2 The core Python language I

2.1 The Python shell

This chapter introduces the syntax, structure and data types of the Python programming language. The first few sections do not involve writing much beyond a few statements of Python code and so can be followed using the Python *shell*. This is an interactive environment: the user enters Python statements that are executed immediately after the *Enter* key is pressed.

The steps for accessing the "native" Python shell differ by operating system. To start it from the command line, first open a terminal using the instructions from Section 1.4 and type `python`.

To exit the Python shell, type `exit()`.

When you start the Python shell, you will be greeted by a message (which will vary depending on your operating system and precise Python version). On my system, the message reads:

```
Python 3.3.5 |Anaconda 2.0.1 (x86\_64)| (default, Mar 10 2014, 11:22:25)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The three chevrons (`>>>`) are the *prompt*, which is where you will enter your Python commands. Note that this book is concerned with Python 3, so you should check that the Python version number reported on the first line is `Python 3.X.Y` where the precise values of the minor version numbers `X` and `Y` should not be important.

Many Python distributions come with a slightly more advanced shell called *IDLE*, which features tab-completion, and *syntax highlighting* (Python keywords are colored specially when you type them). We will pass over the use of this application in favor of the newer and more advanced *IPython* environment, discussed in Chapter 5.

It is also possible for many installations (especially on Windows) to start a Python shell directly from an application installed when you install the Python interpreter itself. Some installations even add a shortcut icon to your Desktop which will open a Python shell when you click on it.

2.2 Numbers, variables, comparisons and logic

2.2.1 Types of numbers

Among the most basic Python objects are the numbers, which come in three *types*: integers (type: `int`), floating point numbers (type: `float`) and complex numbers (type: `complex`).

Integers

Integers are whole numbers such as 1, 8, -72 and 3847298893721407. In Python 3, there is no limit to their magnitude (apart from the availability of your computer's memory).[1] Integer arithmetic is exact.

Floating point numbers

Floating point numbers are the representation of real numbers such as 1.2, -0.36 and 1.67263×10^{-7}. They do not, in general, have the exact value of the real number they represent, but are stored in binary to a certain precision (on most systems, to the equivalent of 15–16 decimal places),[2] as explained in Section 9.1. For example, the number $\frac{4}{3}$ is stored as the binary equivalent of $1.33333333333333325931846502\cdots$, which is nearly (but not quite) the same as the infinitely repeating decimal representation of $\frac{4}{3} = 1.3333\cdots$. Moreover, even numbers that do have an exact *decimal* representation may not have an exact *binary* representation: for example $1/10$ is represented by the binary number equivalent to $0.10000000000000000555111512\cdots$. Because of this finite precision, floating point arithmetic is not exact but, with care, it is "good enough" for most scientific applications.

Any single number containing a period ('`.`') is considered by Python to specify a floating point number. Scientific notation is supported using '`e`' or '`E`' to separate the significand (mantissa) from the exponent: for example, `1.67263e-7` represents the number 1.67263×10^{-7}.

Complex numbers

Complex numbers such as $4+3j$ consist of a real and an imaginary part (denoted by `j` in Python), each of which is itself represented as a floating point number (even if specified without a period). Complex number arithmetic is therefore not exact but subject to the same finite precision considerations as `floats`.

A complex number may be specified either by "adding" a real number to an imaginary one (denoted by the `j` suffix), as in `2.3 + 1.2j` or by separating the real and imaginary parts in a call to `complex`, as in `complex(2.3, 1.2)`.

[1] In Python 2, there were two kinds of integer: "simple" integers (system-dependent, but usually stored in either 32 or 64 bits and "long" integers (of any size), indicated with the suffix `L`.

[2] This corresponds to the implementation of the IEEE 754 double-precision standard.

Example E2.1 Typing a number at the Python shell prompt simply echoes the number back to you:

```
>>> 5
5
>>> 5.
5.0
>>> 0.10
❶ 0.1
>>> 0.0001
0.0001
>>> 0.0000999
❷ 9.99e-05
```

Note that the Python interpreter displays numbers in a standard way. For example:

❶ The internal representation of 0.1 discussed earlier is rounded to '`0.1`', which is the shortest number with this representation.

❷ Numbers smaller in magnitude than 0.0001 are displayed in scientific notation.

A number of one type can be created from a number of another type with the relevant *constructor*:

```
>>> float(5)
5.0
>>> int(5.2)
5
>>> int(5.9)
❶ 5
>>> complex(3.)
❷ (3+0j)
❸ >>> complex(0., 3.)
3j
```

❶ Note that a floating point number is *rounded down* in casting it into an integer.

❷ Constructing a `complex` object from a `float` generates a complex number with the imaginary part equal to zero.

❸ To generate a pure imaginary number, you have to explicitly pass two numbers to `complex` with the first, real part, equal to zero.

2.2.2 Using the Python shell as a calculator

Basic arithmetic

With the three basic number types described earlier, it is possible to use the Python shell as a simple calculator using the operators given in Table 2.1. These are *binary* operators in that they act on two numbers (the *operands*) to produce a third (e.g., `2**3` evaluates to `8`).

Python 3 has two types of division: floating point division (`/`) always returns a floating point (or complex) number result, even if it acts on integers. Integer division (`//`) *always rounds down* the result to the nearest integer; the type of the resulting number is an `int`

Table 2.1 Basic Python arithmetic operators

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	floating point division
`//`	integer division
`%`	modulus (remainder)
`**`	exponentiation

only if both of its operands are `ints`; otherwise it returns a `float`. Some examples should make this clearer:

Regular floating point division with (`/`):

```
>>> 2.7 / 2
1.35
>>> 9 / 2
4.5
>>> 8 / 4
2.0
```

The last operation returns a `float` even though both operands are `ints`.[3]

Integer division with (`//`):

```
>>> 8 // 4
2
>>> 9 // 2
4
>>> 2.7 // 2
1.0
```

Note that `//` can perform integer arithmetic (rounding down) on floating point numbers. The modulus operator gives the remainder of an integer division:

```
>>> 9 % 2
1
>>> 4.5 % 3
1.5
```

Again, the number returned is an `int` only if both of the operands are `ints`.

Operator precedence

Arithmetic operations can be strung together in a sequence, which naturally raises the question of *precedence*: for example, does `2 + 4 * 3` evaluate to `14` (as `2 + 12`) or `18` (as `6 * 3`)? Table 2.2 shows that the answer is `14`: multiplication has a higher precedence than addition and is evaluated first. These precedence rules are overridden by the use of *parentheses*: for example, `(2 + 4) * 3 = 18`.

[3] This is a major difference from Python 2, in which the `/` operator performed integer division on two integers.

Table 2.2 Python arithmetic operator precedence

`**`	(highest precedence)
`*`, `/`, `//`, `%`	
`+`, `-`	(lowest precedence)

Operators of equal precedence are evaluated left to right with the exception of exponentiation (`**`), which is evaluated right to left (that is, "top down" when written using the conventional superscript notation). For example,

```
>>> 6 / 2 / 4          # the same as 3 / 4
0.75
>>> 6 / (2 / 4)        # the same as 6 / 0.5
12.0
>>> 2**2**3            # the same as 2**(2**3) == 2**8
256
>>> (2**2)**3          # the same as 4**3
64
```

In examples such as these, the text following the hash symbol, `#`, is a comment that is ignored by the interpreter. We shall sometimes use comments in this to explain more about a statement, but it is not necessary to type it in if you try out the code.

Methods and attributes of numbers

Python numbers are *objects* (in fact, everything in Python is an object) and have certain *attributes*, accessed using the "dot" notation: `<object>.<attribute>` (this use of the period has nothing to do with the decimal point appearing in a floating point number). Some attributes are simple values: for example, complex number objects have the attributes `real` and `imag` which are the real and imaginary (floating point) parts of the number:

```
>>> (4+5j).real
4.0
>>> (4+5j).imag
5.0
```

Other attributes are *methods*: callable functions that act on their object in some way.[4] For example, complex numbers have a method, `conjugate`, which returns the complex conjugate:

```
>>> (4+5j).conjugate()
(4-5j)
```

Here, the empty parentheses indicate that the method is to be *called*, that is, the function to calculate the complex conjugate is to be run on the number $4 + 5j$; if we omit them, as in `(4+5j).conjugate`, we are referring to the method itself (without calling it) – this method is itself an object!

[4] In this book, we will use the terms *method* and *function* interchangeably. In Python, everything is an object and the distinction is not as meaningful as it is in some other languages.

Integers and floating point numbers don't actually have very many attributes that it makes sense to use in this way, but if you're curious you can find out how many bits an integer takes up in memory by calling its `bit_length` method. For example,

```
>>> (3847298893721407).bit_length()
52
```

Note that Python allocates as much memory as is necessary to exactly represent the integer.

Mathematical functions

Two of the mathematical functions that are provided "by default" as so-called *built-ins* are `abs` and `round`.

`abs` returns the absolute value of a number as follows:

```
>>> abs(-5.2)
5.2
>>> abs(-2)
2
>>> abs(3+4j)
5.0
```

This is an example of *polymorphism*: the same function, `abs`, does different things to different objects. If passed a real number, x, it returns $|x|$, the non-negative magnitude of that number, without regard to sign; if passed a complex number, $z = x + iy$, it returns the modulus, $|z| = \sqrt{x^2 + y^2}$.

The `round` function (with one argument) rounds a floating point number to the nearest integer:

```
>>> round(-9.62)
-10
>>> round(7.5)
8
>>> round(4.5)
4
```

Note that in Python 3, this function employs *Banker's rounding*: if a number is mid way between two integers, then the even integer is returned.[5]

Python is a very modular language: functionality is available in packages and modules that are *imported* if they are needed but are not loaded by default: this keeps the memory required to run a Python program to a minimum and improves performance. For example, many useful mathematical functions are provided by the `math` module, which is imported with the statement

```
>>> import math
```

The `math` module concerns itself with floating point and integer operations (for functions of complex numbers, there is another module, called `cmath`). These are called

[5] In Python 2 the `round()` function rounds *away from zero* when two integers are equally close: thus `round(2.5)` is `3` but `round(-2.5)` is `-3`.

Table 2.3 Some functions provided by the `math` module. *Angular arguments are assumed to be in radians.*

`math.sqrt(x)`	$\sqrt{x}$
`math.exp(x)`	e^x
`math.log(x)`	$\ln x$
`math.log(x, b)`	$\log_b x$
`math.log10(x)`	$\log_{10} x$
`math.sin(x)`	$\sin(x)$
`math.cos(x)`	$\cos(x)$
`math.tan(x)`	$\tan(x)$
`math.asin(x)`	$\arcsin(x)$
`math.acos(x)`	$\arccos(x)$
`math.atan(x)`	$\arctan(x)$
`math.sinh(x)`	$\sinh(x)$
`math.cosh(x)`	$\cosh(x)$
`math.tanh(x)`	$\tanh(x)$
`math.asinh(x)`	arsinh(x)
`math.acosh(x)`	arcosh(x)
`math.atanh(x)`	artanh(x)
`math.hypot(x, y)`	The Euclidean norm, $\sqrt{x^2 + y^2}$
`math.factorial(x)`	$x!$
`math.erf(x)`	The error function at x
`math.gamma(x)`	The gamma function at x, $\Gamma(x)$
`math.degrees(x)`	Converts x from radians to degrees
`math.radians(x)`	Converts x from degrees to radians

by passing one (or sometimes more than one) number to them inside parentheses (the numbers are said to act as *arguments* to the function being called). For example,

```
>>> import math
>>> math.exp(-1.5)
0.22313016014842982
>>> math.cos(0)
1.0
>>> math.sqrt(16)
4.0
```

A complete list of the mathematical functions provided by the `math` module is available in the online documentation;[6] the more commonly used ones are listed in Table 2.3.

The `math` module also provides two very useful nonfunction attributes: `math.pi` and `math.e` give the values of π and e, the base of the natural logarithm, respectively.

It is possible to import the math module with '`from math import *`' and access its functions directly:

```
>>> from math import *
>>> cos(pi)
-1.0
```

[6] http://docs.python.org/3/library/math.html.

However, although this may be convenient for interacting with the Python shell, it is not recommended in Python programs. There is a danger of name conflicts (particularly if many modules are imported in this way), and makes it difficult to know which function comes from which module. Importing with `import math` keeps the functions bound to their module's *namespace*: thus, even though `math.cos` requires more typing it makes for code that is much easier to understand and maintain.

Example E2.2 As might be expected, mathematical functions can be strung together in a single expression:

```
>>> import math
>>> math.sin(math.pi/2)
1.0
>>> math.degrees(math.acos(math.sqrt(3)/2))
30.000000000000004
```

Note the finite precision here: the exact answer is $\arccos(\sqrt{3}/2) = 30°$.

The fact that the `int` function rounds down in casting a floating point number to an integer can be used to find the number of digits a positive integer has:

```
>>> int(math.log10(9999)) + 1
4
>>> int(math.log10(10000)) + 1
5
```

2.2.3 Variables

What is a variable?

When an object, such as a `float`, is created in a Python program or using the Python shell, memory is allocated for it: the location of this memory within the computer's architecture is called its *address*. The actual value of an object's address isn't actually very useful in Python, but if you're curious you can find it out by calling the `id` built-in method:

```
>>> id(20.1)
4297273888              # for example
```

This number refers to a specific location in memory that has been allocated to hold the `float` object with the value `20.1`.

For anything beyond the most basic usage, it is necessary to store the objects that are involved in a calculation or algorithm and to be able to refer to them by some convenient and meaningful name (rather than an address in memory). This is what *variables* are for.[7] A variable name can be assigned ("bound") to any object and used to identify that object in future calculations. For example,

[7] In Python, it is arguably better to talk of object *identifiers* or *identifier names* rather than variables, but we will not be too strict about this.

```
>>> a = 3
>>> b = -0.5
>>> a * b
-1.5
```

In this snippet, we create the `int` object with the value `3` and assign the variable name `a` to it. We then create the `float` object with the value `-0.5` and assign `b` to it. Finally, the calculation `a * b` is carried out: the values of `a` and `b` are multiplied together and the result returned. This result isn't assigned to any variable, so after being output to the screen it is thrown away. That is, the memory required to store the result, a `float` with the value `-1.5`, is allocated for long enough for it to be displayed to the user, but then it is gone.[8] If we need the result for some subsequent calculation, we should assign it to another variable:

```
>>> c = a * b
>>> c
-1.5
```

Note that we did not have to *declare* the variables before we assign them (tell Python that the variable name `a` is to refer to an integer, `b` is to refer to a floating point number, etc.), as is necessary in some computer languages. Python is a *dynamically typed* language and the necessary object type is inferred from its definition: in the absence of a decimal point, the number `3` is assumed to be an `int`; `-0.5` looks like a floating point number and so Python defines `b` to be a `float`.[9]

Variable names

There are some rules about what makes a valid variable name:

- Variable names are *case-sensitive*: `a` and `A` are different variables;
- Variable names can contain any letter, the underscore character ('_') and any digit (0–9) ...
- ... but must not *start with* a digit;
- A variable name must not be the same as one of the *reserved keywords* given in Table 2.4;
- The built-in constant names `True`, `False` and `None` cannot be assigned as variable names.

Most of the reserved keywords are pretty unlikely choices for variable names, with the exception of `lambda`. Python programmers often use `lam` if they need to use it. A good text editor will highlight the keywords as you type your program, so this is unlikely to cause confusion.

It is possible to give a variable the same name as a built-in function (e.g., `abs` and `round`), but that built-in function will no longer be available after such an assignment,

[8] Actually in an interactive Python session the result of the last calculation is stored in the special variable called _ (the underscore), so it isn't really thrown away until overwritten by the *next* calculation.

[9] This is sometimes called *duck-typing* after the phrase attributed to James Whitcomb Riley: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

Table 2.4 Python 3 reserved keywords

`and`	`assert`	`break`	`class`	`continue`
`def`	`del`	`elif`	`else`	`except`
`finally`	`for`	`from`	`global`	`if`
`import`	`in`	`is`	`lambda`	`nonlocal`
`not`	`or`	`pass`	`print`	`raise`
`return`	`try`	`while`	`yield`	

Note: in Python 2, `exec` is a keyword but `nonlocal` is not.

so this is probably best avoided – luckily, most have names that are unlikely to be chosen in practice.[10]

In addition to the rules mentioned earlier, there are certain style considerations that dictate good practice in naming variables:

- Variable names should be meaningful (`area` is better than `a`) ...
- ... but not too long (`the_area_of_the_triangle` is unwieldy);
- Generally, don't use `I` (uppercase `i`), `l` (lowercase `L`) or the uppercase letter `O`: they look too much like the digits 1 and 0;
- The variable names `i`, `j` and `k` are usually used as integer counters;
- Use lowercase names, with words separated by underscores rather than 'CamelCase': for example, `mean_height` and not `MeanHeight`.[11]

These and many other rules and conventions are codified in a style guide called PEP8 which forms part of the Python documentation[12] (see also Section 9.3.1).

Breaking these style rules will not result in your program failing to run, but it might make it harder to maintain and debug – the person you help might be yourself!

Example E2.3 *Heron's formula* gives the area, A, of a triangle with sides a, b, c as:

$$A = \sqrt{s(s-a)(s-b)(s-c)} \text{ where } s = \tfrac{1}{2}(a+b+c).$$

For example,

```
>>> a = 4.503
>>> b = 2.377
>>> c = 3.902
>>> s = (a + b + c) / 2
❶ >>> area = math.sqrt(s * (s - a) * (s - b) * (s - c))
>>> area
4.63511081571606
```

❶ Don't forget to `import math` if you haven't already in this Python session.

[10] For a complete list of built-in function names, see http://docs.python.org/3/library/functions.html.

[11] CamelCase in Python is usually reserved for class names: see Section 4.6.2.

[12] http://legacy.python.org/dev/peps/pep-0008/.

Table 2.5 Python comparison operators

==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Example E2.4 The data type and memory address of the object referred to by a variable name can be found with the built-ins `type` and `id`:

```
>>> type(a)
<class 'float'>
>>> id(area)
4298539728      # for example
```

2.2.4 Comparisons and logic

Operators

The main comparison operators that are used in Python to compare objects (such as numbers) are given in Table 2.5.

The result of a comparison is a *boolean* object (of type `bool`) which has exactly one of two values: `True` or `False`. These are built-in constant keywords and cannot be reassigned to other values.[13] For example,

```
>>> 7 == 8
False
>>> 4 >= 3.14
True
```

Python is able, as far as possible without ambiguity, to compare objects of different types: the integer `4` is promoted to a `float` for comparison with the number `3.14`.

Note the importance of the difference between `==` and `=`. The single equals sign is an *assignment*, which does not return a value: the statement `a=7` assigns the variable `a` to the integer object `7` and that is all, whereas the expression `a==7` is a test: it returns `True` or `False` depending on the value of `a`.[14]

Care should be taken in comparing floating point numbers for equality. Because they are not stored exactly and calculations involving them frequently leads to a loss of precision, this can give unexpected results to the unwary. For example,

```
>>> a = 0.01
>>> b = 0.1**2
>>> a == b
False
```

[13] In Python 2, however, unhelpful assignments such as `True = False` were allowed.

[14] In some languages, such as C, assignment returns the value of whatever is being assigned, which can lead to some nasty and hard-to-find bugs when `=` is mistakenly used as a comparison operator.

In this example, 0.01 cannot be represented exactly as a floating point number but is (on my system) stored as a binary number equivalent to 0.010000000000000000208; the result of squaring the floating point representation of 0.1 on the other hand is 0.010000000000000000194, and these two numbers are not the same. See Section 9.1 for more information.

Logic operators

Comparisons can be modified and strung together with the logic operator keywords `and`, `not` and `or`. See Tables 2.6, 2.7 and 2.8. For example,

```
>>> 7.0 > 4 and -1 >= 0      # equivalent to True and False
False
>>> 5 < 4 or 1 != 2          # equivalent to False or True
True
```

In compound expressions such as these, the comparison operators are evaluated first, and then the logic operators in order of precedence: `not`, `and`, `or`. This precedence is overridden with parentheses, as for arithmetic. Thus,

```
>>> not 7.5 < 0.9 or 4 == 4
True
>>> not (7.5 < 0.9 or 4 == 4)
False
```

Table 2.6 Truth table for the `not` operator

P	not P
True	False
False	True

Table 2.7 Truth table for the `and` operator

P	Q	P and Q
True	True	True
False	True	False
True	False	False
False	False	False

Table 2.8 Truth table for the `or` operator

P	Q	P or Q
True	True	True
False	True	True
True	False	True
False	False	False

The truth tables for the logic operators are given below; note that, in common with most languages `or` in Python is the *inclusive or* variant for which `A or B` is `True` if both `A` and `B` are `True`, rather than the *exclusive or* operator (`A xor B` is `True` only if one but not both of `A` and `B` are `True`).

◊ Boolean equivalents and conditional assignment

In a logic test expression, it is not always necessary to make an explicit comparison to obtain a boolean value: Python will try to convert an object to a `bool` type if needed. For numeric objects, 0 evaluates to `False` and any nonzero value is `True`:

```
>>> a = 0
>>> a or 4 < 3      # same as: False or 4 < 3
False
>>>
>>> not a+1         # same as: not True
False
```

In this last example, addition has higher precedence than the logic operator `not`, so `a+1` is evaluated first to give `1`. This corresponds to boolean `True`, and so the whole expression is equivalent to `not True`. To explicitly convert an object to a boolean object, use the `bool` constructor:

```
>>> bool(-1)
True
>>> bool(0.0)
False
```

In fact, the `and` and `or` operators always return one of their operands and not just its `bool` equivalent. So, for example:

```
>>> a = 0
❶ >>> a-2 or a
-2
❷ >>> 4 > 3 and a-2
-2
❸ >>> 4 > 3 and a
0
```

Logic expressions are evaluated left to right, and those involving `and` or `or` are *short-circuited*: the second expression is only evaluated if necessary to decide the truth value of the whole expression. The three examples presented here can be analyzed as follows:
❶ In the first example, `a−2` is evaluated first: this is equal to −2, which is equivalent to `True`, so the `or` condition is fulfilled and the operand evaluating to `True` is returned immediately: −2.
❷ `4 > 3` is `True`, so the second expression must be evaluated to establish the truth of the `and` condition. `a−2` is equal to −2, which is also equivalent to `True`, so the `and` condition is fulfilled and −2 (as the most recently evaluated expression) is returned.
❸ In the last case, `a` is 0 which is equivalent to `False`: the `and` condition evaluates to `False` because of this, and so the return value is 0.

Python's special value, `None`

Python defines a single value, `None`, of the special type, `NoneType`. It is used to represent the absence of a defined value, for example, where no value is possible or relevant. This is particularly helpful in avoiding arbitrary default values (such as `0`, `1` or `−99`) for bad or missing data.

In a boolean comparison, `None` evaluates to `False`, but to test whether or not a variable, `x`, is equal to `None`, use

```
if x is None
```

and

```
if x is not None
```

rather than the shortcuts `if x` and `if not x`.[15]

Example E2.5 A common Python idiom is to assign a variable using the return value of a logic expression:

```
>>> a = 0
>>> b = a or -1
>>> b
-1
```

That is (for `a` understood to be an integer): "set `b` equal to the value of `a` unless `a==0`, in which case set `b` equal to −1."

2.2.5 Immutability and identity

The objects presented so far, such as integers and booleans, are *immutable*. Immutable objects do not change after they are created, though a variable name may be reassigned to refer to a different object from the one it was originally assigned to. For example, consider the assignments:

```
>>> a = 8
>>> b = a
```

The first line creates the integer object with value `8` in memory, and assigns the name `a` to it. The second line assigns the name `b` to the same object. You can see this by inspecting the address of the object refered to by each name:

```
>>> id(a)
4297273504
>>> id(b)
4297273504
```

Thus, `a` and `b` are references to the same integer object. Now suppose `a` is reassigned to a new number object:

---

[15] Recall that `not x` also evaluates to `True` if `x` is any of `0`, `False` or the empty string and so is not a very reliable way to test specifically if `x` is not set to `None`.

```
>>> a = 3.14
>>> a
3.14
>>> b
8
>>> id(a)
4298630152
>>> id(b)
4297273504
```

Note that the value of `b` has not changed: this variable still refers to the original `8`. The variable `a` now refers to a new, `float` object with the value `3.14` located at a new address. This is what is meant by immutability: it is not the "variable" that cannot change but the immutable object itself. This is illustrated in Figure 2.1.

A more convenient way to establish if two variables refer to the same object is to use the `is` operator, which determines object *identity*:

```
>>> a = 2432
>>> b = a
>>> a is b
True
>>> c = 2432
>>> c is a
False
>>> c == a
True
```

Here, the assignment `c = 2432` creates an entirely new integer object so `c is a` evaluates as `False`, even though `a` and `c` have the same *value*. That is, the two variables refer to different objects with the same value.

It is often necessary to change the value of a variable in some way, such as

```
>>> a = 800
>>> a = a + 1
>>> a
801
```

The integers `800` and `801` are immutable: the line `a = a + 1` creates a *new* integer object with the value `801` (the right-hand side is evaluated first) and assigns it to the

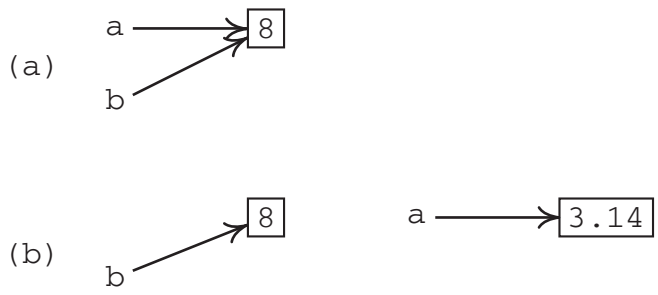

Figure 2.1 (a) Two variables referring to the same integer; (b) after reassigning the value of `a`.

variable name `a` (the old `800` is forgotten[16] unless some other variable refers to it). That is, `a` points to a different address before and after this statement.

This reassignment of a variable by an arithmetic operation on its value is so common that there is a useful shorthand notation: the *augmented assignment* `a += 5` is the same as `a = a + 5`. The operators `-=`, `*=`, `/=`, `//=`, `%=` work in the same way. C-style increment and decrement operations such as `a++` for `a += 1` are *not* supported in Python, however.[17]

Example E2.6 Python provides the operator `is not`: it is more natural to use `c is not a` than `not c is a`.

```
>>> a = 8
>>> b = a
>>> b is a
True
>>> b /= 2
>>> b is not a
True
```

♢ **Example E2.7** Given the previous discussion, it might come as a surprise to find that

```
>>> a = 256
>>> b = 256
>>> a is b
True
```

This happens because Python keeps a *cache* of commonly used, small integer objects (on my system, the numbers `−5–256`). To improve performance, the assignment `a = 256` attaches the variable name `a` to the existing integer object without having to allocate new memory for it. Because the same thing happens with `b`, the two variables in this case do, in fact, point to the same object. By contrast,

```
>>> a = 257
>>> b = 257
>>> a is b
False
```

2.2.6 Exercises

Questions

Q2.2.1 Predict the result of the following expressions and check them using the Python shell.

a. `2.7 / 2`
b. `2 / 4 - 1`

[16] That is, the memory assigned for it by Python is reclaimed ("garbage-collected") for general use.

[17] Assignment and augmented assignment in Python are statements not expressions and so do not return a value and cannot be chained together.

c. `2 // 4 - 1`
d. `(2 + 5) % 3`
e. `2 + 5 % 3`
f. `3 * 4 // 6`
g. `3 * (4 // 6)`
h. `3 * 2 ** 2`
i. `3 ** 2 * 2`

Q2.2.2 The operators listed in Table 2.1 are all *binary* operators: they take two operands (numbers) and return a single value. The symbol - is also used as a *unary* operator, which returns the negative value of the single operand on which it acts. For example,

```
>>> a = 4
>>> b = -a
>>> b
-4
```

Note that the expression `b = -a` (which sets the variable `b` to the negative value of `a`) is different from the expression `b -= a` (which subtracts `a` from `b` and stores the result in `b`). The unary – operator has a higher precedence than `*`, `/` and `%` but a lower precedence than exponentiation (`**`), so that, for example `-2 ** 4` is `-16` (i.e., $-(2^4)$, not $(-2)^4$).

Predict the result of the following expressions and check them using the Python shell.

a. `-2 ** 2`
b. `2 ** -2`
c. `-2 ** -2`
d. `2 ** 2 ** 3`
e. `2 ** 3 ** 2`
f. `-2 ** 3 ** 2`
g. `(-2) ** 3 ** 2`
h. `(-2) ** 2 ** 3`

Q2.2.3 Predict and explain the results of the following statements.

a. `9 + 6j / 2`
b. `complex(4, 5).conjugate().imag`
c. `complex(0, 3j)`
d. `round(2.5)`
e. `round(-2.5)`
f. `abs(complex(5, -4)) == math.hypot(4,5)`

Q2.2.4 Determine the value of i^i as a real number, where $i = \sqrt{-1}$.

Q2.2.5 Explain the (surprising?) behavior of the following short code:

```
>>> d = 8
>>> e = 2
>>> from math import *
```

```
>>> sqrt(d ** e)
16.88210319127114
```

Q2.2.6 Formally, the integer division, `a // b` is defined as the *floor* of a/b (sometimes written $\lfloor \frac{a}{b} \rfloor$) – that is, the largest integer less than or equal to `a / b`. The modulus or remainder, `a % b` (written a mod b), is then

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor .$$

Use these definitions to predict the result of the following expressions and check them using the Python shell.

a. `7 // 4`
b. `7 % 4`
c. `-7 // 4`
d. `-7 % 4`
e. `7 // -4`
f. `7 % -4`
g. `-7 // -4`
h. `-7 % -4`

Q2.2.7 If two adjacent sides of a regular, six-sided die have the values a and b when viewed side-on and read left to right, the value on the top of the die is given by $3(a^3b - ab^3) \mod 7$.

Determine the value on the top of the die if (a) $a = 2, b = 6$, (b) $a = 3, b = 5$.

Q2.2.8 How many times must a sheet of paper (thickness, $t = 0.1$ mm but otherwise any size required) be folded to reach the moon (distance from Earth, $d = 384,400$ km)?

Q2.2.9 Predict the results of the following expressions and check them using the Python shell.

a. `not 1 < 2 or 4 > 2`
b. `not (1 < 2 or 4 > 2)`
c. `1 < 2 or 4 > 2`
d. `4 > 2 or 10/0 == 0`
e. `not 0 < 1`
f. `1 and 2`
g. `0 and 1`
h. `1 or 0`
i. `type(complex(2, 3).real) is int`

Q2.2.10 Explain why the following expression does not evaluate to 100.

```
>>> 10^2
8
```

Hint: refer to the Python documentation for *bitwise* operators.

Problems

P2.2.1 There is no exclusive-or operator provided "out of the box" by Python, but one can be constructed from the existing operators. Devise two different ways of doing this. The truth table for the xor operator is given in Table 2.9.

P2.2.2 Some fun with the `math` module:

a. What is special about the numbers $\sin\left(2017\sqrt[5]{2}\right)$ and $(\pi + 20)^i$?

b. What happens if you try to evaluate an expression, such as e^{1000}, which generates a number larger than the largest floating point number that can be represented in the default double precision? What if you restrict your calculation to integer arithmetic (e.g., by evaluating 1000!)?

c. What happens if you try to perform an undefined mathematical operation such as division by zero?

d. The maximum representable floating point number in IEEE 754 double precision is about 1.8×10^{308}. Calculate the length of the hypotenuse of a right angled triangle with opposite and adjacent sides 1.5×10^{200} and 3.5×10^{201} (i) using the `math.hypot()` function directly and (ii) without using this function.

P2.2.3 Some languages provide a `sign(a)` function which returns -1 if its argument, `a`, is negative and 1 otherwise. *Python does not provide such a function*, but the `math` module does include a function `math.copysign(x, y)`, which returns the absolute value of `x` with the sign of `y`. How would you use this function in the same way as the missing `sign(a)` function?

P2.2.4 The World Geodetic System is a set of international standards for describing the shape of the Earth. In the latest WGS-84 revision, the Earth's *geoid* is approximated to a reference ellipsoid that takes the form of an oblate spheroid with semi-major and semi-minor axes $a = 6378137.0$ m and $c = 6356752.314245$ m respectively.

Use the formula for the surface area of an oblate spheroid,

$$S_{\text{obl}} = 2\pi a^2 \left(1 + \frac{1 - e^2}{e}\text{atanh}(e)\right), \quad \text{where } e^2 = 1 - \frac{c^2}{a^2},$$

to calculate the surface area of this reference ellipsoid and compare it with the surface area of the Earth assumed to be a sphere with radius 6371 km.

Table 2.9 Truth table for the `xor` operator

P	Q	P xor Q
True	True	False
False	True	True
True	False	True
False	False	False

2.3 Python objects I: strings

2.3.1 Defining a string object

A Python string object (of type `str`) is an ordered, immutable sequence of characters. To define a variable containing some constant text (a *string literal*), enclose the text in either single or double quotes:

```
>>> greeting = "Hello, Sir!"
>>> bye = 'À bientôt'
```

Strings can be concatenated using either the + operator or by placing them next to each other on the same line:

```
>>> 'abc' + 'def'
'abcdef'
>>> 'one ' 'two' ' three'
'one two three'
```

Python doesn't place any restriction on the length of a line, so a string literal can be defined in a single, quoted block of text. However, for ease of reading, it is usually a good idea to keep the lines of your program to a fixed maximum length (79 characters is recommended). To break up a string over two or more lines of code, use the line continuation character, '\' or (better) enclose the string literal in parentheses:

```
>>> long_string = 'We hold these truths to be self-evident,'\
...               ' that all men are created equal...'

