4 The core Python language II

This chapter continues the introduction to the core Python language started in Chapter 2 with a description of Python error handling with exceptions, the data structures known as dictionaries and sets, some convenient and efficient idioms to achieve common tasks, and a survey of some of the modules provided in the Python standard library. Finally, we present a brief introduction to *object-oriented programming* with Python.

4.1 Errors and exceptions

Python distinguishes between two types of error: *Syntax errors* and other *exceptions*. Syntax errors are mistakes in the grammar of the language and are checked for before the program is executed. Exceptions are *runtime* errors: conditions usually caused by attempting an invalid operation on an item of data. The distinction is that syntax errors are always fatal: there is nothing the Python compiler can do for you if your program does not conform to the grammar of the language. Exceptions, however, are conditions that arise during the running of a Python program (such as division by zero) and a mechanism exists for "catching" them and handling the condition gracefully without stopping the program's execution.

4.1.1 Syntax errors

Syntax errors are caught by the Python compiler and produce a message indicating where the error occurred. For example,

```
>>> for lambda in range(8):
  File "<stdin>", line 1
    for lambda in range(8):
              ^
SyntaxError: invalid syntax
```

Because `lambda` is a reserved keyword, it cannot be used as a variable name. Its occurrence where a variable name is expected is therefore a syntax error. Similarly,

```
>>> for f in range(8:
  File "<stdin>", line 1
    for f in range(8:
                   ^
SyntaxError: invalid syntax
```

The syntax error here occurs because a single argument to the `range` built-in must be given as an integer between parentheses: the colon breaks the syntax of calling functions and so Python complains of a syntax error.

Because a line of Python code may be split within an open bracket ("`()`", "`[]`", or "`{}`"), a statement split over several lines can sometimes cause a `SyntaxError` to be indicated somewhere other than the location of the true bug. For example,

```
>>> a = [1, 2, 3, 4,
... b = 5
  File "<stdin>", line 4
    b = 5
    ^
SyntaxError: invalid syntax
```

Here, the statement `b = 5` is syntactically valid: the error arises from failing to close the square bracket of the previous list declaration (the Python shell indicates that a line is a continuation of a previous one with the initial ellipsis ("`...`").

There are two special types of `SyntaxError` that are worth mentioning: an `IndentationError` occurs when a block of code is improperly indented and `TabError` is raised when a tabs and spaces are mixed inconsistently to provide indentation.[1]

Example E4.1 A common syntax error experienced by beginner Python programmers is in using the assignment operator "`=`" instead of the equality operator "`==`" in a conditional expression:

```
>>> if a = 5:
  File "<stdin>", line 1
    if a = 5:
         ^
SyntaxError: invalid syntax
```

This assignment `a = 5` does not return a value (it simply assigns the integer object `5` to the variable name `a`) and so there is nothing corresponding to `True` or `False` that the `if` statement can use: hence the `SyntaxError`. This contrasts with the C language in which an assignment returns the value of the variable being assigned (and so the statement `a = 5` evaluates to `true`). This behavior is the source of many hard-to-find bugs and security vulnerabilities and its omission from the Python language is by design.

4.1.2 Exceptions

An exception occurs when an syntactically correct expression is executed and causes a *runtime error*. There are different types of built-in exception, and custom exceptions can be defined by the programmer if required. If an exception is not "caught" using the `try ... except` clause described later, Python produces a (usually helpful) error message. If the exception occurs within a function (which may have been called, in turn, by

[1] This error can be avoided by using only spaces to indent code.

another function, and so on), the message returned takes the form of a *stack traceback*: the history of function calls leading to the error is reported so that its location in the program execution can be determined.

Some built-in exceptions will be familiar from your use of Python so far.

NameError

```
>>> print('4z = ', 4*z)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
```

A `NameError` exception occurs when a variable name is used that hasn't been defined: the `print` statement here is valid, but Python doesn't know what to print for `z`.

ZeroDivisionError

```
>>> a, b = 0, 5
>>> b / a

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
```

Division by zero is not mathematically defined.

TypeError and ValueError

A `TypeError` is raised if an object of the wrong type is used in an expression or function. For example,

```
>>> '00' + 7

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Python is a (fairly) strongly typed language, and it is not possible to add a string to an integer.[2]

A `ValueError`, on the other hand, occurs when the object involved has the correct *type* but an invalid *value*:

```
>>> float('hello')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'hello'
```

[2] Unlike in, say, Javascript or PHP, where it seems anything goes.

Table 4.1 Common Python exceptions

Exception	Cause and description
`FileNotFoundError`	Attempting to open a file or directory that does not exist – this exception is a particular type of `OSError`.
`IndexError`	Indexing a sequence (such as a list or string) with a subscript that is out of range.
`KeyError`	Indexing a dictionary with a key that does not exist in that dictionary (see Section 4.2.2).
`NameError`	Referencing a local or global variable name that has not been defined.
`TypeError`	Attempting to use an object of an inappropriate type as an argument to a built-in operation or function.
`ValueError`	Attempting to use an object of the correct type but with an incompatible value as an argument to a built-in operation or function.
`ZeroDivisionError`	Attempting to divide by zero (either explicitly (using '/' or '//') or as part of a modulo operation '`%`').
`SystemExit`	Raised by the `sys.exit` function (see Section 4.4.1) – if not handled, this function causes the Python interpreter to exit.

The `float` built-in does take a string as its argument, so `float('hello')` is not a `TypeError`: the exception is raised because the particular string '`hello`' does not evaluate to a meaningful floating point number. More subtly,

