Warning

This documentation is for an old version of IPython. You can find docs for newer versions here.

The IPython Notebook

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 IPython 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 documentation for directions on how to install the notebook and its dependencies.

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.

Table Of Contents

The IPython Notebook
- Introduction
  - Main features of the web application
  - Notebook documents
- Starting the notebook server
  - Creating a new notebook document
  - Opening notebooks
- Notebook user interface
- Structure of a notebook document
  - Code cells
  - Markdown cells
  - Raw cells
  - Heading cells
- Basic workflow
  - Keyboard shortcuts
- Plotting
- Configuring the IPython Notebook
- Installing new kernels
- Signing Notebooks
- Importing `.py` files

Previous topic

The IPython notebook

Next topic

The Jupyter Notebook Format

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

Notebook documents

Notebook documents contains the inputs and outputs of a 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 new nbconvert command.

Furthermore, any `.ipynb` notebook document available from a public URL can be shared via the IPython 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 IPython 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

Starting the notebook server

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

```
ipython 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 IPython 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 `ipython 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.

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.

Opening notebooks

An open notebook has **exactly one** interactive session connected to an IPython 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 underlying IPython kernel. The notebook server always prints to the terminal the full details of how to connect to each kernel, with messages such as the following:

```
[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. You can also request this connection data by running the `%connect_info` magic. This will print the same ID information as well as the content of the JSON data structure it contains.

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:

```
$ ipython qtconsole --existing 87f7d2c0
```

Without an ID, `--existing` will connect to the most recently started kernel. This can also be done by running the `%qtconsole` magic in the notebook.

See also

Decoupled two-process model

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

Structure of a notebook document

The notebook consists of a sequence of cells. A cell is a multi-line text input field, and its contents can be executed by using `Shift-Enter`, or by clicking either the “Play” button the toolbar, or `Cell | Run` in the menu bar. The execution behavior of a cell is determined the cell’s type. There are four types of cells: **code cells**, **markdown cells**, **raw cells** and **heading cells**. Every cell starts off being a **code cell**, but its type can be changed by using a dropdown 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.

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

Basic Output example notebook

Rich Display System example notebook

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.

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

Markdown Cells example notebook

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.

Heading cells

You can provide a conceptual structure for your computational document as a whole using different levels of headings; there are 6 levels available, from level 1 (top level) down to level 6 (paragraph). These can be used later for constructing tables of contents, etc. As with Markdown cells, a heading cell is replaced by a rich text rendering of the heading when the cell is executed.

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 IPython Notebook example notebook

Basic Output example notebook

a warning about doing "roundtrip" conversions.

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.

Plotting

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

To set this up, before any plotting is performed you must execute the `%matplotlib` magic command. This performs the necessary behind-the-scenes setup for IPython to work correctly hand in hand with `matplotlib`; it does *not*, however, actually execute any Python `import` commands, that is, no names are added to the namespace.

If the `%matplotlib` magic is called without an argument, the output of a plotting command is displayed using the default `matplotlib` backend in a separate window. Alternatively, the backend can be explicitly requested using, for example:

```
%matplotlib gtk
```

A particularly interesting backend, provided by IPython, is the `inline` backend. This is available only for the IPython Notebook and the IPython QtConsole. It can be invoked as follows:

```
%matplotlib inline
```

With this backend, the output of plotting commands is displayed *inline* within the notebook, directly below the code cell that produced it. The resulting plots will then also be stored in the notebook document.

See also

Plotting with Matplotlib example notebook

Configuring the IPython Notebook

The notebook server can be run with a variety of command line arguments. To see a list of available options enter:

```
$ ipython notebook --help
```

Defaults for these options can also be set by creating a file named `ipython_notebook_config.py` in your IPython *profile folder*. The profile folder is a subfolder of your IPython directory; to find out where it is located, run:

```
$ ipython locate
```

To create a new set of default configuration files, with lots of information on available options, use:

```
$ ipython profile create
```

See also

Overview of the IPython configuration system, in particular Profiles.

Securing a notebook server

Running a public notebook server

Installing new kernels

Running the notebook makes the current python installation available as a kernel. Other python installations (different python versions, virtualenv or conda environments) can be installed as kernels by following these steps:

- make sure that the desired python installation is active (e.g. activate the environment) and ipython is installed
- run once `ipython kernelspec install-self --user` (or `ipython2 ...` or `ipython3 ...` if you want to install specific python versions)

The last command installs a kernel spec file for the current python installation in `~/.ipython/kernels/`. Kernel spec files are JSON files, which can be viewed and changed with a normal text editor.

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.

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:

```
$ ipython 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:

```
$ ipython trust --reset
```

Importing `.py` files

`.py` files will be imported as a notebook with the same basename, but an `.ipynb` extension, located in the notebook directory. The notebook created will have just one cell, which will contain all the code in the `.py` file. You can later manually partition this into individual cells using the `Edit | Split Cell` menu option, or the `Ctrl-m -` keyboard shortcut.

Note that `.py` scripts obtained from a notebook document using *nbconvert* maintain the structure of the notebook in comments. Reimporting such a script back into a notebook will preserve this structure.

Warning

While in simple cases you can "roundtrip" a notebook to Python, edit the Python file, and then import it back without loss of main content, this is in general *not guaranteed to work*. First, there is extra metadata saved in the notebook that may not be saved to the `.py` format. And as the notebook format evolves in complexity, there will be attributes of the notebook that will not survive a roundtrip through the Python form. You should think of the Python format as a way to output a script version of a notebook and the import capabilities as a way to load existing code to get a notebook started. But the Python version is *not* an alternate notebook format.

See also

The Jupyter Notebook Format