2 Getting started with *IPython*

2.1 Generalities

This sounds like software produced by Apple, but it is in fact a Python interpreter on steroids. It has been designed and written by scientists with the aim of offering very fast exploration and construction of code with minimal typing effort, and offering appropriate, even maximal, on-screen help when required. Documentation and much more is available on the website.[1] This chapter is a brief introduction to the essentials of using *IPython*. A more extended discursive treatment can be found in, e.g., Rossant (2013).

IPython comes with three different user interfaces, *terminal*, *qtconsole* and *notebook*. If you have installed the recommended EPD distribution, then all three should be available to you. For the last two, additional software might be needed if you have used a different distribution. You can check what versions are available for your installation by issuing (at the command line) first `ipython` followed by the "return" key (RET). You can escape from *IPython* by typing `exit` followed by RET in the interpreter. Next try out the command `ipython qtconsole` following a similar strategy. Finally, try out the command `ipython notebook`. This should open in a new browser window. To escape from the third, you need CTL-`c` at the command line, plus closing the browser window. What is the difference between them?

Built into *IPython* is the GNU *readline* utility. This means that on the interpreter's current line, the left- and right-arrow keys move the cursor appropriately, and deletion and insertion are straightforward. Just about the only additional commands I ever use are CTL-`a` and CTL-`e` which move the cursor to the start and end of the line, and CTL-`k` which deletes the material to the right of the cursor. The up- and down-arrow keys replicate the previous and next command lines respectively, so that they can be edited. However, there is no way to edit two different lines simultaneously in *terminal* mode.

Both *qtconsole* and *notebook* mode remove this restriction, allowing multiline editing, which is a great convenience. All three modes allow access to the Python scientific graphics facility *matplotlib* explored in Chapter 5. This displays graphics in new windows, which themselves can be edited interactively. The latter two *IPython* modes also allow in-line graphics, i.e., the graphics are displayed in the console or browser windows. This looks neat but, alas, the interactivity has been lost. The *notebook* mode offers facilities which both emulate and transcend the Mathematica "notebook" concept. Besides including in-line graphics, we can insert text before and after the code snippets.

[1] It can be found at `www.ipython.org`.

https://doi.org/10.1017/CBO9781107447875.003 Published online by Cambridge University Press

Both RTF and LaTeX are supported. This is clearly a significant facility for science education. Tantalizingly, the *notebook* mode is still in very active development. You need to check the website cited above for up-to-date details. For this reason, we shall concentrate on *terminal* mode which is central to understanding and using all three modes.

2.2 Tab completion

While using the *IPython* interpreter, *tab completion* is always present. This means that whenever we start typing a Python-related name, we can pause and press the TAB key, to see a list of names valid in this context, which agree with the characters already typed. As an example, suppose we need to type `import matplotlib`.[2] Typing `i`TAB reveals 15 possible completions. By inspection, only one of them has second letter `m`, so that `im`TAB will complete to `import`. Augmenting this to `import m`TAB shows 25 possibilities, and by inspection we see that we need to complete the command by `import matp`TAB to complete the desired line.

That example was somewhat contrived. Here is a more compulsive reason for using tab completion. When developing code, we tend, lazily, to use short names for variables, functions etc. (In early versions of Fortran, we were indeed restricted to six or eight characters, but nowadays the length can be arbitrary.) Short names are not always meaningful ones, and the danger is that if we revisit the code in six months time, the intent of the code may no longer be self evident. By using meaningful names of whatever length is needed, we can avoid this trap. Because of tab completion, the long name only has to be typed once.

2.3 Introspection

IPython has the ability to inspect just about any Python construct, including itself, and to report whatever information its developers have chosen to make available. This facility is called *introspection*. It is accessed by the single character `?`. The easiest way to understand it is to use it, and so you are recommended to fire up the interpreter, e.g., by typing `ipython` followed by a RET on the command line. *IPython* will respond with quite a lengthy header, followed by an input line labelled `In [1]:`.

Now that you are in *IPython*, you can try out introspection by typing `?` (followed by RET) on the input line. *IPython* responds by issuing in pager mode a summary of all of the facilities available. If you exit this, see Section A.2.6, the command `quickref` (hint: use tab completion) gives a more concise version. Very careful study of both documents is highly recommended.

