# 5 IPython and IPython Notebook

The IPython shell and the related interactive, browser-based IPython Notebook provide a powerful interface to the Python language. IPython has several advantages over the native Python shell, including easy interaction with the operating system, introspection and tab completion. IPython Notebook increasingly is being adopted by scientists to share their data and the code they write to analyze it in a standardized manner that aids reproducibility and visualization.

## 5.1 IPython

### 5.1.1 Installing IPython

Comprehensive details on installing IPython are available at the IPython website: see http://ipython.org/install.html, but a summary is provided here.

IPython is included in the Continuum Anaconda and Enthought Canopy Python distributions. To update to the current version within Anaconda, use the `conda` package manager:

```
conda update conda
conda update ipython
```

With Canopy, use

```
enpkg ipython
```

If you are not using these distributions but already have Python installed, there are several alternative options. If you have the `pip` package manager:

```
pip install ipython
pip install "ipython[notebook]"
```

It is also possible to manually download the latest IPython version from the github repository at https://github.com/ipython/ipython/releases and compile and install from its top-level source directory with

```
python setup.py install
```

### 5.1.2 Using the IPython shell

To start an interactive IPython shell session from the command line, simply type `ipython`. You should be greeted with a message similar to this one:

```
Python 3.3.5 |Anaconda 2.0.1 (x86_64)| (default, Mar 10 2014, 11:22:25)
Type "copyright", "credits" or "license" for more information.

IPython 2.1.0 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

(The precise details of this message will depend on the setup of your system.) The prompt `In [1]:` is where you type your Python statements and replaces the native Python `>>>` shell prompt. The counter in square brackets increments with each Python statement or code block. For example,

```
In [1]: 4+5
Out[1]: 9
In [2]: print(1)
1
In [3]: for i in range(4):
   ...:     print(i, end='')
   ...:
0123
In [4]:
```

To exit the IPython shell, type `quit` or `exit`. Unlike with the native Python shell, no parentheses are required.[1]

#### Help commands

As listed in the welcome message, there are various helpful commands to obtain information about using IPython:

- Typing a single '`?`' outputs an overview of the usage of IPython's main features (page down with the space bar or `f`; page back up with `b`; exit the help page with `q`).
- `%quickref` provides a brief reference summary of each of the main IPython commands and "magics" (see Section 5.1.3).
- `help()` or `help(`*`object`*`)` invokes Python's own help system (interactively or for *`object`* if specified).
- Typing one question mark after an object name provides information about that object: see below.

[1] Some find this alone a good reason to use IPython.

Possibly the most frequently used help functionality provided by IPython is the *introspection* provided by the `object?` syntax. For example,

```
In [4]: a = [5, 6]
In [5]: a?
Type:        list
String form: [5, 6]
Length:      2
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

Here, the command `a?` gives details about the object `a`: its string representation (which would be produced by, for example, `print(a)`), its length (equivalent to `len(a)` and the docstring associated with the class of which it is an instance: since `a` is a list, this provides brief details of how to instantiate a `list` object.[2]

The `?` syntax is particularly useful as a reminder of the arguments that a function or method takes. For example,

```
In [6]: import pylab
In [7]: pylab.linspace?

String form: <function linspace at 0x10432d560>
File:        /Users/christian/anaconda/envs/py33/lib/python3.3/site-packages/numpy/
             core/function_base.py
Definition:  pylab.linspace(start, stop, num=50, endpoint=True, retstep=False)
Docstring:
Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the
interval [`start`, `stop` ].

The endpoint of the interval can optionally be excluded.

Parameters
----------
start : scalar
    The starting value of the sequence.
stop : scalar
    The end value of the sequence, unless `endpoint` is set to False.
    In that case, the sequence consists of all but the last of ``num + 1``
    evenly spaced samples, so that `stop` is excluded. Note that the step
    size changes when `endpoint` is False.
num : int, optional
    Number of samples to generate. Default is 50.
endpoint : bool, optional
    If True, `stop` is the last sample. Otherwise, it is not included.
    Default is True.
retstep : bool, optional
    If True, return (`samples`, `step`), where `step` is the spacing
    between samples.
```

[2] This is what is meant by introspection: Python is able to inspect its own objects and provide information about them.

```
Returns
-------
...

See Also
--------
...

Examples
--------
...
```

For some objects, the syntax `object??` returns more advanced information such as the location and details of its source code.

### Tab completion

Just as with many command line shells, IPython supports tab completion: start typing the name of an object or keyword, press the <TAB> key, and it will autocomplete it for you or provide a list of options if more than one possibility exists. For example,

```
In [8]: w<TAB>
%%writefile  %who         %who_ls      %whos        while        with

In [8]: w
```

If you resume typing until the word becomes unambiguous (e.g., add the letters `hi`) and then press <TAB> again: it will be autocompleted to `while`. The options with percent signs in front of them are "magic functions," described in Section 5.1.3.

### History

You may already have used the native Python shell's command history functionality (pressing the up and down arrows through previous statements typed during your current session). IPython stores both the commands you enter and the output they produce in the special variables `In` and `Out` (these are, in fact, a list and a dictionary respectively, and correspond to the prompts at the beginning of each input and output). For example,

```
  In [9]: d = {'C': 'Cador', 'G': 'Galahad', 'T': 'Tristan', 'A': 'Arthur'}
  In [10]: for a in 'ACGT':
     ....:     print(d[a])
     ....:
  Arthur
  Cador
  Galahad
  Tristan
  In [11]: d = {'C': 'Cytosine', 'G': 'Guanine', 'T': 'Thymine', 'A': 'Adenine'}
❶ In [12]: In[10]
  Out[12]: "for a in 'ACGT':\n    print(d[a])\n    "
❷ In [13]: exec(In[10])
  Adenine
  Cytosine
  Guanine
  Thymine
```

❶ Note that `In[10]` simply holds the string version of the Python statement (here a `for` loop) that was entered at index 10.
❷ To actually execute the statement (with the *current* dictionary `d`), we must send it to Python's `exec` built-in (see also the `%rerun` magic, Section 5.1.3).

