# 3 A short Python tutorial

Although Python is a small language it is a very rich one. It is very tempting, when writing a textbook, to spell out all of the ramifications, concept by concept. The obvious example is the introductory tutorial from the originator of Python, Guido van Rossum. This is available in electronic form as the tutorial in your Python documentation or on-line[1] or as hard copy, van Rossum and Drake Jr. (2011). It is relatively terse at 150 printed pages, and does not mention *numpy*. My favourite textbook, Lutz (2009) runs to over 1200 pages, a marathon learning curve, and only mentions *numpy* in passing. It is excellent at explaining the features in detail, but is too expansive for a first course in Python. A similar criticism can be applied to two books with a more scientific orientation, Langtangen (2008) and Langtangen (2009), both around 700 pages with a significant overlap between them. I recommend these various books among many others for reference, but not for learning the language.

Very few people would learn a foreign language by first mastering a grammar textbook and then memorizing a dictionary. Most start with a few rudiments of grammar and a tiny vocabulary. Then by practice they gradually extend their range of constructs and working vocabulary. This allows them to comprehend and speak the language very quickly, and it is the approach to learning Python that is being adopted here. The disadvantage is that the grammar and vocabulary are diffused throughout the learning process, but this is ameliorated by the existence of textbooks, such as those cited in the first paragraph.

## 3.1 Typing Python

Although the narrative can simply be read, it is extremely helpful to have the *IPython* interpreter to hand. For longer code snippets, you may choose to use an editor as well, so that you can save the code. Your choices are described in Section A.2.1 of Appendix A. Some of the code snippets shown here have numbered lines. It is a good idea to type in these (omitting of course the line numbers to the left of the box) and terminating lines with the "return" (RET) key. You are strongly encouraged to try out your own experiments in the interpreter after saving each code snippet.

Every programming language includes *blocks* of code, which consist of one or more lines of code forming a syntactic whole. Python uses rather fewer parentheses () and

[1] It is available at `http://docs.python.org/2/tutorial`.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

braces {} than other languages, and instead uses indentation as a tool for formatting blocks. After any line ending in a colon : a block is required, and it is differentiated from the surrounding code by being consistently indented. Although the amount is not specified, the unofficial standard is four spaces. *IPython* and any Python-aware text editor will do this automatically. To revert to the original indentation level, use the RET key to enter a totally empty line. Removing braces improves readability, but the disadvantage is that each line in a block must have the same indentation as the one before, or a syntax error will occur.

Python allows two forms of *comments*. A hash symbol # indicates that the rest of the current line is a comment, or more precisely a "tweet". A "documentation string" or *docstring* can run over many lines and include any printable character. It is delimited by a pair of triple quotes, e.g.

```
""" This is a very short docstring. """
```

For completeness, we note that we may place several statements on the same line provided we separate them with semicolons, but we should think about readability. Long statements can be broken up with the continuation symbol \. More usefully, if a statement includes a pair of brackets (), we can split the line at any point between them without the need for the continuation symbol. Here are simple examples.

```
a=4; b=5.5; c=1.5+2j; d='a'
e=6.0*a-b*b+\
       c**(a+b+c)
f=6.0*a-b*b+c**(
       a+b+c)
```

## 3.2 Objects and identifiers

Python deals exclusively with objects and identifiers. An *object* may be thought of as a region of computer memory containing both some data and information associated with those data. For a simple object, this information consists of its *type*, and its *identity*,[2] i.e., the location in memory, which is of course machine-dependent. The identity is therefore of no interest for most users. They need a machine-independent method for accessing objects. This is provided by an *identifier*, a label which can be attached to objects. It is made up of one or more characters. The first must be a letter or underscore, and any subsequent characters must be digits, letters or underscores. Identifiers are case-sensitive, `x` and `X` are different identifiers. (Identifiers which have leading and/or trailing underscores have specialized uses, and should be avoided by the beginner.) We must avoid using predefined words, e.g., **`list`**, and should always try to use meaningful identifiers. However, the choice between say `xnew`, `x_new` and `xNew` is a matter of taste. Consider the following code, which should be typed in the interpreter window.

[2] An unfortunate choice of name, not to be confused with the about-to-be-defined identifiers.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

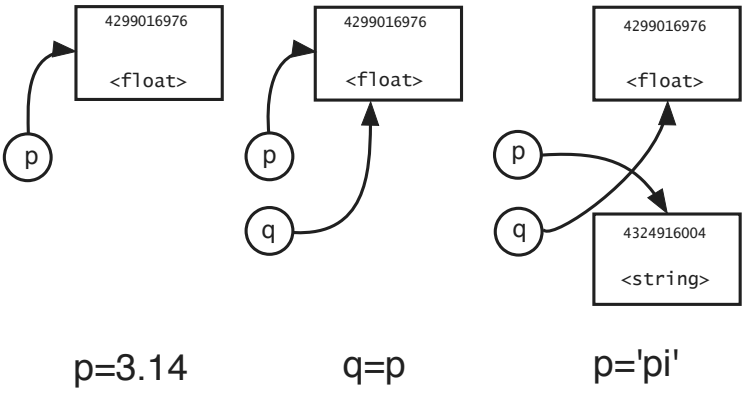


**Figure 3.1** A schematic representation of assignments in Python. After the first command `p=3.14`, the float object `3.14` is created and identifier `p` is assigned to it. Here the object is depicted by its *identity*, a large number, its address in the memory of my computer (highly machine-dependent) where the data are stored, and the *type*. The second command `q=p` assigns identifier `q` to the same object. The third command `p='pi'` assigns `p` to a new "string" object, leaving `q` pointing to the original float object.

```
1 p=3.14
2 p
3 q=p
4 p='pi'
5 p
6 q
```

Note that we never declared the *type* of the object referred to by the identifier `p`. We would have had to declare `p` to be of type "double" in C and "real*8" in Fortran. This is no accident or oversight. A fundamental feature of Python is that the "type" belongs to the object, not to the identifier.[3]

Next in line 3, we set `q=p`. The right-hand side is replaced by whatever object `p` pointed to, and `q` is a new identifier which points to this object, see Figure 3.1. No equality of identifiers `q` and `p` is implied here! Notice that in line 4, we reassign the identifier `p` to a "string" object. However, the original float object is still pointed to by the identifier `q`, see Figure 3.1, and this is confirmed by the output of lines 5 and 6. Suppose we were to reassign the identifier `q`. Then, unless in the interim another identifier had been assigned to `q`, the original "float" object would have no identifier assigned to it and so becomes inaccessible to the programmer. Python will detect this automatically and silently free up the computer memory, a process known as *garbage collection*.

[3] The curious can find the type of an object with identifier `p` with the command **`type`**`(p)` and its identity with **`id`**`(p)`.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

Because of its importance in what follows we emphasize the point that the basic building block in Python is, in pseudocode,

```
<identifier>=<object>
```

which will appear over and over again. As we have already stated earlier, the type of an object "belongs" to the object and not to any identifier assigned to it.

Since we have introduced a "float" , albeit informally, we turn next to a simple class of objects.

## 3.3 Numbers

Python contains three simple types of number objects, and we introduce a fourth, not so simple, one.

### 3.3.1 Integers

Python refers to integers as **int**s . Although early versions only supported integers in the range $[-2^{31}, 2^{31}-1]$, in recent versions the range is considerably larger and is limited only by the availability of memory.

The usual operations of addition (+), subtraction (−) and multiplication (∗) are of course available. There is a slight problem with division, for if $p$ and $q$ are integers, $p/q$ may not be an integer. We may assume without loss of generality that $q > 0$ in which case there exist unique integers $m$ and $n$ with

$$p = mq + n, \qquad \text{where} \quad 0 \leqslant n < q.$$

Then *integer division* in Python is defined by `p//q`, which returns `m`. The *remainder* $n$ is available as `p%q`. *Exponentiation* $p^q$ is also available as `p**q`, and can produce a real number if $q < 0$.

### 3.3.2 Real numbers

Floating-point numbers are available as **float**s . In most installations, the default will be approximately 16 digits of precision for floats in the range $(10^{-308}, 10^{308})$. These correspond to *doubles* in the C-family of languages and *real*8* in the Fortran family. The notation for float constants is standard, e.g.

```
-3.14, -314e-2, -314.0e-2, -0.00314E3
```

all represent the same float.