However, scientists are impatient folk, and the purpose of this chapter is to get them up and running with the most useful features. Therefore, we need to type in some Python code, which newcomers will have to take on trust until they have mastered Chapters 3 and 4. For example, please type in (spaces optional)

[2] *matplotlib* is essential to scientific graphics and forms the main topic of Chapter 5.

https://doi.org/10.1017/CBO9781107447875.003 Published online by Cambridge University Press

```
a=3
b=2.7
c=12 + 5j
s='Hello World!'
L=[a, b, c, s, 77.77]
```

The first two lines mean that `a` refers to an integer and `b` to a float (floating-point number). Python uses the engineers' convention whereby $\sqrt{-1} = j$. (Mathematicians would have preferred $\sqrt{-1} = i$.) Then `c` refers to a complex number.[3] Typing each of `a`, `b` and `c` on a line by itself reveals the value of the object to which the identifier refers. Now try `c?` on a line by itself. This confirms that `c` does indeed refer to a complex number, shows how it will be displayed and points out that another way of creating it would have been `c=complex(12,5)`. Next try `c.` immediately followed by TAB. The interpreter will immediately offer three possible completions. What do they mean? Here it is almost obvious, but try `c.real?`. (Using tab completion, you don't need to type `eal`.) This reveals that `c.real` is a float with the value 12, i.e., the real part of the complex number. The newcomer might like to check out `c.imag`. Next try out `c.conjugate?`. (Again only five keystrokes plus RETURN are needed!) This reveals that `c.conjugate` is a function, to be used, e.g., as `cc = c.conjugate()`.

The notation here might seem rather strange to users of say Fortran or C, where `real(c)` or `conjugate(c)` might have been expected. The change in syntax comes about because Python is an object-oriented language. The object here is `12+5j`, referred to as `c`. Then `c.real` is an enquiry as to the value of a component of the object. It changes nothing. However, `c.conjugate()` either alters the object or, as here, creates a new one, and is hence a function. This notation is uniform for all objects, and is discussed in more detail in Section 3.10.

Returning to the snippet, typing `s` by itself on a line prints a string. We can confirm this by the line `s?`, and `s.` followed by a TAB reveals 38 possible completions associated with string objects. The reader should use introspection to reveal what some of them do. Similarly, `L.?` shows that `L` is a list object with nine available completions. Do try a few of them out! As a general rule, the use of introspection and tab completion anywhere in Python code should generate focused documentation. There is a further introspection command `??` which, where appropriate, will reveal the original source code of a function, and examples will be given later in Section 2.6. (The object functions we have so far encountered are built-in, and were not coded in Python!)

2.4 History

If you look at the output from the code snippet in the previous section, you will see that *IPython* employs a history mechanism which is very similar to that in Mathematica notebooks. Input lines are labelled `In[1]`, `In[2]`, ..., and if input `In[n]` produces any

[3] Note that in the code snippet there is no multiplication sign (*) between 5 and `j`.

https://doi.org/10.1017/CBO9781107447875.003 Published online by Cambridge University Press

output, it is labelled `Out[n]`. As a convenience, the past three input lines are available as `_i`, `_ii` and `_iii`, and the corresponding output lines are available as `_`, `__` and `___`. In practice though, you can insert the content of a previous input line into the current one by navigating using ↑ (or CTL-`p`) and ↓ (or CTL-`n`), and this is perhaps the most common usage. Unusually, but conveniently, history persists. If you close down *IPython* (using `exit`) and then restart it, the history of the previous session is still available via the arrow keys. There are many more elaborate things you can do with the history mechanism, try the command `%history?`.

2.5 Magic commands

The *IPython* window expects to receive valid Python commands. However, it is very convenient to be able to type commands which control either the behaviour of *IPython* or that of the underlying operating system. Such commands, which coexist with Python ones, are called *magic* commands. A very long very detailed description can be found by typing `%magic` in the interpreter, and a compact list of available commands is given by typing `%lsmagic`. (Do not forget tab completion!) You can get very helpful documentation on each command by using introspection. The rest of this section is devoted to explaining "magic".