There are a couple of further shortcuts: the alias `_iN` is the same as `In[N]`, `_N` is the same as `Out[N]`, and the two most recent outputs are returned by the variables `_` and `__` respectively.

To view the contents of the history, use the `%history` or `%hist` magic function. By default only the entered statements are output; it is often more useful to output the line numbers as well, which is acheived using the `-n` option:

```
In [14]: %history -n
   1: 4+5
   2: print(1)
   3:
for i in range(4):
    print(i)
   4: a = [5, 6]
   5: a?
   6: import pylab
   7: pylab.linspace?
   8: d = {'C': 'Cador', 'G': 'Galahad', 'T': 'Tristan', 'A': 'Arthur'}
  10:
for a in 'ACGT':
    print(d[a])
  11: d = {'C': 'Cytosine', 'G': 'Guanine', 'T': 'Thymine', 'A': 'Adenine'}
  12: In[10]
  13: exec(In[10])
  14: %history -n
```

To output a specific line or range of lines, refer to them by number and/or number range when calling `%history`:

```
In [15]: %history 4
a = [5, 6]

In [16]: %history -n 2-5
   2: print(1)
   3:
for i in range(4):
    print(i)
   4: a = [5, 6]
   5: a?

In [17]: %history -n 1-3 7 12-14
   1: 4+5
   2: print(1)
   3:
for i in range(4):
    print(i)
   7: pylab.linspace?
  12: In[10]
  13: exec(In[10])
  14: %history -n
```

This syntax is also used by several other IPython magic functions (see the following section). The `%history` function can also take an additional option: `-o` displays the output as well as the input.

Pressing `CTRL-R` brings up a prompt, the somewhat cryptic `(reverse-i-search)`':`, from which you can search within your command history.[3]

### Interacting with the operating system

IPython makes it easy to execute operating system commands from within your shell session: any statement preceded by an exclamation mark, `!`, is sent to the operating system command line (the "system shell") instead of being executed as a Python statement. For example, you can delete files, list directory contents and even execute other programs and scripts:

```
In [11]: !pwd               # return the current working directory
/Users/christian/research
In [12]: !ls                # list the files in this directory
Meetings          Papers            code              books
databases         temp-file
In [13]: !rm temp-file      # delete temp-file

In [14]: !ls
Meetings          Papers            code              books
databases
```

Note that, for technical reasons,[4] the `cd` (Unix-like systems) and `chdir` (Windows) commands must be executed as IPython magic functions:

```
In [15]: %cd /       # Change into root directory
In [16]: !ls
Applications      Volumes           usr               Library
bin               net               Network           cores
opt               www               System            dev
private           sbin              Users             home
In [17]: %cd ~/temp   # Change directory to temp within user's home directory
In [18]: !ls
output.txt        test.py           readme.txt        utils
zigzag.py
```

If you use Windows and want to include a drive letter (such as `C:`) in the directory path you should enclose the path in quotes: `%cd 'C:\My Documents'`.

Help, via `!`*`command`*`?`, and tab completion, as described in Section 5.1.2, work within operating system commands.

You can pass the values of Python variables to operating system commands by prefixing the variable name with a dollar sign, `$`:

```
In [19]: python_script = 'zigzag.py'
In [20]: !ls $python_script
```

[3] This functionality may be familiar to users of the bash shell.

[4] System commands executed via the `!`*`command`* method spawn their own shell, which is discarded immediately afterward; changing a directory occurs only in this spawned shell and is not reflected in the one running IPython.

```
zigzag.py
In [21]: text_files = '*.txt'
```
❶
```
In [22]: text_file_list = !ls $text_files
In [23]: text_file_list
output.txt   readme.txt
In [24]: readme_file = text_file_list[1]
In [25]: !cat $readme_file
This is the file readme.txt
Each line of the file appears as an item
in a list when returned from !cat readme.txt
```
❷
```
In [26]: readme_lines = !cat $readme_file

In [27]: readme_lines
Out[28]:
['This is the file readme.txt',
 'Each line of the file appears as an item',
 'in a list when returned from !cat readme.txt']
```

❶ Note that the output of a system command can be assigned to a Python variable, here a list of the `.txt` files in the current directory.
❷ The `cat` system command returns the contents of the text file; IPython splits this output on the newline character and assigns the resulting list to `readme_lines`. See also Section 5.1.3

### 5.1.3 IPython magic functions

IPython provides many “magic” functions (or simply *magics*, those commands prefixed with `%`) to speed up coding and experimenting within the IPython shell. Some of the more useful ones are described in this section; for more advanced information the reader is referred to the IPython documentation.[5] IPython makes a distinction between *line magics*: those whose arguments are given on a single line, and *cell magics* (prefixed by two percent signs, `%%`): those which act on a series of Python commands. An example is given in Section 5.1.3 where we describe the `%%timeit` cell magic.

A list of currently available magic functions can be obtained by typing `%lsmagic`.

The magic function `%automagic` toggles the “automagic” setting: its default is `ON` meaning that typing the name of a magic function without the `%` will also execute that function, unless you have bound the name as a Python identifier (variable name) to some object. The same principle applies to system commands:

```
In [x]: ls
output.txt         test.py            readme.txt         utils
zigzag.py
In [x]: ls = 0
In [x]: ls          # Now ls is an integer; !ls will still work
Out[x]: 0
```

Table 5.1 summarizes some useful IPython magics; the following subsections explain more fully the less straightforward ones.