The usual arithmetic rules for addition, subtraction, multiplication, division and exponentiation are available for floats. For the first three, mixed mode operations are implemented seamlessly, e.g, if addition of an int and a float is requested, then the int is automatically *widened* to be a float. The same applies for division if one operand is an

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

int and the other is a float. However, if both are ints, e.g., ±1/5, what is the result? Earlier versions (< 3.0) of Python adopt integer division, `1/5=0` and `-1/5=-1`, while versions ⩾ 3.0 use real division `1/5=0.2`, `-1/5=-0.2`. This is a potential pitfall which is easily avoided. Either use integer division `//` or widen one of the operands to ensure an unambiguous result.

*Widening* of an int to a float is available explicitly, e.g., **`float`**`(4)` will return either `4.` or `4.0`. *Narrowing* of a float to an int is defined by the following algorithm. If $x$ is real and positive, then there exist an integer $m$ and a real $y$ such that

$$x = m + y \qquad \text{where} \quad 0 \leqslant y < 1.0.$$

In Python, this narrowing is implemented via **`int`**`(x)`, which returns `m`. If $x$ is negative, then **`int`**`(x)=-`**`int`**`(-x)`, succinctly described as "truncation towards zero", e.g., **`int`**`(1.4)=1` and **`int`**`(-1.4)=-1`.

In programming languages, we expect a wide range of familiar mathematical functions to be available. Fortran builds in the most commonly used ones, but in the C-family of languages, we need to import them with a statement such as `#include math.h` at the top of the programme. Python also requires a module to be imported, and as an example we consider the `math` module. (Modules are defined in Section 3.4.) Suppose first that we do not know what the `math` module contains. The following snippet first loads the module and then lists the identifiers of its contents.

```
import math
dir(math)
```

To find out more about the individual objects, we can either consult the written documentation or use the built-in help, e.g., in *IPython*

```
math.atan2?              # or help(math.atan2)
```

If one is already familiar with the contents, then a quick-and-dirty-fix is to replace the **`import`** command above by

```
from math import *
```

anywhere in the code before invoking the functions. Then the function mentioned above is available as `atan2(y,x)` rather than `math.atan2(y,x)`, which at first sight looks appealing, but see Section 3.4. Note that unlike C, the **`import`** command can occur anywhere in the programme before its contents are needed.

### 3.3.3 Boolean numbers

For completeness, we include here Boolean numbers or `bool` which are a subset of the ints with two possible values `True` and `False`, roughly equivalent to 1 (one) and 0 (zero).

Suppose that `box` and `boy` refer to bools. Then expressions such as "**`not`** `box`", "`box` **`and`** `boy`" and "`box` **`or`** `boy`" take on their usual meanings.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

The standard *equality* operators are defined for ints and floats `x`, `y`, e.g., `x==y` (equality), `x!=y` (inequality). As a simple exercise, to remind you of the limitations of Python *floats*, guess the result of the following line, next type it and then explain the result.

```
math.tan(math.pi/4.0)==1.0
```

However, for the comparison operators "`x>y`", "`x>=y`", "`x<y`" and "`x<=y`", widening takes place if necessary. Unusually, but conveniently, *chaining* of comparison operators is allowed, e.g., "`0<=x<1<y>z`" is equivalent to

```
(0<=x) and (x<1) and (1<y) and (y>z)
```

Note that there is no comparison between `x` and `z` in this example.

### 3.3.4 Complex numbers

We have introduced three classes of numbers which form the simplest types of Python object. These are the basis for classes of numbers that are more complicated. For example *rational* numbers can be implemented in terms of pairs of integers. For our purposes, a probably more useful class is that of complex numbers which is implemented in terms of a pair of real numbers. Whereas mathematicians usually denote $\sqrt{-1}$ by $i$, many engineers prefer $j$, and Python has adopted the latter approach. Thus a Python `complex` number can be defined explicitly by, e.g., `c=1.5-0.4j`. Note carefully the syntax: the `j` (or equivalently `J`) follows the float with no intervening "$*$". Alternatively, a pair of floats `a` and `b` can be widened to a complex number via `c=complex(a,b)`. We can narrow a complex; e.g., with `c` as in the last sentence, `c.real` returns `a` and `c.imag` returns `b`. Another useful object is `c.conjugate()`, which returns the complex conjugate of `c`. The syntax of complex attributes will be explained in Section 3.10.

The five basic arithmetic operations work for Python complex numbers, and in mixed mode widening is done automatically. The library of mathematical functions for complex arguments is available. We need to import from `cmath` rather than `math`. However, for obvious reasons, the comparison operations involving ordering described above, are not defined for complex numbers, although the equality and inequality operators are available.

You have now seen enough of Python to use the interpreter as a sophisticated five function calculator, and you are urged to try out a few examples of your own.

## 3.4 Namespaces and modules

While Python is running, it needs to keep a list of those identifiers which have been assigned to objects. This list is called a *namespace*, and as a Python object it too has an identifier. For example, while working in the interpreter, the namespace has the unmemorable name `__main__`.

One of the strengths of Python is its ability to include files of objects, functions

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

etc., written either by you or someone else. To enable this inclusion, suppose you have created a file containing objects, e.g., `obj1`, `obj2` that you want to reuse. The file should be saved as, e.g., `foo.py`, where the `.py` ending is mandatory. (Note that with most text editors you need this ending for the editor to realize that it is dealing with Python code.) This file is then called a *module*.

This module can be imported into subsequent sessions via

```
import foo
```

(When the module is first imported, it is compiled into bytecode and written back to storage as a file `foo.pyc`. On subsequent imports, the interpreter loads this precompiled bytecode unless the modification date of `foo.py` is more recent, in which case a new version of the file `foo.pyc` is generated automatically.)

One effect of this import is to make the namespace of the module available as `foo`. Then the objects from `foo` are available with, e.g., identifiers `foo.obj1` and `foo.obj2`. If you are absolutely sure that `obj1` and `obj2` will not clash with identifiers in the current namespace, you can import them via

```
from foo import obj1, obj2
```

and then refer to them as `obj1` etc.

The "quick-and-dirty-fix" of Section 3.3.2 is equivalent to the line

```
from foo import *
```

which imports *everything* from the module `foo`'s namespace. If an identifier `obj1` already existed, it will be overwritten by this import process. For example, suppose we had an identifier `gamma` referring to a float. Then

```
from math import *
```

overwrites this and `gamma` now refers to the (real) gamma-function. A subsequent

```
from cmath import *
```

overwrites `gamma` with the (complex) gamma-function! Note too that import statements can appear anywhere in Python code, and so chaos is lurking if we use this option.

Except for quick, exploratory work in the interpreter, it is far better to modify the import statements as, e.g.,

```
import math as re
import cmath as co
```

so that in the example above `gamma`, `re.gamma` and `co.gamma` are all available.

We now have sufficient background to explain the mysterious code line

```
if __name__ == "__main__"
```

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

which occurred in both of the snippets in Section 2.6. The first instance occurred in a file `fib.py`. Now if we import this module into the interpreter, its name is `fib` and not `__main__` and so the lines after this code line will be ignored. However, when developing the functions in the module, it is normal to make the module available directly, usually via the `%run` command. Then, as explained at the start of this section, the contents are read into the `__main__` namespace. Then the **`if`** condition of the code line is satisfied and the subsequent lines will be executed. In practice, this is incredibly convenient. While developing a suite of objects, e.g., functions, we can keep the ancillary test functions nearby. In production mode via **`import`**, these ancillary functions are effectively "commented out".