```
>>> int('7.0')

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '7.0'
```

A string that looks like a `float` cannot be directly cast to `int`: to obtain the result probably intended, use `int(float('7.0'))`.

Table 4.1 provides a list of the more commonly encountered built-in exceptions and their descriptions.

Example E4.2 When an exception is raised but not *handled* (see Section 4.1.3), Python will issue a *traceback report* indicating where in the program flow it occurred. This is particularly useful when an error occurs within nested functions or within imported modules. For example, consider the following short program:[3]

```
# exception-test.py
import math

def func(x):
    def trig(x):
```

[3] Note the use of `f.__name__` to return a string representation of a function's name in this program; for example, `math.sin.__name__` is `'sin'`.

```
        for f in (math.sin, math.cos, math.tan):
            print('{f}({x}) = {res}'.format(f=f.__name__, x=x, res=f(x)))
    def invtrig(x):
        for f in (math.asin, math.acos, math.atan):
❶           print('{f}({x}) = {res}'.format(f=f.__name__, x=x, res=f(x)))
    trig(x)
❷   invtrig(x)

❸ func(1.2)
```

The function `func` passes its argument, `x`, to its two nested functions. The first, `trig`, is unproblematic but the second, `invtrig`, is expected to fail for `x` out of the domain (range of acceptable values) for the inverse trigonometric function, `asin`:

```
sin(1.2) = 0.9320390859672263
cos(1.2) = 0.3623577544766736
tan(1.2) = 2.5721516221263183
Traceback (most recent call last):
  File "exception-test.py", line 14, in <module>
    func(1.2)
  File "exception-test.py", line 12, in func
    invtrig(x)
  File "exception-test.py", line 10, in invtrig
    print('{f}({x}) = {res}'.format(f=f.__name__, x=x, res=f(x)))
ValueError: math domain error
```

Following the traceback backward shows that the `ValueError` exception was raised within `invtrig` (line 10, ❶), which was called from within `func` (line 12, ❷), which was itself called by the `exception-test.py` module (i.e., program) at line 14, ❸.

4.1.3 Handling and raising exceptions

Handling exceptions

Often, a program must manipulate data in a way which might cause an exception to be raised. Assuming such a condition is not to cause the program to exit with an error but to be handled "gracefully" in some sense (an invalid data point ignored, division by a zero value skipped, and so on), there are two approaches to this situation: check the value of the data object before using it, or "handle" any exception that is raised before resuming execution. The Pythonic approach is the latter, summed up in the expression *It is Easier to Ask Forgiveness than to seek Permission* (EAFP).

To catch an exception in a block of code, write the code within a `try:` clause and handle any exceptions raised in an `except:` clause. For example,

```
try:
    y = 1 / x
    print('1 /', x, ' = ',y)
except ZeroDivisionError:
    print('1 / 0 is not defined.')
# ... more statements
```

No check is required: we go ahead and calculate `1/x` and handle the error arising from division by zero if necessary. The program execution continues after the `except` block

whether the `ZeroDivisionError` exception was raised or not. If a different exception is raised (e.g., a `NameError` because `x` is not defined), then this will not be caught – it is an *unhandled exception* and will trigger an error message.

To handle more than one exception in a single `except` block, list them in a tuple (which must be within brackets).

```
try:
    y = 1. / x
    print('1 /', x, ' = ',y)
except (ZeroDivisionError, NameError):
    print('x is zero or undefined!)
# ... more statements
```

To handle each exception separately, use more than one `except` clause:

```
try:
    y = 1. / x
    print('1 /', x, ' = ',y)
except ZeroDivisionError:
    print('1 / 0 is not defined.')
except NameError:
    print('x is not defined')
# ... more statements
```

Warning: You may come across the following type of construction:

```
try:
    [do something]
except:                 # Don't do this!
    pass
```

This will execute the statements in the `try` block and ignore *any* exceptions raised – it is very unwise to do this as it makes code very hard to maintain and debug (errors, whatever their cause, are silently supressed). Always catch specific exceptions and handle them appropriately, allowing any other exceptions to "bubble up" to be handled (or not) by any other `except` clauses.

The `try ... except` statement has two more optional clauses (which must follow any `except` clauses if they are used). Statements in a block following the `finally` keyword are *always* executed, whether an exception was raised or not. Statements in a block following the `else` keyword are executed if an exception was *not* raised (see Example E4.5).

◊ Raising exceptions

Usually an exception is raised by the Python interpreter as a result of some behavior (anticipated or not) by the program. But sometimes it is desirable for a program to raise a particular exception if some condition is met. The `raise` keyword allows a program to force a specific exception and customize the message or other data associated with it. For example,

```
if n % 2:
    raise ValueError('n must be even!')
# statements here may proceed, knowing n is even ...
```

A related keyword, `assert`, evaluates a conditional expression and raises an `AssertionError` exception if that expression is not `True`. This is useful to check that some essential condition holds at a specific point in your program's execution and is often helpful in debugging.