[5] http://ipython.org/documentation.html.

**Table 5.1** Useful IPython line magics

| Magic | Description |
|---|---|
| `%alias` | Create an alias to a system command. |
| `%alias_magic` | Create an alias to an existing IPython magic. |
| `%bookmark` | Interact with IPython's directory bookmarking system. |
| `%cd` | Change the current working directory. |
| `%dhist` | Output a list of visited directories. |
| `%edit` | Create or edit Python code within a text editor and then execute it. |
| `%env` | List the system environment variables, such as `$HOME`. |
| `%history` | List the input history for this IPython session. |
| `%load` | Read in code from a provided file and make it available for editing. |
| `%macro` | Define a named macro from previous input for future reexecution. |
| `%paste` | Paste input from the clipboard: use this in preference to, for example, `CTRL-V`, to handle code indenting properly. |
| `%pylab` | Activate the `pylab` library within the current session for interactive plotting. |
| `%recall` | Place one or more input lines from the command history at the current input prompt. |
| `%rerun` | Reexecute previous input from the numbered command history. |
| `%reset` | Reset the namespace for the current IPython session. |
| `%run` | Execute a named file as a Python script within the current session. |
| `%save` | Save a set of input lines or macro (defined with `%macro`) to a file with a given name. |
| `%sx` or `!!` | Shell execute: run a given shell command and store its output. |
| `%timeit` | Time the execution of a provided Python statement. |
| `%who` | Output all the currently defined variables. |
| `%who_ls` | As for `%who`, but return the variable names as a list of strings. |
| `%whos` | As for `%who`, but provides more information about each variable. |

## Aliases and bookmarks

A system shell command can be given an *alias*: a shortcut for a shell command that can be called as its own magic. For example, on Unix-like systems we could define the following alias to list only the directories on the current path:

```
In [x]: %alias lstdir ls -d */
In [x]: %lstdir
Meetings/         Papers/           code/             books/
databases/
```

Now typing `%lstdir` has the same effect as `!ls -d */`. If `%automagic` is `ON` this alias can also simply be called with `lstdir`.

The magic `%alias_magic` provides a similar functionality for IPython magics. For example, if you want to use `%h` as an alias to `%history`, type:

```
In [x]: %alias_magic h history
```

When working on larger projects it is often necessary to switch between different directories. IPython has a simple system for maintaining a list of bookmarks which act as shortcuts to different directories. The syntax for this magic function is

`%bookmark <name> [directory]`

If *[directory]* is omitted, it defaults to the current working directory.

```
In [x]: %bookmark py ~/research/code/python
In [x]: %bookmark www /srv/websites
In [x]: %cd py
/Users/christian/research/code/python
```

It may happen that a directory with the same name as your bookmark is within the current working directory. In that case, this directory takes precedence and you must use `%cd -b <name>` to refer to the bookmark.

A few more useful commands include:

- `%bookmark -l`: list all bookmarks
- `%bookmark -d <name>`: remove bookmark `<name>`
- `%bookmark -r`: remove all bookmarks

## Timing code execution

The IPython magic `%timeit <statement>` times the execution of the *single-line* statement `<statement>`. The statement is executed $N$ times in a loop, and each loop is repeated $R$ times. $N$ is a suitable, usually large, number chosen by IPython to yield meaningful results and $R$ is, by default, 3. The average time per loop for the best of the $R$ repetitions is reported. For example, to profile the sorting of a random arrangement of the numbers 1–100:

```
In [x]: import random
In [x]: numbers = list(range(1,101))
In [x]: random.shuffle(numbers)
In [x]: %timeit sorted(numbers)
100000 loops, best of 3: 13.2 µs per loop
```

Obviously the execution time will depend on the system (processor speed, memory, etc.). The aim of repeating the execution many times is to allow for variations in speed due to other processes running on the system. You can select $N$ and $R$ explicitly by passing values to the options `-n` and `-r` respectively:

```
In [x]: %timeit -n 10000 -r 5 sorted(numbers)
10000 loops, best of 5: 11.2 µs per loop
```

The cell magic, `%%timeit` enables one to time a *multiline block* of code. For example, a naive algorithm to find the factors of an integer `n` can be examined with

```
In [x]: n = 150
In [x]: %%timeit
factors = set()
for i in range(1, n+1):
    if not n % i:
        factors.add(n // i)
   ....:

100000 loops, best of 3: 16.3 µs per loop
```

## Recalling and rerunning code

To reexecute one or more lines from your IPython history, use `%rerun` with a line number or range of line numbers:

```
In [1]: import math
In [2]: angles = [0, 30, 60, 90]
In [3]: for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))
    .....:
sin(  0) =  0.00000
sin( 30) =  0.50000
sin( 45) =  0.70711
sin( 60) =  0.86603
sin( 90) =  1.00000

In [4]: angles = [15, 45, 75]
In [5]: %rerun 3
=== Executing: ===
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))

=== Output: ===
sin( 15) =  0.25882
sin( 45) =  0.70711
sin( 75) =  0.96593

In [6]: %rerun 2-3
=== Executing: ===
angles = [0, 30, 45, 60, 90]
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))

=== Output: ===
sin(  0) =  0.00000
sin( 30) =  0.50000
sin( 45) =  0.70711
sin( 60) =  0.86603
sin( 90) =  1.00000
```

The similar magic function `%recall` places the requested lines at the command prompt but does not execute them until you press Enter, allowing you to modify them first if you need to.

