

This view provides a convenient way to track notebooks that you start as you navigate the file system in a long running notebook server.

### 15.2.2 Overview of the Notebook UI

If you create a new notebook or open an existing one, you will be taken to the notebook user interface (UI). This UI allows you to run code and author notebook documents interactively. The notebook UI has the following main areas:

- Menu
- Toolbar
- Notebook area and cells
Fig. 15.3: Jupyter dashboard showing one notebook with a running kernel

Fig. 15.4: Buttons: Duplicate, rename, shutdown, delete, new, refresh
Fig. 15.5: Jupyter dashboard running tab
The notebook has an interactive tour of these elements that can be started in the “Help:User Interface Tour” menu item.

### 15.2.3 Modal editor

Starting with IPython 2.0, the Jupyter Notebook has a modal user interface. This means that the keyboard does different things depending on which mode the Notebook is in. There are two modes: edit mode and command mode.

#### Edit mode

Edit mode is indicated by a green cell border and a prompt showing in the editor area:

```
In [1]: a = 10
```

![Fig. 15.6: Jupyter cell with green border](image)

When a cell is in edit mode, you can type into the cell, like a normal text editor. Enter edit mode by pressing `Enter` or using the mouse to click on a cell’s editor area.

#### Command mode

Command mode is indicated by a grey cell border with a blue left margin:

```
In [1]: a = 10
```

![Fig. 15.7: Jupyter cell with blue & grey border](image)

When you are in command mode, you are able to edit the notebook as a whole, but not type into individual cells. Most importantly, in command mode, the keyboard is mapped to a set of shortcuts that let you perform notebook and cell actions efficiently. For example, if you are in command mode and you press `c`, you will copy the current cell - no modifier is needed.

Don’t try to type into a cell in command mode; unexpected things will happen! Enter command mode by pressing `Esc` or using the mouse to click outside a cell’s editor area.

### 15.2.4 Mouse navigation

All navigation and actions in the Notebook are available using the mouse through the menubar and toolbar, which are both above the main Notebook area:

![Fig. 15.8: Jupyter notebook menus and toolbar](image)
The first idea of mouse based navigation is that **cells can be selected by clicking on them**. The currently selected cell gets a grey or green border depending on whether the notebook is in edit or command mode. If you click inside a cell’s editor area, you will enter edit mode. If you click on the prompt or output area of a cell you will enter command mode.

If you are running this notebook in a live session (not on http://nbviewer.jupyter.org) try selecting different cells and going between edit and command mode. Try typing into a cell.

The second idea of mouse based navigation is that **cell actions usually apply to the currently selected cell**. Thus if you want to run the code in a cell, you would select it and click the button in the toolbar or the “Cell:Run” menu item. Similarly, to copy a cell you would select it and click the button in the toolbar or the “Edit:Copy” menu item. With this simple pattern, you should be able to do most everything you need with the mouse.

Markdown cells have one other state that can be modified with the mouse. These cells can either be rendered or unrendered. When they are rendered, you will see a nice formatted representation of the cell’s contents. When they are unrendered, you will see the raw text source of the cell. To render the selected cell with the mouse, click the button in the toolbar or the “Cell:Run” menu item. To unrerender the selected cell, double click on the cell.

### 15.2.5 Keyboard Navigation

The modal user interface of the Jupyter Notebook has been optimized for efficient keyboard usage. This is made possible by having two different sets of keyboard shortcuts: one set that is active in edit mode and another in command mode.

The most important keyboard shortcuts are **Enter**, which enters edit mode, and **Esc**, which enters command mode.

In edit mode, most of the keyboard is dedicated to typing into the cell’s editor. Thus, in edit mode there are relatively few shortcuts. In command mode, the entire keyboard is available for shortcuts, so there are many more. The Help->“Keyboard Shortcuts” dialog lists the available shortcuts.

We recommend learning the command mode shortcuts in the following rough order:

1. **Basic navigation**: enter, shift-enter, up/k, down/j
2. **Saving the notebook**: s
3. **Change Cell types**: y, m, 1-6, t
4. **Cell creation**: a, b
5. **Cell editing**: x, c, v, d, z
6. **Kernel operations**: i, 0 (press twice)

### 15.3 Running Code

First and foremost, the Jupyter Notebook is an interactive environment for writing and running code. The notebook is capable of running code in a wide range of languages. However, each notebook is associated with a single kernel. This notebook is associated with the IPython kernel, therefor runs Python code.

#### 15.3.1 Code cells allow you to enter and run code

Run a code cell using **Shift-Enter** or pressing the button in the toolbar above:
In [2]: a = 10
In [3]: print(a)
10

There are two other keyboard shortcuts for running code:

- Alt-Enter runs the current cell and inserts a new one below.
- Ctrl-Enter run the current cell and enters command mode.

### 15.3.2 Managing the Kernel

Code is run in a separate process called the Kernel. The Kernel can be interrupted or restarted. Try running the following cell and then hit the button in the toolbar above.

In [4]: import time
time.sleep(10)

If the Kernel dies you will be prompted to restart it. Here we call the low-level system libc.time routine with the wrong argument via ctypes to segfault the Python interpreter:

In [5]: import sys
from ctypes import CDLL
   # This will crash a Linux or Mac system
   # equivalent calls can be made on Windows
   
   # Uncomment these lines if you would like to see the segfault
   
   # dll = 'dylib' if sys.platform == 'darwin' else 'so.6'
   # libc = CDLL("libc.%s" % dll)
   # libc.time(-1) # BOOM!!

### 15.3.3 Cell menu

The “Cell” menu has a number of menu items for running code in different ways. These includes:

- Run and Select Below
- Run and Insert Below
- Run All
- Run All Above
- Run All Below

### 15.3.4 Restarting the kernels

The kernel maintains the state of a notebook’s computations. You can reset this state by restarting the kernel. This is done by clicking on the in the toolbar above.
15.3.5 sys.stdout and sys.stderr

The stdout and stderr streams are displayed as text in the output area.

```
In [6]: print("hi, stdout")
hi, stdout
```

```
In [7]: from __future__ import print_function
    print('hi, stderr', file=sys.stderr)
hi, stderr
```

15.3.6 Output is asynchronous

All output is displayed asynchronously as it is generated in the Kernel. If you execute the next cell, you will see the output one piece at a time, not all at the end.

```
In [8]: import time, sys
    for i in range(8):
        print(i)
        time.sleep(0.5)
```

15.3.7 Large outputs

To better handle large outputs, the output area can be collapsed. Run the following cell and then single- or double-click on the active area to the left of the output:

```
In [9]: for i in range(50):
    print(i)
```
Beyond a certain point, output will scroll automatically:

```python
In [10]: for i in range(500):
    ...:     print(2**i - 1)
```