```
>>> assert 2==2         # [silence]: 2==2 is True so nothing happens
>>>
>>> assert 1==2         # Will raise the AssertionError

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

The syntax `assert` *`expr1`*`,` *`expr2`* passes *`expr2`* (typically an error message) to the `AssertionError`:

```
>>> assert 1==2, 'One does not equal two'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: One does not equal two
```

Python is a dynamically typed language and arguments of any type can be legally passed to a function, even if that function is expecting a particular type. It is sometimes necessary to check that an argument object is of a suitable type before using it, and `assert` could be used to do this.

Example E4.3 The following function returns a string representation of a two-dimensional (2D) or three-dimensional (3D) vector, which must be represented as a `list` or `tuple` containing two or three items.

```
>>> def str_vector(v):
...     assert type(v) is list or type(v) is tuple,\
...                 'argument to str_vector must be a list or tuple'
...     assert len(v) in (2,3),\
...                 'vector must be 2D or 3D in str_vector'
...     unit_vectors = ['i','j','k']
...     s = []
...     for i, component in enumerate(v):
...         s.append('{}{}'.format(component, unit_vectors[i]))
❶ ...     return '+'.join(s).replace('+-', '-')
```

❶ `replace('+-', '-')` here converts, for example, `'4i+-3j'` into `'4i-3j'`.

Example E4.4 As another example, suppose you have a function that calculates the vector (cross) product of two vectors represented as `list` objects. This product is only defined for three-dimensional vectors, so calling it with lists of any other length is an error.

```
>>> def cross_product(a, b):
...     assert len(a) == len(b) == 3, 'Vectors a, b must be three-dimensional'
...     return [a[1]*b[2] - a[2]*b[1],
...             a[2]*b[0] - a[0]*b[2],
...             a[0]*b[1] - a[1]*b[0]]
...
>>> cross_product([1, 2, -1], [2, 0, -1, 3])    # Oops

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cross_product
AssertionError: Vectors a, b must be three-dimensional
>>> cross_product([1, 2, -1], [2, 0, -1])
[-2, -1, -4]
```

Example E4.5 The following code gives an example of the use of a `try ... except ... else ... finally` clause:

```
# try-except-else-finally.py

def process_file(filename):
    try:
        fi = open(filename, 'r')
    except IOError:
        print('Oops: couldn\'t open {} for reading'.format(filename))
        return
    else:
❶       lines = fi.readlines()
        print('{} has {} lines.'.format(filename, len(lines)))
        fi.close()
    finally:
❷       print('   Done with file {}'.format(filename))

    print('The first line of {} is:\n{}'.format(filename, lines[0]))
    # further processing of the lines ...
    return

process_file('sonnet0.txt')
process_file('sonnet18.txt')
```

❶ Within the `else` block, the contents of the file are only read if the file was successfully opened.

❷ Within the `finally` block, '`Done with file` *`filename`*' is printed whether the file was successfully opened or not.

Assuming that the file `sonnet0.txt` does not exist but that `sonnet18.txt` does, running this program prints:

```
Oops: couldn't open sonnet0.txt for reading
   Done with file sonnet0.txt
sonnet18.txt has 14 lines.
   Done with file sonnet18.txt
The first line of sonnet18.txt is:
Shall I compare thee to a summer's day?
```

4.1.4 Exercises

Questions

Q4.1.1 What is the point of `else`? Why not put statements in this block inside the original `try` block?

Q4.1.2 What is the point of the `finally` clause? Why not put any statements you want executed after the `try` block (regardless of whether or not an exception has been raised) after the entire `try ... except` clause?

Hint: see what happens if you modify Example E4.5 to put the statements in the `finally` clause after the `try` block.

Problems

P4.1.1 Write a program to read in the data from the file `swallow-speeds.txt` (available at scipython.com/ex/ada) and use it to calculate the average air-speed velocity of an (unladen) African swallow. Use exceptions to handle the processing of lines that do not contain valid data points.

P4.1.2 Adapt the function of Example E4.3, which returns a vector in the following form:

```
>>> print(str_vector([-2, 3.5])
-2i + 3.5j
>>> print(str_vector((4, 0.5, -2))
4i + 0.5j - 2k
```

to raise an exception if any element in the vector array does not represent a real number.

P4.1.3 Python follows the convention of many computer languages in choosing to define $0^0 = 1$. Write a function, `powr(a, b)`, which behaves the same as the Python expression `a**b` (or, for that matter, `math.pow(a,b)`) but raises a `ValueError` if `a` and `b` are both zero.