>>> long_string = ('We hold these truths to be self-evident,'
...                ' that all men are created equal...')
```

This defines the variable `long_string` to hold a single line of text (with no carriage returns). The concatenation does not insert spaces so they need to be included explicitly if they are wanted. The spaces lining up the opening quotes in this example are optional but make the code easier to read.

If your string consists of a repetition of one or more characters, the * operator can be used to concatenate them the required number of times:

```
>>> 'a'*4
'aaaa'
>>> '-o-'*5
'-o--o--o--o--o-'
```

The *empty string* is defined simply as `s = ''` (two single quotes) or `s = ""`.

Finally, the built-in function, `str` converts an object passed as its argument into a string according to a set of rules defined by the object itself:

```
>>> str(42)
'42'
>>> str(3.4e5)
'340000.0'
>>> str(3.4e20)
'3.4e+20'
```

For finer control over the formatting of the string representation of numbers, see Section 2.3.7.

Example E2.8 Strings concatenated with the '+' operator can repeated with '*', but only if enclosed in parentheses:

```
>>> ('a'*4 + 'B')*3
'aaaaBaaaaBaaaaB'
```

2.3.2 Escape sequences

The choice of quotes for strings allows one to include the quote character itself inside a string literal – just define it using the other quote:

```
>>> verse = 'Quoth the Raven "Nevermore."'
```

But what if you need to include both quotes in a string? Or to include more than one line in the string? This case is handled by special *escape sequences* indicated by a backslash, \. The most commonly used escape sequences are listed in Table 2.10. For example,

```
>>> sentence = "He said, \"This parrot's dead.\""
❶ >>> sentence
'He said, "This parrot\'s dead."'
❷ >>> print(sentence)
He said, "This parrot's dead."
>>> subjects = 'Physics\nChemistry\nGeology\nBiology'
>>> subjects
'Physics\nChemistry\nGeology\nBiology'
>>> print(subjects)
Physics
Chemistry
Geology
Biology
```

❶ Note that just typing a variable's name at the Python shell prompt simply echoes its literal value back to you (in quotes).

Table 2.10 Common Python escape sequences

Escape sequence	Meaning
\'	Single quote (')
\"	Double quote (")
\n	Linefeed (LF)
\r	Carriage return (CR)
\t	Horizontal tab
\b	Backspace
\\	The backslash character itself
\u, \U, \N{}	Unicode character (see Section 2.3.3)
\x	Hex-encoded byte

❷ To produce the desired string including the proper interpretation of special characters, pass the variable to the `print` built-in function (see Section 2.3.6).

On the other hand, if you want to define a string to include character sequences such as '`\n`' *without them being escaped*, define a *raw string* prefixed with `r`:

```
>>> rawstring = r'The escape sequence for a new line is \n.'
>>> rawstring
'The escape sequence for a new line is \\n.'
>>> print(rawstring)
The escape sequence for a new line is \n.
```

When defining a block of text including several line endings it is often inconvenient to use `\n` repeatedly. This can be avoided by using *triple-quoted strings*: new lines defined within strings delimited by `"""` and `'''` are preserved in the string:[18]

```
a = """one
two
three"""
>>> print(a)
one
two
three
```

This is often used to create "docstrings" which document blocks of code in a program (see Section 2.7.1).

Example E2.9 The `\x` escape denotes a character encoded by the single-byte hex value given by the subsequent two characters. For example, the capital letter '`N`' has the value 78, which is `4e` in hex. Hence,

```
>>> '\x4e'
'N'
```

The *backspace* "character" is encoded as hex `08`, which is why `'\b'` is equivalent to `'\x08'`:

```
>>> 'hello\b\b\b\bgoodbye'
'hello\x08\x08\x08\x08\x08goodbye'
```

Sending this string to the `print()` function outputs the string formed by the sequence of characters in this string literal:

```
>>> print('hello\b\b\b\bgoodbye')
goodbye
```

2.3.3 Unicode

Python 3 strings are composed of *Unicode* characters. Unicode is a standard describing the representation of more than 100,000 characters in just about every human language, as well as many other specialist characters such as scientific symbols. It does this by

[18] It is generally considered better to use three *double* quotes, `"""`, for this purpose.

assigning a number (*code point*) to every character; the numbers that make up a string are then encoded as a sequence of bytes.[19] For a long time, there was no agreed encoding standard, but the *UTF-8* encoding, which is used by Python 3 by default, has emerged as the most widely used today.[20] If your editor will not allow you to enter a character directly into a string literal, you can use its 16- or 32-bit hex value or its Unicode character name as an escape sequence:

```
>>> '\u00E9' # 16-bit hex value
'é'
>>> '\u000000E9' # 32-bit hex value
'é'
>>> '\N{LATIN SMALL LETTER E WITH ACUTE}' # by name
'é'
```

Example E2.10 Providing your editor or terminal allows it, and you can type them at your keyboard or paste them from elsewhere (e.g, a web browser or word processor), Unicode characters can be entered directly into string literals:

```
>>> creams = 'Crème fraîche, crème brûlée, crème pâtissière'
```

Python even supports Unicode variable names, so identifiers can use non-ASCII characters:

```
>>> Σ = 4
>>> crème = 'anglaise'
```

Needless to say, because of the potential difficulty in entering non-ASCII characters from a standard keyboard and because many distinct characters look very similar, this is not a good idea.

2.3.4 Indexing and slicing strings

Indexing (or "subscripting") a string returns a single character at a given location. Like all sequences in Python, strings are indexed with the first character having the index 0; this means that the final character in a string consisting of n characters is indexed at $n - 1$. For example,

```
>>> a = "Knight"
>>> a[0]
'K'
>>> a[3]
'g'
```

The character is returned in a `str` object of length 1. A non-negative index counts forward from the start of the string; there is a handy notation for the index of a string counting backward: a negative index, starting at -1 (for the final character) is used. So,

```
>>> a = "Knight"
>>> a[-1]
't'
```

[19] For a list of code points, see the official Unicode website's code charts at www.unicode.org/charts/.

[20] UTF-8 encoded Unicode encompasses the venerable 8-bit encoding of the ASCII character set (e.g., `A=65`).

```
>>> a[-4]
'i'
```

It is an error to attempt to index a string outside its length (here, with index greater than 5 or less than −6): Python raises an `IndexError`:

```
>>> a[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Slicing a string, `s[i:j]`, produces a substring of a string between the characters at two indexes, *including* the first (`i`) but *excluding* the second (`j`). If the first index is omitted, 0 is assumed; if the second is omitted, the string is sliced to its end. For example,

```
>>> a = "Knight"
>>> a[1:3]
'ni'
>>> a[:3]
'Kni'
>>> a[3:]
'ght'
>>> a[:]
'Knight'
```

This can seem confusing at first, but it ensures that the length of a substring returned as `s[i:j]` has length `j-i` (for positive `i`, `j`) and that `s[:i] + s[i:] == s`. Unlike indexing, slicing a string outside its bounds does not raise an error:

```
>>> a= "Knight"
>>> a[3:10]
'ght'
>>> a[10:]
''
```

To test if a string contains a given substring, use the `in` operator:

```
>>> 'Kni' in 'Knight':
True
>>> 'kni' in 'Knight':
False
```

Example E2.11 Because of the nature of slicing, `s[m:n]`, n-m is always the length of the substring. In other words, to return `r` characters starting at index `m`, use `s[m:m+r]`. For example,

```
>>> s = 'whitechocolatespaceegg'
>>> s[:5]
'white'
>>> s[5:14]
'chocolate'
>>> s[14:19]
'space'
>>> s[19:]
'egg'
```

Example E2.12 The optional, third number in a slice specifies the *stride*. If omitted, the default is 1: return every character in the requested range. To return every *k*th letter, set the stride to `k`. Negative values of `k` reverse the string. For example,

```
>>> s = 'King Arthur'
>>> s[::2]
'Kn rhr'
>>> s[1::2]
'igAtu'
>>> s[-1:4:-1]
'ruhtrA'
```

This last slice can be explained as a selection of characters from the last (index `-1`) down to (but not including) character at index `4`, with stride `-1` (select every character, in the reverse direction).

A convenient way of reversing a string is to slice between default limits (by omitting the first and last indexes) with a stride of `-1`:

```
>>> s[::-1]
'ruhtrA gniK'
```

2.3.5 String methods

Python strings are *immutable* objects, and so it is not possible to change a string by assignment – for example, the following is an error:

```
>>> a = 'Knight'
>>> a[0] = 'k'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

New strings *can* be constructed from existing strings, but only as new objects. For example,

```
>>> a += ' Templar'
>>> print(a)
Knight Templar
>>> b = 'Black ' + a[:6]
>>> print(b)
Black Knight
```

To find the number of characters a string contains, use the `len` built-in method:

```
>>> a = 'Earth'
>>> len(a)
5
```

String objects come with a large number of methods for manipulating and transforming them. These are accessed using the usual dot notation we have met already – some of the more useful ones are listed in Table 2.11. In this and similar tables, text in *italics* is intended to be replaced by a specific value appropriate to the use of the method; italic text in `[`*`square brackets`*`]` denotes an optional argument.

Table 2.11 Some common string methods

Method	Description
`center(`*`width`*`)`	Return the string centered in a string with total number of characters *`width`*.
`endswith(`*`suffix`*`)`	Return `True` if the string ends with the substring *`suffix`*.
`startswith(`*`prefix`*`)`	Return `True` if the string starts with the substring *`prefix`*.
`index(`*`substring`*`)`	Return the lowest index in the string containing *`substring`*.
`lstrip([`*`chars`*`])`	Return a copy of the string with any of the leading characters specified by `[`*`chars`*`]` removed. If `[`*`chars`*`]` is omitted, any leading whitespace is removed.
`rstrip([`*`chars`*`])`	Return a copy of the string with any of the trailing characters specified by `[`*`chars`*`]` removed. If `[`*`chars`*`]` is omitted, any trailing whitespace is removed.
`strip([`*`chars`*`])`	Return a copy of the string with leading and trailing characters specified by `[`*`chars`*`]` removed. If `[`*`chars`*`]` is omitted, any leading and trailing whitespace is removed.
`upper()`	Return a copy of the string with all characters in uppercase.
`lower()`	Return a copy of the string with all characters in lowercase.
`title()`	Return a copy of the string with all words starting with capitals and other characters in lowercase.
`replace(`*`old, new`*`)`	Return a copy of the string with each substring *`old`* replaced with *`new`*.
`split([`*`sep`*`])`	Return a `list` (see Section 2.4.1) of substrings from the original string which are separated by the string *`sep`*. If *`sep`* is not specified, the separator is taken to be any amount of whitespace.
`join([`*`list`*`])`	Use the string as a separator in joining a `list` of strings.
`isalpha()`	Return `True` if all characters in the string are alphabetic and the string is not empty; otherwise return `False`.
`isdigit()`	Return `True` if all characters in the string are digits and the string is not empty; otherwise return `False`.

Because these methods each return a new string (remember that strings are immutable objects), they can be chained together:

```
>>> s = '-+-Python Wrangling for Beginners'
>>> s.lower().replace('wrangling', 'programming').lstrip('+-')
'python programming for beginners'
```

Example E2.13 Here are some possible manipulations using string methods:

```
>>> a = 'java python c++ fortran'
>>> a.isalpha()
❶ False
>>> b = a.title()
>>> b
'Java Python C++ Fortran'
>>> c = b.replace(' ', '!\n')
>>> c
```

```
'Java!\nPython!\nC++!\nFortran'
>>> print(c)
Java!
Python!
C++!
Fortran!
>>> c.index('Python')
❷ 6
>>> c[6:].startswith('Py')
True
>>> c[6:12].isalpha()
True
```

❶ `a.isalpha()` is `False` because of the spaces and `'++'`.
❷ Note that `\n` is a single character.

2.3.6 The `print` function

One of the most obvious changes between Python 2 and Python 3 is in the way that `print` works. In the older version of Python, `print` was a *statement* that output the string representation of a list of objects, separated by spaces:

```
>>> ans = 6
>>> print 'Solve:', 2, 'x =', ans, 'for x'            # Python 2 only!
Solve: 2 x = 6 for x
```

(There was also a special syntax for redirecting the output to a file.) Python 3 adopts a more consistent and flexible approach: `print` is a built-in *function* (just like the others we have met such as `len` and `round`.) It takes a list of objects, and also, optionally, arguments `end` and `sep`, that specify which characters should end the string and which characters should be used to separate the printed objects respectively. Omitting these additional arguments gives the same result as the old `print` statement: the object fields are separated by a *single space* and the line is ended with a *newline* character.[21] For example,

```
>>> ans = 6
>>> print('Solve:', 2, 'x =', ans, 'for x')
Solve: 2 x = 6 for x
>>> print('Solve: ', 2, 'x = ', ans, ' for x', sep='', end='!\n')
Solve: 2x = 6 for x!
❶ >>> print()

>>> print('Answer: x =', ans/2)
Answer: x = 3.0
```

❶ Note that `print()` with no arguments just prints the default newline `end` character.

To suppress the newline at the end of a printed string, specify `end` to be the empty string: `end=''`:

[21] The specific newline character used depends on the operating system: for example, on a Mac it is '`\n`', on Windows it is two characters: '`\r\n`'.

```
>>> print('A line with no newline character', end='')
A line with no newline character>>>
```

The chevrons, `>>>`, at the end of this line form the prompt for the next Python command to be entered.

Example E2.14 `print` can be used to create simple text tables:

```
>>> heading = '| Index of Dutch Tulip Prices |'
>>> line = '+' + '-'*16 + '-'*13 + '+'
>>> print(line, heading, line,
...       '|    Nov 23 1636 |        100 |',
...       '|    Nov 25 1636 |        673 |',
...       '|    Feb  1 1637 |       1366 |', line, sep='\n')
...
+-----------------------------+
| Index of Dutch Tulip Prices |
+-----------------------------+
|    Nov 23 1636 |        100 |
|    Nov 25 1636 |        673 |
|    Feb  1 1637 |       1366 |
+-----------------------------+
```

2.3.7 String formatting

Introduction to Python 3 string formatting

In its simplest form, it is possible to use a string's `format` method to insert objects into it. The most basic syntax is

```
>>> '{} plus {} equals {}'.format(2, 3, 'five')
2 plus 3 equals five
```

Here, the `format` method is called on the string literal with the arguments `2`, `3` and `'five'` which are interpolated, in order, into the locations of the *replacement fields*, indicated by braces, `{}`. Replacement fields can also be numbered or named, which helps with longer strings and allows the same value to be interpolated more than once:[22]

```
>>> '{1} plus {0} equals {2}'.format(2, 3, 'five')
'3 plus 2 equals five'
>>> '{num1} plus {num2} equals {answer}'.format(num1=2, num2=3, answer='five')
'2 plus 3 equals five'
>>> '{0} plus {0} equals {1}'.format(2, 2+2)
'2 plus 2 equals 4'
```

Note that numbered fields can appear in any order and are indexed starting at 0.

Replacement fields can be given a minimum size within the string by the inclusion of an integer length after a colon as follows:

```
>>> '=== {0:12} ==='.format('Python')
'=== Python       ==='
```

[22] This type of string formatting was introduced into Python 2 as well, although only Python 2.7 supports *unnamed* replacement fields denoted by empty braces, `{}`.

If the string is too long for the minimum size, it will take up as many characters as needed (overriding the replacement field size specified):

```
>>> 'A number: <{0:2}>'.format(-20)
'A number: <-20>'       # -20 won't fit into 2 characters: 3 are used anyway
```

By default, the interpolated string is aligned to the left; this can be modified to align to the right or to center the string. The single characters `<`, `>` and `^` control the alignment:

```
>>> '=== {0:<12} ===' .format('Python')
'=== Python       ==='
>>> '=== {0:>12} ===' .format('Python')
'===       Python ==='
>>> '=== {0:^12} ===' .format('Python')
'===    Python    ==='
```

In these examples, the field is padded with spaces, but this fill character can also be specified. For example, to pad with hyphens in the last example, specify

```
>>> '=== {0:-^12} ===' .format('Python')
'=== ---Python--- ==='
```

It is even possible to pass the minimum field size as a parameter to be interpolated. Just replace the field size with a reference in braces as follows:

```
>>> a = 15
>>> 'This field has {0} characters: ==={1:>{2}}===.'.format(a, 'the field', a)
'This field has 15 characters: ===      the field===.'
```

Or with named interpolation:

```
>>> 'This field has {w} characters: ==={0:>{w}}===.'.format('the field', w=a)
'This field has 15 characters: ===      the field===.'
```

In each case, the second format specifier here has been taken to be `:>15`.

To insert the brace characters themselves into a formatted string, they must be doubled up: use '{{' and '}}'.

Formatting numbers

The Python 3 string `format` method provides a powerful way to format numbers.

The specifiers '`d`', '`b`', '`o`', '`x`'/'`X`' indicate a decimal, binary, octal and lowercase/uppercase hex *integer* respectively:

```
>>> a = 254
>>> 'a = {0:5d}'.format(a)      # decimal
'a =   254'
>>> 'a = {0:10b}'.format(a)     # binary
'a =   11111110'
>>> 'a = {0:5o}'.format(a)      # octal
'a =   364'
>>> 'a = {0:5x}'.format(a)      # hex (lowercase)
'a =    fe'
>>> 'a = {0:5X}'.format(a)      # hex (uppercase)
'a =    FE'
```

Numbers can be padded with zeros to fill out the specified field size by prefixing the minimum width with a `0`:

```
>>> a = 254
>>> 'a = {a:05d}'.format(a=a)
'a = 00254'
```

By default, the sign of a number is only output if it is negative. This behavior can also be customized by specifying, before the minimum width:

- '+': always output a sign;
- '-': only output a negative sign, the default; or
- ' ': output a leading space only if the number is positive.

This last option enables columns of positive and negative numbers to be lined up nicely:

```
>>> print('{0: 5d}\n{1: 5d}\n{2: 5d}'.format(-4510, 1001, -3026))
-4510
 1001
-3026
>>> a = -25
>>> b = 12
>>> s = '{0:+5d}\n{1:+5d}\n= {2:+3d}'.format(a, b, a+b)
>>> print(s)
  -25
  +12
= -13
```

There are also format specifiers for floating point numbers, which can be output to a chosen precision if desired. The most useful options are 'f': fixed-point notation, 'e'/'E': exponent (i.e., "scientific" notation), and 'g'/'G': a general format which uses scientific notation for very large and very small numbers.[23] The desired precision (number of decimal places) is specified as '.p' after the minimum field width. Some examples:

```
>>> a = 1.464e-10
>>> '{0:g}'.format(a)
'1.464e-10'
>>> '{0:10.2E}'.format(a)
'  1.46E-10'
>>> '{0:15.13f}'.format(a)
'0.0000000001464'
>>> '{0:10f}'.format(a)
'  0.000000'
```

❶ Note that Python will not protect you from this kind of rounding to zero if not enough space is provided for a fixed-point number.

Older C-style formatting

Python 3 also supports the less powerful, C-style format specifiers that are still in widespread use. In this formulation the replacement fields are specified with the minimum width and precision specifiers following a % sign. The objects whose values are to be interpolated are then given after the end of the string, following another % sign. They

[23] More specifically, the g/G specifier acts like f/F for numbers between 10^{-4} and 10^{p} where p is the desired precision (which defaults to 6), and acts like e/E otherwise.

must be enclosed in parentheses if there is more than one of them. The same letters for the different output types are used as earlier; strings must be specified explicitly with '%s'. For example,

```
>>> kB = 1.3806504-23
>>> 'Here\'s a number: %10.2e' % kB
"Here's a number:   1.38e-23"
>>> 'The same number formatted differently: %7.1e and %12.6e' % (kB, kB)
'The same number formatted differently: 1.4e-23 and 1.380650e-23'
>>> '%s is %g J/K' % ("Boltzmann's constant", kB)
"Boltzmann's constant is 1.38065e-23 J/K"
```

Example E2.15 Python can produce string representations of numbers for which thousands are separated by commas:

```
>>> '{:11,d}'.format(1000000)
'  1,000,000'
>>> '{:11,.1f}'.format(1000000.)
'1,000,000.0'
```

Here is another table, produced using several different string methods:

```
title = '|' + '{:^51}'.format('Cereal Yields (kg/ha)') + '|'
line = '+' + '-'*15 + '+' + ('-'*8 + '+')*4
row = '| {:<13} |' + ' {:6,d} |'*4
header = '| {:^13s} |'.format('Country') + (' {:^6d} |'*4).format(1980, 1990,
                                                                  2000, 2010)
print('+' + '-'*(len(title)-2) + '+',
      title,
      line,
      header,
      line,
      row.format('China', 2937, 4321, 4752, 5527),
      row.format('Germany', 4225, 5411, 6453, 6718),
      row.format('United States', 3772, 4755, 5854, 6988),
      line,
      sep='\n')
```

```
+-------------------------------------------------+
|               Cereal Yields (kg/ha)             |
+---------------+--------+--------+--------+------+
|    Country    |  1980  |  1990  |  2000  | 2010 |
+---------------+--------+--------+--------+------+
| China         |  2,937 |  4,321 |  4,752 | 5,527 |
| Germany       |  4,225 |  5,411 |  6,453 | 6,718 |
| United States |  3,772 |  4,755 |  5,854 | 6,988 |
+---------------+--------+--------+--------+------+
```

2.3.8 Exercises

Questions

Q2.3.1 Slice the string `s='seehemewe'` to produce the following substrings:

a. `'see'`
b. `'he'`

c. `'me'`
d. `'we'`
e. `'hem'`
f. `'meh'`
g. `'wee'`

Q2.3.2 Write a single-line expression for determining if a string is a palindrome (reads the same forward as backward).

Q2.3.3 Predict the results of the following statements and check them using the Python shell.