## 3.5 Container objects

The usefulness of computers is based in large part on their ability to carry out repetitive tasks very quickly. Most programming languages therefore provide container objects, often called arrays, which can store large numbers of objects of the same type, and retrieve them via an indexing mechanism. Mathematical vectors would correspond to one-dimensional arrays, matrices to two-dimensional arrays etc. It may come as a surprise to find that the Python core language has no array concept. Instead, it has container objects which are much more general, *lists*, *tuples*, *strings* and *dictionaries*. It will soon become clear that we can simulate an array object via a *list*, and this is how numerical work in Python used to be done. Because of the generality of *lists*, such simulations took a great deal longer than equivalent constructions in Fortran or C, and this gave Python a deservedly poor reputation for its slowness in numerical work. Developers produced various schemes to alleviate this, and they have now standardized on the *numpy* add-on module to be described in Chapter 4. Arrays in *numpy* have much of the versatility of Python *lists*, but are implemented behind the scenes as arrays in C significantly reducing, but not quite eliminating, the speed penalty. However, in this section we describe the core container objects in sufficient detail for much scientific work. Numerical arrays are deferred to the next chapter, but the reader particularly interested in numerics will need to understand the content of this section, because the ideas carry forward into the next chapter.

### 3.5.1 *Lists*

Consider typing the code snippet

```
[1,4.0,'a']
u=[1,4.0,'a']
v=[3.14,2.78,u,42]
v
len(v)
len?              # or help(len)
```

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

```
7  v*2
8  v+u
9  v.append('foo')
10 v
```

Line 1 is our first instance of a Python *list*, an ordered sequence of Python objects separated by commas and surrounded by square brackets. It is itself a Python object, and can be assigned to a Python identifier, as in line 2. Unlike arrays, there is no requirement that the elements of a *list* be all of the same type. In lines 3 and 4, we see that in creating the *list* an identifier is replaced by the object it refers to, e.g., one *list* can be an element in another. The beginner should consult Figure 3.1 again. It is the object, not the identifier, which matters. In line 5, we invoke a Python function which returns the length of the *list*, here 4. (Python functions will be discussed in Section 3.8. In the meantime we can find what **`len`** does by typing the line **`len`**`?` in *IPython*.) We can replicate *lists* by constructions like line 7, and concatenate *lists* as in line 8. We can append items to the ends of *lists* as in line 9. Here `v.append()` is another useful function. You should try `v.append?` or **`help`**`(v.append)` to see a description of it. Incidentally, **`list.`** followed by TAB completion or **`help(list)`** will give a catalogue of functions intrinsic to *lists*. Ignore those starting with double underscores and look particularly at those at the end of the catalogue, since these are likely to be the ones most useful for beginners.

### 3.5.2 *List* indexing

We can access elements of `u` by *indexing*, `u[i]` where $i \in [0, len(u))$ is an integer. Note that indexing starts with `u[0]` and ends with **`len`**`(u)-1]`. So far, this is very similar to what is available for arrays in, e.g., C or Fortran. However, a Python *list* such as `u` "knows" its length, and so we could also index the elements, in reverse order, by `u[`**`len`**`(u)-k]` where $k \in (0, len(u)]$, which Python abbreviates to `u[-k]`. This turns out to be very convenient. For example, not only is the first element of any *list* `w` referred to by `w[0]`, but the last element is `w[-1]`. The middle line of Figure 3.2 shows both sets of indices for a *list* of length 8. Using the code snippet above, you might like to guess the objects corresponding to `v[1]` and `v[-3]`, and perhaps use the interpreter to check your answers.

At first sight, this may appear to be a trivial enhancement, but it becomes very powerful when coupled to the concepts of slicing and mutability, which we address next. Therefore, it is important to make sure you understand clearly what negative indices represent.

### 3.5.3 *List* slicing

Given a *list* `u`, we can form more *lists* by the operation of *slicing*. The simplest form of a slice is `u[start:end]`, which is a *list* of length `end-start`, as shown in Figure 3.2. If the slice occurs on the right-hand side of an assignation, then a **new** *list* is created. For example, `su=u[2:6]` generates a new *list* with four elements, where `su[0]` is initialized

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

with `u[2]`. If the slice occurs on the left, no new *list* is generated. Instead, it allows us to change a block of values in an existing *list*. There are important new constructs here which may well be unfamiliar to C and Fortran users.

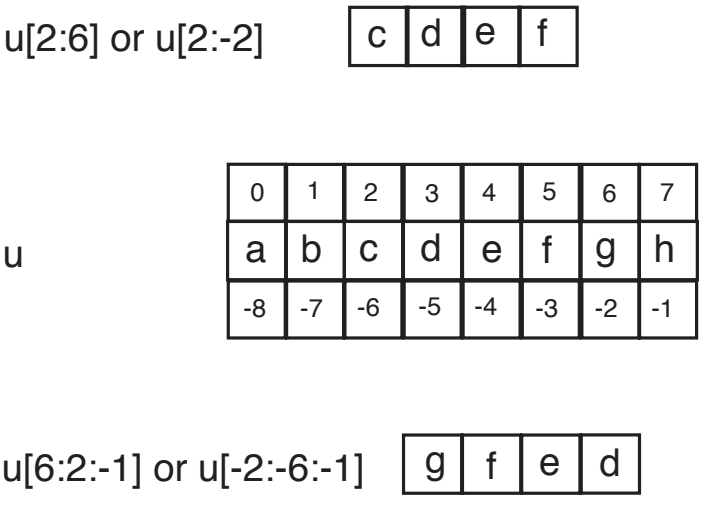


**Figure 3.2** Indices and slicing for a *list* `u` of length 8. The middle line shows the contents of `u` and the two sets of indices by which the elements can be addressed. The top line shows the contents of a slice of length 4 with conventional ordering. The bottom line shows another slice with reversed ordering.

Consider the simple example below which illustrates the possibilities, and carry out further experiments of your own.

```
u=[0,1,2,3,4,5,6,7]
su=u[2:4]
su
su[0]=17
su
u
u[2:4]=[3.14,'a']
u
```

If `start` is zero, it may be omitted, e.g., `u[ :-1]` is a copy of `u` with the last element omitted. The same applies at the other end, `u[1: ]` is a copy with the first element omitted and `u[:]` is a copy of `u`. In these examples, we are assuming that the slice occurs on the right-hand side of an assignation. The more general form of *slicing* is `su = u[start:end:step]`. Then `su` contains the elements `u[start]`, `u[start+step]`, `u[start+2*step]`, ..., as long as the index is less than `end`. Thus with the list `u` chosen as in the example above we would have `u[2:-1:2]=[2,4,6]`. A particularly useful

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

choice is `step=-1`, which allows traversal of the *list* in the reverse direction. See Figure 3.2 for an example.

### 3.5.4 *List* mutability

For any container object `u`, it may be possible to modify an element or a slice, without any apparent operation applied to the object's identifier. Such objects are said to be *mutable*. As an example, consider a politician's promises. In particular, *lists* are mutable. There is a trap here for the unwary. Consider the code

```
1  a=4
2  b=a
3  b='foo'
4  a
5  b
6  u=[0,1,4,9,16]
7  v=u
8  v[2]='foo'
9  v
10 u
```

The first five lines should be comprehensible: `a` is assigned to the object `4`; so is `b`. Then `b` is assigned to the object `'foo'`, and this does not change `a`. In line 6, `u` is assigned to a *list* object and so is `v` in line 7. Because *lists* are mutable, we may change the second element of the *list* object in line 8. Line 9 shows the effect. But `u` is pointing to the same object (see Figure 3.1) and it too shows the change in line 10. While the logic is clear, this may not be what was intended, for `u` was never changed explicitly.

It is important to remember the assertion made above: a slice of a *list* is always a **new** object, even if the dimensions of the slice and the original *list* agree. Therefore, compare lines 6–10 of the code snippet above with

```
1  u=[0,1,4,9,16]
2  v=u[ : ]
3  v[2]='foo'
4  v
5  u
```

Now line 2 makes a slice object, which is a *copy*[4] of the object defined in line 1. Changes to the `v`-*list* do not alter the `u`-*list* and vice-versa.

*Lists* are very versatile objects, and there exist many Python functions which can generate them. We shall discuss *list* generation at many points in the rest of this book.