4.2 Python objects III: dictionaries and sets

A *dictionary* in Python is a type of "associative array" (also known as a "hash" in some languages). A dictionary can contain any objects as its *values*, but unlike sequences such as lists and tuples, in which the items are indexed by an integer starting at 0, each item in a dictionary is indexed by a unique *key*, which may be any *immutable* object.[4] The dictionary therefore exists as a number of *key-value* pairs, which do not have any particular order. Dictionaries themselves are *mutable* objects.

[4] Actually, dictionary keys can be any *hashable* object: a hashable object in Python is one with a special method for generating a particular integer from any instance of that object; the idea is that instances (which may be large and complex) that compare as equal should have hash numbers that also compare as equal so they can be rapidly looked up in a *hash table*. This is important for some data structures and for optimizing the speed of algorithms involving their objects.

4.2.1 Defining and indexing a dictionary

An dictionary can be defined by giving *key: value* pairs between braces:

```
>>> height = {'Burj Khalifa': 828., 'One World Trade Center': 541.3,
              'Mercury City Tower': -1., 'Q1': 323.,
              'Carlton Centre': 223., 'Gran Torre Santiago': 300.,
              'Mercury City Tower': 339.}
>>> height
{'Q1': 323.0,
 'Burj Khalifa': 828.0,
 'Carlton Centre': 223.0,
 'One World Trade Center': 541.3,
 'Mercury City Tower': 339.0,
 'Gran Torre Santiago': 300.0}
```

The command `print(height)` will return the dictionary in the same format (between braces), but in no particular order. If the same key is attached to different values (as `'Mercury City Tower'` is here), only the most recent value survives: the keys in a dictionary are unique.

An individual item can be retreived by indexing it with its key, either as a literal (`'Q1'`) or with a variable equal to the key:

```
>>> height['One World Trade Center']
541.3
>>> building = 'Carlton Centre'
>>> height[building]
223.0
```

Items in a dictionary can also be *assigned* by indexing it in this way:

```
height['Empire State Building'] = 381.
height['The Shard'] = 306.
```

An alternative way of defining a dictionary is to pass a sequence of `(key, value)` pairs to the `dict` constructor. If the keys are simple strings (of the sort that could be used as variable names), the pairs can also be specified as keyword arguments to this constructor:

```
>>> ordinal = dict([(1, 'First'), (2, 'Second'), (3, 'Third')])
>>> mass = dict(Mercury=3.301e23, Venus=4.867e24, Earth=5.972e24)
>>> ordinal[2]     # NB 2 here is a key, not an index
'Second'
>>> mass['Earth']
5.972e+24
```

A `for`-loop iteration over a dictionary returns the dictionary *keys* (in no particular order):

```
>>> for c in ordinal:
...     print(c, ordinal[c])
...
3 Third
1 First
2 Second
```

Example E4.6 A simple dictionary of roman numerals:

```
>>> numerals = {'one':'I', 'two':'II', 'three':'III', 'four':'IV', 'five':'V',
                'six':'VI', 'seven':'VII', 'eight':'VIII',
                1: 'I', 2: 'II', 3: 'III', 4:'IV', 5: 'V', 6:'VI', 7:'VII',
                8:'VIII'}
>>> for i in ['three', 'four', 'five', 'six']:
...     print(numerals[i], end=' ')
...
III IV V VI
>>> for i in range(8,0,-1):
...     print(numerals[i], end=' ')
VIII VII VI V IV III II I
```

Note that even though the keys are stored in an arbitrary order, the dictionary can be indexed in any order. Note also that although the dictionary *keys* must be unique, the dictionary *values* need not be.

4.2.2 Dictionary methods

`get()`

Indexing a dictionary with a key that does not exist is an error:

```
>>> mass['Pluto']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Pluto'
```

However, the useful method `get()` can be used to retrieve the value, given a key if it exists, or some default value if it does not. If no default is specified, then `None` is returned. For example,

```
>>> print(mass.get('Pluto'))
None
>>> mass.get('Pluto', -1)
-1
```

`keys`, `values` and `items`

The three methods, `keys`, `values` and `items`, return respectively, a dictionary's keys, values and key-value pairs (as tuples). In previous versions of Python, each of these were returned in a list, but for most purposes this is wasteful of memory: calling `keys`, for example, required all of the dictionary's keys to to be copied as a list, which in most cases was simply iterated over. That is, storing a whole new copy of the dictionary's keys is not usually necessary. Python 3 solves this by returning an iterable object, which accesses the dictionary's keys one by one, without copying them to a list. This is faster and saves memory (important for very large dictionaries). For example,

```
>>> planets = mass.keys()
>>> print(planets)
dict_keys(['Venus', 'Mercury', 'Earth'])
>>> for planet in planets:
...     print(planet, mass[planet])
```

```
...
Venus 4.867e+24
Mercury 3.301e+23
Earth 5.972e+24
```

A `dict_keys` object can be iterated over any number of times, but it is not a list and cannot be indexed or assigned:

```
>>> planets = mass.keys()
>>> planets[0]


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_keys' object is not subscriptable
```

If you really do want a list of the dictionary's keys, simply pass the `dict_keys` object to the `list` constructor (which takes any kind of sequence and makes a `list` out of it):

```
>>> planet_list = list(mass.keys())
>>> planet_list[0]
'Venus'
❶ >>> planet_list[1] = 'Jupiter'
>>> planet_list
['Venus', 'Jupiter', 'Earth']
```

❶ This last assignment only changes the `planet_list` list; it doesn't alter the original dictionary's keys.