<table>
<thead>
<tr>
<th>i</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>63</td>
</tr>
<tr>
<td>7</td>
<td>127</td>
</tr>
<tr>
<td>8</td>
<td>255</td>
</tr>
<tr>
<td>9</td>
<td>511</td>
</tr>
<tr>
<td>10</td>
<td>1023</td>
</tr>
<tr>
<td>11</td>
<td>2047</td>
</tr>
<tr>
<td>12</td>
<td>4095</td>
</tr>
<tr>
<td>13</td>
<td>8191</td>
</tr>
<tr>
<td>14</td>
<td>16383</td>
</tr>
<tr>
<td>15</td>
<td>32767</td>
</tr>
<tr>
<td>16</td>
<td>65535</td>
</tr>
<tr>
<td>17</td>
<td>131071</td>
</tr>
<tr>
<td>18</td>
<td>262143</td>
</tr>
<tr>
<td>19</td>
<td>524287</td>
</tr>
<tr>
<td>20</td>
<td>1048575</td>
</tr>
<tr>
<td>21</td>
<td>2097151</td>
</tr>
<tr>
<td>22</td>
<td>4194303</td>
</tr>
</tbody>
</table>
8388607
16777215
33554431
67108863
134217727
268435455
536870911
1073741823
2147483647
4294967295
8589934591
17179869183
34359738367
68719476735
137438953471
274877906943
549755813887
1099511627775
219902355551
439804611103
879609322007
17592186044415
35184372088831
70368744177663
14073748835327
281474976710655
562949953421311
1125899906842623
2251799813685247
4503599627370495
9007199254740991
18014398509481983
36028797018963967
72057594037927935
144115188075855871
288230376151711743
576460752303423487
1152921504606846975
2305843009213693951
4611686018427387903
9223372036854775807
18446744073709551615
36893488147419103231
73786976294838206463
147573952589676412927
295147905179352825855
590295810358705651711
1180591620717411303423
2361183241434822606847
4722366482869645213695
9444732965739290427391
18889465931478580854783
37778931862957161709567
75557863725914323419135
151115727451828646838271
302231454903657293676543
604462909807314587353087
1208925819614629174706175
2417851639229258349412351

Chapter 15. Examples
15.3. Running Code

Jupyter Notebook Documentation, Release 5.3.0.dev0
Chapter 15. Examples
10.4 Markdown Cells

Text can be added to Jupyter Notebooks using Markdown cells. Markdown is a popular markup language that is a superset of HTML. Its specification can be found here:

https://daringfireball.net/projects/markdown/

10.4.1 Markdown basics

You can make text italic or bold.

You can build nested itemized or enumerated lists:

- One
  - Sublist
    * This
  - Sublist - That - The other thing
- Two
- Three
- Sublist

Now another list:

1. Here we go
   (a) Sublist
   (b) Sublist
2. There we go
3. Now this

You can add horizontal rules:

Here is a blockquote:

Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren’t special enough to break the rules. Although practicality beats purity. Errors should never pass silently. Unless explicitly silenced. In the face of ambiguity, refuse the temptation to guess. There should be one– and preferably only one –obvious way to do it. Although that way may not be obvious at first unless you’re Dutch. Now is better than never. Although never is often better than right now. If the implementation is hard to explain, it’s a bad idea. If the implementation is easy to explain, it may be a good idea. Namespaces are one honking great idea – let’s do more of those!
And shorthand for links:

Jupyter’s website

### 15.4.2 Headings

You can add headings by starting a line with one (or multiple) `#` followed by a space, as in the following example:

```
# Heading 1
# Heading 2
## Heading 2.1
## Heading 2.2
```

### 15.4.3 Embedded code

You can embed code meant for illustration instead of execution in Python:

```python
def f(x):
    """a docstring"
    return x**2
```

or other languages:

```javascript
if (i=0; i<n; i++) {
    printf("hello $d\n", i);
    x += 4;
}
```

### 15.4.4 LaTeX equations

Courtesy of MathJax, you can include mathematical expressions both inline: $e^{i\pi} + 1 = 0$ and displayed:

$$e^x = \sum_{i=0}^{\infty} \frac{1}{i!}x^i$$

Inline expressions can be added by surrounding the latex code with $:$

$e^{i\pi} + 1 = 0$

Expressions on their own line are surrounded by $$:

$$e^x=\sum_{i=0}^{\infty}\frac{1}{i!}x^i$$

### 15.4.5 GitHub flavored markdown

The Notebook webapp supports Github flavored markdown meaning that you can use triple backticks for code blocks:

```python
print "Hello World"
```

```javascript
```

### 15.4. Markdown Cells

93
console.log("Hello World")
```
Gives:
```
print "Hello World"
console.log("Hello World")
And a table like this:

<table>
<thead>
<tr>
<th>This</th>
<th>is</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>table</td>
</tr>
</tbody>
</table>

A nice HTML Table:

15.4.6 General HTML

Because Markdown is a superset of HTML you can even add things like HTML tables:

Header 1
Header 2
row 1, cell 1
row 1, cell 2
row 2, cell 1
row 2, cell 2

15.4.7 Local files

If you have local files in your Notebook directory, you can refer to these files in Markdown cells directly:

[subdirectory/]<filename>

For example, in the images folder, we have the Python logo:

<img src="../images/python_logo.svg" />

and a video with the HTML5 video tag:

<video controls src="../images/animation.m4v" />

These do not embed the data into the notebook file, and require that the files exist when you are viewing the notebook.
Security of local files

Note that this means that the Jupyter notebook server also acts as a generic file server for files inside the same tree as your notebooks. Access is not granted outside the notebook folder so you have strict control over what files are visible, but for this reason it is highly recommended that you do not run the notebook server with a notebook directory at a high level in your filesystem (e.g. your home directory).

When you run the notebook in a password-protected manner, local file access is restricted to authenticated users unless read-only views are active.

15.5 Keyboard Shortcut Customization

Starting with Jupyter Notebook 5.0, you can customize the command mode shortcuts from within the Notebook Application itself.

Head to the “Help” menu and select the “Edit keyboard Shortcuts” item. A dialog will guide you through the process of adding custom keyboard shortcuts.

Keyboard shortcut set from within the Notebook Application will be persisted to your configuration file. A single action may have several shortcuts attached to it.

15.6 Keyboard Shortcut Customization (Pre Notebook 5.0)

Starting with IPython 2.0 keyboard shortcuts in command and edit mode are fully customizable. These customizations are made using the Jupyter JavaScript API. Here is an example that makes the r key available for running a cell:

```javascript
In [1]: %%javascript
   Jupyter.keyboard_manager.command_shortcuts.add_shortcut('r', {
       help : 'run cell',
       help_index : 'zz',
       handler : function (event) {
           IPython.notebook.execute_cell();
           return false;
       }
   });
<IPython.core.display.Javascript object>
```

“By default the keypress r, while in command mode, changes the type of the selected cell to raw. This shortcut is overridden by the code in the previous cell, and thus the action no longer be available via the keypress r.”

