# Getting Started in IPython and Jupyter

In writing Python code for data science, I generally go between three modes of working: I use the IPython shell for trying out short sequences of commands, the Jupyter Notebook for longer interactive analysis and for sharing content with others, and interactive development environments (IDEs) like Emacs or VSCode for creating reusable Python packages. This chapter focuses on the first two modes: the IPython shell and the Jupyter Notebook. Use of an IDE for software development is an important third tool in the data scientist's repertoire, but we will not directly address that here.

## Launching the IPython Shell

The text in this part, like most of this book, is not designed to be absorbed passively. I recommend that as you read through it, you follow along and experiment with the tools and syntax we cover: the muscle memory you build through doing this will be far more useful than the simple act of reading about it. Start by launching the IPython interpreter by typing **`ipython`** on the command line; alternatively, if you've installed a distribution like Anaconda or EPD, there may be a launcher specific to your system (we'll discuss this more fully in Help and Documentation in IPython (01.01-Help-And-Documentation.ipynb)).

Once you do this, you should see a prompt like the following:

```
Python 3.9.2 (v3.9.2:1a79785e3e, Feb 19 2021, 09:06:10)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.21.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

With that, you're ready to follow along.

## Launching the Jupyter Notebook

The Jupyter Notebook is a browser-based graphical interface to the IPython shell, and builds on it a rich set of dynamic display capabilities. As well as executing Python/IPython statements, notebooks allow the user to include formatted text, static and dynamic visualizations, mathematical equations, JavaScript widgets, and much more. Furthermore, these documents can be saved in a way that lets other people open them and execute the code on their own systems.

Though you'll view and edit Jupyter notebooks through your web browser window, they must connect to a running Python process in order to execute code. You can start this process (known as a "kernel") by running the following command in your system shell:

```
$ jupyter lab
```

This command will launch a local web server that will be visible to your browser. It immediately spits out a log showing what it is doing; that log will look something like this:

```
$ jupyter lab
[ServerApp] Serving notebooks from local directory: /Users/jakevdp/Py
thonDataScienceHandbook
[ServerApp] Jupyter Server 1.4.1 is running at:
[ServerApp] http://localhost:8888/lab?token=dd852649
[ServerApp] Use Control-C to stop this server and shut down all kerne
ls (twice to skip confirmation).
```

Upon issuing the command, your default browser should automatically open and navigate to the listed local URL; the exact address will depend on your system. If the browser does not open automatically, you can open a window and manually open this address.

# Help and Documentation in IPython

If you read no other section in this chapter, read this one: I find the tools discussed here to be the most transformative contributions of IPython to my daily workflow.

When a technologically minded person is asked to help a friend, family member, or colleague with a computer problem, most of the time it's less a matter of knowing the answer than of knowing how to quickly find an unknown answer. In data science it's the same: searchable web resources such as online documentation, mailing list threads, and Stack Overflow answers contain a wealth of information, even (especially?) about topics you've found yourself searching on before. Being an effective practitioner of data science is less about memorizing the tool or command you should use for every possible situation, and more about learning to effectively find the information you don't know, whether through a web search engine or another means.

One of the most useful functions of IPython/Jupyter is to shorten the gap between the user and the type of documentation and search that will help them do their work effectively. While web searches still play a role in answering complicated questions, an amazing amount of information can be found through IPython alone. Some examples of the questions IPython can help answer in a few keystrokes include:

- How do I call this function? What arguments and options does it have?
- What does the source code of this Python object look like?
- What is in this package I imported?
- What attributes or methods does this object have?

Here we'll discuss the tools provided in the IPython shell and Jupyter Notebook to quickly access this information, namely the `?` character to explore documentation, the `??` characters to explore source code, and the Tab key for autocompletion.

## Accessing Documentation with ?

The Python language and its data science ecosystem are built with the user in mind, and one big part of that is access to documentation. Every Python object contains a reference to a string, known as a *docstring*, which in most cases will contain a concise summary of the object

and how to use it. Python has a built-in `help` function that can access this information and prints the results. For example, to see the documentation of the built-in `len` function, you can do the following:

```
In [1]: help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

Depending on your interpreter, this information may be displayed as inline text or in a separate pop-up window.