Similar methods exist for retrieving a dictionary's values and items (key-value pairs): the objects returned are `dict_values` and `dict_items`.

For example,

```
>>> mass.items()
dict_items([('Venus', 4.867e+24), ('Mercury', 3.301e+23), ('Earth', 5.972e+24)])
>>> mass.values()
dict_values([4.867e+24, 3.301e+23, 5.972e+24])
>>> for planet_data in mass.items():
...     print(planet_data)
...
('Venus', 4.867e+24)
('Mercury', 3.301e+23)
('Earth', 5.972e+24)
```

Example E4.7 A Python dictionary can act as a kind of simple database. The following code stores some information about some astronomical objects in a dictionary of tuples, keyed by the object name, and manipulates them to produce a list of planet densities.

Listing 4.1 Astronomical data

```
# eg4-astrodict.py
import math

# Mass (in kg) and radius (in km) for some astronomical bodies
body = {'Sun': (1.988e30, 6.955e5),
        'Mercury': (3.301e23, 2440.),
        'Venus': (4.867e+24, 6052.),
```

```
        'Earth': (5.972e24, 6371.),
        'Mars': (6.417e23, 3390.),
        'Jupiter': (1.899e27, 69911.),
        'Saturn': (5.685e26, 58232.),
        'Uranus': (8.682e25, 25362.),
        'Neptune': (1.024e26, 24622.)
       }

planets = list(body.keys())
# The sun isn't a planet!
planets.remove('Sun')

def calc_density(m, r):
    """ Returns the density of a sphere with mass m and radius r. """
    return m / (4/3 * math.pi * r**3)

rho = {}
for planet in planets:
    m, r = body[planet]
    # calculate the density in g/cm3
    rho[planet] = calc_density(m*1000, r*1.e5)

❶ for planet, density in sorted(rho.items()):
    print('The density of {0} is {1:3.2f} g/cm3'.format(planet, density))
```

❶ `sorted(rho.items())` returns a `list` of the `rho` dictionary's key-value pairs, sorted by key.

The output is

```
The density of Earth is 5.51 g/cm3
The density of Jupiter is 1.33 g/cm3
The density of Mars is 3.93 g/cm3
The density of Mercury is 5.42 g/cm3
The density of Neptune is 1.64 g/cm3
The density of Saturn is 0.69 g/cm3
The density of Uranus is 1.27 g/cm3
The density of Venus is 5.24 g/cm3
```

◊ Keyword arguments

In Section 2.7, we discussed the syntax for passing arguments to functions. In that description, it was assumed that the function would always know what arguments could be passed to it and these were listed in the function definition. For example,

```
def func(a, b, c):
```

Python provides a couple of useful features for handling the case where it is not necessarily known what arguments a function will receive. Including `*args` (after any "formally defined" arguments) places any additional positional argument into a `tuple`, `args`, as illustrated by the following code:

```
>>> def func(a, b, *args):
...     print(args)
...
```

```
>>> func(1, 2, 3, 4, 'msg')
(3, 4, 'msg')
```

That is, inside `func`, in addition to the formal arguments `a=1` and `b=2`, the arguments `3`, `4` and `'msg'` are available as the items of the `tuple` `args`. This `tuple` can be arbitrarily long. Python's own `print` built-in function works in this way: it takes an arbitrary number of arguments to output as a string, followed by some optional keyword arguments:

```
def print(*args, sep=' ', end='\n', file=None):
```

It is also possible to collect arbitrary *keyword arguments* (see Section 2.7.2) to a function inside a dictionary by using the `**kwargs` syntax in the function definition. Python takes any keyword arguments not specified in the function definition and packs them into the dictionary `kwargs`. For example,

```
>>> def func(a, b, **kwargs):
...     for k in kwargs:
...         print(k, '=', kwargs[k])
...
>>> func(1, b=2, c=3, d=4, s='msg')
d = 4
s = msg
c = 3
```

One can also use `*args` and `**kwargs` when *calling* a function, which can be convenient, for example, with functions that take a large number of arguments:

```
>>> def func(a, b, c, x, y, z):
...     print(a, b, c)
...     print(x, y, z)
...
>>> args = [1,2,3]
>>> kwargs = {'x': 4, 'y': 5, 'z': 'msg'}
>>> func(*args, **kwargs)
1 2 3
4 5 msg
```