If you find yourself reexecuting a series of statements frequently, you can define a named macro to invoke them. Specify line numbers as before:

```
In [7]: %macro sines 3
Macro 'sines' created. To execute, type its name (without quotes).
=== Macro contents: ===
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))
```

```
In [8]: angles = [-45, -30, 0, 30, 45]
In [9]: sines
sin(-45) = -0.70711
sin(-30) = -0.50000
sin(  0) =  0.00000
sin( 30) =  0.50000
sin( 45) =  0.70711
```

## Loading, executing and saving code

To load code from an external file into the current IPython session, use

```
%load <filename>
```

If you want only certain lines from the input file, specify them after the `-r` option. This magic enters the lines at the command prompt, so they can be edited before being executed.

To load *and execute* code from a file, use

```
%run <filename>
```

Pass any command line options after *filename*; by default IPython treats them the same way that the system shell would. There are a few additional options to `%run`:

- `-i`: Run the script in the current IPython namespace instead of an empty one (i.e., the program will have access to variables defined in the current IPython session);
- `-e`: Ignore `sys.exit()` calls and `SystemExit` exceptions;
- `-t`: Output timing information at the end of execution (pass an integer to the additional option `-N` to repeat execution that number of times).

For example, to run `my_script.py` 10 times from within IPython with timing information:

```
In [x]: %run -t -N10 my_script.py
```

To save a range of input lines or a macro to a file, use `%save`. Line numbers are specified using the same syntax as `%history`. A `.py` extension is added if you don't add it yourself, and confirmation is sought before overwriting an existing file. For example,

```
In [x]: %save sines1 1 8 3
The following commands were written to file `sines1.py`:
import math
angles = [-45, -30, 0, 30, 45]
for angle in angles:
    print('sin({:3d}) = {:8.5f}'.format(angle, math.sin(math.radians(angle))))

In [x]: %save sines2 1-3
The following commands were written to file `sines2.py`:
import math
angles = [0, 30, 60, 90]
for angle in angles:
    print('sin({:3d}) = {:8.5f}'.format(angle, math.sin(math.radians(angle))))
```

Finally, to *append* to a file instead of overwriting it, use the `-a` option:

```
%save -a <filename> <line numbers>
```

### Capturing the output of a shell command

The IPython magic `%sx` *`command`*, equivalent to `!!`*`command`* executes the shell command *`command`* and returns the resulting output as a list (split into semantically useful parts on the new line character so there is one item per line). This list can be assigned to a variable to be manipulated later. For example,

```
In [x]: current_working_directory = %sx pwd
In [x]: current_working_directory
['/Users/christian/temp']
In [x]: filenames = %sx ls
In [x]: filenames
Out[x]:
['output.txt',
 'test.py',
 'readme.txt',
 'utils',
 'zigzag.py']
```

Here, `filenames` is a list of individual filenames.

The returned object is actually an `IPython.utils.text.SList` string list object. Among the useful additional features provided by `SList` are a native method for splitting each string into fields delimited by whitespace: `fields`; for sorting on those fields: `sort`; and for searching within the string list: `grep`. For example,

```
In [x]: files = %sx ls -l
In [x]: files
['total 8',
 '-rw-r--r--  1 christian  staff     93  5 Nov 16:30 output.txt',
 '-rw-r--r--  1 christian  staff  23258  5 Nov 16:31 readme.txt',
 '-rw-r--r--  1 christian  staff    218  5 Nov 16:32 test.py',
 'drwxr-xr-x  2 christian  staff     68  5 Nov 16:32 utils',
 '-rw-r--r--  1 christian  staff    365  5 Nov 16:20 zigzag.py']
In [x]: del files[0]    # strip non-file line 'total 8'
In [x]: files.fields()
Out[x]:
[['-rw-r--r--', '1', 'christian', 'staff', '93', '5', 'Nov', '16:30', 'output.txt'],
 ['-rw-r--r--', '1', 'christian', 'staff', '23258', '5', 'Nov', '16:31', 'readme.txt'],
 ...
 ['-rw-r--r--', '1', 'christian', 'staff', '365', '5', 'Nov', '16:20', 'zigzag.py']]

In [x]: ['{} last modified at {} on {} {}'.format(f[8], f[7], f[5], f[6])
                                    for f in files.fields()]
Out[x]:
['output.txt last modified at 16:30 on 5 Nov',
 'readme.txt last modified at 16:31 on 5 Nov',
 'test.py last modified at 16:32 on 5 Nov',
 'utils last modified at 16:32 on 5 Nov',
 'zigzag.py last modified at 16:20 on 5 Nov']
```

The `fields` method can also take arguments specifying the indexes of the fields to output; if more than one index is given the fields are joined by spaces:

```
In [x]: files.fields(0)      # First field in each line of files
Out[x]: ['-rw-r--r--', '-rw-r--r--', '-rw-r--r--', 'drwxr-xr-x', '-rw-r--r--']
```

```
In [x]: files.fields(-1)    # Last field in each line of files
Out[x]: ['output.txt', 'readme.txt', 'test.py', 'utils', 'zigzag.py']

In [x]: files.fields(8,7,5,6)
Out[x]:
['output.txt 16:30 5 Nov',
 'readme.txt 16:31 5 Nov',
 'test.py 16:32 5 Nov',
 'utils 16:32 5 Nov',
 'zigzag.py 16:20 5 Nov']
```