```
>>> days = 'Sun Mon Tues Weds Thurs Fri Sat'
```

a. `print(days[days.index('M'):])`
b. `print(days[days.index('M'):days.index('Sa')].rstrip())`
c. `print(days[6:3:-1].lower()*3)`
d. `print(days.replace('rs', '').replace('s ', ' ')[::4])`
e. `print(' -*- '.join(days.split()))`

Q2.3.4 What is the output of the following code? How does it work?

```
>>> suff = 'thstndrdthththththththth'
>>> n = 1
>>> print('{:d}{:s}'.format(n, suff[n*2:n*2+2]))
>>> n = 3
>>> print('{:d}{:s}'.format(n, suff[n*2:n*2+2]))
>>> n = 5
>>> print('{:d}{:s}'.format(n, suff[n*2:n*2+2]))
```

Q2.3.5 Consider the following (incorrect) tests to see if the string 's' has one of two values. Explain how these statements are interpreted by Python and give a correct alternative.

```
>>> s = 'eggs'
>>> s == ('eggs' or 'ham')
True

>>> s == ('ham' or 'eggs')
False
```

Problems

P2.3.1 a. Given a string representing a base-pair sequence (i.e., containing only the letters A, G, C and T), determine the fraction of G and C bases in the sequence. (*Hint:* strings have a `count` method, returning the number of occurrences of a substring.)

b. Using only string methods, devise a way to determine if a nucleotide sequence is a palindrome in the sense that it is equal to its own complementary sequence read backward. For example, the sequence TGGATCCA is palindromic because

its complement is ACCTAGGT which is the same as the original sequence backward. The complementary base pairs are (A, T) and (C, G).

P2.3.2 The table that follows gives the names, symbols, values, uncertainties and units of some physical constants.

Defining variables of the form

Name	Symbol	Value	Uncertainty	Units
Boltzmann constant	k_B	$1.3806504 \times 10^{-23}$	2.4×10^{-29}	$\mathrm{J\,K^{-1}}$
Speed of light	c	299792458	(def)	$\mathrm{m\,s^{-1}}$
Planck constant	h	$6.62606896 \times 10^{-34}$	3.3×10^{-41}	J s
Avogadro constant	N_A	$6.02214179 \times 10^{23}$	3×10^{16}	$\mathrm{mol^{-1}}$
Electron magnetic moment	μ_e	$-9.28476377 \times 10^{-24}$	2.3×10^{-31}	J/T
Gravitational constant	G	6.67428×10^{-11}	6.7×10^{-15}	$\mathrm{N\,m^2\,kg^{-2}}$

```
kB = 1.3806504e-23  # J/K
kB_unc = 2.4e-29    # uncertainty
kB_units = 'J/K'
```

use the string object's `format` method to produce the following output:

a. `kB = 1.381e-23 J/K`

b. `G = 0.0000000000667428 Nm^2/kg^2`

c. Using the same format specifier for each line,

```
kB   =  1.3807e-23 J/K
mu_e = -9.2848e-24 J/T
N_A  =  6.0221e+23 mol-1
c    =  2.9979e+08 m/s
```

d. Again, using the same format specifier for each line,

```
===    G = +6.67E-11 [Nm^2/kg^2]    ===
===   μe = -9.28E-24 [      J/T]    ===
```

Hint: the Unicode codepoint for the lowercase Greek letter mu is `U+03BC`.

e. (Harder). Produce the output below, in which the uncertainty (one standard deviation) in the value of each constant is expressed as a number in parentheses relative the preceding digits: that is, $6.62606896(33) \times 10^{-34}$ means $6.62606896 \times 10^{-34} \pm 3.3 \times 10^{-41}$.

```
G = 6.67428(67)e-11 Nm2/kg2
mu_e = -9.28476377(23)e-24 J/T
```

P2.3.3 Given the elements of a 3×3 matrix as the nine variables `a11`, `a12`, ... `a33`, produce a string representation of the matrix using formatting methods, (a) assuming the matrix elements are (possibly negative) real numbers to be given to one decimal place; (b) assuming the matrix is a permutation matrix with integer entries taking the values 0 or 1 only. For example,

```
>>> print(s_a)
[  0.0  3.4 -1.2 ]
[ -1.1  0.5 -0.2 ]
[  2.3 -1.4 -0.7 ]
>>> print(s_b)
[ 0 0 1 ]
[ 0 1 0 ]
[ 1 0 0 ]
```

P2.3.4 Find the Unicode code points for the planet symbols listed on the NASA website (http://solarsystem.nasa.gov/multimedia/display.cfm?IM_ID=167) which mostly fall within the hex range 2600–26FF: Miscellaneous Symbols (www.unicode.org/charts/PDF/U2600.pdf) and output a list of planet names and symbols.

2.4 Python objects II: lists, tuples and loops

2.4.1 Lists

Initializing and indexing lists

Python provides data structures for holding an ordered list of objects. In some other languages (e.g., C and Fortran) such a data structure is called an *array* and can hold only one type of data (e.g., an array of integers); the core array structures in Python, however, can hold a mixture of data types.

A Python *list* is an ordered, *mutable* array of objects. A list is constructed by specifying the objects, separated by commas, between square brackets, `[]`. For example,

```
>>> list1 = [1, 'two', 3.14, 0]
>>> list1
[1, 'two', 3.14, 0]
>>> a = 4
>>> list2 = [2, a, -0.1, list1, True]
>>> list2
[2, 4, -0.1, [1, 'two', 3.14, 0], True]
```

Note that a Python list can contain references to any type of object: strings, the various types of numbers, built-in constants such as the boolean value `True`, and even other lists. It is not necessary to declare the size of a list in advance of using it. An empty list can be created with `list0 = []`.

An item can be retrieved from the list by indexing it (remember Python indexes start at 0):

```
>>> list1[2]
3.14
>>> list2[-1]
True
>>> list2[3][1]
'two'
```

This last example retrieves the second (index: `1`) item of the fourth (index: `3`) item of `list2`. This is valid because the item `list2[3]` happens to be a list (the one also identified by the variable name `list1`), and `list1[1]` is the string `'two'`. In fact, since strings can also be indexed:

```
>>> list2[3][1][1]
'w'
```

To test for membership of a list, the operator `in` is used, as for strings:

```
>>> 1 in list1
True
>>> 'two' in list2:
False
```

This last expression evaluates to `False` because `list2` does not contain the string literal `'two'` even though it contains `list1` which does: the `in` operator does not recurse into lists-of-lists when it tests for membership.

Lists and mutability

Python lists are the first *mutable* object we have encountered. Unlike strings, which cannot be altered once defined, the items of a list can be reassigned:

```
>>> list1
[1, 'two', 3.14, 0]
>>> list1[2] = 2.72
>>> list1
[1, 'two', 2.72, 0]
>>> list2
[2, 4, -0.1, [1, 'two', 2.72, 0], True]
```

Note that not only has `list1` been changed, but `list2` (which contains `list1` as an item) *has also changed*.[24] This behavior catches a lot of people out to begin with, particularly if a list needs to be copied to a different variable.

```
>>> q1 = [1, 2, 3]
>>> q2 = q1
>>> q1[2] = 'oops'
>>> q1
[1, 2, 'oops']
>>> q2
[1, 2, 'oops']
```

Here, the variables `q1` and `q2` refer to the *same list*, stored in the same memory location, and because lists are mutable, the line `q1[2] = 'oops'` actually changes one of the stored values at that location; `q2` still points to the same location and so it appears to have changed as well. In fact, there is only one list (referred to by two variable names) and it is changed once. In contrast, integers are *immutable*, so the following does not change the value of `q[2]`:

```
>>> a = 3
>>> q = [1, 2, a]
>>> a = 4
>>> q
[1, 2, 3]
```

[24] Actually, it hasn't changed: it only ever contained a series of references to objects: the reference to `list1` is the same, even though the references within `list1` have changed.

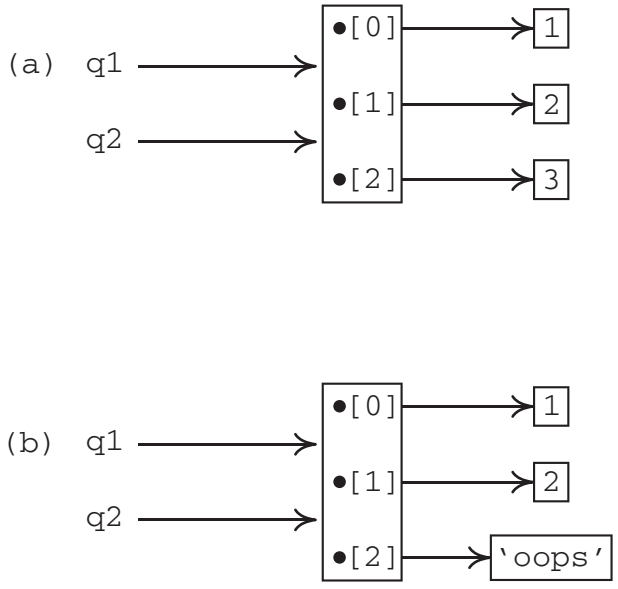

Figure 2.2 Two variables referring to the same list: (a) on initialization and (b) after setting `q1[2] = 'oops'`.

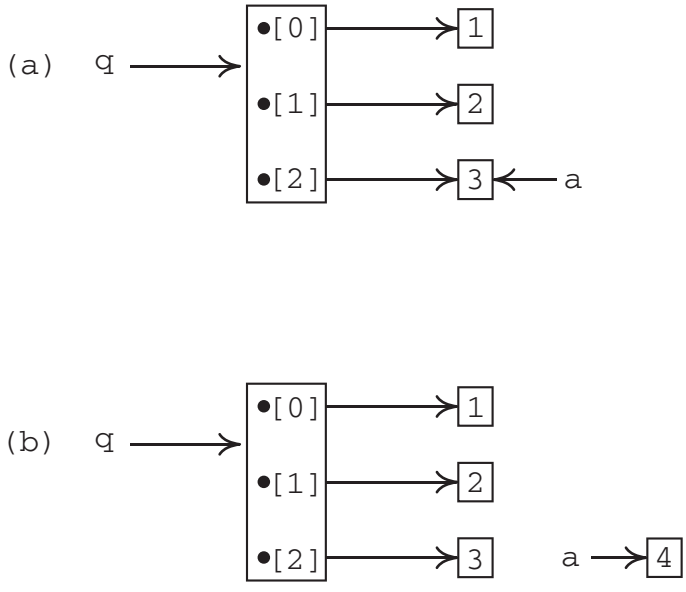

Figure 2.3 A list defined with `q = [1, 2, a]` where `a=3`: (a) on initialization and (b) after changing the value of a with `a=4`.

The assignment `a=4` creates a whole new integer object, quite independent of the original `3` that ended up in the list `q`. This original integer object isn't changed by the assignment (integers are immutable) and so the list is unchanged. This distinction is illustrated by Figures 2.2, 2.3 and 2.4.

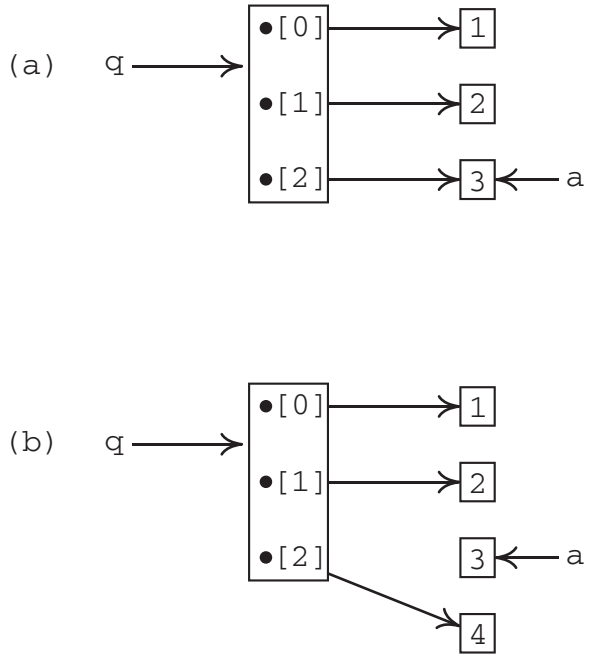

Figure 2.4 A list defined with `q = [1, 2, a]` where `a=3`: (a) on initialization and (b) after changing the value of q with `q[2]=4`.

Lists can be *sliced* in the same way as string sequences:

```
>>> q1 = [0., 0.1, 0.2, 0.3, 0.4, 0.5]
>>> q1[1:4]
[0.1, 0.2, 0.3]
>>> q1[::-1]        # return a reversed copy of the list
[0.5, 0.4, 0.3, 0.2, 0.1, 0.0]
>>> q1[1::2]        # striding: returns elements at 1, 3, 5
[0.1, 0.3, 0.5]
```

Taking a slice *copies the data* to a new list. Hence,

```
>>> q2 = q1[1:4]
>>> q2[1] = 99      # only affects q2
>>> q2
[0.1, 99, 0.3]
>>> q1
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

List methods

Just as for strings, Python lists come with a large number of useful methods, summarized in Table 2.12. Because `list` objects are mutable, they can grow or shrink *in place*, that is, without having to copy the contents to a new object, as we had to do with strings. The relevant methods are

- `append`: add an item to the end of the list;

Table 2.12 Some common list methods

Method	Description
`append(element)`	Append *element* to the end of the list.
`extend(list2)`	Extend the list with the elements from *list2*.
`index(element)`	Return the lowest index of the list containing *element*.
`insert(index, element)`	Insert *element* at index *index*.
`pop()`	Remove and return the last element from the list.
`reverse()`	Reverse the list in place.
`remove(element)`	Remove the first occurrence of *element* from the list.
`sort()`	Sort the list in place.
`copy()`	Return a copy of the list.
`count(element)`	Return the number of elements equal to *element* in the list.

- `extend`: add one or more objects by copying them from another list;[25]
- `insert`: insert an item at a specified index and
- `remove`: remove a specified item from the list.

```
>>> q = []
>>> q.append(4)
>>> q
[4]
>>> q.extend([6, 7, 8])
>>> q
[4, 6, 7, 8]
>>> q.insert(1, 5)  # insert 5 at index 1
>>> q
[4, 5, 6, 7, 8]
>>> q.remove(7)
>>> q
[4, 5, 6, 8]
>>> q.index(8)
3   # the item 8 appears at index 3
```

Two useful list methods are `sort` and `reverse`, which sort and reverse the list *in place*. That is, they change the list object, but *do not return a value*:

```
>>> q = [2, 0, 4, 3, 1]
>>> q.sort()
>>> q
[0, 1, 2, 3, 4]
>>> q.reverse()
>>> q
[4, 3, 2, 1, 0]
```

If you do want a sorted *copy* of the list, leaving it unchanged, you can use the `sorted` built-in function:

[25] Actually, any Python object that forms a *sequence* that can be iterated over (e.g., a string) can be used as the argument to `extend`

```
>>> q = ['a', 'e', 'A', 'c', 'b']
>>> sorted(q)
['A', 'a', 'b', 'c', 'e']   # returns a new list
>>> q
['a', 'e', 'A', 'c', 'b']   # the old list is unchanged
```

By default, `sort()` and `sorted()` order the items in an array in *ascending order*. Set the optional argument `reverse=True` to return the items in descending order:

```
>>> q = [10, 5, 5, 2, 6, 1, 67]
>>> sorted(q, reverse=True)
[67, 10, 6, 5, 5, 2, 1]
```

Python 3, unlike Python 2, does not allow direct comparisons between strings and numbers, so it is an error to attempt to sort a list containing a mixture of such types:

```
>>> q = [5, '4', 2, 8]
>>> q.sort()
TypeError: unorderable types: str() < int()
```

Example E2.16 The methods `append` and `pop` make it very easy to use a `list` to implement the data structure known as a *stack*:

```
>>> stack = []
>>> stack.append(1)
>>> stack.append(2)
>>> stack.append(3)
>>> stack.append(4)
>>> print(stack)
[1, 2, 3, 4]
>>> stack.pop()
4
>>> print(stack)
[1, 2, 3]
```

The end of the list is the top of the stack from which items may be added or removed (think of a stack of dinner plates).