4.2.3 Sets

A `set` is an *unordered* collection of *unique* items. As with dictionary keys, elements of a set must be hashable objects. A set is useful for removing duplicates from a sequence and for determining the union, intersection and difference between two collections. Because they are unordered, `set` objects cannot be indexed or sliced, but they can be iterated over, tested for membership, and they support the `len` built-in. A `set` is created by listing its elements between braces ({ . . . }) or by passing an iterable to the `set()` constructor:

```
>>> s = set([1, 1, 4, 3, 2, 2, 3, 4, 1, 3, 'surprise!'])
>>> s
{1, 2, 'surprise!', 3, 4}
>>> len(s)                  # cardinality of the set
5
```

```
>>> 2 in s, 6 not in s      # membership, nonmembership
(True, True)
>>> for item in s:
...     print(item)
...
1
2
surprise!
3
4
```

The `set` method `add` is used to add elements to the set. To remove elements there are several methods: `remove` removes a specified element but raises a `KeyError` exception if the element is not present in the set; `discard()` does the same but does not raise an error in this case. Both methods take (as a single argument) the element to be removed. `pop` (with no argument) removes an *arbitrary* element from the set and `clear` removes *all* elements from the set:

```
>>> s = {2,-2,0}
>>> s.add(1)
>>> s.add(-1)
❶ >>> s.add(1.0)
>>> s
{0, 1, 2, -1, -2}
>>> s.remove(1)
>>> s
{0, 2, -1, -2}
>>> s.discard(3)    # OK - does nothing
>>> s
{0, 2, -1, -2}
>>> s.pop()
0                   # (for example)
>>> s
{2, -1, -2}
>>> s.clear()
set()               # the empty set
```

❶ This statement will not add a new member to the set, even though the existing `1` is an integer and the item we're adding is a `float`. The test `1 == 1.0` is `True`, so `1.0` is considered to be already in the set.

`set` objects have a wide range of methods corresponding to the properties of mathematical sets; the most useful are illustrated in Table 4.2, which uses the following terms from set theory:

- The *cardinality* of a set, $|A|$, is the number of elements it contains.
- Two sets are *equal* if they both contain the same elements.
- Set A is a *subset* of set B ($A \subseteq B$) if all the elements of A are also elements of B; set B is said to be a *superset* of set A.
- Set A is a *proper subset* of B ($A \subset B$) if it is a subset of B but not equal to B; in this case, set B is said to be a *proper superset* of A.
- The *union* of two sets ($A \cup B$) is the set of all elements from both of them.
- The *intersection* of two sets ($A \cap B$) is the set of all elements they have in common.

Table 4.2 `set` methods

Method	Description
`isdisjoint(other)`	Is `set` disjoint with `other`?
`issubset(other)`, `set <= other`	Is `set` a subset of `other`?
`set < other`	Is `set` a proper subset of `other`?
`issuperset(other)`, `set >= other`	Is `set` a superset of `other`?
`set > other`	Is `set` a proper superset of `other`?
`union(other)` `set | other | ...`	The union of `set` and `other`(s)
`intersection(other)`, `set & other & ...`	The intersection of `set` and `other`(s)
`difference(other)`, `set - other - ...`	The difference of `set` and `other`(s)
`symmetric_difference(other)`, `set ^ other ^ ...`	The symmetric difference of `set` and `other`(s)

- The *difference* of set A and set B ($A \backslash B$) is the set of elements in A that are not in B.
- The *symmetric difference* of two sets, $A \triangle B$, is the set of elements in either but not in both.
- Two sets are said to be *disjoint* if they have no elements in common.

There are two forms for most `set` expressions: the operator-like syntax requires all arguments to be `set` objects, whereas explicit method calls will convert any iterable argument into a `set`.

```
>>> A = set((1, 2, 3))
>>> B = set((1, 2, 3, 4))
>>> A <= B
True
>>> A.issubset((1, 2, 3, 4))  # OK: (1, 2, 3, 4) is turned into a set
True
```

Some more examples:

```
>>> C, D = set((3, 4, 5, 6)), set((7, 8, 9))
>>> B | C                # union
{1, 2, 3, 4, 5, 6}
>>> A | C | D            # union of three sets
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> A & C                # intersection
{3}
>>> C & D
set()                   # the empty set
>>> C.isdisjoint(D)
True
>>> B - C                # difference
{1, 2}
>>> B ^ C                # symmetric difference
{1, 2, 5, 6}
```

◊ frozensets

sets are mutable objects (items can be added to and removed from a set); because of this they are *unhashable* and so cannot be used as dictionary keys or as members of other sets.

```
>>> a = set((1,2,3))
>>> b = set(('q', (1,2), a))

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>>
```

(In the same way, lists cannot be dictionary keys or set members.) There is, however, a frozenset object which is a kind of immutable (and hashable) set.[5] frozensets are fixed, unordered collections of unique objects and *can* be used as dictionary keys and set members.

```
>>> a = frozenset((1,2,3))
>>> b = set(('q', (1,2), a))     # OK: the frozenset a is hashable
>>> b.add(4)                     # OK: b is a regular set
>>> a.add(4)                     # Not OK: frozensets are immutable

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Example E4.8 A *Mersenne prime*, M_i, is a prime number of the form $M_i = 2^i - 1$. The set of Mersenne primes less than n may be thought of as the intersection of the set of all primes less than n, P_n, with the set, A_n, of integers satisfying $2^i - 1 < n$.

The following program returns a list of the Mersenne primes less than 1000000.

Listing 4.2 The Mersenne primes