There are a couple of points to mention about this API:

- The help_index field is used to sort the shortcuts in the Keyboard Shortcuts help dialog. It defaults to zz.
- When a handler returns false it indicates that the event should stop propagating and the default action should not be performed. For further details about the event object or event handling, see the jQuery docs.
- If you don’t need a help or help_index field, you can simply pass a function as the second argument to add_shortcut.

```javascript
In [2]: %%javascript
   Jupyter.keyboard_manager.command_shortcuts.addShortcut('r', function (event) {
       IPython.notebook.execute_cell();
       return false;
   });
```
Likewise, to remove a shortcut, use `remove_shortcut`:

```
In [3]: %%javascript
    
    Jupyter.keyboard_manager.command_shortcuts.remove_shortcut('r');
```

If you want your keyboard shortcuts to be active for all of your notebooks, put the above API calls into your `custom.js` file.

Of course we provide name for majority of existing action so that you do not have to re-write everything, here is for example how to bind `r` back to it’s initial behavior:

```
In [4]: %%javascript
    
    Jupyter.keyboard_manager.command_shortcuts.add_shortcut('r', 'jupyter-notebook:change-cell-to-raw');
```

### 15.7 Embracing web standards

One of the main reasons why we developed the current notebook web application was to embrace the web technology. By being a pure web application using HTML, Javascript, and CSS, the Notebook can get all the web technology improvement for free. Thus, as browser support for different media extend, the notebook web app should be able to be compatible without modification.

This is also true with performance of the User Interface as the speed of Javascript VM increases.

The other advantage of using only web technology is that the code of the interface is fully accessible to the end user and is modifiable live. Even if this task is not always easy, we strive to keep our code as accessible and reusable as possible. This should allow us - with minimum effort - development of small extensions that customize the behavior of the web interface.

#### 15.7.1 Tampering with the Notebook application

The first tool that is available to you and that you should be aware of are browser “developers tool”. The exact naming can change across browser and might require the installation of extensions. But basically they can allow you to inspect/modify the DOM, and interact with the javascript code that runs the frontend.

- In Chrome and Safari, Developer tools are in the menu View > Developer > Javascript Console
- In Firefox you might need to install Firebug

Those will be your best friends to debug and try different approaches for your extensions.

**Injecting JS**

**Using magics**

The above tools can be tedious for editing edit long JavaScript files. Therefore we provide the `%%javascript` magic. This allows you to quickly inject JavaScript into the notebook. Still the javascript injected this way will not survive reloading. Hence, it is a good tool for testing and refining a script.
You might see here and there people modifying CSS and injecting JS into the notebook by reading file(s) and publishing them into the notebook. Not only does this often break the flow of the notebook and make the re-execution of the notebook broken, but it also means that you need to execute those cells in the entire notebook every time you need to update the code.

This can still be useful in some cases, like the %autosave magic that allows you to control the time between each save. But this can be replaced by a JavaScript dropdown menu to select the save interval.

```
In [ ]: ## you can inspect the autosave code to see what it does.
%autosave??
```

**custom.js**

To inject Javascript we provide an entry point: custom.js that allows the user to execute and load other resources into the notebook. Javascript code in custom.js will be executed when the notebook app starts and can then be used to customize almost anything in the UI and in the behavior of the notebook.

custom.js can be found in the ~/.jupyter/custom/custom.js. You can share your custom.js with others.

### Back to theory

```
In [ ]: from jupyter_core.paths import jupyter_config_dir
   jupyter_dir = jupyter_config_dir()
   jupyter_dir

and custom.js is in

In [ ]: import os.path
   custom_js_path = os.path.join(jupyter_dir, 'custom', 'custom.js')

In [ ]: # my custom js
   if os.path.isfile(custom_js_path):
       with open(custom_js_path) as f:
           print(f.read())
   else:
       print("You don't have a custom.js file")
```

Note that custom.js is meant to be modified by user. When writing a script, you can define it in a separate file and add a line of configuration into custom.js that will fetch and execute the file.

**Warning**: even if modification of custom.js takes effect immediately after browser refresh (except if browser cache is aggressive), creating a file in static/ directory needs a server restart.

#### 15.7.2 Exercise :

- Create a custom.js in the right location with the following content:

```javascript
alert("hello world from custom.js")
```

- Restart your server and open any notebook.

- Be greeted by custom.js

Have a look at default custom.js, to see its content and for more explanation.
For the quick ones:

We’ve seen above that you can change the autosave rate by using a magic. This is typically something I don’t want to type every time, and that I don’t like to embed into my workflow and documents. (readers don’t care what my autosave time is). Let’s build an extension that allows us to do it.

Create a dropdown element in the toolbar (DOM `Jupyter.toolbar.element`), you will need

- `Jupyter.notebook.set_autosave_interval(milliseconds)`
- know that 1 min = 60 sec, and 1 sec = 1000 ms

```javascript
var label = jQuery('<label/>').text('AutoScroll Limit:');
var select = jQuery('<select/>')
  .append(jQuery('<option/>').attr('value', '2').text('2min (default)'))
  .append(jQuery('<option/>').attr('value', undefined).text('disabled'))
  .append(jQuery('<option/>').attr('value', time_m[0]).text(thr+'min'));

select.change(function() {
  var val = jQuery(this).val() // val will be the value in [2]
  // TODO
  // this will be called when dropdown changes
});
```

```javascript
var time_m = [1,5,10,15,30];
for (var i=0; i < time_m.length; i++) {
  var ts = time_m[i];
  select.append($('<option/>').attr('value', ts).text(thr+'min'));
}
```

A non-interactive example first

I like my cython to be nicely highlighted

```python
Jupyter.config.cell_magic_highlight['magic_text/x-cython'] = {}
Jupyter.config.cell_magic_highlight['magic_text/x-cython'].reg = ['/^%%cython/']
```

text/x-cython is the name of CodeMirror mode name, magic_ prefix will just patch the mode so that the first line that contains a magic does not screw up the highlighting. reg is a list or regular expression that will trigger the change of mode.
Get more documentation

Sadly, you will have to read the js source file (but there are lots of comments) and/or build the JavaScript documentation using yuidoc. If you have node and yui-doc installed:

```bash
$ cd ~/jupyter/notebook/notebook/static/notebook/js/
$ yuidoc . --server
warn: (yuidoc): Failed to extract port, setting to the default :3000
info: (yuidoc): Starting YUIDoc@0.3.45 using YUI@3.9.1 with NodeJS@0.10.15
info: (yuidoc): Starting YUIDoc with the following options:
info: (yuidoc):
| port: 3000, |
| nocode: false, |
| paths: [ '.' ], |
| server: true, |
| outdir: './out' |
info: (server): Starting server: http://127.0.0.1:3000
```

and browse http://127.0.0.1:3000 to get documentation

Some convenience methods

By browsing the documentation you will see that we have some convenience methods that allows us to avoid re-inventing the UI every time:

```javascript
Jupyter.toolbar.add_buttons_group([{
    'label': 'run qtconsole',
    'icon': 'fa-terminal', // select your icon from
    // http://fortawesome.io/icons/
    'callback': function() {Jupyter.notebook.kernel.execute('%qtconsole')}
    // add more button here if needed.
});
```

with a lot of icons you can select from.