[4] For the sake of completeness, we should note that this is a *shallow copy*. If `u` contains an element which is mutable, e.g., another *list* `w`, the corresponding element of `v` still accesses the original `w`. To guard against this, we need a *deep copy* to obtain a distinct but exact copy of both `u` and its current contents, e.g., `v=u.copy()`.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

### 3.5.5 *Tuples*

The next container to be discussed is the *tuple*. Syntactically it differs from a *list* only by using () instead of [] as delimiters, and indexing and slicing work as for *lists*. However, there is a fundamental difference. We cannot change the values of its elements, a *tuple* is *immutable*. At first sight, the *tuple* would appear to be entirely redundant. Why not use a *list* instead? The rigidity of a *tuple* however has an advantage. We can use a *tuple* where a scalar quantity is expected, and in many cases we can drop the brackets () when there is no ambiguity, and indeed this is the commonest way of utilizing *tuples*. Consider the snippet below, where we have written a *tuple* assignation in two different ways.

```
(a,b,c,d)=(4,5.0,1.5+2j,'a')
a,b,c,d = 4,5.0,1.5+2j,'a'
```

The second line shows how we can make multiple scalar assignments with a single assignation operator. This becomes extremely useful in the common case where we need to swap two objects, or equivalently two identifiers, say `a` and `L1`. The conventional way to do this is

```
temp=a
a=L1
L1=temp
```

This would work in any language, assuming `temp`, `a` and `L1` all refer to the same type. However

```
a,L1 = L1,a
```

does the same job in Python, is clearer, more concise and works for arbitrary types.

Another use, perhaps the most important one for *tuples*, is the ability to pass a variable number of arguments to a function, as discussed in Section 3.8.4. Finally, we note a feature of the notation which often confuses the beginner. We sometimes need a *tuple* with only one element, say `foo`. The construction `(foo)` strips the parentheses and leaves just the element. The correct *tuple* construction is `(foo,)`.

### 3.5.6 *Strings*

Although we have already seen *string*s in passing, we note that Python regards them as immutable container objects for alphanumeric characters. There is no comma separator between items. The delimeters can be either single quotes or double quotes, but not a mixture. The unused delimiter can occur within the *string*, e.g.

```
s1="It's time to go"
s2=' "Bravo!" he shouted.'
```

Indexing and slicing work in the same way as for *lists*. *Strings* will turn out to be very useful in producing formatted output from the print function (see Section 3.8.6).

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

### 3.5.7 *Dictionaries*

As we have seen, a *list* object is an ordered collection of objects, A *dictionary* object is an unordered collection. Instead of accessing the elements by virtue of their position, we have to assign a keyword, an immutable object, usually a *string*, which identifies the element. Thus a *dictionary* is a collection of pairs of objects, where the first item in the pair is a *key* to the second. A key-object pair is written as `key:object`. We fetch items via keys rather than position. The *dictionary* delimeters are the braces { }. Here is a simple example that illustrates the basics.

```
1 empty={}
2 parms={'alpha':1.3,'beta':2.74}
3 #parms=dict(alpha=1.3,beta=2.74)
4 parms['gamma']=0.999
5 parms
```

The commented-out line 3 is an alternative to line 2. This illustrates the main numerical usage of *dictionaries*, to pass around an unknown and possibly variable number of parameters. Another important use will be keyword arguments in functions (see Section 3.8.5). A more sophisticated application is discussed in Section 7.5.2.

*Dictionaries* are extremely versatile, have many other properties and have plenty of applications in contexts which are more general, see the textbooks for details.

## 3.6 Python *if* statements

Normally, Python executes statements in the order that they are written. The **`if`** statement is the simplest way to modify this behaviour, and exists in every programming language. In its simplest form the Python syntax is

```
if < Boolean expression >:
    <block 1>
<block 2>
```

Here the expression must produce a `True` or `False` result. If `True`, then block1 is executed, followed by block 2, the rest of the programme; if the expression is `False`, then only block 2 is executed. Note that the **`if`** statement ends with a colon : indicating that a block must follow. The absence of delimeters such as braces makes it much easier to follow the logic, but the price required is careful attention to the indentation. Any Python-aware editor should take care of this automatically.

Here is a very simple example that also illustrates *strings* in action.

```
x=0.47
if 0 < x < 1:
    print "x lies between zero and one."
y=4
```

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

A simple generalization is

```
if < Boolean expression >:
    <block 1>
else:
    <block 2>
<block 3>
```

which executes either block 1 or block 2 followed by block 3.

We can chain **if** statements and there is a convenient abbreviation **elif**. Note that all of the logical blocks must be present. If nothing is required in a particular block, it should consist of the single command **pass**, e.g.

```
if < Boolean expression 1>:
    <block 1>
elif <Boolean expression 2>:
    <block 2>
elif <Boolean expression 3>:
    pass
else:
    <block 4>
<block 5>
```

Here the truth of Boolean expression 1 leads to the execution of blocks 1 and 5. If expression 1 is false and expression 2 is true, we get blocks 2 and 5. However if expressions 1 and 2 are false but 3 is true, only block 5 is executed, but if all three expressions are false, both blocks 4 and 5 are executed.

A situation which arises quite often is a construction with terse expressions, e.g.

```
if x>=0:
    y=f
else:
    y=g
```

As in the C-family of languages, there is an abbreviated form. The snippet above can be shortened in Python with no loss of clarity to

```
y=f if x>=0 else g
```

## 3.7 Loop constructs

Computers are capable of repeating sequences of actions with great speed, and Python has two loop constructs, **for** and **while** loops.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

### 3.7.1 The Python *for* loop

This, the simplest loop construct, exists in all programming languages, as `for` loops in the C-family and as `do` loops in Fortran. The Python *loop* construct is a generalized, sophisticated evolution of these. Its simplest form is

```
for <iterator> in <iterable>:
    <block>
```

Here `<iterable>` is any container object. The `<iterator>` is any quantity which can be used to access, term by term, the elements of the container object. If `iterable` is an ordered container, e.g., a *list* `a`, then `iterator` could be an integer `i` within the range of the *list*. The block above would include references to, e.g., `a[i]`.

This sounds very abstract and needs to be elucidated. Many of the conventional C and Fortran uses will be postponed to Chapter 4 because the core Python being described here can offer only a very inefficient implementation of them. We start with a simple, but unconventional example.[5]

```
c = 4
for c in "Python":
    print c

c
```

Using `c` as the loop iterator here overwrites any previous use of `c` as an identifier. For each character in the *string* iterable, the block is executed, which here merely prints its value. When there are no more characters, the loop exits, and `c` refers to its last loop value.

At first sight it, looks as though `<iterator>` and `<iterable>` have to be single objects but, as will become normal, we can circumvent this requirement by using *tuples*. For example, suppose `Z` is a list of *tuples* each of length 2. Then a loop with two iterators can be constructed via

```
for (x,y) in Z:
    <block>
```

and is perfectly permissible. For another generalization, consider the **`zip`** function introduced in Section 4.4.1.

Before we can exhibit usage that is more traditional, we need to introduce the built-in Python **`range`** function. Its general form is

```
range(start,end,step)
```

which generates the *list* of integers `[start,start+step,start+2*step,...]` as long as each integer is less than `end`. (We have seen this before in the discussion of slicing in

[5] If you are using a version of Python ⩾ 3.0, this code will fail. See Section 3.8.6 to find out why, and the trivial change needed to repair the snippet.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

Section 3.5.3.) Here, `step` is an optional argument, which defaults to one, and `start` is optional, defaulting to zero. Thus **`range`**`(4)` yields `[0,1,2,3]`. Consider the following example

```
L=[1,4,9,16,25,36]
for it in range(len(L)):
    L[it]+=1
L
```

Note that the loop is set up at the execution of the **`for`** statement. The block can change the iterator, but not the loop. Try the simple example

```
for it in range(4):
    it*=2
    print it
it
```

It should be emphasized that although the loop bodies in these examples are trivial, this need not be so. Python offers two different ways of dynamically altering the flow of control while the loop is executing. In a real-life situation, they could of course both occur in the same loop.

### 3.7.2 The Python *continue* statement

Consider the schematic example

```
for <iterator> in <iterable>:
    <block1>
    if <test1>:
        continue
    <block2>
<block5>
```

Here `<test1>` returns a Boolean value, presumably as a result of actions in `<block1>`. On each pass, it is checked, and if `True` control passes to the top of the loop, so that `<iterator>` is incremented, and the next pass commences. If `<test1>` returns `False`, then `<block2>` is executed, after which control passes to the top of the loop. At the termination of the loop (in the usual way), `<block5>` is executed.