The `sort` method provided by `SList` objects can sort by a given field, optionally converting the field from a string to a number if required (so that, for example, `10 > 9`). Note that this method returns a new `SList` object.

```
In [x]: files.sort(4)      # Sort alphanumerically by size (not useful)
Out[x]:
['-rw-r--r--  1 christian  staff    218  5 Nov 16:32 test.py',
 '-rw-r--r--  1 christian  staff  23258  5 Nov 16:31 readme.txt',
 '-rw-r--r--  1 christian  staff    365  5 Nov 16:20 zigzag.py',
 'drwxr-xr-x  2 christian  staff     68  5 Nov 16:32 utils',
 '-rw-r--r--  1 christian  staff     93  5 Nov 16:30 output.txt']

In [x]: files.sort(4, nums=True)      # Sort numerically by size (useful)
Out[x]:
['drwxr-xr-x  2 christian  staff     68  5 Nov 16:32 utils',
 '-rw-r--r--  1 christian  staff     93  5 Nov 16:30 output.txt',
 '-rw-r--r--  1 christian  staff    218  5 Nov 16:32 test.py',
 '-rw-r--r--  1 christian  staff    365  5 Nov 16:20 zigzag.py',
 '-rw-r--r--  1 christian  staff  23258  5 Nov 16:31 readme.txt']
```

The `grep` method returns items from the `SList` containing a given string;[6] to search for a string in a given field only, use the `field` argument:

```
In [x]: files.grep('txt')             # Search for lines containing 'txt'
Out[x]:
['-rw-r--r--  1 christian  staff     93  5 Nov 16:30 output.txt',
 '-rw-r--r--  1 christian  staff  23258  5 Nov 16:31 readme.txt']

In [x]: files.grep('16:32', field=7)    # Search file files created at 16:32
Out[x]:
['-rw-r--r--  1 christian  staff    218  5 Nov 16:32 test.py',
 'drwxr-xr-x  2 christian  staff     68  5 Nov 16:32 utils']
```

**Example E5.1** RNA encodes the amino acids of a peptide as a sequence of *codons*, with each codon consisting of three nucleotides chosen from the 'alphabet': U (uracil), C (cytosine), A (adenine) and G (guanine).

The Python script, `codon_lookup.py`, available at scipython.com/eg/aab, creates a dictionary, `codon_table`, mapping codons to amino acids where each amino acid is identified by its one-letter abbreviation (e.g., `R` = arginine). The stop codons, signaling termination of RNA translation, are identified with the single asterisk character, `*`.

[6] In fact, its name implies it will match *regular expressions* as well, but we will not expand on this here.

The codon AUG signals the start of translation within a nucleotide sequence as well as coding for the amino acid methionine.

This script can be executed within IPython with `%run codon_lookup.py` (or loaded and then executed with `%load codon_lookup.py` followed by pressing Enter:

```
In [x]: %run codon_lookup.py
In [x]: codon_table
Out[x]:
{'GCG': 'A',
 'UAA': '*',
 'GGU': 'G',
 'UCU': 'S',
      ...
 'ACA': 'T',
 'ACC': 'T'}
```

Let's define a function to translate an RNA sequence. Type `%edit` and enter the following code in the editor that appears.

```
def translate_rna(seq):
    start = seq.find('AUG')
    peptide = []
    i = start
    while i < len(seq)-2:
        codon = seq[i:i+3]
        a = codon_table[codon]
        if a == '*':
            break
        i += 3
        peptide.append(a)
    return ''.join(peptide)
```

When you exit the editor it will be executed, defining the function, `translate_rna`:

```
IPython will make a temporary file named: /var/folders/fj/yv29fhm91v7_6g
7sqsy1z2940000gp/T/ipython_edit_thunq9/ipython_edit_dltv_i.py
Editing... done. Executing edited code...
Out[x]: "def translate_rna(seq):\n    start = seq.find('AUG')\n
peptide = []\
n    i = start\n    while i < len(seq)-2:\n        codon = seq[i:i+3]\n        a
 = codon_table[codon]\n        if a == '*':\n            break\n        i += 3\n
       peptide.append(a)\n    return ''.join(peptide)\n"
```

Now feed the function an RNA sequence to translate:

```
In[x]: seq = 'CAGCAGCUCAUACAGCAGGUAAUGUCUGGUCUCGUCCCCGGAUGUCGCUACCCACGAG
ACCCGUAUCCUACUUUCUGGGGAGCCUUUACACGGCGGUCCACGUUUUUCGCUACCGUCGUUUUCCCGGUGC
CAUAGAUGAAUGUU'
In [x]: translate_rna(seq)
Out[x]: 'MSGLVPGCRYPRDPYPTFWGAFTRRSTFFATVVFPVP'
```

To read in a list of RNA sequences (one per line) from a text file, `seqs.txt`, and translate them, one could use `%sx` with the system command `cat` (or, on Windows, the command `type`):

```
In [x]: seqs = %sx cat seqs.txt
In [x]: for seq in seqs:
   ...:     print(translate_rna(seq))
   ...:
MHMLDENLYDLGMKACHEGTNVLDKWRNMARVCSCDYQFK
MQGSDGQQESYCTLPFEVSGMP
MPVEWRTMQFQRLERASCVKDSTFKNTGSFIKDRKVSGISQDEWAYAMSHQMQPAAHYA
MIVVTMCQ
MGQCMRFAPGMHGMYSSFHPQHKEITPGIDYASMNEVETAETIRPI
```

### 5.1.4 Exercises

#### Problems

**P5.1.1** Improve on the algorithm to find the number of factors of an integer given in Section 5.1.3 by (a) looping the trial factor, `i`, up to no greater than the square root of `n` (why is it not necessary to test values of `i` greater than this?), and (b) using a generator (see Section 4.3.5). Compare the execution speed of these alternatives using the `%timeit` IPython magic.

**P5.1.2** Using the fastest algorithm from the previous question, devise a short piece of code to determine the *highly composite numbers* less than 100000 and use the `%%timeit` cell magic to time its execution. A highly composite number is a positive integer with more factors than any smaller positive integer, for example: $1, 2, 4, 6, 12, 24, 36, 48, \cdots$.