15.7.3 Cell Metadata

The most requested feature is generally to be able to distinguish an individual cell in the notebook, or run a specific action with them. To do so, you can either use Jupyter.notebook.get_selected_cell(), or rely on CellToolbar. This allows you to register a set of actions and graphical elements that will be attached to individual cells.

Cell Toolbar

You can see some example of what can be done by toggling the Cell Toolbar selector in the toolbar on top of the notebook. It provides two default presets that are Default and slideshow. Default allows the user to edit the metadata attached to each cell manually.
First we define a function that takes at first parameter an element on the DOM in which to inject UI element. The second element is the cell this element was registered with. Then we will need to register that function and give it a name.

Register a callback

```javascript
In [ ]: %%javascript
    var CellToolbar = Jupyter.CellToolbar
    var toggle = function(div, cell) {
        var button_container = $(div)
        // let's create a button that shows the current value of the metadata
        var button = $('<button/>').addClass('btn btn-mini').text(String(cell.metadata.foo));
        // On click, change the metadata value and update the button label
        button.click(function() {
            var v = cell.metadata.foo;
            cell.metadata.foo = !v;
            button.text(String(!v));
        })
        // add the button to the DOM div.
        button_container.append(button);
    }
    // now we register the callback under the name foo to give the
    // user the ability to use it later
    CellToolbar.register_callback('tuto.foo', toggle);
```

Registering a preset

This function can now be part of many preset of the CellToolBar.

```javascript
In [ ]: %%javascript
    Jupyter.CellToolbar.register_preset('Tutorial 1', ['tuto.foo', 'default.rawedit'])
    Jupyter.CellToolbar.register_preset('Tutorial 2', ['slideshow.select', 'tuto.foo'])
```

You should now have access to two presets:

- Tutorial 1
- Tutorial 2

And check that the buttons you defined share state when you toggle preset. Also check that the metadata of the cell is modified when you click the button, and that when saved on reloaded the metadata is still available.

Exercise:

Try to wrap the all code in a file, put this file in `{jupyter_dir}/custom/<a-name>.js`, and add

```javascript
require(['custom/<a-name>']);
```

in custom.js to have this script automatically loaded in all your notebooks.

Require is provided by a javascript library that allow you to express dependency. For simple extension like the previous one we directly mute the global namespace, but for more complex extension you could pass a callback to require([...], <callback>) call, to allow the user to pass configuration information to your plugin.
In Python lang,

```javascript
require(['a/b', 'c/d'], function (e, f){
    e.something()
    f.something()
})
```

could be read as

```python
import a.b as e
import c.d as f

e.something()

f.something()
```

See for example @damianavila “ZenMode” plugin:

```javascript
require(['custom/zenmode/main'], function (zenmode){
    zenmode.background('images/back12.jpg');
})
```

For the quickest

Try to use the following to bind a dropdown list to `cell.metadata.difficulty.select`.

It should be able to take the 4 following values:

- <None>
- Easy
- Medium
- Hard

We will use it to customize the output of the converted notebook depending on the tag on each cell.

```
In [1]: # %load soln/celldiff.js
```

## 15.8 Importing Jupyter Notebooks as Modules

It is a common problem that people want to import code from Jupyter Notebooks. This is made difficult by the fact that Notebooks are not plain Python files, and thus cannot be imported by the regular Python machinery.

Fortunately, Python provides some fairly sophisticated hooks into the import machinery, so we can actually make Jupyter notebooks importable without much difficulty, and only using public APIs.

```
In [1]: import io, os, sys, types
In [1]: from IPython import get_ipython
    from nbformat import read
    from IPython.core.interactiveshell import InteractiveShell
```

Import hooks typically take the form of two objects:

1. a Module Loader, which takes a module name (e.g. 'IPython.display'), and returns a Module
2. a Module Finder, which figures out whether a module might exist, and tells Python what Loader to use
In [ ]: def find_notebook(fullname, path=None):
    """find a notebook, given its fully qualified name and an optional path
    This turns "foo.bar" into "foo/bar.ipynb"
    and tries turning "Foo_Bar" into "Foo Bar" if Foo_Bar
does not exist.
    """
    name = fullname.rsplit('.', 1)[-1]
    if not path:
        path = ['']
    for d in path:
        nb_path = os.path.join(d, name + "ipynb")
        if os.path.isfile(nb_path):
            return nb_path
    # let import Notebook_Name find "Notebook Name.ipynb"
    nb_path = nb_path.replace("_", " ")
    if os.path.isfile(nb_path):
        return nb_path

15.8.1 Notebook Loader

Here we have our Notebook Loader. It’s actually quite simple - once we figure out the filename of the module, all it
does is:

1. load the notebook document into memory
2. create an empty Module
3. execute every cell in the Module namespace

Since IPython cells can have extended syntax, the IPython transform is applied to turn each of these cells into their
pure-Python counterparts before executing them. If all of your notebook cells are pure-Python, this step is unnecessary.

In [ ]: class NotebookLoader(object):
    """Module Loader for Jupyter Notebooks"
    ""
    def __init__(self, path=None):
        self.shell = InteractiveShell.instance()
        self.path = path

    def load_module(self, fullname):
        """import a notebook as a module"
        path = find_notebook(fullname, self.path)
        print ("importing Jupyter notebook from %s" % path)

        # load the notebook object
        with io.open(path, 'r', encoding='utf-8') as f:
            nb = read(f, 4)

        # create the module and add it to sys.modules
        # if name in sys.modules:
        #     return sys.modules[name]
        mod = types.ModuleType(fullname)
        mod.__file__ = path
        mod.__loader__ = self
        mod.__dict__['get_ipython'] = get_ipython
        sys.modules[fullname] = mod
# extra work to ensure that magics that would affect the user_ns
# actually affect the notebook module's ns
save_user_ns = self.shell.user_ns
self.shell.user_ns = mod.__dict__

try:
    for cell in nb.cells:
        if cell.cell_type == 'code':
            # transform the input to executable Python
            code = self.shell.input_transformer_manager.transform_cell(cell.source)
            # run the code in the module
            exec(code, mod.__dict__)
finally:
    self.shell.user_ns = save_user_ns
return mod

## 15.8.2 The Module Finder

The finder is a simple object that tells you whether a name can be imported, and returns the appropriate loader. All
this one does is check, when you do:

```python
import mynotebook
```

it checks whether mynotebook.ipynb exists. If a notebook is found, then it returns a NotebookLoader.