### 3.7.3 The Python *break* statement

The **`break`** statement allows for premature ending of the loop and, optionally via the use of an **`else`** clause, different outcomes. The basic syntax is

```
for <iterator> in <iterable>:
    <block1>
    if <test2>:
```

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

```
        break
    <block2>
else:
    <block4>
<block5>
```

If on any pass `test2` produces a `True` value, the loop is exited and control passes to `<block5>`. If `<test2>` gives `False`, then the loop terminates in the usual way, and control passes first to `<block4>` and ultimately to `<block5>`. The author finds the flow of control here to be counterintuitive, but the use of the **`else`** clause in this context is both optional and rare. Here is a simple example.

```
y=107
for x in range(2,y):
   if y%x == 0:
       print y, " has a factor ", x
       break
else:
    print y, " is prime."
```

### 3.7.4 *List* comprehensions

A task which arises surprisingly often is the following. We have a *list* `L1`, and need to construct a second *list* `L2` whose elements are a fixed function of the corresponding elements of the first. The conventional way to do this is via a **`for`** loop. For example, let us produce an itemwise squared *list*.

```
L1=[2,3,5,7,11,14]
L2=[]                # Empty list
for i in range(len(L1)):
    L2.append(L1[i]**2)

L2
```

However, Python can execute the loop in a single line via a *list comprehension*.

```
L1=[2,3,5,7,11,14]
L2=[x**2 for x in L1]
L2
```

Not only is this shorter, it is faster, especially for long *lists*, because there is no need to construct the explicit for-loop.

*List* comprehensions are considerably more versatile than this. Suppose we want to build `L2`, but only for the odd numbers in `L1`.

```
L2=[x*x for x in L1 if x%2]
```

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

Suppose we have a *list* of points in the plane, where the coordinates of the points are stored in *tuples*, and we need to form a *list* of their Euclidean distances from the origin.

```
import math
lpoints=[(1,0),(1,1),(4,3),(5,12)]
ldists=[math.sqrt(x*x+y*y) for (x,y) in lpoints]
```

Next suppose that we have a rectangular grid of points with the $x$-coordinates in one *list* and the $y$-coordinates in the other. We can build the distance *list* with

```
l_x=[0,2,3,4]
l_y=[1,2]
l_dist=[math.sqrt(x*x+y*y) for x in l_x for y in l_y]
```

*List* comprehension is a Python feature which, despite its initial unfamiliarity, is well worth mastering.

### 3.7.5 Python *while* loops

The other extremely useful loop construct supported by Python is the **while** loop. In its simplest form, the syntax is

```
while <test>:
    <block1>
<block2>
```

Here <test> is an expression which yields a Boolean object. If it is True, then <block1> is executed. Otherwise, control passes to <block2>. Each time <block1> reaches its end, <test> is re-evaluated, and the process repeated. Thus the construct

```
while True :
    print "Type Control-C to stop this!"
```

will, without outside intervention, run for ever.

Just as with **for** loops, **else**, **continue** and **break** clauses are available, and the last two work in the same way, either curtailing execution of the loop or exiting it. Note in particular the code snippet above can become extremely useful with a **break** clause These were discussed in Section 3.7.3.

## 3.8 Functions

A *function* (or *subroutine*) is a device which groups together a sequence of statements that can be executed an arbitrary number of times within a programme. To increase generality, we may supply input arguments which can change between invocations. A function may, or may not, return data.

In Python, a function is, like everything else, an *object*. We explore first the basic

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

syntax and the concept of scope and only finally in Sections 3.8.2–3.8.5 the nature of input arguments. (This may seem an illogical order, but the variety of input arguments is extremely rich.)

### 3.8.1 Syntax and scope

A Python function may be defined *anywhere* in a programme before it is actually used. The basic syntax is outlined in the following piece of pseudocode

```
def <name>(<arglist>):
   <body>
```

The **def** denotes the start of a function definition; <name> assigns an identifier or name to address the object. The usual rules on identifier names apply, and of course the identifier can be changed later. The brackets () are mandatory. Between them, we may insert zero, one or more variable names, separated by commas, which are called *arguments*. The final colon is mandatory.

Next follows the *body* of the function, the statements to be executed. As we have seen already, such blocks of code have to be indented. The conclusion of the function body is indicated by a return to the same level of indentation as was used for the **def** statement. In rare circumstances we may need to define a function, while postponing the filling-out of its body. In this preliminary phase, the body should be the single statement **pass**. It is not mandatory, but conventional and highly recommended, to include a *docstring*, a description of what the function does, between the definition and body. This is enclosed in a pair of triple quotes, and can be in free format extending over one or more lines. The inclusion of a docstring may seem to be an inessential frippery, but it is not. The author of the **len** function included one which you accessed in Section 3.5.1 via **len**?. Treat your readership (most likely **you**) with the same respect!

The body of the function definition introduces a new private namespace which is destroyed when the execution of the body code terminates. When the function is invoked, this namespace is populated by the identifiers introduced as arguments in the **def** statement, and will point to whatever the arguments pointed at when the function was invoked. New identifiers introduced in the body also belong in this namespace. Of course, the function is defined within a namespace which contains other external identifiers. Those which have the same names as the function arguments or identifiers already defined in the body do not exist in the private namespace, because they have been replaced by those private arguments. The others are visible in the private namespace, but it is strongly recommended not to use them unless the user is absolutely sure that they will point to the same object on every invocation of the function. In order to ensure portability when defining functions, try to use only the identifiers contained in the argument list and those which you have defined within, and intrinsic to, the private namespace.

Usually we require the function to produce some object or associated variable, say `y`, and this is done with a line **return** `y`. The function is exited after such a statement and so the **return** statement will be the last executed, and hence usually the last statement

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

in the function body. In principle, y should be a scalar, but this is easily circumvented by using a *tuple*. For example, to return three quantities, say u, v and w one should use a *tuple*, e.g., **return** (u,v,w) or even **return** u,v,w. If however there is no return statement Python inserts an invisible **return** None statement. Here None is a special Python variable which refers to an empty or void object, and is the "value" returned by the function. In this way, Python avoids the Fortran dichotomy of *functions* and *subroutines*.

Here are some simple toy examples to illustrate a number of points. It is worthwhile typing them into the interpreter so that after verifying the points made here you can experiment further.

```
1 def add_one(x):
2     """ Takes x and returns x + 1. """
3     x=x+1
4     return x
5
6 x=23
7 add_one?                # or help(add_one)
8 add_one(0.456)
9 x
```

In lines 1–4, we define the function add_one(x). There is just one argument x introduced in line 1, and the object which it references is changed in line 3. In line 6, we introduce an integer object referenced by x. The next line tests the docstring and the one after that the function. The final line checks the value of x, which remains unchanged at 23 despite the fact that we implicitly assigned x to a float in line 8.

Next consider a faulty example

```
1 def add_y(x):
2     """ Adds y to x and returns x+y. """
3     return x+y
4
5 add_y(0.456)
6 y=1
7 add_y(0.456)
```

In line 5, private x is assigned to 0.456, and line 3 looks for a private y. None is found, and so the function looks for an identifier y in the enclosing namespace. None can be found and so Python stops with an error. After y has been introduced in line 6, the second invocation of add_y in line 7 works as expected. However, this is non-portable behaviour. We can only use the function inside namespaces where a y has already been defined. There are a few cases where this condition will be satisfied, but in general this type of function should be avoided.

Better coding is shown in the following example which also shows how to return multiple values via a *tuple*, and also that functions are objects.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

```
def add_x_and_y(x, y):
    """ Add x and y and return them and their sum. """
    z = x+y
    return x,y,z

a,b,c = add_x_and_y(1,0.456)
a
b
c
f=add_x_and_y
f(1,0.456)
add_x_and_y
f
```

The identifier z is private to the function, and is lost after the function has been left. Because we assigned c to the object pointed to by z, the object itself is not lost when the identifier z disappears. Lines 10–13 show that functions are objects, and we can assign new identifiers to them. (It may be helpful to look again at Figure 3.1 at this point.)