## 5.2 IPython Notebook

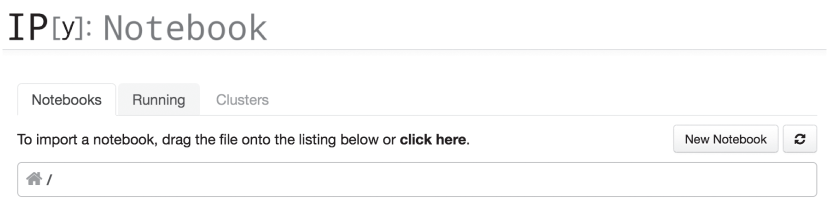
IPython Notebook provides an interactive environment for Python programming within a web browser. Its main advantage over the more traditional console-based approach of the IPython shell is that Python code can be combined with documentation (including in rendered LaTeX), images and even rich media such as embedded videos. IPython notebooks are increasingly being used by scientists to communicate their research by including the computations carried out on data as well as simply the results of those computations. The format makes it easy for researchers to collaborate on a project and for others to validate their findings by reproducing their calculations on the same data. Note that from version 4, the IPython Notebook project has been reformulated as Jupyter with bindings for other languages as well as Python.

### 5.2.1 IPython notebook basics

#### Starting the IPython notebook server

If you have IPython notebook installed, the server that runs the browser-based interface to IPython can be started from the command line with

```
ipython notebook
```



To import a notebook, drag the file onto the listing below or **click here**.

**Figure 5.1** The IPython notebook index page.

This will open a web browser window at the URL of the local IPython notebook application. By default this is http://127.0.0.1:8888 though it will default to a different port if 8888 is in use.

The notebook index page (Figure 5.1) contains a list of the notebooks currently available in the directory from which the notebook server was started. This is also the default directory to which notebooks will be saved (with the extension `.ipynb`), so it is a good idea to execute the above command somewhere convenient in your directory hierarchy for the project you are working on.

The index page contains three tabs: *Notebooks* lists the IPython notebooks and subdirectories within the current working directory, *Running* lists those notebooks that are currently active within your session (even if they are not open in a browser window); *Clusters* provides an interface to IPython's parallel computing engine: we will not cover this topic in this book.

From the index page, one can start a new notebook (by clicking on "New Notebook") or open an existing notebook (by clicking on its name). To import an existing notebook into the index page, either click where indicated at the top of the page or drag the notebook file into the index listing from elsewhere on your operating system.

To stop the notebook server, press CTRL-C in the terminal window it was started from (and confirm at the prompt).

### Editing an IPython notebook

To start a new notebook, click the "New Notebook" button. This opens a new browser tab containing the interface where you will write your code and connects it to an IPython *kernel*, the process responsible for executing the code and communicating the results back to the browser.

The new notebook document (Figure 5.2) consists of a *title bar*, a *menu bar* and a *tool bar*, under which is an IPython prompt where you will type the code and markup (e.g., explanatory text and documentation) as a series of *cells*.

In the title bar the name of the first notebook you open will probably be "Untitled0"; click on it to rename it to something more informative. The menu bar contains options for saving, copying, printing, rearranging and otherwise manipulating the notebook document. The tool bar consists of series of icons that act as shortcuts for common operations that can also be achieved through the menu bar.

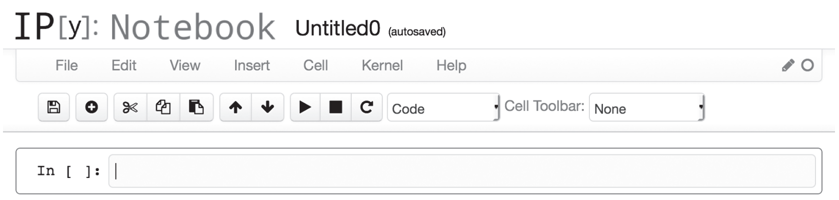


**Figure 5.2** IPython with a new notebook document.

There are four types of input cells where you can write the content for your notebook:

- Code cells: the default type of cell, this type of cell consists of executable code. As far as this chapter is concerned, the code you write here will be Python, but IPython Notebook (now called Jupyter) does provide a mechanism of executing code written in other languages such as Julia and R.
- Heading cells: six levels of heading (from top-level section titles to paragraph-level text). When "executed" this type of cell produces a rich-text rendering of their contents at an appropriate font size.
- Markdown cells: this type of cell allows for a rich form of documentation for your code. When executed, the input to a markdown cell is converted into HTML, which can include mathematical equations, font effects, lists, tables, embedded images and videos – see Section 5.2.1.
- Raw cells: input into this type of cell is not changed by the notebook – its content and formatting is preserved exactly.

### Running cells

Each cell can consist of more than one line of input, and the cell is not interpreted until you "run" (i.e., execute) it. This is achieved either by selecting the appropriate option from the menu bar (under the "Cell" drop-down submenu), by clicking the "Run cell" "play" button on the tool bar, or through the following keyboard shortcuts:

- `Shift-Enter`: Execute the cell, showing any output, and then *move the cursor* onto the cell below. If there is no cell below, a new, empty one will be created.
- `CTRL-Enter`: Execute the cell in place, but *keep the cursor* in the current cell. Useful for quick "disposable" commands to check if a command works or for retrieving a directory listing.
- `Alt-Enter`: Execute the cell, showing any output, and then *insert and move the cursor to a new cell* immediately beneath it.

Two other keyboard shortcuts are useful. When editing a cell the arrow keys navigate the *contents* of the cell (*edit mode*); from this mode, pressing `Esc` enters *command mode* from which the arrow keys navigate *through* the cells. To reenter edit mode on a selected cell, press `Enter`.