Any extra logic is just for resolving paths within packages.

```python
In [ ]: class NotebookFinder(object):
    """Module finder that locates Jupyter Notebooks"""
    def __init__(self):
        self.loaders = {}

    def find_module(self, fullname, path=None):
        nb_path = find_notebook(fullname, path)
        if not nb_path:
            return
        key = path
        if path:
            # lists aren’t hashable
            key = os.path.sep.join(path)

        if key not in self.loaders:
            self.loaders[key] = NotebookLoader(path)
        return self.loaders[key]
```

## 15.8.3 Register the hook

Now we register the NotebookFinder with sys.meta_path

```python
In [ ]: sys.meta_path.append(NotebookFinder())
```

After this point, my notebooks should be importable.

Let’s look at what we have in the CWD:

```python
In [ ]: ls nbpackage
```

### 15.8. Importing Jupyter Notebooks as Modules 103
So I should be able to import nbpackage.mynotebook.

In [ ]: import nbpackage.mynotebook

Aside: displaying notebooks

Here is some simple code to display the contents of a notebook with syntax highlighting, etc.

In [ ]:

```python
from pygments import highlight
from pygments.lexers import PythonLexer
from pygments.formatters import HtmlFormatter
from IPython.display import display, HTML

formatter = HtmlFormatter()
lexer = PythonLexer()

# publish the CSS for pygments highlighting
display(HTML(''
    %s
'' % formatter.get_style_defs()))
```

In [ ]:

def show_notebook(fname):
    '''display a short summary of the cells of a notebook'''
    with io.open(fname, 'r', encoding='utf-8') as f:
        nb = read(f, 4)
        html = []
        for cell in nb.cells:
            html.append("<h4>%s</h4>" % cell.cell_type)
            if cell.cell_type == 'code':
                html.append(highlight(cell.source, lexer, formatter))
            else:
                html.append("<pre>%s</pre>" % cell.source)
        display(HTML(''\n'.join(html)))

show_notebook(os.path.join("nbpackage", "mynotebook.ipynb"))

So my notebook has some code cells, one of which contains some IPython syntax.

Let’s see what happens when we import it

In [ ]: from nbpackage import mynotebook

Hooray, it imported! Does it work?

In [ ]: mynotebook.foo()

Hooray again!

Even the function that contains IPython syntax works:

In [ ]: mynotebook.has_ip_syntax()

15.8.4 Notebooks in packages

We also have a notebook inside the nb package, so let’s make sure that works as well.

In [ ]: ls nbpackage/nbs
Note that the __init__.py is necessary for nb to be considered a package, just like usual.

```python
In [ ]: show_notebook(os.path.join("nbpackage", "nbs", "other.ipynb"))
```

```python
In [ ]: from nbpackage.nbs import other
    other.bar(5)
```

So now we have importable notebooks, from both the local directory and inside packages.

I can even put a notebook inside IPython, to further demonstrate that this is working properly:

```python
In [ ]: import shutil
    from IPython.paths import get_ipython_package_dir

    utils = os.path.join(get_ipython_package_dir(), 'utils')
    shutil.copy(os.path.join("nbpackage", "mynotebook.ipynb"),
                os.path.join(utils, "inside_ipython.ipynb"))
```

and import the notebook from IPython.utils

```python
In [ ]: from IPython.utils import inside_ipython
    inside_ipython.whatsmyname()
```

This approach can even import functions and classes that are defined in a notebook using the %%cython magic.

### 15.9 Connecting to an existing IPython kernel using the Qt Console

#### 15.9.1 The Frontend/Kernel Model

The traditional IPython (ipython) consists of a single process that combines a terminal based UI with the process that runs the users code.

While this traditional application still exists, the modern Jupyter consists of two processes:

- Kernel: this is the process that runs the users code.
- Frontend: this is the process that provides the user interface where the user types code and sees results.

Jupyter currently has 3 frontends:

- Terminal Console (jupyter console)
- Qt Console (jupyter qtconsole)
- Notebook (jupyter notebook)

The Kernel and Frontend communicate over a ZeroMQ/JSON based messaging protocol, which allows multiple Frontends (even of different types) to communicate with a single Kernel. This opens the door for all sorts of interesting things, such as connecting a Console or Qt Console to a Notebook’s Kernel. For example, you may want to connect a Qt console to your Notebook’s Kernel and use it as a help browser, calling ?? on objects in the Qt console (whose pager is more flexible than the one in the notebook).

This Notebook describes how you would connect another Frontend to an IPython Kernel that is associated with a Notebook. The commands currently given here are specific to the IPython kernel.

#### 15.9.2 Manual connection

To connect another Frontend to a Kernel manually, you first need to find out the connection information for the Kernel using the %connect_info magic:
In [1]: %connect_info
{
    "hb_port": 58928,
    "key": "83c5bc10-42515a2d3e6dad0c975f6de4",
    "stdin_port": 34423,
    "signature_scheme": "hmac-sha256",
    "kernel_name": "",
    "shell_port": 38241,
    "iopub_port": 42193,
    "transport": "tcp",
    "control_port": 51755,
    "ip": "127.0.0.1"
}

Paste the above JSON into a file, and connect with:
   $> jupyter <app> --existing <file>
or, if you are local, you can connect with just:
   $> jupyter <app> --existing /tmp/tmp45alz_em.json
or even just:
   $> jupyter <app> --existing
if this is the most recent Jupyter kernel you have started.

You can see that this magic displays everything you need to connect to this Notebook’s Kernel.

15.9.3 Automatic connection using a new Qt Console

You can also start a new Qt Console connected to your current Kernel by using the %qtconsole magic. This will detect the necessary connection information and start the Qt Console for you automatically.

In [2]: a = 10

In [3]: %qtconsole

The Markdown parser included in the Jupyter Notebook is MathJax-aware. This means that you can freely mix in mathematical expressions using the MathJax subset of Tex and LaTeX. Some examples from the MathJax site are reproduced below, as well as the Markdown+TeX source.