```
import math

def primes(n):
    """ Return a list of the prime numbers <= n. """

    sieve = [True] * (n // 2)
    for i in range(3, int(math.sqrt(n))+1, 2):
        if sieve[i//2]:
            sieve[i*i//2::i] = [False] * ((n - i*i - 1) // (2*i) +1)
    return [2] + [2*i+1 for i in range(1, n // 2) if sieve[i]]

n = 1000000

❶ P = set(primes(n))
```

---

[5] In a sense, they are to sets what tuples are to lists.

```
# A list of integers 2^i-1 <= n
A = []
❷ for i in range(2, int(math.log(n+1, 2))+1):
    A.append(2**i - 1)

# The set of Mersenne primes as the intersection of P and A
M = P.intersection(A)

# Output as a sorted list of M
❸ print(sorted(list(M)))
```

The prime numbers are produced in a list by the function `primes`, which implements an optimized version of the Sieve of Eratosthenes algorithm (see Exercise P2.5.8); this is converted into the `set`, `P` (❶). We can take the intersection of this set with any iterable object using the `intersection` method, so there is no need to explicitly convert our second list of integers, `A`, (❷) into a `set`.

❸ Finally, the set of Mersenne primes we create, `M`, is an unordered collection, so for output purposes we convert it into a sorted `list`.

For $n = 1000000$, This output is

```
[3, 7, 31, 127, 8191, 131071, 524287]
```

4.2.4 Exercises

Questions

Q4.2.1 Write a one-line Python program to determine if a string is a *pangram* (a string that contains each letter of the alphabet at least once).

Q4.2.2 Write a function, using `set` objects, to remove duplicates from an ordered `list`. For example,

```
>>> remove_dupes([1,1,2,3,4,4,4,5,7,8,8,9])
[1, 2, 3, 4, 5, 7, 8, 9]
```

Q4.2.3 Predict and explain the effect of the following statements:

```
>>> set('hellohellohello')
>>> set(['hellohellohello'])
>>> set(('hellohellohello'))
>>> set(('hellohellohello',))
>>> set(('hello', 'hello', 'hello'))
>>> set(('hello', ('hello', 'hello')))
>>> set(('hello', ['hello', 'hello']))
```

Q4.2.4 If `frozenset` objects are *immutable*, how is this possible?

```
>>> a = frozenset((1,2,3))
>>> a |= {2,3,4,5}
>>> print(a)
frozenset([1, 2, 3, 4, 5])
```

Table 4.3 Resistor color codes

Color	Abbreviation	Significant figures	Multiplier	Tolerance
Black	`bk`	0	1	–
Brown	`br`	1	10	±1%
Red	`rd`	2	10^2	±2%
Orange	`or`	3	10^3	–
Yellow	`yl`	4	10^4	±5%
Green	`gr`	5	10^5	±0.5%
Blue	`bl`	6	10^6	±0.25%
Violet	`vi`	7	10^7	±0.1%
Gray	`gy`	8	10^8	±0.05%
White	`wh`	9	10^9	–
Gold	`au`	–	–	±5%
Silver	`ag`	–	–	±10%
None	`--`	–	–	±20%

Problems

P4.2.1 The values and tolerances of older resistors are identified by four colored bands: the first two indicate the first two significant figures of the resistance in ohms, the third denotes a decimal multiplier (number of zeros), and the fourth indicates the tolerance. The colors and their meanings for each band are listed in Table 4.3.

For example, a resistor with colored bands violet, yellow, red, green has value $74 \times 10^2 = 7400\ \Omega$ and tolerance ±0.5%.

Write a program that defines a function to translate a list of four color abbreviations into a resistance value and a tolerance. For example,