Example E2.17 The string method, `split` generates a list of substrings from a given string, split on a specified separator:

```
>>> s = 'Jan Feb Mar Apr May Jun'
>>> s.split()       # By default, splits on whitespace
['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
>>> s = "J. M. Brown AND B. Mencken AND R. P. van't Rooden"
>>> s.split(' AND ')
['J. M. Brown', 'B. Mencken', "R. P. van't Rooden"]
```

2.4.2 Tuples

The `tuple` object

A `tuple` may be thought of as an immutable `list`. Tuples are constructed by placing the items inside parentheses:

```
>>> t = (1, 'two', 3.)
>>> t
(1, 'two', 3.0)
```

Tuples can be indexed and sliced in the same way as lists but, being immutable, they cannot be appended to, extended, or have elements removed from them:

```
>>> t = (1, 'two', 3.)
>>> t[1]
'two'
>>> t[2] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Although a tuple itself is immutable, it may *contain* references to mutable objects such as lists. Hence,

```
>>> t = (1, ['a', 'b', 'd'], 0)
>>> t[1][2] = 'c'   # OK to change the list within the tuple
>>> t
(1, ['a', 'b', 'c'], 0)
```

An empty `tuple` is created with empty parentheses: `t0 = ()`. To create a `tuple` containing only one item (a *singleton*), however, it is not sufficient to enclose the item in parentheses (which could be confused with other syntactical use of parentheses); instead, the lone item is given a trailing comma: `t = ('one',)`.

Uses of `tuples`

In some circumstances, particularly for simple assignments such as those in the previous section, the parentheses around a tuple's items are not required:

```
>>> t = 1, 2, 3
>>> t
(1, 2, 3)
```

This usage is an example of *tuple packing*. The reverse, *tuple unpacking* is a common way of assigning multiple variables in one line:

```
>>> a, b, c = 97, 98, 99
>>> b
98
```

This method of assigning multiple variables is commonly used in preference to separate assignment statements either on different lines or (very un-Pythonically) on a single line, separated by semicolons:

```
a = 97; b = 98; c = 99      # Don't do this!
```

Tuples are useful where a sequence of items cannot or should not be altered. In the previous example, the `tuple` object only exists in order to assign the variables `a`, `b` and `c`. The values to be assigned: `97`, `98` and `99` are packed into a tuple for the purpose of this statement (to be unpacked into the variables), but once this has happened, the tuple object itself is destroyed. As another example, a function (Section 2.7) may return more

than one object: these objects are returned packed into a tuple. If you need any further persuading, `tuples` are slightly faster for many uses than `lists`.

Example E2.18 In an assignment using the '=' operator the right-hand side expression is evaluated first. This provides a convenient way to swap the values of two variables using tuples:

```
a, b = b, a
```

Here, the right-hand side is packed into a tuple object, which is then unpacked into the variables assigned on the left-hand side. This is more convenient than using a temporary variable:

```
t = a
a = b
b = t
```

2.4.3 Iterable objects

Examples of iterable objects

Strings, lists and tuples are all examples of data structures that are *iterable* objects: they are ordered sequences of items (characters in the case of strings, or arbitrary objects in the case of lists and tuples) which can be taken one at a time. One way of seeing this is to use the alternative method of initializing a list (or tuple) using the built-in constructor methods `list()` and `tuple()`. These take any iterable object and generate a list and a tuple respectively from its sequence of items. For example,

```
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
>>> tuple([1, 'two', 3])
(1, 'two', 3)
```

Because the data elements are *copied* in the construction of a new object using these constructor methods, `list` is another way of creating an independent `list` object from another:

```
>>> a = [5, 4, 3, 2, 1]
>>> b = a          # b and a refer to the same list object
>>> b is a
True
>>> b = list(a)    # create an entirely new list object with the same contents as a
>>> b is a
False
```

Because slices also return a copy of the object references from a sequence, the idiom `b = a[:]` is often used in preference to `b = list(a)`.

`any` and `all`

The built-in function `any` tests whether any of the items in an iterable object are equivalent to `True`; `all` tests whether all of them are. For example,

```
>>> a = [1, 0, 0, 2, 3]
>>> any(a), all(a)
(True, False)          # some (but not all) of a's items are equivalent to True
>>> b = [[], False, 0.]
>>> any(b), all(b)
(False, False)         # none of b's items is equivalent to True
```

◊ * syntax

It is sometimes necessary to call a function with arguments taken from a list or other sequence. The * syntax, used in a function call unpacks such a sequence into positional arguments to the function (see also Section 2.7). For example, the `math.hypot` function takes two arguments, `a` and `b`, and returns the quantity $\sqrt{a^2 + b^2}$. If the arguments you wish to use are in a list or tuple, the following will fail:

```
>>> t = [3, 4]
>>> math.hypot(t)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: hypot expected 2 arguments, got 1
```

We tried to call `math.hypot()` with a single argument (the list object `t`), which is an error. We could index the list explicitly to retrieve the two values we need:

```
>>> t = [3, 4]
>>> math.hypot(t[0], t[1])
5.0
```

but a more elegant method is to *unpack* the list into arguments to the function with `*t`:

```
>>> math.hypot(*t)
5.0
```

`for` loops

It is often necessary to take the items in an iterable object one by one and do something with each in turn. Other languages, such as C, require this type of *loop* to refer to each item in turn by its integer index. In Python this is possible, but the more natural and convenient way is with the idiom:

`for` *item* `in` *iterable object*`:`

which yields each element of the iterable object in turn to be processed by the subsequent block of code. For example,

```
>>> fruit_list = ['apple', 'melon', 'banana', 'orange']
>>> for fruit in fruit_list:
...     print(fruit)
...
apple
melon
banana
orange
```

Each item in the `list` object `fruit_list` is taken in turn and assigned to the variable `fruit` for the block of statements following the ':' – each statement in this block must be indented by the same amount of whitespace. Any number of spaces or tab characters

could be used, but it is **strongly recommended to use four spaces** to indent code.[26] Loops can be nested – the inner loop block needs to be indented by the same amount of whitespace again as the outer loop (i.e. eight spaces):

```
>>> fruit_list = ['apple', 'melon', 'banana', 'orange']
>>> for fruit in fruit_list:
...     for letter in fruit:
...         print(letter, end='.')
...     print()
...
a.p.p.l.e.
m.e.l.o.n.
b.a.n.a.n.a.
o.r.a.n.g.e.
```

In this example, we iterate over the string items in `fruit_list` one by one, and for each string (fruit name), iterate over its letters. Each letter is printed followed by a full stop (the body of the inner loop). The last statement of the outer loop, `print()` forces a new line after each fruit.

Example E2.19 We have already briefly met the string method `join`, which takes a sequence of string objects and joins them together in a single string:

```
>>> ', '.join( ('one', 'two', 'three') )
'one, two, three'
>>> print('\n'.join(reversed(['one', 'two', 'three'])))
three
two
one
>>> ' '.join('hello')
'h e l l o'
```

Recall that strings are themselves iterable sequences, so the last statement joins the letters of `'hello'` with a single space.

The `range` type

Python provides an efficient method of referring to a sequence of numbers that forms a simple arithmetic progression: $a_n = a_0 + nd$ for $n = 0, 1, 2, \cdots$. In such a sequence, each term is spaced by a constant value, the *stride*, d. In the simplest case, one simply needs an integer counter which runs in steps of one from an initial value of zero: $0, 1, 2, \cdots, N-1$. It would be possible to create a list to hold each of the values, but for most purposes this is wasteful of memory: it is easy to generate the next number in the sequence without having to store all of the numbers at the same time.

[26] The use of whitespace as part of the syntax of Python is one of its most contentious aspects. Some people used to languages such as C and Java which delimit code blocks with braces ({ . . . }) find it an anathema; others argue that code is almost always indented consistently to make it readable even when this isn't enforced by the grammar of the language and consider it less harmful.

Representing such arithmetic progressions for iterating over is the purpose of the `range` type. A `range` object can be constructed with up to three arguments defining the first integer, the integer to stop at and the stride (which can be negative).

```
range([a0=0], n, [stride=1])
```

The notation describing the `range` constructor here means that if the initial value, `a0`, is not given it is taken to be `0`; `stride` is also optional and if it is not given it is taken to be `1`. Some examples:

```
>>> a = range(5)           # 0,1,2,3,4
>>> b = range(1,6)         # 1,2,3,4,5
>>> c = range(0,6,2)       # 0,2,4
>>> d = range(10, 0, -2)   # 10,8,6,4,2
```

In Python 3, the object created by `range` *is not a list*.[27] Rather it is an iterable object that can produce integers on demand: `range` objects can be indexed, cast into lists and tuples, and iterated over:

```
>>> c[1]        # i.e. the second element of 0,2,4
2
>>> c[0]
0
>>> list(d)     # make a list from the range
[10, 8, 6, 4, 2]
>>> for x in range(5):
...     print(x)
0
1
2
3
4
```

Example E2.20 The *Fibonacci sequence* is the sequence of numbers generated by applying the rules:

$$a_1 = a_2 = 1, \quad a_i = a_{i-1} + a_{i-2}.$$

That is, the ith Fibonacci number is the sum of the previous two: $1, 1, 2, 3, 5, 8, 13, \cdots$.

We present two ways of generating the Fibonacci series. First, by appending to a list:

Listing 2.1 Calculating the Fibonacci series in a `list`

```
# eg2-i-fibonacci.py
# Calculates and stores the first n Fibonacci numbers

n = 100
fib = [1, 1]
for i in range(2, n+1):
    fib.append(fib[i-1] + fib[i-2])
print(fib)
```

[27] In Python 2, `range` returned a list and a second method, `xrange`, created the equivalent to Python 3's `range` object.

Alternatively, we can generate the series without storing more than two numbers at a time as follows:

Listing 2.2 Calculating the Fibonacci series without storing it

```
# eg2-ii-fibonacci.py
# Calculates the first n Fibonacci numbers

n = 100
# Keep track of the two most recent Fibonacci numbers
a, b = 1, 1
print(a, b, end='')
for i in range(2, n+1):
    # The next number (b) is a+b, and a becomes the previous b
    a, b = b, a+b
    print(' ', b, end='')
```

enumerate

Because `range` objects can be used to produce a sequence of integers, it is tempting to use them to provide the indexes of lists or tuples when iterating over them in a `for` loop:

```
>>> mammals = ['kangaroo', 'wombat', 'platypus']
>>> for i in range(len(mammals)):
        print(i, ':', mammals[i])
0 : kangaroo
1 : wombat
2 : platypus
```

This works, of course, but it is more natural to avoid the explicit construction of a `range` object (and the call to the `len` built-in) by using `enumerate`. This method takes an iterable object and produces, for each item in turn, a tuple `(count, item)`, consisting of a counting index and the item itself:

```
>>> mammals = ['kangaroo', 'wombat', 'platypus']
>>> for i, mammal in enumerate(mammals):
        print(i, ':', mammal)
0 : kangaroo
1 : wombat
2 : platypus
```

Note that each `(count, item)` tuple is unpacked in the `for` loop into the variables `i` and `mammal`. It is also possible to set the starting value of `count` to something other than 0 (although then it won't be the index of the item in the original list, of course):

```
>>> list(enumerate(mammals, 4))
[(4, 'kangaroo'), (5, 'wombat'), (6, 'platypus')]
```

◊ zip

What if you want to iterate over two (or more) sequences at the same time? This is what the `zip` built-in function is for: it creates an iterator object in which each item is a tuple of items taken in turn from the sequences passed to it:

```
>>> a = [1, 2, 3, 4]
>>> b = ['a', 'b', 'c', 'd']
>>> zip(a,b)
<builtins.zip at 0x104476998>
>>> for pair in zip(a,b):
...     print(pair)
...
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'd')
>>> list(zip(a,b))      # convert to list
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

A nice feature of `zip` is that it can be used to *unzip* sequences of tuples as well:

```
>>> z = zip(a,b)        # zip
>>> A, B = zip(*z)      # unzip
>>> print(A, B)
(1, 2, 3, 4) ('a', 'b', 'c', 'd')
>>> list(A) == a, list(B) == b
(True, True)
```

`zip` does not copy the items into a new object, so it is memory-efficient and fast; but this means that you only get to iterate over the zipped items once and you can't index it:[28]

```
>>> z = zip(a, b):
>>> z[0]
TypeError: 'zip' object is not subscriptable

>>> for pair in z:
...     x = 0       # just some dummy operation performed on each iteration
...
>>> for pair in z:
...     print(pair)
...
        # (nothing: we've already exhausted the iterator z)
>>>
```

2.4.4 Exercises

Questions

Q2.4.1 Predict and explain the outcome of the following statements using the variables

```
s = 'hello'
a = [4, 10, 2]
```

a. `print(s, sep='-')`

[28] This is another difference between Python 2 and Python 3: in the older version of Python, `zip` returned a `list` of tuples.

b. `print(*s, sep='-')`

c. `print(a)`

d. `print(*a, sep='\thinspace\!\!')`

e. `list(range(*a))`

Q2.4.2 A `list` could be used as a simple representation of a polynomial, $P(x)$, with the items as the coefficients of the successive powers of x, and their indexes as the powers themselves. Thus, the polynomial $P(x) = 4 + 5x + 2x^3$ would be represented by the list `[4, 5, 0, 2]`. Why does the following attempt to differentiate a polynomial fail to produce the correct answer?

```
>>> P = [4, 5, 0, 2]
>>> dPdx = []
>>> for i, c in enumerate(P[1:]):
...     dPdx.append(i*c)
>>> dPdx
[0, 0, 4]             # wrong!
```

How can this code be fixed?

Q2.4.3 Given an ordered list of test scores, produce a list associating each score with a *rank* (starting with 1 for the highest score). Equal scores should have the same rank. For example, the input list `[87, 75, 75, 50, 32, 32]` should produce the list of rankings `[1,2,2,4,5,5]`.

Q2.4.4 Use a `for` loop to calculate π from the first 20 terms of the *Madhava series*:

$$\pi = \sqrt{12}\left(1 - \frac{1}{3 \cdot 3} + \frac{1}{5 \cdot 3^2} - \frac{1}{7 \cdot 3^3} + \cdots\right).$$

Q2.4.5 For what iterable sequences, `x`, does the expression

`any(x) and not all(x)`

evaluate to `True`?

Q2.4.6 Explain why `zip(*z)` is the inverse of `z = zip(a, b)` – that is, while `z` pairs the items: `(a0, b0), (a1, b1), (a2, b2), ...`, `zip(*z)` separates them again: `(a0, a1, a2, ...), (b0, b1, b2, ...)`.

Q2.4.7 Sorting a list of tuples arranges them in order of the first element in each tuple first. If two or more tuples have the same first element, they are ordered by the second element, and so on:

```
>>> sorted([(3,1), (1,4), (3,0), (2, 2), (1, -1)])
[(1, -1), (1, 4), (2, 2), (3, 0), (3, 1)]
```

This suggests a way of using `zip` to sort one list using the elements of another. Implement this method on the data below to produce an ordered list of the average amount of sunshine in hours in London by month. Output the sunniest month first.

Jan	Feb	Mar	Apr	May	Jun
44.7	65.4	101.7	148.3	170.9	171.4
Jul	Aug	Sep	Oct	Nov	Dec
176.7	186.1	133.9	105.4	59.6	45.8

Problems

P2.4.1 Write a short Python program which, given an array of integers, `a`, calculates an array of the same length, `p`, in which `p[i]` is the product of all the integers in `a` *except* `a[i]`. So, for example, if `a = [1,2,3]`, then `p` is `[6, 3, 2]`.

P2.4.2 The *Hamming distance* between two equal-length strings is the number of positions at which the characters are different. Write a Python routine to calculate the Hamming distance between two strings, `s1` and `s2`.

P2.4.3 Using a tuple of strings naming the digits 0–9, create a Python program which outputs the representation of π as read aloud to 8 decimal places:

```
three point one four one five nine two six five
```

P2.4.4 Write a program to output a nicely formatted depiction of the first eight rows of Pascal's Triangle.

P2.4.5 A DNA sequence encodes each amino acid making up a protein as a three-nucleotide sequence called a *codon*. For example, the sequence fragment AGTCTTATATCT contains the codons (AGT, CTT, ATA, TCT) if read from the first position (*"frame"*). If read in the second frame it yields the codons (GTC, TTA, TAT) and in the third (TCT, TAT, ATC).

Write some Python code to extract the codons into a list of 3-letter strings given a sequence and `frame` as an integer value (0, 1 or 2).

P2.4.6 The factorial function, $n! = 1 \cdot 2 \cdot 3 \cdot \cdots (n-1)n$ is the product of the first n positive integers and is provided by the `math` module's `factorial` method. The *double factorial* function, $n!!$, is the product of the positive *odd* integers up to and including n (which must itself be odd):

$$n!! = \prod_{i=1}^{(n+1)/2} (2i-1) = 1 \cdot 3 \cdot 5 \cdots (n-2) \cdot n.$$

Write a routine to calculate $n!!$ in Python.