### 15.10 Motivating Examples

#### 15.10.1 The Lorenz Equations

**Source**

\begin{align}
\dot{x} & = \sigma(y-x) \\
\dot{y} & = \rho x - y - xz \\
\dot{z} & = -\beta z + xy
\end{align}

**Display**
15.10.2 The Cauchy-Schwarz Inequality

Source

\begin{equation*}
\left( \sum_{k=1}^n a_k b_k \right)^2 \leq \left( \sum_{k=1}^n a_k^2 \right) \left( \sum_{k=1}^n b_k^2 \right)
\end{equation*}

Display

\[(\sum_{k=1}^n a_k b_k)^2 \leq (\sum_{k=1}^n a_k^2)(\sum_{k=1}^n b_k^2)\]

15.10.3 A Cross Product Formula

Source

\begin{equation*}
\mathbf{V}_1 \times \mathbf{V}_2 = \begin{vmatrix}
\mathbf{i} & \mathbf{j} & \mathbf{k} \\
\frac{\partial X}{\partial u} & \frac{\partial Y}{\partial u} & 0 \\
\frac{\partial X}{\partial v} & \frac{\partial Y}{\partial v} & 0
\end{vmatrix}
\end{equation*}

Display

\[\mathbf{V}_1 \times \mathbf{V}_2 = \begin{vmatrix}
\mathbf{i} & \mathbf{j} & \mathbf{k} \\
\frac{\partial X}{\partial u} & \frac{\partial Y}{\partial u} & 0 \\
\frac{\partial X}{\partial v} & \frac{\partial Y}{\partial v} & 0
\end{vmatrix}\]

15.10.4 The probability of getting (k) heads when flipping (n) coins is

Source

\begin{equation*}
P(E) = \binom{n}{k} p^k (1-p)^{n-k}
\end{equation*}

Display

\[P(E) = \binom{n}{k} p^k (1-p)^{n-k}\]

15.10. Motivating Examples
15.10.5 An Identity of Ramanujan

Source

\begin{equation*}
\frac{1}{\sqrt{\phi \sqrt{5}}-\phi} e^{\frac{2\pi}{5}} = 1+\frac{e^{-2\pi}} {1+\frac{e^{-4\pi}} {1+\frac{e^{-6\pi}} {1+\frac{e^{-8\pi}} {1+\ldots} } } }
\end{equation*}

Display

\[
\frac{1}{\sqrt{\phi \sqrt{5}}-\phi} e^{\frac{2\pi}{5}} = 1+\frac{e^{-2\pi}} {1+\frac{e^{-4\pi}} {1+\frac{e^{-6\pi}} {1+\frac{e^{-8\pi}} {1+\ldots} } } }
\]

15.10.6 A Rogers-Ramanujan Identity

Source

\begin{equation*}
1 + \frac{q^2}{(1-q)} + \frac{q^6}{(1-q)(1-q^2)} + \cdots = \prod_{j=0}^{\infty} \frac{1}{(1-q^{5j+2})(1-q^{5j+3})}, \quad \text{for } |q|<1.
\end{equation*}

Display

\[
1 + \frac{q^2}{(1-q)} + \frac{q^6}{(1-q)(1-q^2)} + \cdots = \prod_{j=0}^{\infty} \frac{1}{(1-q^{5j+2})(1-q^{5j+3})}, \quad \text{for } |q|<1.
\]

15.10.7 Maxwell’s Equations

Source

\begin{align*}
\nabla \times \vec{B} - \frac{1}{c} \frac{\partial \vec{E}}{\partial t} &= \frac{4\pi}{c} \vec{j} \\
\nabla \cdot \vec{E} &= 4 \pi \rho \\
\nabla \times \vec{E} + \frac{1}{c} \frac{\partial \vec{B}}{\partial t} &= \vec{0} \\
\nabla \cdot \vec{B} &= 0
\end{align*}
15.10.8 Equation Numbering and References

Equation numbering and referencing will be available in a future version of the Jupyter notebook.

15.10.9 Inline Typesetting (Mixing Markdown and TeX)

While display equations look good for a page of samples, the ability to mix math and formatted text in a paragraph is also important.

Source

This expression $\sqrt{3x-1}+(1+x)^2$ is an example of a TeX inline equation in a Markdown-formatted sentence.

Display

This expression $\sqrt{3x-1}+(1+x)^2$ is an example of a TeX inline equation in a Markdown-formatted sentence.

15.10.10 Other Syntax

You will notice in other places on the web that $$ are needed explicitly to begin and end MathJax typesetting. This is not required if you will be using TeX environments, but the Jupyter notebook will accept this syntax on legacy notebooks.

15.10.11 Source

$$
\begin{array}{c}
y_1 \\
y_2 \\
z_{3,4}
\end{array}
$$

$$
\begin{array}{c}
y_1 \\
y_2 \\
y_3
\end{array}
$$

15.10. Motivating Examples
15.10.12 Display

\begin{equation}
\begin{align*}
x' &= x \sin \phi + z \cos \phi \\
z' &= -x \cos \phi + z \sin \phi
\end{align*}
\end{equation}

(15.1)

\begin{align*}
y_1 \\
y_2 t_i \\
z_{3,4} \\
y_1 \\
y_2 t_i \\
y_3
\end{align*}

\begin{align*}
x' &= x \sin \phi + z \cos \phi \\
z' &= -x \cos \phi + z \sin \phi
\end{align*}

\begin{align*}
x &= 4
\end{align*}
In [1]:
def foo():
    return "foo"

In [2]:
def has_ip_syntax():
    listing = !ls
    return listing

In [4]:
def whatsmyname():
    return __name__
This notebook just defines `bar`

```
In [2]: def bar(x):
    ...:     return "bar" * x
```
A summary of changes in the Jupyter notebook. For more detailed information, see [GitHub](https://github.com).

**Tip:** Use `pip install notebook --upgrade` or `conda upgrade notebook` to upgrade to the latest release.

We strongly recommend that you upgrade to version 9+ of pip before upgrading `notebook`.

**Tip:** Use `pip install pip --upgrade` to upgrade pip. Check pip version with `pip --version`.

### 18.1 5.2.1

- Fix invisible CodeMirror cursor at specific browser zoom levels (PR #2983).
- Fix nbconvert handler causing broken export to PDF (PR #2981).
- Fix the prompt_area argument of the output area constructor. (PR #2961).
- Handle a compound extension in new_untitled (PR #2949).
- Allow disabling offline message buffering (PR #2916).