In all of these examples, the objects used as arguments have been immutable, and so have not been changed by the function's invocation. This is not quite true if the argument is a mutable container, as is shown in the following example.

```
L = [0,1,2]
id(L)
def add_with_side_effects(M):
    """ Increment first element of list. """
    M[0]+=1

add_with_side_effects(L)
L
id(L)
```

The *list* L has not been changed. However, its contents can, and have been changed, with no assignment operator (outside the function body). This is a *side effect* which is harmless in this context, but can lead to subtle hard-to-identify errors in real-life code. The cure is to make a private copy

```
L = [0,1,2]
id(L)
def add_without_side_effects(M):
    """ Increment first element of list. """
    MC=M[ : ]
    MC[0]+=1
    return MC
```

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

```
8
9  L = add_without_side_effects(L)
10 L
11 id(L)
```

In some situations, there will be an overhead in copying a long *list*, which may detract from the speed of the code, and so there is a temptation to avoid such copying. However, before using functions with side effects remember the adage "premature optimization is the root of all evil", and use them with care.

### 3.8.2 Positional arguments

This is the conventional case, common to all programming languages. As an example, consider

```
def foo1(a,b,c):
    <body>
```

Every time `foo1` is called, precisely three arguments have to be supplied. An example might be `y=foo1(3,2,1)`, and the obvious substitution by order takes place. Another way of calling the function might be `y=foo1(c=1,a=3,b=2)`, which allows a relaxation of the ordering. It is an error to supply any other number of arguments.

### 3.8.3 Keyword arguments

Another form of function definition specifies *keyword* arguments, e.g.

```
def foo2(d=21.2,e=4,f='a'):
    <block>
```

In calling such a function we can give all arguments, or omit some, in which case the default value will be taken from the **`def`** statement. For example, invoking `foo2(f='b')` will use the default values `d=21.2` and `e=4` so as to make up the required three arguments. Since these are keyword arguments, the order does not matter.

It is possible to combine both of these forms of argument in the same function, provided all of the positional arguments come before the keyword ones. For example

```
def foo3(a,b,c,d=21.2,e=4,f='a')
    <block>
```

Now in calling this function, we must give between three and six arguments, and the first three refer to the positional arguments.

### 3.8.4 Variable number of positional arguments

It frequently happens that we do not know in advance how many arguments will be needed. For example, imagine designing a `print` function, where you cannot specify

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

in advance how many items are to be printed. Python uses *tuples* to resolve this issue. Here is an example to illustrate syntax, method and usage.

```
1  def average(*args):
2      """ Return mean of a non-empty tuple of numbers. """
3      print args
4      sum=0.0
5      for x in args:
6          sum+=x
7      return sum/len(args)
8
9  average(1,2,3,4)
10 average(1,2,3,4,5)
```

It is traditional, but not obligatory, to call the *tuple* `args` in the definition. The asterisk is however mandatory. Line 3 is superfluous, and is simply to illustrate that the arguments supplied really do get wrapped into a *tuple*. Note that by forcing **`sum`** to be real in line 4, we ensure that the division in line 7 works as we would expect, even though the denominator is an integer.

### 3.8.5 Variable number of keyword arguments

Python has no problem coping with a function which takes a certain fixed number of positional arguments, followed by an arbitrary number of positional arguments, followed by an arbitrary number of keyword arguments, since the ordering "positional before keyword" is preserved. As the previous section showed, the additional positional arguments are packed into a *tuple* (identified by an asterisk). The additional keyword arguments are wrapped into a *dictionary* (identified by a repeated asterisk). The following example exemplifies the process.

```
1  def show(a, b, *args, **kwargs):
2     print a, b, args, kwargs
3
4  show(1.3,2.7,3,'a',4.2,alpha=0.99,gamma=5.67)
```

Beginners are not likely to want to use proactively all of these types of arguments. However, they will see them occasionally in the docstrings of library functions, and so it is useful to know what they are.

### 3.8.6 The Python *print* function

Until the last two sections, we have not needed a print function. Working in the interpreter we can "print" any Python object by simply typing its identifier. However, once we start to write functions and programmes, we do need such a function. There is a small complication here. In versions of Python $< 3.0$, **`print`** looks like a command, and would be invoked by, e.g.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

```
print <things to be printed>
```

In versions of Python ⩾ 3.0, **print** is implemented as a function, and the code line above becomes

```
print(<things to be printed>)
```

At the time of writing, *numpy* and its extensions are only available using the earlier versions. Since earlier versions of Python may eventually be deprecated, numerical users face a potential obsolescence dilemma, which is however resolved easily in one of two ways. The **print** function expects a variable number of arguments, implicitly a *tuple*. If we use the second of the two code lines above with earlier versions of Python, then the print command sees an explicit *tuple* because of the () and prints <things to be printed> surrounded by brackets. If the extra brackets prove annoying, then the alternative approach is to include the line

```
from __future__ import print_function
```

at the top of the code, which will remove them when using the Python ⩾ 3.0 **print** function within Python < 3.0 code.

The print command expects a *tuple* as its argument. Thus assuming `it` refers to an integer, and `y` to a float, we might write, dropping the *tuple* delimeters, simply

```
print "After iteration ",it,", the solution was ",y
```

Here we have no control over the formatting of the two numbers, and this is a potential source of problems, e.g., when producing a table, where uniformity is highly desirable. A first attempt at a more sophisticated approach is

```
print "After iteration %d, the solution was %f" % (it,y)
```

Here the *string* argument is printed with the `%d` term replaced by an integer and the `%f` term replaced by a float, both chosen from the final *tuple* in the argument slot. As written, the output of this version is identical to that of the previous line, but we can improve significantly on that. The format codes summarized below are based on those defined for the `printf` function in the C-family of languages.

We start by considering the format of the integer `it`, which we assume takes the value `41`. If we replace `%d` in the code line with `%5d`, the output will be right justified in a field of 5 characters, i.e., ␣␣␣41, where the character ␣ denotes a space. Similarly `%-5d` will produce left justified output 41␣␣␣. Also `%05d` will give 00041. If the number is negative, then a minus sign counts as a character in the field. This can lead to untidiness when printing both positive and negative integers. We can enforce the printing of a leading plus or minor sign by choosing `%+5d` for right justification and `%+-5d` for left justification. Of course, there is nothing special about the field width 5. It can be replaced by any suitable choice. Indeed, if the exact representation of the integer requires more characters than have been specified, then Python overrides the format instruction so as to preserve accuracy.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

For the formatting of floats, there are three basic possibilities. Suppose `y` takes the value `123.456789`. If we replace `%f` by `%.3f`, the output will be 123.457, i.e., the value is truncated to three decimal places. The format code `%10.3f` prints it right justified in a field of width 10, i.e., ␣␣␣123.457, while `%-10.3f` does the same, left justified. As with integer formatting, a plus sign immediately after the percent sign forces the printing of a leading plus or minus sign. Also `%010.3f` replaces the leading spaces by zeros.

Clearly, the `%f` format will lose precision if `y` is very large or very small, and we consider the case `z=1234567.89`. On the printed page, we might write $z = 1.23456789 \times 10^6$ and in Python `z=1.23456789e6`. Now consider the print format code `%13.4e`, which applied to this `z` produces ␣␣␣1.2346e+06. There are precisely four integers after the decimal point, and the number is printed in a field thirteen characters wide. In this representation, only ten characters are needed and the number is right justified with three space to its left. Just as above, `%-13.4e` produces left justification and `%+13.4e` produces a leading sign, and only two spaces. (`%+-13.4e` does the same with left justification.) If the width is less than the minimum ten required here, then Python will increase it to ten. Finally, replacing 'e' by 'E' in these codes ensures an upper case character, e.g., `%+-13.4E` produces +1.2346E+06␣␣.

Sometimes we need to print a float whose absolute value can vary widely, but we wish to display a particular number of significant digits. With `z` as above, `%.4g` will produce 1.235e+06, i.e., correct to four significant digits. Note that `%g` defaults to `%.6g`. Python will use whichever of the 'e' and 'f' formats is the shorter. Also `%.4G` chooses between the 'E' and 'f' formats.

For completeness, we note that *string* variables are also catered for, e.g., `%20s` will print a *string* in a field of at least 20 characters, with left padding by spaces in needed.