As a bonus exercise, extend the formula to allow for even n as follows:

$$n!! = \prod_{i=1}^{n/2} (2i) = 2 \cdot 4 \cdot 6 \cdots (n-2) \cdot n.$$

P2.4.7 *Benford's Law* is an observation about the distribution of the frequencies of the first digits of the numbers in many different data sets. It is frequently found that the first

digits are not uniformly distributed, but follow the logarithmic distribution

$$P(d) = \log_{10}\left(\frac{d+1}{d}\right).$$

That is, numbers starting with 1 are more common than those starting with 2, and so on, with those starting with 9 the least common. The probabilities follow:

1	0.301
2	0.176
3	0.125
4	0.097
5	0.079
6	0.067
7	0.058
8	0.051
9	0.046

Benford's Law is most accurate for data sets which span several orders of magnitude, and can be proved to be exact for some infinite sequences of numbers.

1 Demonstrate that the first digits of the first 500 Fibonacci numbers (see Example E2.20) follow Benford's Law quite closely.

2 The length of the amino acid sequences of 500 randomly chosen proteins are provided in the file `ex2-4_e_ii_protein_lengths.py` which can be downloaded from scipython.com/ex/aba. This file contains a list, `naa`, which can be imported at the start of your program with

```
from ex2-4_e_ii_protein_lengths import naa
```

To what extent does the distribution of protein lengths obey Benford's Law?

2.5 Control flow

Few computer programs are executed in a purely linear fashion, one statement after another as written in the source code. It is more likely that during the program execution, data objects are inspected and blocks of code executed conditionally on the basis of some test carried out on them. Thus, all practical languages have the equivalent of an *if-then-(else)* construction. This section explains the syntax of Python's version of this clause and covers a further kind of loop: the `while` loop.

2.5.1 `if ... elif ... else`

The `if ... elif ... else` construction allows statements to be executed conditionally, depending on the result of one or more logical tests (which evaluate to the boolean values `True` or `False`):

```
if <logical expression 1>:
    <statements 1>
elif <logical expression 2>:
    <statements 2>
...
else:
    <statements>
```

That is, if *<logical expression 1>* evaluates to `True`, *<statements 1>* are executed; otherwise, if *<logical expression 2>* evaluates to `True`, *<statements 2>* are executed, and so on; if none of the preceding logical expressions evaluate to `True`, the statements in the block of code following `else:` are executed. These statement blocks are indented with whitespace, as for the `for` loop. For example,

```
for x in range(10):
    if x <= 3:
        print(x, 'is less than or equal to three')
    elif x > 5:
        print(x, 'is greater than five')
    else:
        print(x, 'must be four or five, then')
```

produces the output:

```
0 is less than or equal to three
1 is less than or equal to three
2 is less than or equal to three
3 is less than or equal to three
4 must be four or five, then
5 must be four or five, then
6 is greater than five
7 is greater than five
8 is greater than five
9 is greater than five
```

It is not necessary to enclose test expressions such as `x <= 3` in parentheses, as it is in C, for example, but the colon following the test is mandatory. The test expressions don't, in fact, have to evaluate explicitly to the boolean values `True` and `False`: as we have seen, other data types are taken to be equivalent to `True` unless they are `0` (`int`) or `0.` (`float`), the empty string, `''`, empty list, `[]`, the empty tuple, `()`, and so forth or Python's special type, `None` (see Section 2.2.4). Consider:

```
for x in range(10):
    if x % 2:
        print(x, 'is odd!')
    else:
        print(x, 'is even!')
```

This works because `x % 2 = 1` for odd integers, which is equivalent to `True` and `x % 2 = 0` for even integers, which is equivalent to `False`.

There is **no** `switch ... case ... finally` construction in Python – equivalent control flow can be achieved with `if ... elif ... endif` or with *dictionaries* (see Section 4.2).

Example E2.21 In the Gregorian calendar a year is a *leap year* if it is divisible by 4 with the exceptions that years divisible by 100 are *not* leap years unless they are also divisible by 400. The following Python program determines if `year` is a leap year.

Listing 2.3 Determining if a year is a leap year

```
year = 1900

if not year % 400:
    is_leap_year = True
elif not year % 100:
    is_leap_year = False
elif not year % 4:
    is_leap_year = True
else:
    is_leap_year = False

s_ly = 'is a' if is_leap_year else 'is not a'
print('{:4d} {:s} leap year'.format(year, s_ly))
```

Hence the output:

```
1900 is not a leap year
```

2.5.2 `while` loops

Whereas a `for` loop is established for a fixed number of iterations, statements within the block of a `while` loop execute only and as long as some condition holds:

```
>>> i = 0
>>> while i < 10:
...     i += 1
...     print(i, end='.')
...
>>> print()
1.2.3.4.5.6.7.8.9.10.
```

The counter `i` is initialized to `0`, which is less than 10 so the `while` loop begins. On each iteration, `i` is incremented by one and its value printed. When `i` reaches `10`, on the following iteration `i < 10` is `False`: the loop ends and execution continues after the loop, where `print()` outputs a newline.

Example E2.22 A more interesting example of the use of a `while` loop is given by this implementation of Euclid's algorithm for finding the greatest common divisor of two numbers, $\gcd(a, b)$:

```
>>> a, b = 1071, 462
>>> while b:
...     a, b = b, a % b
...
>>> print(a)
21
```

The loop continues until `b` divides `a` exactly; on each iteration, `b` is set to the remainder of `a//b` and then `a` is set to the old value of `b`. Recall that the integer 0 evaluates as boolean `False` so `while b:` is equivalent to `while b != 0:`.

2.5.3 More control flow: `break`, `pass`, `continue` and `else`

`break`

Python provides three further statements for controlling the flow of a program. The `break` command, issued inside a loop, immediately ends that loop and moves execution to the statements following the loop:

```
x = 0
while True:
    x += 1
    if not (x % 15 or x % 25):
        break
print(x, 'is divisible by both 15 and 25')
```

The `while` loop condition here is (literally) always `True` so the only escape from the loop occurs when the `break` statement is reached. This occurs only when the counter `x` is divisible by both `15` and `25`. The output is therefore:

```
75 is divisible by both 15 and 25
```

Similarly, to find the index of the first occurrence of a negative number in a list:

```
alist = [0, 4, 5, -2, 5, 10]
for i, a in enumerate(alist):
    if a < 0:
        break
print(a, 'occurs at index', i)
```

Note that after escaping from the loop, the variables `i` and `a` have the values that they had within the loop at the `break` statement.

`continue`

The `continue` statement acts in a similar way to `break` but instead of breaking out of the containing loop, it immediately forces the next iteration of the loop without completing the statement block for the current iteration. For example,

```
for i in range(1, 11):
    if i % 2:
        continue
    print(i, 'is even!')
```

prints only the even integers $2, 4, 6, 8, 10$: if `i` is not divisible by 2 (and hence `i % 2` is 1, equivalent to `True`), that loop iteration is canceled and the loop resumed with the next value of `i` (the `print` statement is skipped).

`pass`

The `pass` command does nothing. It is useful as a "stub" for code that has not yet been written but where a statement is syntactically required by Python's whitespace convention.

```
>>> for i in range(1, 11):
...     if i == 6:
...         pass    # do something special if i is 6
...     if not i % 3:
...        print(i, 'is divisible by 3')
...
3 is divisible by 3
6 is divisible by 3
9 is divisible by 3
```

If the `pass` statement had been `continue` the line `6 is divisible by 3` would not have been printed: execution would have returned to the top of the loop and `i=7` instead of continuing to the second `if` statement.

◊ else

A `for` or `while` loop may be followed by an `else` block of statements, which will be executed only if the loop finished "normally" (that is, *without* the intervention of a `break`). For `for` loops, this means these statements will be executed after the loop has reached the end of the sequence it is iterating over; for `while` loops, they are executed when the `while` condition becomes `False`. For example, consider again our program to find the first occurrence of a negative number in a list. This code behaves rather oddly if there aren't any negative numbers in the list:

```
>>> alist = [0, 4, 5, 2, 5, 10]
>>> for i, a in enumerate(alist):
...     if a < 0:
...         break
...
>>> print(a, 'occurs at index', i)
10 occurs at index 5
```

It outputs the index and number of the last item in the list (whether it is negative or not). A way to improve this is to notice when the `for` loop runs through every item without encountering a negative number (and hence the `break`) and output a message:

```
>>> alist = [0, 4, 5, 2, 5, 10]
... for i, a in enumerate(alist):
...     if a < 0:
...         print(a, 'occurs at index', i)
...         break
... else:
...    print('no negative numbers in the list')
...

no negative numbers in the list
```

As another example, consider this (not particularly elegant) routine for finding the largest factor of a number `a > 2`:

```
a = 1013
b = a - 1
while b != 1:
    if not a % b:
```

```
        print('the largest factor of', a, 'is', b)
        break
    b -= 1
else:
    print(a, 'is prime!')
```

`b` is the largest factor not equal to `a`. The `while` loop continues as long as `b` is not equal to `1` (in which case `a` is prime) and decrements `b` after testing if `b` divides `a` exactly; if it does, `b` is the highest factor of `a`, and we `break` out of the `while` loop.

Example E2.23 A simple "turtle" virtual robot lives on an infinite two-dimensional plane on which its location is always an integer pair of (x, y) coordinates. It can face only in directions parallel to the x and y axes (i.e. 'North,' 'East,' 'South' or 'West') and it understands four commands:

- `F`: move forward one unit;
- `L`: turn left (counterclockwise) by 90°;
- `R`: turn right (clockwise) by 90°;
- `S`: stop and exit.

The following Python program takes a list of such commands as a string and tracks the turtle's location. The turtle starts at $(0, 0)$, facing in the direction $(1, 0)$ ('East'). The program ignores (but warns about) invalid commands and reports when the turtle crosses its own path.

Listing 2.4 A virtual turtle robot

```
# eg2-turtle.py
commands = 'FFFFFLFFFLFFFFRRRFXFFFFFFS'

# Current location, current facing direction
x, y = 0, 0
dx, dy = 1, 0
# Keep track of the turtle's location in the list of tuples, locs
locs = [(0, 0)]

❶ for cmd in commands:
    if cmd == 'S':
        # Stop command
        break
    if cmd == 'F':
        # Move forward in the current direction
        x += dx
        y += dy
        if (x, y) in locs:
            print('Path crosses itself at: ({}, {})'.format(x,y))
        locs.append((x,y))
        continue
    if cmd in 'LR':
        # Turn to the left (counterclockwise) or right (clockwise)
        # L => (dx, dy): (1,0) -> (0, 1) -> (-1,0) -> (0,-1) -> (1,0)
        # R => (dx, dy): (1,0) -> (0,-1) -> (-1,0) -> (0, 1) -> (1,0)
        sgn = 1
```

```
            if dy != 0:
                sgn = -1
            if cmd == 'R':
                sgn = -sgn
            dx, dy = sgn * dy, sgn * dx
            continue
        # if we're here it's because we don't recognize the command: warn
        print('Unknown command:', cmd)
❷   else:
        # We exhausted the commands without encountering an S for STOP
        print('Instructions ended without a STOP')

    # Plot a path of asterisks
    # First find the total range of x and y values encountered
❸   x, y = zip(*locs)
    xmin, xmax = min(x), max(x)
    ymin, ymax = min(y), max(y)
    # The grid size needed for the plot is (nx, ny)
    nx = xmax - xmin + 1
    ny = ymax - ymin + 1
    # Reverse the y-axis so that it decreases *down* the screen
    for iy in reversed(range(ny)):
        for ix in range(nx):
            if (ix+xmin, iy+ymin) in locs:
                print('*', end='')
            else:
                print(' ', end='')
        print()
```

❶ We can iterate over the string `commands` to take its characters one at a time.

❷ Note that the `else:` clause to the `for` loop is only executed if we do not `break` out of it on encountering a STOP command.

❸ We unzip the list of tuples, `locs`, into separate sequences of the *x* and *y* coordinates with `zip(*locs)`.

The output produced from the commands given is:

```
Unknown command: X
Path crosses itself at: (1, 0)
 *****
 *   *
 *   *
******
 *
 *
 *
 *
```

2.5.4 Exercises

Questions

Q2.5.1 Write a Python program to normalize a list of numbers, `a`, such that its values lie between 0 and 1. Thus, for example, the list `a = [2,4,10,6,8,4]` becomes `[0.0, 0.25, 1.0, 0.5, 0.75, 0.25]`.

Hint: use the built-ins `min`, and `max` which return the minimum and maximum values in a sequence respectively; for example, `min(a)` returns `2` in the earlier mentioned list.

Q2.5.2 Write a `while` loop to calculate the *arithmetic-geometric mean* (AGM) of two positive real numbers, x and y, defined as the limit of the sequences:

$$a_{n+1} = \tfrac{1}{2}(a_n + b_n)$$
$$b_{n+1} = \sqrt{a_n b_n},$$

starting with $a_0 = x$, $b_0 = y$. Both sequences converge to the same number, denoted $\mathrm{agm}(x, y)$. Use your loop to determine *Gauss's constant*, $G = 1/\mathrm{agm}(1, \sqrt{2})$.

Q2.5.3 The game of "Fizzbuzz" involves counting, but replacing numbers divisible by 3 with the word '*Fizz*,' those divisible by 5 with '*Buzz*,' and those divisible by both 3 and 5 with '*FizzBuzz*.' Write a program to play this game, counting up to 100.

Q2.5.4 Straight-chain alkanes are hydrocarbons with the general stoichiometric formula C_nH_{2n+2}, in which the carbon atoms form a simple chain: for example, butane, C_4H_{10} has the structural formula that may be depicted $H_3CCH_2CH_2CH_3$. Write a program to output the structural formula of such an alkane, given its stoichiometry (assume $n > 1$). For example, given `stoich='C8H18'`, the output should be

```
H3C-CH2-CH2-CH2-CH2-CH2-CH2-CH3
```

Problems

P2.5.1 Modify your solution to Problem P2.4.4 to output the first 50 rows of Pascal's triangle, but instead of the numbers themselves, output an asterisk if the number is odd and a space if it is even.

P2.5.2 The *iterative weak acid* approximation determines the hydrogen ion concentration, $[H^+]$ of an acid solution from the acid dissociation constant, K_a, and the acid concentration, c, by successive application of the formula

$$[H^+]_{n+1} = \sqrt{K_a \left(c - [H^+]_n\right)},$$

starting with $[H^+]_0 = 0$. The iterations are continued until $[H^+]$ changes by less than some predetermined, small tolerance value.

Use this method to determine the hydrogen ion concentration, and hence the pH ($= -\log_{10}[H^+]$) of a $c = 0.01$ M solution of acetic acid ($K_a = 1.78 \times 10^{-5}$). Use the tolerance `TOL = 1.e-10`.

P2.5.3 The *Luhn algorithm* is a simple checksum formula used to validate credit card and bank account numbers. It is designed to prevent common errors in transcribing the number, and detects all single-digit errors and almost all transpositions of two adjacent digits. The algorithm may be written as the following steps:

1. Reverse the number.
2. Treating the number as an array of digits, take the even-indexed digits (where the indexes *start at 1*) and double their values. If a doubled digit results in a number greater than 10, add the two digits (e.g., the digit 6 becomes 12 and hence 1 + 2 = 3).
3. Sum this modified array.
4. If the sum of the array modulo 10 is 0 the credit card number is valid.

Write a Python program to take a credit card number as a string of digits (possibly in groups, separated by spaces) and establish if it is valid or not. For example, the string `'4799 2739 8713 6272'` is a valid credit card number, but any number with a single digit in this string changed is not.

P2.5.4 *Hero's method* for calculating the square root of a number, S, is as follows: starting with an initial guess, x_0, the sequence of numbers $x_{n+1} = \frac{1}{2}(x_n + S/x_n)$ are successively better approximations to $\sqrt{S}$. Implement this algorithm to estimate the square root of 2117519.73 to two decimal places and compare with the "exact" answer provided by the `math.sqrt` method. For the purpose of this exercise, start with an initial guess, $x_0 = 2000$.

P2.5.5 Write a program to determine the tomorrow's date given a string representing today's date, `today`, as either "`D/M/Y`" or "`M/D/Y`." Cater for both British and US-style dates when parsing `today` according to the value of a boolean variable `us_date_style`. For example, when `us_date_style` is `False` and `today` is `'3/4/2014'`, tomorrow's date should be reported as `'4/4/2014'`.[29] (*Hint:* use the algorithm for determining if a year is a leap year, which is provided in the example to Section 2.5.1.)

P2.5.6 Write a Python program to determine $f(n)$, the number of trailing zeros in $n!$, using the special case of *de Polignac's formula*:

$$f(n) = \sum_{i=1} \left\lfloor \frac{n}{5^i} \right\rfloor,$$

where $\lfloor x \rfloor$ denotes the *floor* of x, the largest integer less than or equal to x.

P2.5.7 The *hailstone sequence* starting at an integer $n > 0$ is generated by the repeated application of the three rules:

- if $n = 1$, the sequence ends;
- if n is even, the next number in the sequence is $n/2$;
- if n is odd, the next number in the sequence is $3n + 1$.

a. Write a program to calculate the hailstone sequence starting at 27.
b. Let the *stopping time* be the number of numbers in a given hailstone sequence. Modify your hailstone program to return the stopping time instead of the numbers

[29] In practice, it would be better to use Python's `datetime` library (described in Section 4.5.3), but avoid it for this exercise.

themselves. Adapt your program to demonstrate that the hailstone sequences started with $1 \leq n \leq 100$ agree with the *Collatz conjecture* (that all hailstone sequences stop eventually).

P2.5.8 The algorithm known as the *Sieve of Eratosthenes* finds the prime numbers in a list $2, 3, \cdots, n$. It may be summarized as follows, starting at $p = 2$, the first prime number:

Step 1. Mark all the multiples of p in the list as nonprime (that is, the numbers mp where $m = 2, 3, 4, \cdots$: these numbers are *composite*.

Step 2. Find the first unmarked number greater than p in the list. If there is no such number, stop.

Step 3. Let p equal this new number and return to Step 1.

When the algorithm stops, the unmarked numbers are the primes.