Thanks to the following contributors:

- Steven Silvester ([blink1073](https://github.com/blink1073))
- Grant Nestor ([gnestor](https://github.com/gnestor))
- Jason Grout ([jasongrout](https://github.com/jasongrout))
- Min RK ([minrk](https://github.com/minrk))
- M Pacer ([mpacer](https://github.com/mpacer))

See the 5.2.1 milestone on GitHub for a complete list of pull requests involved in this release.
18.2 5.2.0

- Allow setting token via jupyter_token env (PR #2921).
- Fix some errors caused by raising 403 in get_current_user (PR #2919).
- Register contents_manager.files_handler_class directly (PR #2917).
- Update viewable_extensions (PR #2913).
- Show edit shortcuts modal after shortcuts modal is hidden (PR #2912).
- Improve edit/view behavior (PR #2911).
- The root directory of the notebook server should never be hidden (PR #2907).
- Fix notebook require config to match tools/build-main (PR #2888).
- Give page constructor default arguments (PR #2887).
- Fix codemirror.less to match codemirror’s expected padding layout (PR #2880).
- Add x-xsrftoken to access-control-allow-headers (PR #2876).
- Buffer messages when websocket connection is interrupted (PR #2871).
- Load locale dynamically only when not en-us (PR #2866).
- Changed key strength to 2048 bits (PR #2861).
- Resync jsversion with python version (PR #2860).
- Allow copy operation on modified, read-only notebook (PR #2854).
- Update error handling on apihandlers (PR #2853).
- Test python 3.6 on travis, drop 3.3 (PR #2852).
- Avoid base64-literals in image tests (PR #2851).
- Upgrade xterm.js to 2.9.2 (PR #2849).
- Changed all python variables named file to file_name to not override built_in file (PR #2830).
- Add more doc tests (PR #2823).
- Typos fix (PR #2815).
- Rename and update license [ci skip] (PR #2810).
- Travis builds doc (PR #2808).
- Pull request i18n (PR #2804).
- Factor out output_prompt_function, as is done with input prompt (PR #2774).
- Use rfc5987 encoding for filenames (PR #2767).
- Added path to the resources metadata, the same as in from_filename(…) in nbconvert.exporters.py (PR #2753).
- Make “extrakeys” consistent for notebook and editor (PR #2745).
- Bidi support (PR #2357).

Special thanks to samarsultan and the Arabic Competence and Globalization Center Team at IBM Egypt for adding RTL (right-to-left) support to the notebook!

See the 5.2 milestone on GitHub for a complete list of issues and pull requests involved in this release.
18.3  5.1.0

- Preliminary i18n implementation (PR #2140).
- Expose URL with auth token in notebook UI (PR #2666).
- Fix search background style (PR #2387).
- List running notebooks without requiring `--allow-root` (PR #2421).
- Allow session of type other than notebook (PR #2559).
- Fix search background style (PR #2387).
- Fix some Markdown styling issues (PR #2571), (PR #2691) and (PR #2534).
- Remove keymaps that conflict with non-English keyboards (PR #2535).
- Add session-specific favicons (notebook, terminal, file) (PR #2452).
- Add /api/shutdown handler (PR #2507).
- Include metadata when copying a cell (PR #2349).
- Stop notebook server from command line (PR #2388).
- Improve “View” and “Edit” file handling in dashboard (PR #2449) and (PR #2402).
- Provide a promise to replace use of the `app_initialized.NotebookApp` event (PR #2710).
- Fix disabled collapse/expand output button (PR #2681).
- Cull idle kernels using `--MappingKernelManager.cull_idle_timeout` (PR #2215).
- Allow read-only notebooks to be trusted (PR #2718).

See the 5.1 milestone on GitHub for a complete list of issues and pull requests involved in this release.

18.4  5.0.0

This is the first major release of the Jupyter Notebook since version 4.0 was created by the “Big Split” of IPython and Jupyter.

We encourage users to start trying JupyterLab in preparation for a future transition.

We have merged more than 300 pull requests since 4.0. Some of the major user-facing changes are described here.

18.4.1  File sorting in the dashboard

Files in the dashboard may now be sorted by last modified date or name (PR #943):
18.4.2 Cell tags

There is a new cell toolbar for adding cell tags (PR #2048):

```python
In [4]:
import numpy as np
print(np.__version__)

1.12.0

In [5]:
a = np.arange(15).reshape(3, 5)
a

Out[5]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [6]:
a.shape

Out[6]: (3, 5)
```

Cell tags are a lightweight way to customise the behaviour of tools working with notebooks; we’re working on building support for them into tools like nbconvert and nbval. To start using tags, select Tags in the View > Cell Toolbar menu in a notebook.

The UI for editing cell tags is basic for now; we hope to improve it in future releases.

18.4.3 Table style

The default styling for tables in the notebook has been updated (PR #1776).

Before:
18.4.4 Customise keyboard shortcuts

You can now edit keyboard shortcuts for *Command Mode* within the UI (PR #1347):
18.4.5 Other additions

- You can copy and paste cells between notebooks, using Ctrl-C and Ctrl-V (Cmd-C and Cmd-V on Mac).
- It’s easier to configure a password for the notebook with the new `jupyter notebook password` command (PR #2007).
- The file list can now be ordered by `last modified` or by `name` (PR #943).
- Markdown cells now support attachments. Simply drag and drop an image from your desktop to a markdown cell to add it. Unlike relative links that you enter manually, attachments are embedded in the notebook itself. An unreferenced attachment will be automatically scrubbed from the notebook on save (PR #621).
- Undoing cell deletion now supports undeleting multiple cells. Cells may not be in the same order as before their deletion, depending on the actions you did on the meantime, but this should help reduce the impact of accidentally deleting code.
- The file browser now has `Edit` and `View` buttons.
- The file browser now supports moving multiple files at once (PR #1088).
- The Notebook will refuse to run as root unless the `--allow-root` flag is given (PR #1115).
- Keyboard shortcuts are now declarative (PR #1234).
- Toggling line numbers can now affect all cells (PR #1312).
- Add more visible `Trusted` and `Untrusted` notifications (PR #1658).
- The favicon (browser shortcut icon) now changes to indicate when the kernel is busy (PR #1837).
• Header and toolbar visibility is now persisted in nbconfig and across sessions (PR #1769).
• Load server extensions with ConfigManager so that merge happens recursively, unlike normal config values, to make it load more consistently with frontend extensions (PR #2108).
• The notebook server now supports the bundler API from the jupyter_cms incubator project (PR #1579).
• The notebook server now provides information about kernel activity in its kernel resource API (PR #1827).

Remember that upgrading notebook only affects the user interface. Upgrading kernels and libraries may also provide new features, better stability and integration with the notebook interface.

18.5 4.4.0

• Allow override of output callbacks to redirect output messages. This is used to implement the ipywidgets Output widget, for example.
• Fix an async bug in message handling by allowing comm message handlers to return a promise which halts message processing until the promise resolves.

See the 4.4 milestone on GitHub for a complete list of issues and pull requests involved in this release.

18.6 4.3.2

4.3.2 is a patch release with a bug fix for CodeMirror and improved handling of the “editable” cell metadata field.
• Monkey-patch for CodeMirror that resolves #2037 without breaking #1967
• Read-only ("editable": false) cells can be executed but cannot be split, merged, or deleted

See the 4.3.2 milestone on GitHub for a complete list of issues and pull requests involved in this release.

18.7 4.3.1

4.3.1 is a patch release with a security patch, a couple bug fixes, and improvements to the newly-released token authentication.