### 3.8.7 Anonymous functions

It should be obvious that the choice of a name for a function's arguments is arbitrary, $f(x)$ and $f(y)$ refer to the same function. On the other hand, we saw in the code snippet for the function `add_x_and_y` in Section 3.8.1 that we could change the name to `f` with impunity. This is a fundamental tenet of mathematical logic, usually described in the formalism of the *lambda-calculus* or *λ-calculus*. There will occur situations in Python coding where the name of the function is totally irrelevant, and Python mimics the lambda calculus. We could have coded `add_x_and_y` as

```
f = lambda x, y : x, y, x+y
```

or just

```
lambda x, y : x, y, x+y
```

See Sections 4.1.5 and 7.5.3 for more realistic cases of anonymous functions.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

## 3.9 Introduction to Python classes

Classes are extremely versatile structures in Python. Because of this, the documentation is both long and complicated. Examples tend to be drawn from non-scientific data-processing and are often either too simple or unduly complicated. The basic idea is that you may have a fixed data structure or object which occurs frequently, together with operations directly associated to it. The Python *class* encapsulates both the object and its operations Our introductory presentation by way of an scientific example is concise but contains many of those features most commonly used by scientists. In this pedagogic context, we have of course to use integer arithmetic.

Our example is that of fractions, which can be thought of as arbitrary precision representations of real numbers. Python does itself include a `fraction` class, and so to avoid confusion our pedagogic example will be called `Frac`. We have in mind to implement a `Frac` as a pair of integers `num` and `den`, where the latter is non-zero. We have to be careful however for 3/7 and 24/56 are usually regarded as the same number. We looked at this particular problem in Section 2.6, and we will need access to the file `gcd.py` created there. The following snippet, which shows a skeleton `Frac` class, relies on it

```
# File frac.py

import gcd

class Frac:
    """ Fractional class. A Frac is a pair of integers num, den
        (with den!=0) whose GCD is 1.
    """

    def __init__(self,n,d):
        """ Construct a Frac from integers n and d.
            Needs error message if d = 0!
        """
        hcf=gcd.gcd(n, d)
        self.num, self.den = n/hcf, d/hcf

    def __str__(self):
        """ Generate a string representation of a Frac. """
        return "%d/%d" % (self.num,self.den)

    def __mul__(self,another):
        """ Multiply two Fracs to produce a Frac. """
        return Frac(self.num*another.num, self.den*another.den)

    def __add__(self,another):
        """ Add two Fracs to produce a Frac. """
```

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

```
27          return Frac(self.num*another.den+self.den*another.num,
28                      self.den*another.den)
29
30      def to_real(self):
31          """ return floating point value of Frac. """
32          return float(self.num)/float(self.den)
33
34  if __name__=="__main__":
35      a=Frac(3,7)
36      b=Frac(24,56)
37      print "a.num= ",a.num, ", b.den= ",b.den
38      print a
39      print b
40      print "floating point value of a is ", a.to_real()
41      print "product= ",a*b,", sum= ",a+b
```

The first novelty here is the **class** statement in line 5. Note the terminating colon. The actual class is defined through indentation and here extends from line 5 to line 32. Lines 6–8 define the class docstring for online documentation. Within the class. we have defined five class functions, which are indented (because they are in the class) and which have indented bodies (as usual).

The first one, lines 10–15, would in other languages be called a "constructor" function. Its purpose is to turn pairs of integers into `Frac` objects. The mandatory name **__init__** is strange, but, as we shall see, it is never used outside the class definition. The first argument is usually called **self**, and it too never appears outside the function definition. This all looks very unfamiliar, so look next at line 35 of the test suite, `a=Frac(3,7)`. This is asking for a `Frac` object for the integer pair 3,7 to be referenced by the identifier `a`. Implicitly, this line invokes the **__init__** function with **self** replaced by `a` and `n` and `d` replaced by 3 and 7. It then computes `hcf`, the GCD of 3 and 7 (here 1), and in line 15 it computes `a.num` as `n/hcf` and `a.den` as `hcf`. These are accessible in the usual way, and in line 37 the first of them gets printed. The same applies to the assignment in line 36, where **__init__** gets called with **self** replaced by `b`.

Almost every class would benefit from having lines like 38–39, where class objects are "printed". This is the purpose of the class string function `__str__`, which is defined in lines 17–19. When line 38 is executed, `__str__` is invoked with **self** replaced by `a`. Then line 19 returns the *string* `"3/7"`, and this is what is printed.

While these first two class functions are more or less essential, we may define many or few functions to carry out class operations. Let us concentrate on multiplication and addition, according to the standard rules

$$\frac{n_1}{d_1} * \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}, \qquad \frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + d_1 n_2}{d_1 d_2}.$$

Multiplication of two `Frac` objects requires the definition of the class function `__mul__` in lines 21–23. When we invoke `a*b` in line 41, this function is invoked with **self** replaced by the left operand, here `a`, and `another` replaced by the right operand, here `b`.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

Note how line 23 computes the numerator and denominator of the product, and then calls `__init__` to create a new `Frac` object, so that `c=a*b` would create a new `Frac` with identifier `c`. In line 41 though, the new, as yet unnamed `Frac`, is passed immediately to `__str__`. As the code snippet shows, addition is handled in exactly the same way.

For the sake of brevity, we have left the class incomplete. It is a valuable exercise for the beginner to enhance the code gradually in ways which are more advanced.

1 Create division and subtraction class functions called `__div__` and `__sub__`, and test them.
2 It looks a little odd if the denominator turns out to be 1 to print, e.g., 7/1. Amend the `__str__` function so that if `self.den` is 1, then the *string* created is precisely `self.num`, and test that the new version works.
3 The `__init__` code will certainly fail if argument `d` is zero. Think of warning the user.

## 3.10 The structure of Python

Near the end of Section 3.2, we pointed out the relationship between *identifiers* and *objects*, and we now return to this topic. In our pedagogic example of a Python class `Frac` in Section 3.9, we noted that an instance of a `Frac`, e.g., `a=Frac(3,7)` produces an *identifier* `a` referring to an *object* of class `Frac`. Now a `Frac` object contains data, here a pair of integers, together with a set of functions to operate on them. We access the data relevant to the instance via the "dot mechanism", e.g., `a.num`. Similarly, we access the associated functions via, e.g., `a.real()`.

So far, this is merely gathering previously stated facts. However, Python is packed with objects, some very complicated, and this "dot mechanism" is used universally to access the objects' components. We have already seen enough of Python to point out some examples.

Our first example is that of complex numbers described in Section 3.3.4. Suppose we have set `c=1.5-0.4j` or equivalently `c=complex(1.5,-0.4)`. We should regard complex numbers as being supplied by class `Complex`, although to guarantee speed they are hard-coded into the system. Now just as with the `Frac` class, we access the data via `c.real` and `c.imag`, while `c.conjugate()` produces the complex conjugate number `1.5+0.4j`. We say that Python is *object oriented*. Compare this approach with that of a *function oriented* language such as Fortran77, where we would have used `C=CMPLX(1.5,-0.40)`, `REAL(C)`, `AIMAG(C)` and `CONJG(C)`. At this simple level, there is no reason to prefer one approach to the other.

Our next example refers to *modules* described in Section 3.4. Like most other features in Python, a module is an *object*. Thus `import math as re` includes the `math` module and gives it the identifier `re`. We can access data as, e.g., `re.pi` and functions by, e.g., `re.gamma(2)`.

Once you master the concept of the "dot mechanism", understanding Python should

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

become a lot clearer. See, e.g., the discussion of container objects in Section 3.5. All of the more sophisticated packages, e.g., *numpy*, *matplotlib*, *mayavi* and *pandas* rely on it. It is at this level that the *object-oriented* approach pays dividends in offering a uniform environment which is not available in early versions of C or Fortran.

## 3.11 Prime numbers: a worked example

We finish this chapter on "pure" Python by looking at a real problem. The internet has revolutionized communications and has emphasized the need for security of transmission of data. Much of this security is based on the fact that it is exceedingly difficult to decide whether a given integer $n$ which is large, say $n > 10^{100}$, can be written as the product of prime numbers. We look here at a much more fundamental problem, building a list of prime numbers.