Let us start by considering system commands. A harmless example is `pwd` which comes from the Unix operating system where it just prints the name of the current directory (**p**rint **w**orking **d**irectory) and exits. There are usually three ways to achieve this in the *IPython* window. You should try out the following snippet. (Do not type the numbers to the left of the box.)

```
1 !pwd
2 %pwd
3 pwd
4 pwd='hoho'
5 pwd
6 %pwd
7 del pwd
8 pwd
9 %automagic?
```

The first two commands are legitimate because no Python identifier starts with either of the symbols used. The first is interpreted as a system command and returns exactly what a regular system command would give. The second returns the same, but enclosed in apostrophes (indicating a string) and prefaced by a `u`. The `u` indicates that the string is encoded in Unicode, which enables a rich variety of outputs. Unicode was mentioned briefly in Section A.2.1. The `%` prefix indicates a magic command, as we now explain. Up to now, no significance has been assigned to the identifier `pwd`. Line 3 reveals the same output as line 2. Thus in general the % prefix is not needed. Line 4 says that `pwd` is now an identifier for the string `'hoho'`. Now typing `pwd` in line 5 reveals the string.

https://doi.org/10.1017/CBO9781107447875.003 Published online by Cambridge University Press

However, line 6 shows that the original magic command still works. In line 7, we delete all reference to the string. Then line 8 produces the same output as line 3. The "magic" is that `pwd` is a system command unless the user has assigned another value to it, while `%pwd` always works. Line 9 explains what is going on in much more detail.

2.6 The magic %run command

Because this command is so important, we shall present it via a somewhat more realistic example which, inevitably, requires a little preparation. In a later example, in Section 3.9, we shall consider how to implement arbitrary precision real arithmetic via fractions. There is an issue with fractions, e.g., 3/7 and 24/56 are usually regarded as the same number. Thus there is a problem here, to determine the "highest common factor" of two integers a and b, or as mathematicians are wont to say, their "greatest common divisor" (GCD), which can be used to produce a canonical form for the fraction a/b. (By inspection of factors, the GCD of 24 and 56 is 8, which implies 24/56 = 3/7 and no further reduction of the latter is possible.) Inspection of factors is not easily automated, and a little research, e.g., on the web, reveals *Euclid's algorithm*. To express this concisely, we need a piece of jargon. Suppose a and b are integers. Consider long division of a by b. The remainder is called a mod b, e.g., 13 mod 5 = 3, and 5 mod 13 = 5. Now denote the GCD of a and b by $gcd(a, b)$. Euclid's algorithm is most easily described recursively via

$$gcd(a, 0) = a, \qquad gcd(a, b) = gcd(b, a \bmod b), \quad (b \neq 0). \tag{2.1}$$

Try evaluating by hand $gcd(56, 24)$ according this recipe. It's very fast! It can be shown that the most laborious case arises when a and b are consecutive Fibonacci numbers, and so they would be useful for a test case. The *Fibonacci numbers* F_n are defined recursively via

$$F_0 = 0, \quad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}, \quad n \geqslant 2. \tag{2.2}$$

The sequence begins $0, 1, 1, 2, 3, 5, 8, \ldots$.

How do we implement all of this both efficiently and speedily in Python? We start with the Fibonacci question because it looks to be straightforward.

In order to get started with mastering *IPython*, the novice reader is asked to take on trust the next two code snippets. Partial explanation is offered here, but all of the features will be explained more thoroughly in Chapter 3.

Consider typing, using your chosen text editor, the following snippet into a file called `fib.py` in a convenient folder/directory. Unlike the C and Fortran families, *Python* deliberately eschews the use of parentheses () and braces {}. In order to do this, it insists on code being consistently indented. Any code line which ends with a colon (:) requires a new indented block, the unofficial standard being four spaces long. The end of the code block is shown by a return to the former level of indentation, see, e.g., lines 10–12

https://doi.org/10.1017/CBO9781107447875.003 Published online by Cambridge University Press

in the snippet below. *IPython* and any Python-aware editor will recognize the ending colon and do this automatically.[4]. See also section 3.1.

```
1  # File: fib.py Fibonacci numbers
2
3  """ Module fib: Fibonacci numbers.
4     Contains one function fib(n).
5  """
6
7  def fib(n):
8      """ Returns n'th Fibonacci number. """
9      a,b=0,1
10     for i in range(n):
11         a,b=b,a+b
12     return a
13
14 ######################################################
15 if __name__ == "__main__":
16     for i in range(1001):
17         print "fib(",i,") = ",fib(i)
```