Because finding help on an object is so common and useful, IPython and Jupyter introduce the `?` character as a shorthand for accessing this documentation and other relevant information:

```
In [2]: len?
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

This notation works for just about anything, including object methods:

```
In [3]: L = [1, 2, 3]
In [4]: L.insert?
Signature: L.insert(index, object, /)
Docstring: Insert object before index.
Type:      builtin_function_or_method
```

or even objects themselves, with the documentation from their type:

```
In [5]: L?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.
```

Importantly, this will even work for functions or other objects you create yourself! Here we'll define a small function with a docstring:

```
In [6]: def square(a):
  ....:     """Return the square of a."""
  ....:     return a ** 2
  ....:
```

Note that to create a docstring for our function, we simply placed a string literal in the first line. Because docstrings are usually multiple lines, by convention we used Python's triple-quote notation for multiline strings.

Now we'll use the `?` to find this docstring:

```
In [7]: square?
Signature: square(a)
Docstring: Return the square of a.
File:      <ipython-input-6>
Type:      function
```

This quick access to documentation via docstrings is one reason you should get in the habit of always adding such inline documentation to the code you write!

## Accessing Source Code with ??

Because the Python language is so easily readable, another level of insight can usually be gained by reading the source code of the object you're curious about. IPython and Jupyter provide a shortcut to the source code with the double question mark ( `??` ):

```
In [8]: square??
Signature: square(a)
Source:
def square(a):
    """Return the square of a."""
    return a ** 2
File:      <ipython-input-6>
Type:      function
```

For simple functions like this, the double question mark can give quick insight into the under-the-hood details.

If you play with this much, you'll notice that sometimes the `??` suffix doesn't display any source code: this is generally because the object in question is not implemented in Python, but in C or some other compiled extension language. If this is the case, the `??` suffix gives the same output as the `?` suffix. You'll find this particularly with many of Python's built-in objects and types, including the `len` function from earlier:

```
In [9]: len??
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

Using `?` and/or `??` is a powerful and quick way of finding information about what any Python function or module does.

## Exploring Modules with Tab Completion

Another useful interface is the use of the Tab key for autocompletion and exploration of the contents of objects, modules, and namespaces. In the examples that follow, I'll use `<TAB>` to indicate when the Tab key should be pressed.

### Tab completion of object contents

Every Python object has various attributes and methods associated with it. Like the `help` function mentioned earlier, Python has a built-in `dir` function that returns a list of these, but the tab-completion interface is much easier to use in practice. To see a list of all available attributes of an object, you can type the name of the object followed by a period (" `.` ") character and the Tab key:

```
In [10]: L.<TAB>
            append() count    insert   reverse
            clear    extend   pop      sort
            copy     index    remove
```

To narrow down the list, you can type the first character or several characters of the name, and the Tab key will find the matching attributes and methods:

```
In [10]: L.c<TAB>
             clear() count()
             copy()

In [10]: L.co<TAB>
              copy()  count()
```

If there is only a single option, pressing the Tab key will complete the line for you. For example, the following will instantly be replaced with `L.count`:

```
In [10]: L.cou<TAB>
```

Though Python has no strictly enforced distinction between public/external attributes and private/internal attributes, by convention a preceding underscore is used to denote the latter. For clarity, these private methods and special methods are omitted from the list by default, but it's possible to list them by explicitly typing the underscore:

```
In [10]: L._<TAB>
           __add__             __delattr__     __eq__
           __class__           __delitem__     __format__()
           __class_getitem__() __dir__()       __ge__            >
           __contains__        __doc__         __getattribute__
```

For brevity, I've only shown the first few columns of the output. Most of these are Python's special double-underscore methods (often nicknamed "dunder" methods).

### Tab completion when importing

Tab completion is also useful when importing objects from packages. Here we'll use it to find all possible imports in the `itertools` package that start with `co`:

```
In [10]: from itertools import co<TAB>
         combinations()                  compress()
         combinations_with_replacement() count()
```

Similarly, you can use tab-completion to see which imports are available on your system (this will change depending on which third-party scripts and modules are visible to your Python session):

```
In [10]: import <TAB>
             abc                 anyio
             activate_this       appdirs
             aifc                appnope        >
             antigravity         argon2

In [10]: import h<TAB>
             hashlib html
             heapq   http
             hmac
```

**Beyond tab completion: Wildcard matching**

Tab completion is useful if you know the first few characters of the name of the object or attribute you're looking for, but is little help if you'd like to match characters in the middle or at the end of the name. For this use case, IPython and Jupyter provide a means of wildcard matching for names using the `*` character.

For example, we can use this to list every object in the namespace whose name ends with `Warning`:

```
In [10]: *Warning?
BytesWarning                  RuntimeWarning
DeprecationWarning            SyntaxWarning
FutureWarning                 UnicodeWarning
ImportWarning                 UserWarning
PendingDeprecationWarning     Warning
ResourceWarning
```

Notice that the `*` character matches any string, including the empty string.

Similarly, suppose we are looking for a string method that contains the word `find` somewhere in its name. We can search for it this way:

```
In [11]: str.*find*?
str.find
str.rfind
```

I find this type of flexible wildcard search can be useful for finding a particular command when getting to know a new package or reacquainting myself with a familiar one.