Security fix:
• CVE-2016-9971. Fix CSRF vulnerability, where malicious forms could create untitled files and start kernels (no remote execution or modification of existing files) for users of certain browsers (Firefox, Internet Explorer / Edge). All previous notebook releases are affected.

Bug fixes:
• Fix carriage return handling
• Make the font size more robust against fickle browsers
• Ignore resize events that bubbled up and didn’t come from window
• Add Authorization to allowed CORS headers
• Downgrade CodeMirror to 5.16 while we figure out issues in Safari

Other improvements:
• Better docs for token-based authentication
• Further highlight token info in log output when autogenerated

See the 4.3.1 milestone on GitHub for a complete list of issues and pull requests involved in this release.

18.8 4.3.0

4.3 is a minor release with many bug fixes and improvements. The biggest user-facing change is the addition of token authentication, which is enabled by default. A token is generated and used when your browser is opened automatically, so you shouldn’t have to enter anything in the default circumstances. If you see a login page (e.g. by switching browsers, or launching on a new port with --no-browser), you get a login URL with the token from the command `jupyter notebook list`, which you can paste into your browser.

Highlights:

• API for creating mime-type based renderer extensions using `OutputArea.register_mime_type` and `Notebook.render_cell_output` methods. See `mimerender-cookiecutter` for reference implementations and cookiecutter.

• Enable token authentication by default. See `Security in the Jupyter notebook server` for more details.

• Update security docs to reflect new signature system

• Switched from term.js to xterm.js

Bug fixes:

• Ensure variable is set if exc_info is falsey

• Catch and log handler exceptions in `events.trigger`

• Add debug log for static file paths

• Don’t check origin on token-authenticated requests

• Remove leftover print statement

• Fix highlighting of Python code blocks

• `json_errors` should be outermost decorator on API handlers

• Fix remove old nbserver info files

• Fix notebook mime type on download links

• Fix carriage symbol bahvior

• Fix terminal styles

• Update dead links in docs

• If kernel is broken, start a new session

• Include cross-origin check when allowing login URL redirects

Other improvements:

• Allow JSON output data with mime type `application/*+json`

• Allow kernelspecs to have spaces in them for backward compat

• Allow websocket connections from scripts

• Allow `None` for `post_save_hook`

• Upgrade CodeMirror to 5.21
• Upgrade xterm to 2.1.0
• Docs for using comms
• Set dirty flag when output arrives
• Set ws-url data attribute when accessing a notebook terminal
• Add base aliases for nbextensions
• Include @ operator in CodeMirror IPython mode
• Extend mathjax_url docstring
• Load nbextension in predictable order
• Improve the error messages for nbextensions
• Include cross-origin check when allowing login URL redirects

See the 4.3 milestone on GitHub for a complete list of issues and pull requests involved in this release.

18.9 4.2.3

4.2.3 is a small bugfix release on 4.2.

Highlights:
• Fix regression in 4.2.2 that delayed loading custom.js until after notebook_loaded and app_initialized events have fired.
• Fix some outdated docs and links.

See also:
4.2.3 on GitHub.

18.10 4.2.2

4.2.2 is a small bugfix release on 4.2, with an important security fix. All users are strongly encouraged to upgrade to 4.2.2.

Highlights:
• Security fix: CVE-2016-6524, where untrusted latex output could be added to the page in a way that could execute javascript.
• Fix missing POST in OPTIONS responses.
• Fix for downloading non-ascii filenames.
• Avoid clobbering ssl_options, so that users can specify more detailed SSL configuration.
• Fix inverted load order in nbconfig, so user config has highest priority.
• Improved error messages here and there.

See also:
4.2.2 on GitHub.
18.11 4.2.1

4.2.1 is a small bugfix release on 4.2. Highlights:

• Compatibility fixes for some versions of ipywidgets
• Fix for ignored CSS on Windows
• Fix specifying destination when installing nbextensions

See also:

4.2.1 on GitHub.

18.12 4.2.0

Release 4.2 adds a new API for enabling and installing extensions. Extensions can now be enabled at the system-level, rather than just per-user. An API is defined for installing directly from a Python package, as well.

See also:

Distributing Jupyter Extensions as Python Packages

Highlighted changes:

• Upgrade MathJax to 2.6 to fix vertical-bar appearing on some equations.
• Restore ability for notebook directory to be root (4.1 regression)
• Large outputs are now throttled, reducing the ability of output floods to kill the browser.
• Fix the notebook ignoring cell executions while a kernel is starting by queueing the messages.
• Fix handling of url prefixes (e.g. JupyterHub) in terminal and edit pages.
• Support nested SVGs in output.

And various other fixes and improvements.

18.13 4.1.0

Bug fixes:

• Properly reap zombie subprocesses
• Fix cross-origin problems
• Fix double-escaping of the base URL prefix
• Handle invalid unicode filenames more gracefully
• Fix ANSI color-processing
• Send keepalive messages for web terminals
• Fix bugs in the notebook tour

UI changes:

• Moved the cell toolbar selector into the View menu. Added a button that triggers a “hint” animation to the main toolbar so users can find the new location. (Click here to see a screencast)
• Added Restart & Run All to the Kernel menu. Users can also bind it to a keyboard shortcut on action `restart-kernel-and-run-all-cells`.

• Added multiple-cell selection. Users press `Shift-Up/Down` or `Shift-K/J` to extend selection in command mode. Various actions such as cut/copy/paste, execute, and cell type conversions apply to all selected cells.

• Added a command palette for executing Jupyter actions by name. Users press `Cmd/Ctrl-Shift-P` or click the new command palette icon on the toolbar.

• Added a Find and Replace dialog to the Edit menu. Users can also press `F` in command mode to show the dialog.
Other improvements:

- Custom KernelManager methods can be Tornado coroutines, allowing async operations.
- Make clearing output optional when rewriting input with `set_next_input(replace=True)`.
- Added support for TLS client authentication via --NotebookApp.client-ca.
- Added tags to jupyter/notebook releases on DockerHub. latest continues to track the master branch.

See the 4.1 milestone on GitHub for a complete list of issues and pull requests handled.

### 18.14 4.0.x

### 18.14.1 4.0.6

- fix installation of mathjax support files
- fix some double-escape regressions in 4.0.5
- fix a couple of cases where errors could prevent opening a notebook

### 18.14.2 4.0.5

Security fixes for maliciously crafted files.

- CVE-2015-6938: malicious filenames
• CVE-2015-7337: malicious binary files in text editor.
Thanks to Jonathan Kamens at Quantopian and Juan Broullón for the reports.

18.14.3 4.0.4

• Fix inclusion of mathjax-safe extension

18.14.4 4.0.2

• Fix launching the notebook on Windows
• Fix the path searched for frontend config

18.14.5 4.0.0

First release of the notebook as a standalone package.