The details of Python syntax are explained in Chapter 3. For the time being, note that lines 3–5 define a *docstring*, whose purpose will be explained shortly. Lines 7–12 define a Python function. Note the point made above that every colon (:) demands an indentation. Line 7 is the function declaration. Line 8 is the function *docstring*, again soon to be explained. Line 9 introduces identifiers `a` and `b`, which are local to this function, and refer initially to the values 0 and 1 respectively. Next examine line 11, ignoring for the moment its indentation. Here `a` is set to refer to the value that `b` originally referred to. Simultaneously, `b` is set to refer to the sum of values originally referred to by `a` and `b`. Clearly, lines 9 and 11 replicate the calculations implicit in (2.2). Now line 10 introduces a *for-loop* or *do-loop*, explained in Section 3.7.1, which extends over line 11. Here `range(n)` generates a dummy list with n elements, $[0, 1, \ldots, n-1]$, and so line 11 is executed precisely n times. Finally, line 12 exits the function with the return value set to that referred to finally by `a`. Next type one or more blank lines to remove the indentation.

Naturally, we need to provide a test suite to demonstrate that this function behaves as intended. Line 14 is simply a comment. Line 15 will be explained soon. (When typing it, note that there are four pairs of underscores.) Because it is an *if statement* terminated by a colon, all subsequent lines need to be indented. We have already seen the idea behind line 16. We repeat line 17 precisely 1001 times with $i = 0, 1, 2, \ldots, 1000$. It prints a string with four characters, the value of `i`, another string with four characters, and the value of `fib(i)`.

Assuming that you have created and saved this file, make sure that *IPython* is open

[4] Note that to un-indent in *IPython*, you need to type an empty line.

https://doi.org/10.1017/CBO9781107447875.003 Published online by Cambridge University Press

in the same directory. Then within the *IPython* window, issue the (magic) command `run fib`. If your creation was syntactically correct, the response will be 1001 lines of Fibonacci values. If not, the response will indicate the first error encountered. Returning to the editor window, correct and save the source. Then try the `run fib` command again. (Beginners must expect to go through this cycle several times, but it is quick!) Once the programme is verified we can ask how fast is it? Run the programme again but with the enhanced command `run -t fib` and *IPython* will produce timing data. On my machine, the "User time" is 0.05 s., but the "Wall time" is 4.5 s. Clearly, the discrepancy reflects the very large number of characters printed to the screen. To verify this, modify the snippet as follows. Comment out the print statement in line 17 by inserting a *hash* (#) character at the start of the line. Add a new line 18: `fib(i)`, being careful to get the indentation correct. (This evaluates the function, but does nothing with the value.) Now run the program again. On my machine it takes 0.03 s., showing that `fib(i)` is fast, but printing is not. (Don't forget to comment out line 18, and uncomment in line 17!)

We still need to explain the docstrings in lines 3–5 and 8, and the weird line 15. Close down *IPython* (use `exit`) and then reopen a fresh version. Type the single line `import fib`, which reflects the core of the filename. The tail `.py` is not needed. We have imported an object `fib`. What is it? Introspection suggests the command `fib?`, and *IPython*'s response is to print the *docstring* from lines 3–5 of the snippet. This suggests that we find out more about the function `fib.fib`, so try `fib.fib?`, and we are returned the *docstring* from line 8. The purpose of *docstrings*, which are messages enclosed in pairs of triple double-quotes, is to offer online documentation to other users and, just as importantly, you in a few days time! However, introspection has a further trick up its sleeve. Try `fib.fib??` and you will receive a listing of the source code for this function!

You should have noticed that `import fib` did not list the first 1001 Fibonacci numbers. Had we instead, in a separate session, issued the command `run fib`, they would have been printed! Line 15 of the snippet detects whether the file `fib.py` is being imported or run, and responds accordingly without or with the test suite. How it does this is explained in Section 3.4.