Implement the Sieve of Eratosthenes in a Python program and find all the primes under 10000.

P2.5.9 *Euler's totient function*, $\phi(n)$, counts the number of positive integers less than or equal to n that are relatively prime to n. (Two numbers, a and b, are relatively prime if the only positive integer that divides both of them is 1; that is, if $\gcd(a, b) = 1$.)

Write a Python program to compute $\phi(n)$ for $1 \leq n < 100$.

(*Hint:* you could use Euclid's algorithm for the greatest common divisor given in the example to Section 2.5.2.)

P2.5.10 The value of π may be approximated by Monte Carlo methods. Consider region of the xy-plane bounded by $0 \leq x \leq 1$ and $0 \leq y \leq 1$. By selecting a large number of random points within this region and counting the proportion of them lying beneath the function $y = \sqrt{1 - x^2}$ describing a quarter-circle, one can estimate $\pi/4$, this being the area bounded by the axes and $y(x)$. Write a program to estimate the value of π by this method.

Hint: use Python's `random` module. The method `random.random()` generates a (pseudo-)random number between 0. and 1. See Section 4.5.1 for more information.

P2.5.11 Write a program to take a string of text (words, perhaps with punctuation, separated by spaces) and output the same text with the middle letters shuffled randomly. Keep any punctuation at the end of words. For example, the string:

Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

might be rendered:

Four sorce and seevn yeras ago our fhtaers bhrogut ftroh on this cnnoientt a new noitan, cvieecond in lbrteiy, and ddicetead to the ptosoiporin that all men are cetaerd euaql.

Hint: `random.shuffle` shuffles a *list* of items in place. See Section 4.5.1.

P2.5.12 The *electron configuration* of an atom is the specification of the distribution of its electrons in atomic orbitals. An atomic orbital is identified by a *principal quantum number*, $n = 1, 2, 3, \cdots$ defining a *shell* comprised of one or more *subshells* defined

by the *azimuthal quantum number*, $l = 0, 1, 2, \cdots, n-1$. The values $l = 0, 1, 2, 3$ are refered to be the letters s, p, d and f respectively. Thus, the first few orbitals are 1s ($n = 1, l = 0$), 2s ($n = 2, l = 0$), 2p ($n = 2, l = 1$), 3s ($n = 3, l = 0$), and each shell has n subshells. A maximum of $2(2l + 1)$ electrons may occupy a given subshell.

According to the *Madelung rule*, the N electrons of an atom fill the orbitals in order of increasing $n + l$ such that whenever two orbitals have the same value of $n + l$, they are filled in order of increasing n. For example, the ground state of Titanium ($N = 22$) is predicted (and found) to be $1s^2 2s^2 2p^6 3s^2 3p^6 4s^2 3d^2$.

Write a program to predict the electronic configurations of the elements up to Rutherfordium ($N = 104$). The output for Titanium should be

```
Ti: 1s2.2s2.2p6.3s2.3p6.4s2.3d2
```

A Python list containing the element symbols in order may be downloaded from scipython.com/ex/abb.

As a bonus exercise, modify your program to output the configurations using the convention that the part of the configuration corresponding to the outermost *closed shell*, a noble gas configuration, is replaced by the noble gas symbol in square brackets; thus,

```
Ti: [Ar].4s2.3d2
```

the configuration of Argon being `1s2.2s2.2p6.3s2.3p6`.

2.6 File input/output

Until now, data has been hard-coded into our Python programs, and output has been to the console (the terminal). Of course, it will frequently be necessary to input data from an external file and to write data to an output file. To acheive this, Python has `file` objects.

2.6.1 Opening and closing a file

A `file` object is created by opening a file with a given *filename* and *mode*. The filename may be given as an absolute path, or as a path relative to the directory in which the program is being executed. `mode` is a string with one of the values given in Table 2.13. For example, to open a file for text-mode writing:

```
>>> f = open('myfile.txt', 'w')
```

`file` objects are closed with the `close` method: for example, `f.close()`. Python closes any open file objects automatically when a program terminates.

2.6.2 Writing to a file

The `write` method of a `file` object writes a *string* to the file and returns the number of characters written:

```
>>> f.write('Hello World!')
12
```

Table 2.13 File modes

mode argument	Open mode
r	text, read-only (the default)
w	text, write (an existing file with the same name will be overwritten)
a	text, append to an existing file
r+	text, reading and writing
rb	binary, read-only
wb	binary, write (an existing file with the same name will be overwritten)
ab	binary, append to an existing file
rb+	binary, reading and writing

More helpfully, the `print` built-in takes an argument, `file`, to specify where to redirect its output :

```
>>> print(35, 'Cl', 2, sep='', file=f)
```

writes '35Cl2' to the file opened as `file` object `f` instead of to the console.

Example E2.24 The following program writes the first four powers of the numbers between 1 and 1,000 in comma-separated fields to the file `powers.txt`:

```
f = open('powers.txt', 'w')
for i in range(1,1001):
    print(i, i**2, i**3, i**4, sep=', ', file=f)
f.close()
```

The file contents are

```
1, 1, 1, 1
2, 4, 8, 16
3, 9, 27, 81
...
999, 998001, 997002999, 996005996001
1000, 1000000, 1000000000, 1000000000000
```

2.6.3 Reading from a file

To read `n` bytes from a file, call `f.read(n)`. If `n` is omitted, the entire file is read in.[30]

`readline()` reads a single line from the file, up to and including the newline character. The next call to `readline()` reads in the next line, and so on. Both `read()` and `readline()` return an empty string when they reach the end of the file.

To read all of the lines into a list of strings in one go, use `f.readlines()`.

`file` objects are iterable, and looping over a (text) `file` returns its lines one at a time:

[30] To quote the official documentation: "it's your problem if the file is twice as large as your machine's memory."

```
>>> for line in f:
❶ ...     print(line, end='')
...
First line
Second line
...
```

❶ Because `line` retains its newline character when read in, we use `end=''` to prevent `print` from adding another, which would be output as a blank line.

You probably want to use this method if your file is very large unless you really do want to store every line in memory. See Section 4.3.4 concerning Python's `with` statement for more best practice in file handling.

Example E2.25 To read in the numbers from the file `powers.txt` generated in the previous example, the columns must be converted to lists of integers. To do this, each line must be split into its fields and each field explicitly converted to an `int`:

```
f = open('powers.txt', 'r')
squares, cubes, fourths = [], [], []
for line in f.readlines():
    fields = line.split(',')
    squares.append(int(fields[1]))
    cubes.append(int(fields[2]))
    fourths.append(int(fields[3]))
f.close()
n = 500
print(n, 'cubed is', cubes[n-1])
```

The output is

```
500 cubed is 125000000
```

In practice, it is better to use `numpy` (see Chapter 6) to read in data files such as these.

2.6.4 Exercises

Problems

P2.6.1 The coast redwood tree species, *Sequoia sempervirens*, includes some of the oldest and tallest living organisms on Earth. Some details concerning individual trees are given in the tab-delimited text file `redwood-data.txt`, available at scipython.com/ex/abd. (Data courtesy of the Gymnosperm database, www.conifers.org/cu/Sequoia.php)

Write a Python program to read in this data and report the tallest tree and the tree with the greatest diameter.

P2.6.2 Write a program to read in a text file and censor any words in it that are on a list of banned words by replacing their letters with the same number of asterisks. Your program should store the banned words in lowercase but censor examples of these words in any case. Assume there is no punctuation.

Table 2.14 Parameters used in the definition of ESI

i	Parameter	Earth value, $x_{i,\oplus}$	Weight, w_i
1	Radius	1.0	0.57
2	Density	1.0	1.07
3	Escape velocity, v_{esc}	1.0	0.7
4	Surface temperature	288 K	5.58

As a bonus exercise, handle text that contains punctuation. For example, given the list of banned words: `['C', 'Perl', 'Fortran']` the sentence

```
'Some alternative programming languages to Python are C, C++, Perl, Fortran and Java.'
```

becomes

```
'Some alternative programming languages to Python are *, C++, ****, ******* and Java.'
```

P2.6.3 The *Earth Similarity Index* (ESI) attempts to quantify the physical similarity between an astronomical body (usually a planet or moon) and Earth. It is defined by

$$\text{ESI}_j = \prod_{i=1}^{n} \left(1 - \left|\frac{x_{i,j} - x_{i,\oplus}}{x_{i,j} + x_{i,\oplus}}\right|\right)^{w_i/n}$$

where the parameters $x_{i,j}$ are described, and their terrestrial values, $x_{i,\oplus}$ and weights, w_i given in Table 2.14. The radius, density and escape velocities are taken *relative to* the terrestrial values. The ESI lies between 0 and 1, with the values closer to 1 indicating closer similarity to Earth (which has an ESI of exactly 1: Earth is identical to itself!).

The file `ex2-6-g-esi-data.txt` available from scipython.com/ex/abc contains the earlier mentioned parameters for a range of astronomical bodies. Use these data to calculate the ESI for each of the bodies. Which has properties "closest" to those of the Earth?

P2.6.4 Write a program to read in a two-dimensional array of strings into a list of lists from a file in which the string elements are separated by one or more spaces. The number of rows, m, and columns, n, may not be known in advance of opening the file. For example, the text file

```
A B C D
E F G H
I J K L
```

should create an object, `grid`, as

```
[['A', 'B', 'C', 'D'], ['E', 'F', 'G', 'H'], ['I', 'J', 'K', 'L']]
```

Read like this, `grid` contains a list of the array's *rows*. Once the array has been read in, write loops to output the *columns* of the array:

```
[['A', 'E', 'I'], ['B', 'F', 'J'], ['C', 'G', 'K'], ['D', 'H', 'L']]
```

Harder: also output all its diagonals read in one direction:

```
[['A'], ['B', 'E'], ['C', 'F', 'I'], ['D', 'G', 'J'], ['H', 'K'], ['L']]
```

and the other direction:

```
[['D'], ['C', 'H'], ['B', 'G', 'L'], ['A', 'F', 'K'], ['E', 'J'], ['I']]
```

2.7 Functions

A Python *function* is a set of statements that are grouped together and named so that they can be run more than once in a program. There are two main advantages to using functions. First, they enable code to be reused without having to be replicated in different parts of the program; second, they enable complex tasks to be broken into separate procedures, each implemented by its own function – it is often much easier and more maintainable to code each procedure individually than to code the entire task at once.

2.7.1 Defining and calling functions

The `def` statement defines a function, gives it a name, and lists the arguments (if any) that the function expects to receive when called. The function's statements are written in an indented block following this `def`. If at any point during the execution of this statement block a `return` statement is encountered, the specified values are returned to the caller. For example,

```
❶ >>> def square(x):
...     x_squared = x**2
...     return x_squared
...
>>> number = 2
❷ >>> number_squared = square(number)
>>> print(number, 'squared is', number_squared)
2 squared is 4
❸ >>> print('8 squared is', square(8))
8 squared is 64
```

❶ The simple function named `square` takes a single argument, `x`. It calculates `x**2` and returns this value to the caller. Once defined, it can be called any number of times. ❷ In the first example, the return value is assigned to the variable `number_squared`; ❸ in the second example, it is fed straight into the `print` method for output to the console.

To return two or more values from a function, pack them into a tuple. For example, the following program defines a function to return both roots of the quadratic equation $ax^2 + bx + c$ (assuming it *has* two real roots):

```
import math

def roots(a, b, c):
    d = b**2 - 4*a*c
    r1 = (-b + math.sqrt(d)) / 2 / a
    r2 = (-b - math.sqrt(d)) / 2 / a
    return r1, r2

print(roots(1., -1., -6.))
```

When run, this program outputs, as expected:

```
(3.0, -2.0)
```

It is not necessary for a function to explicitly return any object: functions that fall off the end of their indented block without encountering a `return` statement return Python's special value, `None`.

Function definitions can appear anywhere in a Python program, but a function cannot be referenced before it is defined. Functions can even be *nested*, but a function defined inside another is not (directly) accessible from outside that function.

Docstrings

A function *docstring* is a string literal that occurs as the first statement of the function definition. It should be written as a triple-quoted string on a single line if the function is simple, or on multiple lines with an initial one-line summary for more detailed descriptions of complex functions. For example,

```
def roots(a, b, c):
    """Return the roots of ax^2 + bx + c."""
    d = b**2 - 4*a*c
    ...
```

The docstring becomes the special `__doc__` attribute of the function:

```
>>> roots.__doc__
'Return the roots of ax^2 + bx + c.'
```

A docstring should provide details about *how to use the function*: which arguments to pass it and which objects it returns,[31] but should not generally include details of the specific *implementation* of algorithms used by the function (these are best explained in *comments*, preceded by `#`).

Docstrings are also used to provide documentation for classes and modules (see Sections 4.5 and 4.6.2).

Example E2.26 In Python, *functions are "first class" objects*: they can have variable identifiers assigned to them, they can be passed as arguments to other functions, and they can even be returned *from* other functions. A function is given a name when it is defined, but that name can be reassigned to refer to a different object (don't do this unless you mean to!) if desired.

As the following example demonstrates, it is possible for more than one variable name to be assigned to the same function object.

```
>>> def cosec(x):
...     """Return the cosecant of x, cosec(x) = 1/sin(x)."""
...     return 1./math.sin(x)
...
>>> cosec
<function cosec at 0x100375170>
```

[31] For larger projects, docstrings document an application programming interface (API) for the project.

```
>>> cosec(math.pi/4)
1.4142135623730951
❶ >>> csc = cosec
>>> csc
<function cosec at 0x100375170>
>>> csc(math.pi/4)
1.4142135623730951
```

❶ The assignment `csc = cosec` associates the identifier (variable name) `csc` with the same function object as the identifier `cosec`: this function can then be called with `csc()` as well as with `cosec()`.

2.7.2 Default and keyword arguments

Keyword arguments

In the previous example, the arguments have been passed to the function in the order in which they are given in the function's definition (these are called *positional* arguments). It is also possible to pass the arguments in an arbitrary order by setting them explicitly as *keyword arguments*:

```
roots(a=1., c=-6., b=-1.)
roots(b=-1., a=1., c=-6.)
```

If you mix nonkeyword (positional) and keyword arguments the former must come first; otherwise Python won't know to which variable the positional argument corresponds:

```
>>> roots(1., c=6., b=-1.)  # OK
(3.0, -2.0)
>>> roots(b=-1., 1., -6.)   # Oops: which is a and which is c?
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

Default arguments

Sometimes you want to define a function that takes an *optional* argument: if the caller doesn't provide a value for this argument, a default value is used. Default arguments are set in the function definition:

```
>>> def report_length(value, units='m'):
...     return 'The length is {:.2f} {}'.format(value, units)
>>> report_length(33.136, 'ft')
'The length is 33.14 ft'
>>> report_length(10.1)
'The length is 10.10 m'
```

Default arguments are assigned *when the Python interpreter first encounters the function definition*. This can lead to some unexpected results, particularly for mutable arguments. For example,

```
>>> def func(alist = []):
...     alist.append(7)
...     return alist
...
```

```
>>> func()
[7]
>>> func()
[7, 7]
>>> func()
[7, 7, 7]
```

The default argument to the function `func` here is an empty list, but it is the specific empty list assigned when the function is defined. Therefore, each time `func` is called this specific list grows.

Example E2.27 Default argument values are assigned *when the function is defined*. Therefore, if a function is defined with an argument defaulting to the value of some variable, subsequently changing that variable *will not change the default*:

```
>>> default_units = 'm'
>>> def report_length(value, units=default_units):
...     return 'The length is {:.2f} {}'.format(value, units)
...
>>> report_length(10.1)
'The length is 10.10 m'
>>> default_units = 'cubits'
>>> report_length(10.1)
'The length is 10.10 m'
```

The default units used by the function `report_length` are unchanged by the reassignment of the variable name `default_units`: the default value is set to the string object referred to by `default_units` when the `def` statement is encountered by the Python compiler (`'m'`) and cannot be changed subsequently.

2.7.3 Scope

A function can define and use its own variables. When it does so, those variables are *local* to that function: they are not available outside the function. Conversely, variables assigned outside all function `defs` are *global* and are available everywhere within the program file. For example,

```
>>> def func():
...     a = 5
...     print(a,b)
...
>>> b = 6
>>> func()
5 6
```

The function `func` defines a variable `a`, but prints out both `a` and `b`. Because the variable `b` isn't defined in the local scope of the function, Python looks in the global scope, where it finds `b = 6`, so that is what is printed. It doesn't matter that `b` hasn't been defined when the function is *defined*, but of course it must be before the function is *called*.

What happens if a function defines a variable with the same name as a global variable? In this case, within the function the local scope is searched first when resolving variable

names, so it is the object pointed to by the local variable name that is retrieved. For example,

```
>>> def func():
...     a = 5
...     print(a)
...
>>> a = 6
>>> func()
5
>>> print(a)
6
```

Note that the local variable `a` exists only within the body of the function; it just happens to have the same name as the global variable `a`. It disappears after the function exits and it doesn't overwrite the global `a`.

Python's rules for resolving scope can be summarized as "LEGB": first *local* scope, then *enclosing* scope (for nested functions), then *global* scope, and finally *built-ins*–if you happen to give a variable the same name as a built-in function (such as `range` or `len`), then that name resolves to your variable (in local or global scope) and not to the original built-in. It is therefore generally not a good idea to name your variables after built-ins.

◊ The `global` and `nonlocal` keywords

We have seen that it is possible to access variables defined in scopes other than the local function's. Is it possible to *modify* them ("*rebind*" them to new objects)? Consider the distinction between the behavior of the following functions:

```
>>> def func1():
...     print(x)     # OK, providing x is defined in global or enclosing scope
...
>>> def func2():
...     x += 1       # Not OK: can't modify x if it isn't local
...
>>> x = 4
>>> func1()
4
>>> func2()
UnboundLocalError: local variable 'x' referenced before assignment
```

If you really do want to change variables that are defined outside the local scope, you must first declare within the function body that this is your intention with the keywords `global` (for variables in global scope) and `nonlocal` (for variables in enclosing scope, for example, where a function is defined within another). In the previous case:

```
>>> def func2():
...     global x
...     x += 1       # OK now - Python knows we mean x in global scope
...
>>> x = 4
>>> func2()         # No error
>>> x
5
```

The function `func2` really has changed the value of the variable `x` in global scope.

You should think carefully whether it is really necessary to use this technique (would it be better to pass `x` as an argument and `return` its updated value from the function?), Especially in longer programs, variable names in one scope that change value (or even type!) within functions lead to confusing code, behavior that is hard to predict and tricky bugs.

Example E2.28 Take a moment to study the following code and predict the result before running it.

Listing 2.5 Python scope rules

```
# eg2-scope.py

def outer_func():
    def inner_func():
        a = 9
        print('inside inner_func, a is {:d} (id={:d})'.format(a, id(a)))
        print('inside inner_func, b is {:d} (id={:d})'.format(b, id(b)))
        print('inside inner_func, len is {:d} (id={:d})'.format(len,id(len)))

    len = 2
    print('inside outer_func, a is {:d} (id={:d})'.format(a, id(a)))
    print('inside outer_func, b is {:d} (id={:d})'.format(b, id(b)))
    print('inside outer_func, len is {:d} (id={:d})'.format(len,id(len)))
    inner_func()

a, b = 6, 7
outer_func()
print('in global scope, a is {:d} (id={:d})'.format(a, id(a)))
print('in global scope, b is {:d} (id={:d})'.format(b, id(b)))
print('in global scope, len is', len, '(id={:d})'.format(id(len)))
```

This program defines a function, `inner_func` nested inside another, `outer_func`. After these definitions, the execution proceeds as follows:

1. Global variables `a=6` and `b=7` are initialized.
2. `outer_func` is called:
    a. `outer_func` defines a local variable, `len=2`.
    b. The values of `a` and `b` are printed; they don't exist in local scope and there isn't any enclosing scope, so Python searches for and finds them in global scope: their values (6 and 7) are output.
    c. The value of local variable `len` (2) is printed.
    d. `inner_func` is called:
        (1) A local variable, `a=9` is defined.
        (2) The value of this local variable is printed.
        (3) The value of `b` is printed; `b` doesn't exist in local scope so Python looks for it in enclosing scope, that of `outer_func`. It isn't found

there either, so Python proceeds to look in global scope where it is found: the value `b=7` is printed.

(4) The value of `len` is printed: `len` doesn't exist in local scope, but it is in the enclosing scope since `len=2` is defined in `outer_func`: its value is output.

3. After `outer_func` has finished execution, the values of `a` and `b` in global scope are printed.
4. The value of `len` is printed. This is not defined in global scope, so Python searches its own built-in names: `len` is the built-in function for determining the lengths of sequences. This function is itself an object and it provides a short string description of itself when printed.

```
inside outer_func, a is 6 (id=232)
inside outer_func, b is 7 (id=264)
inside outer_func, len is 2 (id=104)
inside inner_func, a is 9 (id=328)
inside inner_func, b is 7 (id=264)
inside inner_func, len is 2 (id=104)
in global scope, a is 6 (id=232)
in global scope, len is <built-in function len> (id=977)
```

Note that in this example `outer_func` has (perhaps unwisely) redefined (*re-bound*) the name `len` to the integer object `2`. This means that the original `len` built-in function is not available within this function (and neither is it available within the enclosed function, `inner_func`).

2.7.4 ◊ Passing arguments to functions

A common question from new users of Python who come to it with a knowledge of other computer languages is, are arguments to functions passed "by value" or "by reference?" In other words, does the function make its own copy of the argument, leaving the caller's copy unchanged, or does it receive a "pointer" to the location in memory of the argument, the contents of which the function *can* change? The distinction is important for languages such as C, but does not fit well into the Python *name-object* model. Python function arguments are sometimes (not very helpfully) said to be "references, passed by value." Recall that everything in Python is an object, and the same object may have multiple identifiers (what we have been loosely calling "variables" up until now). When a name is passed to a function, the "value" that is passed is, in fact, the it points to. Whether the function can change the object or not (from the point of view of the caller) depends on whether the object is mutable or immutable.

A couple of examples should make this clearer. A simple function, `func1`, taking an integer argument, receives a reference to that integer object, to which it attaches a local name (which may or may not be the same as the global name). The function cannot change the integer object (which is immutable), so any reassignment of the local name simply points to a new object: the global name still points to the original integer object.

```
>>> def func1(a):
...     print('func1: a = {}, id = {}'.format(a, id(a)))
...     a = 7 # reassigns local a to the integer 7
...     print('func1: a = {}, id = {}'.format(a, id(a)))
...
>>> a = 3
>>> print('global: a = {}, id = {}'.format(a, id(a)))

global: a = 3, id = 4297242592

>>> func1(a)
func1: a = 3, id = 4297242592
func1: a = 7, id = 4297242720

>>> print('global: a = {}, id = {}'.format(a, id(a)))

global: a = 3, id = 4297242592
```

`func1` therefore prints `3` (inside the function, `a` is initially the local name for the original integer object); it then prints `7` (this local name now points to a new integer object, with a new `id`) – see Figure 2.5. After it returns, the global name `a` still points to the original `3`.

Now consider passing a mutable object, such as a `list` to a function, `func2`. This time, an assignment to the list changes the original object, and these changes persist after the function call.

```
>>> def func2(b):
...     print('func2: b = {}, id = {}'.format(b, id(b)))
...     b.append(7) # add an item to the list
...     print('func2: b = {}, id = {}'.format(b, id(b)))
...
>>> c = [1, 2, 3]
>>> print('global: c = {}, id = {}'.format(c, id(c)))

global: c = [1, 2, 3], id = 4361122448

>>> func2(c)
func2: b = [1, 2, 3], id = 4361122448
func2: b = [1, 2, 3, 7], id = 4361122448
```

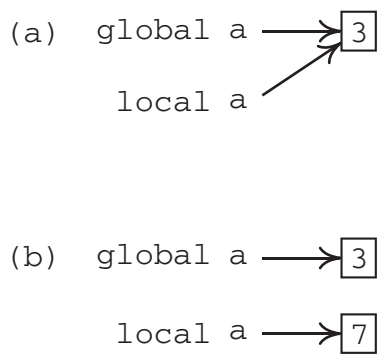

Figure 2.5 Immutable objects. Within `func1`: (a) before reassigning the local variable `a` and (b) after reassigning the value of local variable `a`.

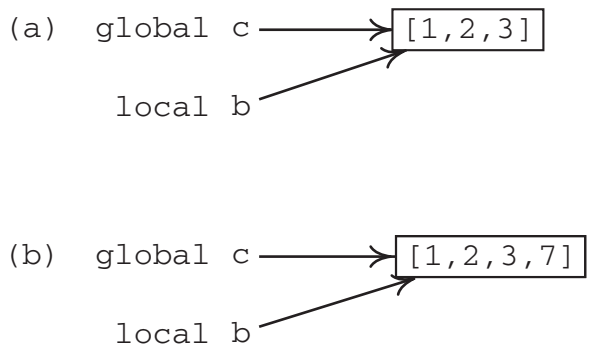

Figure 2.6 Mutable objects. Within `func2`: (a) before appending to the list pointed to by both global variable `c` and local variable `b` and (b) after appending to the list with `b.append(7)`.

```
>>> print('global: c = {}, id = {}'.format(c, id(c)))

global: c = [1, 2, 3, 7], id = 4361122448
```

Note that it doesn't matter what name is given to the list by the function: this name points to the same object, as you can see from its `id`. The relationships between the variable names and objects is illustrated in Figure 2.6.

So are Python arguments passed by value or by reference? The best answer is probably that arguments are passed by value, but that value is a reference to an object (which can be mutable or immutable).

Example E2.29 The *Lazy Caterer's Sequence*, $f(n)$, describes the maximum number of pieces a circular pizza can be divided into with an increasing number of cuts, n. Clearly $f(0) = 1, f(1) = 2$ and $f(2) = 4$). For $n = 3, f(3) = 7$ (the maximum number of pieces are formed if the cuts do not intesect at a common point). It can be shown that the general recursion formula,

$$f(n) = f(n-1) + n,$$

applies. Although there is a closed form for this sequence, $f(n) = \frac{1}{2}(n^2 + n + 2)$, we could also define a function to grow a list of consecutive values in the sequence:

```
>>> def f(seq):
...     seq.append(seq[-1] + n)
...
>>> seq = [1]      # f(0) = 1
>>> for n in range(1,16):
...     f(seq)
...
>>> print(seq)
[1, 2, 4, 7, 11, 16, 22, 29, 37, 46, 56, 67, 79, 92, 106, 121]
```

The list `seq` is mutable and so grows in place each time the function `f()` is called. The `n` referred to within this function is the name found in global scope (the `for` loop counter).

2.7.5 Recursive functions

A function that can call itself is called a *recursive* function. Recursion is not always necessary but can lead to elegant algorithms in some situations.[32] For example, one way to calculate the factorial of an integer $n \geq 1$ is to define the following recursive function:

```
>>> def factorial(n):
...     if n == 1:
...         return 1
...     return n * factorial(n-1)
...
>>> factorial(5)
120
```

Here, a call to `factorial(n)` returns n times whatever is returned by the call to `factorial(n-1)`, which returns $n-1$ times the returned values of `factorial(n-2)` and so on until `factorial(1)` which is 1 by definition. That is, the algorithm makes use of the fact that $n! = n \cdot (n-1)!$ Care should be taken in implementing such recursive algorithms to ensure that they stop when some condition is met.[33]

Example E2.30 The famous *Tower of Hanoi* problem involves three poles, one of which (pole A) is stacked with n differently sized, circular discs, in decreasing order of diameter with the largest at the bottom. The task is to move the stack to the third pole (pole C) by moving one disc at a time in such a way that a larger disc is never placed on a smaller one. It is necessary to use the second pole (pole B) as an intermediate resting place for the discs.

The problem can be solved using the following recursive algorithm. Label the discs D_i with D_1 the smallest disc and D_n the largest.

- Move discs $D_1, D_2, \cdots, D_{n-1}$ from A to B;
- Move disc D_n from A to C;
- Move discs $D_1, D_2, \cdots, D_{n-1}$ from B to C.

The second step is a single move, but the first and last require the movement of a stack of $n-1$ discs from one peg to another – which is exactly what the algorithm itself solves!

In the following code, we identify the discs by the integers $1, 2, 3, \cdots$ stored in one of three lists, `A`, `B` and `C`. The initial state of the system, with all discs on pole A is denoted by, for example, `A = [5,4,3,2,1]` where the first indexed item is the "bottom" of the pole and the last indexed item is the "top." The rules of the problem require that these lists must always be *decreasing* sequences.

---

[32] In fact, because of the overhead involved in making a function call, a recursive algorithm can be expected to be slower than a well-designed iterative one.

[33] In practice, an infinite loop is not possible because of the memory overhead involved in each function call, and Python sets a maximum recursion limit.

Listing 2.6 The Tower of Hanoi problem

```
# eg2-hanoi.py

def hanoi(n, P1, P2, P3):
    """ Move n discs from pole P1 to pole P3. """
    if n == 0:
        # No more discs to move in this step
        return

    global count
    count += 1

    # move n-1 discs from P1 to P2
    hanoi(n-1, P1, P3, P2)

    if P1:
        # move disc from P1 to P3
        P3.append(P1.pop())
        print(A, B, C)

    # move n-1 discs from P2 to P3
    hanoi(n-1, P2, P1, P3)

# Initialize the poles: all n discs are on pole A.
n = 3
A = list(range(n,0,-1))
B, C = [], []

print(A, B, C)
count = 0
hanoi(n, A, B, C)
print(count)
```

Note that the `hanoi` function just moves a stack of discs from one pole to another: lists (representing the poles) are passed into it in some order, and it moves the discs from the pole represented by the first list, known locally as `P1`, to that represented by the third (`P3`). It does not need to know which list is `A`, `B` or `C`.

2.7.6 Exercises

Questions

Q2.7.1 The following small programs each attempt to output the simple sum:

```
   56
  +44
-----
  100
-----
```

Which two programs work as intended? Explain carefully what is wrong with each of the others.

a.
```
def line():
    '-----'

my_sum = '\n'.join(['   56', '  +44', line(), '  100', line()])
print(my_sum)
```

b.
```
def line():
    return '-----'

my_sum = '\n'.join(['   56', '  +44', line(), '  100', line()])
print(my_sum)
```

c.
```
def line():
    return '-----'

my_sum = '\n'.join(['   56', '  +44', line, '  100', line])
print(my_sum)
```

d.
```
def line():
    print('-----')

print('   56')
print('  +44')
print(line)
print('  100')
print(line)
```

e.
```
def line():
    print('-----')

print('   56')
print('  +44')
print(line())
print('  100')
print(line())
```

f.
```
def line():
    print('-----')

print('   56')
print('  +44')
line()
print('  100')
line()
```

Q2.7.2 The following code snippet attempts to calculate the balance of a savings account with an annual interest rate of 5% after 4 years, if it starts with a balance of $100.

```
>>> balance = 100
>>> def add_interest(balance, rate):
...     balance += balance * rate / 100
...
>>> for year in range(4):
...     add_interest(balance, 5)
```

```
...     print('Balance after year {}: ${:.2f}'.format(year+1, balance))
...
Balance after year 1: $100.00
Balance after year 2: $100.00
Balance after year 3: $100.00
Balance after year 4: $100.00
```

Explain why this doesn't work and then provide a working alternative.

Q2.7.3 A *Harshad number* is an integer that is divisible by the sum of its digits (e.g., 21 is divisible by $2 + 1 = 3$ and so is a Harshad number). Correct the following code which should return `True` or `False` if `n` is a Harshad number or not respectively:

```
def digit_sum(n):
    """ Find the sum of the digits of integer n. """

    s_digits = list(str(n))
    dsum = 0
    for s_digit in s_digits:
        dsum += int(s_digit)

def is_harshad(n):
    return not n % digit_sum(n)
```

When run, the function `is_harshad` raises an error:

```
>>> is_harshad(21)
TypeError: unsupported operand type(s) for %: 'int' and 'NoneType'
```

Problems

P2.7.1 The word game Scrabble is played on a 15×15 grid of squares referred to by a row index letter (A – O) and a column index number (1 – 15). Write a function to determine whether a word will fit in the grid, given the position of its first letter as a string (e.g., `'G7'`) a variable indicating whether the word is placed to read *across* or *down* the grid and the word itself.

P2.7.2 Write a program to find the smallest positive integer, n, whose factorial is *not* divisible by the sum of its digits. For example, 6 is not such a number because $6! = 720$ and $7 + 2 + 0 = 9$ divides 720.

P2.7.3 Write two functions which, given two lists of length 3 representing three-dimensional vectors $\mathbf{a}$ and $\mathbf{b}$, calculate the dot product, $\mathbf{a} \cdot \mathbf{b}$ and the vector (cross) product, $\mathbf{a} \times \mathbf{b}$.

Write two more functions to return the scalar triple product, $\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$ and the vector triple product, $\mathbf{a} \times (\mathbf{b} \times \mathbf{c})$.

P2.7.4 A right regular pyramid with height h and a base consisting of a regular n-sided polygon of side length s has a volume, $V = \frac{1}{3}Ah$ and total surface area, $S = A + \frac{1}{2}nsl$ where A is the base area and l the slant height, which may be calculated from the *apothem* of the base polygon, $a = \frac{1}{2}s \cot \frac{\pi}{n}$ as $A = \frac{1}{2}nsa$ and $l = \sqrt{h^2 + a^2}$.

Use these formulas to define a function, `pyramid_AV`, returning V and S when passed values for n, s and h.

P2.7.5 The range of a projectile launched at an angle α and speed v on flat terrain is

$$R = \frac{v^2 \sin 2\alpha}{g},$$

where g is the acceleration due to gravity which may be taken to be $9.81\ \mathrm{m\,s^{-2}}$ for Earth. The maximum height attained by the projectile is given by

$$H = \frac{v^2 \sin^2 \alpha}{2g}.$$

(We neglect air resistance and the curvature and rotation of the Earth.) Write a function to calculate and return the range and maximum height of a projectile, taking α and v as arguments. Test it with the values $v = 10\ \mathrm{m\,s^{-1}}$ and $\alpha = 30°$.

P2.7.6 Write a function, `sinm_cosn`, which returns the value of the following definite integral for integers $m, n > 1$.

$$\int_0^{\pi/2} \sin^n\theta \cos^m\theta \, \mathrm{d}\theta = \begin{cases} \frac{(m-1)!!(n-1)!!}{(m+n)!!}\frac{\pi}{2} & m, n \text{ both even,} \\ \frac{(m-1)!!(n-1)!!}{(m+n)!!} & \text{otherwise.} \end{cases}$$

Hint: for calculating the double factorial, see Exercise P2.4.6.

P2.7.7 Write a function that determines if a string is a palindrome (that is, reads the same backward as forward) *using recursion*.

P2.7.8 *Tetration* may be thought of as the next operator after exponentiation: Thus, where $x \times n$ can be written as the sum $x + x + x + \cdots + x$ with n terms, and x^n is the multiplication of n factors: $x \cdot x \cdot x \cdots x$, the expression written ${}^n x$ is equal to the repeated exponentiation involving n occurrences of x:

$${}^n x = x^{x^{\cdot^{\cdot^{x}}}}$$

For example, ${}^4 2 = 2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65536$. Note that the exponential "tower" is evaluated from top to bottom.

Write a recursive function to calculate ${}^n x$ and test it (for small, positive real values of x and non-negative integers n: tetration generates *very* large numbers)!

How many digits are there in ${}^3 5$? In ${}^5 2$?