The menu bar, under the "Cell" drop-down submenu, provides many ways of running a notebook's cells: usually, you will want to run the current cell individually or run it and all those below it.

## Code cells

You can enter anything into a code cell that you can when writing a Python program in an editor or at the regular IPython shell. Code in a given cell has access to objects defined in other cells (providing they have been run). For example,

```
In [ ]:   n = 10
```

Pressing Shift-Enter or clicking Run Cell executes this statement (defining `n` but producing no output) and opens a new cell underneath the old one:

```
In [1]:   n = 10

In [ ]:
```

Entering the following statements at this new prompt:

```
In [ ]:   sum_of_squares = n * (n+1) * (2*n+1) // 6
          print('1**2 + 2**2 + ... + {}**2 = {}'.format(n,
               sum_of_squares))
```

and executing as before produces output and opens a third empty input cell. The whole notebook document then looks like:

```
In [1]:   n = 10

In [2]:   sum_of_squares = n * (n+1) * (2*n+1) // 6
          print('1**2 + 2**2 + ... + {}**2 = {}'.format(n,
               sum_of_squares))

Out[2]:   1**2 + 2**2 + ... + 10**2 = 385

In [ ]:
```

You can edit the value of `n` in input cell 1 and rerun the entire document to update the output. It is worth noting that it is also possible to set a new value for `n` *after* the calculation in cell 2:

```
In [3]:   n = 15
```

running cell 3 and then cell 2 then leaves the output to cell 2 as

```
Out[2]:   1**2 + 2**2 + ... + 15**2 = 1240
```

even though the cell above still defines `n` to be 10. That is, unless you run the entire document from the beginning, the output does not necessarily reflect the output of a script corresponding to the code cells taken in order.

System commands (those prefixed with ! or !!) and IPython magics can all be used within IPython notebook.

It is also possible to use `pylab` "inline" in the notebook so that plots show up as images embedded in the document. To turn this feature on, use

```
In [x]:   %pylab inline
```

By itself this command imports the `pylab` library we used in Chapter 3 (and a few other things besides), but imports its symbols into the namespace of your interactive session. That is, the `%pylab inline` magic has the effect of `from pylab import *` and you can type, for example, `plot(x, y)` instead of `pylab.plot(x, y)`. To prevent this behavior, we recommend adding the argument `--no-import-all`:

```
In [x]:  %pylab inline --no-import-all
```

This stops `pylab` from polluting your namespace with its own definitions.[7]

## Markdown cells

Markdown cells convert your input text into HTML, applying styles according to a simple syntax illustrated below. The full documentation is at

http://daringfireball.net/projects/markdown/

Here we explain the most useful features. A complete notebook of these examples can be downloaded from scipython.com/book/markdown.

*Basic markdown*

- Simple styles can be applied by enclosing text by asterisks or underscores:

```
In [x]:  Surrounding text by two asterisks denotes
             **bold style**; using one asterisk denotes
             *italic text*, as does _a single
             underscore_.
```

Surrounding text by two asterisks denotes **bold style**; using one asterisk denotes *italic text*, as does *a single underscore*.

- Block quotes are indicated by a single angle bracket, >:

```
In [x]:  > "Climb if you will, but remember that
             courage and strength are nought without
             prudence, and that a momentary negligence
             may destroy the happiness of a lifetime.
             Do nothing in haste; look well to each
             step; and from the beginning think what
             may be the end." - Edward Whymper
```

> “Climb if you will, but remember that courage and strength are nought without prudence, and that a momentary negligence may destroy the happiness of a lifetime. Do nothing in haste; look well to each step; and from the beginning think what may be the end.” – Edward Whymper

- Code *examples* (for illustration rather than execution) are between blank lines and indented by four spaces (or a tab). The following will appear in a monospaced font with the characters as entered:

[7] It is particularly annoying to find your innocent variable names such as `f` clash with `pylab`’s own function calls.

```
In [x]:     n = 57
            while n != 1:
                if n % 2:
                    n = 3*n + 1
                else:
                    n //= 2
```

```
n = 57
while n != 1:
    if n % 2:
        n = 3*n + 1
    else:
        n //= 2
```

- Inline code examples are created by surrounding the text with backticks (`):

```
In [x]:  Here are some Python keywords: `for`, `while`
             and `lambda`.
```

Here are some Python keywords: `for`, `while` and `lambda`.

- New paragraphs are started after a blank line.

*HTML within markdown*

The markdown used by IPython notebooks encompasses HTML, so valid HTML entities and tags can be used directly: for example, the `<em>` tag for emphasis, as can CSS styles to produce effects such as underlined text. Even complex HTML such as tables can be marked up directly.

```
In [x]:  The following <em>Punnett table</em> is <span
             style="text-decoration: underline" >marked
             up</span> in HTML.

         <table style="text-align: center;">
         <tr>
         <th style="border-top:none; border-left:none;"
             rowspan="2" colspan="2"></th>
         <th colspan="2">Male</th>
         </tr>
         <tr>
         <th>A</th>
         <th>a</th>
         </tr>
         <tr>
         <th rowspan="2">Female</th>
         <th>a</th>
         <td style="background: #aaa;">Aa</td>
         <td>aa</td>
         </tr>
         <tr>
         <th>a</th>
         <td style="background: #aaa;">Aa</td>
         <td>aa</td>
         </tr>
         </table>
```

The following *Punnett table* is marked up in HTML.

| | | **Male** | |
|---|---|---|---|
| | | **A** | **a** |
| **Female** | **a** | Aa | aa |
| | **a** | Aa | aa |

*Lists*

Itemized (unnumbered) lists are created using any of the markers `*`, `+` or `-`, and nested sublists are simply indented.