Now we return to our original task, which was to implement the `gcd` function implicit in equation (2.1). Once we recognize that (i) Python has no problem with recursion, and (ii) a mod b is implemented as `a%b`, then a minimal thought solution suggests itself, as in the following snippet. (Ignore for the time being lines 14–18.)

```
1  # File gcd.py Implementing the GCD Euclidean algorithm.
2
3  """ Module gcd: contains two implementations of the Euclid
4     GCD algorithm, gcdr and gcd.
5  """
6
7  def gcdr(a,b):
8     """ Euclidean algorithm, recursive vers., returns GCD. """
9     if b==0:
```

https://doi.org/10.1017/CBO9781107447875.003 Published online by Cambridge University Press

```
10          return a
11      else:
12          return gcdr(b,a%b)
13
14  def gcd(a,b):
15      """ Euclidean algorithm, non-recursive vers., returns GCD. """
16      while b:
17          a,b=b,a%b
18      return a
19
20  ##############################################################
21  if __name__ == "__main__":
22      import fib
23
24      for i in range(984):
25          print i, ' ', gcdr(fib.fib(i),fib.fib(i+1))
```

The only real novelty in this snippet is the **import** `fib` statement in line 22, and we have already discussed its effect above. The number of times the loop in lines 24 and 25 is executed is crucial. As printed, this snippet should run in a fraction of a second. Now change the parameter 984 in line 24 to 985, save the file, and apply `run gcd` again. You should find that the output appears to be in an infinite loop, but be patient. Eventually, the process will terminate with an error statement that the maximum recursion depth has been exceeded. While Python allows recursion, there is a limit on the number of self-calls that can be made.

This limitation may or may not be a problem for you. But it is worth a few moments thought to decide whether we could implement Euclid's algorithm (2.1) without using recursion. I offer one possible solution in the function `gcd` implemented in lines 14–18 of the snippet. Lines 16 and 17 define a *while loop*, note the colon terminating line 16. Between **while** and the colon, Python expects an expression which evaluates to one of the Boolean values `True` or `False`. As long as `True` is found, the loop executes line 17, and then retests the expression. If the test produces `False`, then the loop terminates and control passes to the next statement, line 18. In the expected context, `b` will always be an integer, so how can an integer take a Boolean value? The answer is remarkably simple. The integer value zero is always coerced to `False`, but all non-zero values coerce to `True`. Thus the loop terminates when `b` becomes zero, and then the function returns the value `a`. This is the first clause of (2.1). The transformation in line 17 is the second clause, and so this function implements the algorithm. It is shorter than the recursive function, can be called an arbitrary number of times and, as we shall see, runs faster.

So was the expenditure of thought worthwhile. Using the `run` command, we can obtain revealing statistics. First, edit the snippet to make 980 loops with the `gcdr` function, and save it. Now invoke `run -t gcd` to obtain the time spent. On my machine, the "User time" is 0.31s. Yours will vary, but it is relative timings that matter. The "Wall time" reflects the display overhead and is not relevant here. Next invoke `run -p gcd`,

https://doi.org/10.1017/CBO9781107447875.003 Published online by Cambridge University Press

which invokes the Python profiler. Although you would need to read the documentation to understand every facet of the resulting display, a little scientific intuition can be very useful. This shows that there were 980 direct calls (as expected) of the function `gcdr`, within a total of 480,691 actual calls. The actual time spent within this function was 0.282s. Next there were 1960 calls (as expected) of the function `fib`, and the time expended was 0.087s. Note that these timings cannot be compared with the older `run -t gcd` ones, because the newer ones include the profiler overhead which is significant. However, we can conclude that about 75% of the time was spent in the function `gcdr`.

Next we need to repeat the exercise for the `gcd` function. Amend line 25 of the snippet to replace `gcdr` by `gcd` and resave the file. Now `run -t gcd` to get a "User time" of 0.19s. The other command `run -p gcd` reveals that the 1960 calls of function `fib` took 0.087s. However, the function `gcd` was only called 980 times (as expected) which occupied 0.093s. Thus `gcd` occupied about 50% of the time taken. Very approximately, these relative timings factor out the profiler overhead. Now 75% of the 0.31s. timing for the recursive version is 0.23s., while 50% of the 0.19s. time for the non-recursive version is 0.095s. Thus the expenditure of thought has produced a shortened code which runs in 40% of the time of the "thoughtless code"! There is a moral here.

https://doi.org/10.1017/CBO9781107447875.003 Published online by Cambridge University Press