**Table 3.1** The Sieve of Eratosthenes for primes $\leqslant 18$. We start by writing down in a row the integers from 2 to 18. Starting with the leftmost integer, we delete all multiples of it in line 2. Then we move to the closest remaining integer, here 3, and delete all multiples of 3 in line 3. We continue this process. Clearly, the numbers left are not products of smaller integers; they are the primes.

| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2** | 3 | | 5 | | 7 | | 9 | | 11 | | 13 | | 15 | | 17 | |
| 2 | **3** | | 5 | | 7 | | | | 11 | | 13 | | | | 17 | |
| 2 | 3 | | **5** | | 7 | | | | 11 | | 13 | | | | 17 | |

Recall that an integer $p$ is *prime* if it cannot be written as a product $q \times r$ of integers $q$ and $r$, both greater than 1. Thus the first few primes are 2, 3, 5, 7,.... The problem of determining all primes less than or equal to a given integer $n$ has been studied for millennia, and perhaps the best-known approach is the *sieve of Eratosthenes*. This is illustrated, for the case $n = 18$, in Table 3.1. The caption explains how the process works, and the first three steps are shown in the figure. Notice that since any composite number to be deleted $\leqslant n$, at least one of its factors $\leqslant \sqrt{n}$. In the example, $\sqrt{18} < 5$ and so there is no need to delete multiples of $5, 7, \ldots$ from the sieve.

The following Python function implements, directly, this procedure.

```
1  def sieve_v1(n):
2      """
3        Use Sieve of Eratosthenes to compute list of primes <= n.
4        Version 1
5      """
6      primes=range(2,n+1)
7      for p in primes:
8          if p*p>n:
```

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

```
9             break
10          product=2*p
11          while product<=n:
12              if product in primes:
13                  primes.remove(product)
14              product+=p
15      return len(primes),primes
```

The first five lines are conventional. In line 6, we introduce the top line of Figure 3.1 coded as a Python *list* called `primes`, which is to be sieved. Next we introduce a **`for`** loop, which runs for lines 7–14. Each choice for the loop variable `p` corresponds to a new row in the figure. As we remarked above, we need not consider $p > \sqrt{n}$, and this is tested in lines 8 and 9. The **`break`** command transfers control to the line after the end of the loop, i.e., line 15. (This can be seen from the indentation.) Now look at the **`while`** loop which runs for lines 11–14. Because of the earlier **`break`** statement, it is guaranteed that the **`while`** loop is entered at least once. Then if `product` is still in the *list*, line 13 deletes it. In Section 3.5.1, we saw that `list.append(item)` appends `item` to the **`list`**. Here `list.remove(item)` deletes the first occurrence of `item` in **`list`**. It is an error if `item` is not in **`list`**, and this is why its presence is guaranteed by line 12. (Try `help(list)` to see the methods available for *lists*.) Having potentially removed `2*p`, we next construct `3*p` in line 14 and repeat the process. Once all relevant multiples of `p` have been removed by iteration of the while loop, we return to line 7 and set `p` to be the next prime in the *list*. Finally, we return the list of primes and its length. The latter is an important function in number theory, where it is usually denoted $\pi(n)$.

You are recommended to construct a file called `sieves.py` and type or paste this code into it. Then in *IPython* type

```
from sieves import sieve_v1
sieve_v1?
sieve_v1(18)
```

so as to check that things are working. You can even check the time taken with a command[6]

```
timeit sieve_v1(1000)
```

Inspection of Table 3.2 shows that while the performance of this simple straightforward function is satisfactory for small $n$, the time taken grows unacceptably large for even moderately large values. It is a very useful exercise to see how easily we can improve its performance.

We start by considering the algorithm, which involves two loops. We can hardly avoid iterating over the actual primes, but for each prime $p$ we then loop removing composite numbers $n \times p$, where $n = 2, 3, 4, \ldots$, from the loop. We can make two very simple

[6] The *IPython* `%run -t` introduced in Section 2.6 only works on complete scripts. However, the magic `%timeit` command times individual lines.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

**Table 3.2** The number of primes $\pi(n) \leqslant n$, and the approximate time taken on the author's laptop to compute them using the Python functions given here. The *relative* timings (not the *absolute* ones) are of interest here.

| $n$ | $\pi(n)$ | time_v1(secs) | time_v2(secs) |
|---|---|---|---|
| 10 | 4 | $3.8 \times 10^{-6}$ | $3.2 \times 10^{-6}$ |
| $10^2$ | 25 | $1.1 \times 10^{-4}$ | $8.3 \times 10^{-6}$ |
| $10^3$ | 168 | $8.9 \times 10^{-3}$ | $4.3 \times 10^{-5}$ |
| $10^4$ | 1,229 | $9.3 \times 10^{-1}$ | $3.3 \times 10^{-4}$ |
| $10^5$ | 9,592 | 95 | $3.7 \times 10^{-3}$ |
| $10^6$ | 78,498 | | $4.1 \times 10^{-2}$ |
| $10^7$ | 664,579 | | $4.5 \times 10^{-1}$ |
| $10^8$ | 5,761,455 | | 4.9 |

improvements here. Note that, assuming $p > 2$, any composite number less than $p^2$ will have been removed already from the sieve, and so we may start by removing the composite $p^2$. Further, the composite $p^2 + n \times p$ is even if $n$ is odd, and so was removed from the sieve on the first pass. Therefore, we can improve the algorithm for each prime $p > 2$, by removing just $p^2$, $p^2 + 2p$, $p^2 + 4p$,... This is not the best we could do, e.g., 63 is sieved twice, but it will suffice.[7]

Next we consider the implementation. Here we have been rather wasteful in that five loops are involved. The **for** and **while** loops are explicit. The **if** statement in line 12 involves iterating through the primes *list*. This is repeated in line 13, and after `product` has been identified and discarded, all of the remaining *list* elements have to be shuffled down one position. Consider the following non-obvious re-implementation.

```
1  def sieve_v2(n):
2      """
3          Sieve of Eratosthenes to compute list of primes <= n.
4          Version 2.
5      """
6      sieve = [True]*(n+1)
7      for i in xrange(3,n+1,2):
8          if i*i > n:
9              break
10         if sieve[i]:
11             sieve[i*i: :2*i]=[False]*((n - i*i) // (2*i) + 1)
12     answer = [2] + [i for i in xrange(3,n+1,2) if sieve[i]]
13     return len(answer), answer
```

Here the initial sieve is implemented as a *list* of Booleans, all initialized to `True` in line 6. This will be a block of code, fast to set up, and occupying less memory than a *list* of the same length of integers. The outer loop is a **for** loop running from lines 7 to 11. In line 7, the function **xrange** is new. This behaves just like range, but rather

[7] Sieves which are more elaborate exist, e.g., those of Sundaram and of Atkin.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press

than creating an entire *list* in memory, it generates the members on demand. For large *lists*, this is faster and less memory-hungry. Here the loop covers the odd numbers in the closed interval $[3, n]$. Lines 8 and 9 stop the outer loop once $i > \sqrt{n}$, just as in the earlier version. Now consider lines 10 and 11. Initially, `i` is 3, and `sieve[i]` is `True`. We need to set the sieve elements for $i^2$, $i^2 + 2i$, ... to `False`, the equivalent of discarding them in this implementation. Rather than use a loop, we carry out the operation as a single slice in line 11. (The integer factor on the right gives the dimension of the slice.) At the end of the **`for`** loop, the *list* `sieve` has had set to `False` the items with indices corresponding to all odd composite integers. Finally, in line 12 we build the list of primes. We start with a *list* containing just 2. We use a *list* comprehension to construct a *list* of all odd unsieved numbers, and then we concatenate both *lists*, before returning the result.

Although at first reading it may take some time to understand this version, nothing new (other than **`xrange`**) has been used, the code is 25% shorter and runs considerably faster, as shown in the final column in Table 3.2. Indeed, it delivers the millions of primes less than $10^8$ in a few seconds. The extension to $10^9$ would take a few minutes if tens of gigabytes of memory were available. Clearly, for very large primes, sieve methods are not the most efficient.

Notice too that within the *lists* in this second version, the items all have the same type. Although Python does not impose this restriction, it turns out to be worthwhile to introduce a new object, lists of homogeneous type, and this leads naturally to the *numpy*, the topic of the next chapter.

https://doi.org/10.1017/CBO9781107447875.004 Published online by Cambridge University Press