In [x]:
```
The inner planets and their satellites:

* Mercury
* Venus
* Earth
    * The Moon
+ Mars
    - Phoebus
    - Deimos
```

The inner planets and their satellites:

- Mercury
- Venus
- Earth
  - The Moon
- Mars
  - Phoebus
  - Deimos

Ordered (numbered) lists are created by preceding items by a number followed by a full stop (period) and a space:

In [x]:
```
1. Symphony No. 1 in C major, Op. 21
2. Symphony No. 2 in D major, Op. 36
3. Symphony No. 3 in E-flat major ("Eroica"), Op. 55
```

1. Symphony No. 1 in C major, Op. 21
2. Symphony No. 2 in D major, Op. 36
3. Symphony No. 3 in E-flat major ("Eroica"), Op. 55

*Links*

There are three ways of introducing links into markdown text:

- *Inline* links provide a URL in round brackets after the text to be turned into a link in square brackets. For example,

In [x]:
```
Here is a link to the
[IPython website](http://ipython.org/).
```

Here is a link to the IPython website.

- *Reference* links label the text to turn into a link by placing a name (containing letters, numbers or spaces) in square brackets after it. This name is expected to be defined using the syntax `[name]: url` elsewhere in the document, as in the following example markdown cell.

In [x]:
```
Some important mathematical sequences are the
    [prime numbers][primes],
[Fibonacci sequence][fib] and the [Catalan
    numbers][catalan_numbers].

...

[primes]: http://oeis.org/A000040
[fib]: http://oeis.org/A000045
[catalan_numbers]: http://oeis.org/A000108]
```

Some important mathematical sequences are the primes, Fibonacci sequence and the Catalan numbers.

- *Automatic* links, for which the clickable text is the same as the URL are created simply by surrounding the URL by angle brackets:

In [x]:
```
My website is <http://www.christianhill.co.uk>.
```

My website is http://www.christianhill.co.uk.

If the link is to a file on your local system, give as the URL the path, relative to the notebook directory, prefixed with `files/`:

In [x]:
```
Here is [a local data file](files/data/data0.txt).
```

Here is a a local data file.

Note that links open in a new browser tab when clicked.

### *Mathematics*

Mathematical equations can be written in LaTeX and are rendered using the Javascript library, MathJax. Inline equations are delimited by single dollar signs; "displayed" equations by doubled dollar signs:

In [x]:
```
An inline equation appears within a sentence of
    text, as in the definition of the function
    $f(x) = \sin(x^2)$; displayed equations get
    their own line(s) between lines of text:
$$\int_0^\infty e^{-x^2}dx = \frac{\sqrt{\pi}}{2}.$$
```

An inline equation appears within a sentence of text, as in the definition of the function $f(x) = \sin(x^2)$; displayed equations get their own line(s) between lines of text:

$$\int_0^\infty e^{-x^2}\,dx = \frac{\sqrt{\pi}}{2}.$$

*Images and video*

Links to image files work in exactly the same way as ordinary links (and can be inline or reference links), but are preceded by an exclamation mark, `!`. The text in square brackets between the exclamation mark and the link acts as *alt text* to the image. For example,

```
In [x]:  ![An interesting plot of the Newton
            fractal](/files/images/newton_fractal.png)
         ![A remote link to a star
            image](http://christianhill.co.uk/media/books/
            python/star.svg)
```

Video links must use the HTML5 `<video>` tag, but note that not all browsers support all video formats. For example,

```
In [x]:  <video controls style="width: 500px; margin: 0
            auto; display: block;"
            src="files/diffmap-animated.ogv" />
```

The data constituting images, video and other locally linked content are not *embedded* in the notebook document itself: these files must be provided with the notebook when it is distributed.

## 5.2.2 Converting notebooks to other formats

`nbconvert` is a tool, installed with IPython notebook, to convert notebooks from their native `.ipynb` format[8] to any of several alternative formats. It is run from the (system) command line as

```
ipython nbconvert --to <format> <notebook.ipynb>
```

where *`notebook.ipynb`* is the name of the IPython notebook file to be converted and *`format`* is the desired output format. The default (if no *`format`* is given), is to produce a static HTML file, as described below.

### Conversion to HTML

The command

```
ipython nbconvert <notebook.ipynb>
```

converts `notebook.ipynb` to HTML and produces a file, `notebook.html` in the current directory. This file contains all the necessary headers for a stand-alone HTML page,

[8] This format is, in fact, just a JSON (JavaScript Object Notation) document.

which will closely resemble the interactive view produced by the IPython notebook server, but as a static document.

If you want just the HTML corresponding to the notebook without the header (`<html>`, `<head>`, `<body>` tags, etc.), suitable for embedding in an existing web page, add the `--template basic` option.

Any supporting files, such as images, are automatically placed in a directory with the same base name as the notebook but suffixes with `_files`. For example, `ipython nbconvert mynotebook.ipynb` generates `mynotebook.html` and the directory `mynotebook_files`.

### Conversion to LaTeX

To export the notebook as a LaTeX document, use

```
ipython nbconvert --to latex <notebook.ipynb>
```

To automatically run `pdflatex` on the `notebook.tex` file generated to produce a PDF file, add the option --`post pdf`.

### Conversion to markdown

```
ipython nbconvert --to markdown <notebook.ipynb>
```

converts the whole notebook into markdown (see Section 5.2.1): cells that are already in markdown are unaffected and code cells are placed in triple-backtick (` ``` `) blocks.

### Conversion to Python

The command

```
ipython nbconvert --to python <notebook.ipynb>
```

converts `notebook.ipynb` into an executable Python script. If any of the notebook's code cells contain IPython magic functions, this script may only be executable from within an IPython session. Markdown and other text cells are converted to comments in the generated Python script code.