```
In [x]: print(get_resistor_value(['vi', 'yl', 'rd', 'gr']))
Out[x]: (7400, 0.5)
```

P4.2.2 The novel *Moby-Dick* is out of copyright and can be downloaded as a text file from the Project Gutenberg website at www.gutenberg.org/2/7/0/2701/. Write a program to output the 100 words most frequently used in the book by storing a count of each word encountered in a dictionary.

Hint: use Python's string methods to strip out any punctuation. It suffices to replace any instances of the following characters with the empty string: `!?":;,()'.*[]`. When you have a dictionary with words as the keys and the corresponding word counts as the values, create a list of (*count*, *word*) tuples and sort it.

Bonus exercise: compare the frequencies of the top 2000 words in *Moby-Dick* with the prediction of *Zipf's Law*:

$$\log f(w) = \log C - a \log r(w),$$

where $f(w)$ is the number of occurrences of word w, $r(w)$ is the corresponding *rank* (1 = most common, 2 = second most common, etc.) and C and a are constants. In the

traditional formulation of the law, $C = \log f(w_1)$ and $a = 1$, where w_1 is the most common word, such that $r(w_1) = 1$.

P4.2.3 *Reverse notation* (RPN) (or *postfix* notation) is a notation for mathematical expressions in which each operator follows all of its operands (in contrast to the more familiar *infix* notation, in which the operator appears *between* the operands it acts on). For example, the infix expression `5+6` is written in RPN as `5 6 +`. The advantage of this approach is that parentheses are not necessary: to evaluate `(3+7) / 2`, it may be written as `3 7 + 2 /`. An RPN expression is evaluated left to right with the intermediate values pushed onto a *stack* – a last-in, first-out list of values – and retrieved (popped) from the stack when needed by an operator. Thus, the expression `3 7 + 2 /` proceeds with `3` and then `7` pushed to the stack (with 7 on top). The next token is `+`, so the values are retrieved, added, and the result, `10` pushed onto the (now empty) stack. Next, `2` is pushed to the stack. The final token `/` pops the two values, `10` and `2` from the stack, and divides them to give the result, `5`.

Write a program to evaluate an RPN expression consisting of space-delimited tokens (the operators `+ - * / **` and numbers).

Hint: parse the expression into a list of string tokens and iterate over it, converting and pushing the numbers to the stack (which may be implemented by appending to a `list`). Define functions to carry out the operations by retrieving values from the stack with `pop`. Note that Python does not provide a `switch...case` syntax, but these function objects can be the values in a dictionary with the operator tokens as the keys.

P4.2.4 Use the dictionary of Morse code symbols available from scipython.com/ex/adb to write a program that can translate a message to and from Morse code, using spaces to delimit individual Morse code "letters" and slashes ('/') to delimit words. For example, `'PYTHON 3'` becomes `'.-. -.- - .... -- -. / ...-'`

P4.2.5 The file `shark-species.txt`, available at scipython.com/ex/adc, contains a list of extant shark species arranged in a hierachy by order, family, genus and species (with the species given as *binomial name : common name*). Read the file into a data structure of nested dictionaries, which can be accessed as follows:

```
>>> sharks['Lamniformes']['Lamnidae']['Carcharodon']['C. carcharias']
Great white shark
```

4.3 Pythonic idioms: "syntactic sugar"

Many computer languages provide syntax to make common tasks easier and clearer to code. Such *syntactic sugar* consists of constructs that could be removed from the language without affecting the language's functionality. We have already seen one example in so-called *augmented assignment*: `a += 1` is equivalent to `a = a + 1`. Another

example is negative indexing of sequences: `b[-1]` is equivalent to and more convenient than `b[len(b)-1]`.

4.3.1 Comparison and assignment shortcuts

If more than one variable is to be assigned to the same object, the shortcut

```
x = y = z = -1
```

may be used. Note that if *mutable* objects are assigned this way, the variable names will all refer to the same object, not to distinct copies of it (recall Section 2.4.1).

Similarly, as was shown in Section 2.4.2, multiple assignments to different objects can be acheived in a single line by *tuple unpacking*:

```
a, b, c = x + 1, 'hello', -4.5
```

The tuple on the right-hand side of this expression (parentheses are optional in this case) is unpacked in the assignment to the variable names on the left-hand side. This single line is thus equivalent to the three lines

```
a = x + 1
b = 'hello'
c = -4.5
```

In expressions such as these the right-hand side is evaluated first and then assigned to the left-hand side. As we have seen in Section 2.4.2, this provides a very useful way of swapping the value of two variables without the need for a temporary variable:

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

Comparisons may also be chained together in a natural way:

```
if a == b == 3:
    print('a and b both equal 3')
if -1 < x < 1:
    print('x is between -1 and 1')
```

Python supports *conditional assignment*: a variable name can be set to one value or another depending on the outcome of an `if ... else` expression on the same line as the assignment. For example,

```
y = math.sin(x)/x if x else 1
```

Short examples such as this one, in which the potential division by zero is avoided (recall that `0` evaluates to `False`) are benign enough, but the idiom should be avoided for anything more complex in favor of a more explicit construct such as

```
try:
    y = math.sin(x)/x
except ZeroDivisionError:
    y = 1
```

4.3.2 List comprehension

A list comprehension in Python is a construct for creating a list based on another iterable object in a single line of code. For example, given a list of numbers, `xlist`, a list of the squares of those numbers may be generated as follows:

```
>>> xlist = [1, 2, 3, 4, 5, 6]
>>> x2list = [x**2 for x in xlist]
>>> x2list
[1, 4, 9, 16, 25, 36]
```

This is a faster and syntactically nicer way of creating the same list with a block of code within a `for` loop:

```
>>> x2list = []
>>> for x in xlist:
...     x2list.append(x**2)
```

List comprehensions can also contain conditional statements:

```
>>> x2list = [x**2 for x in xlist if x % 2]
>>> x2list
[1, 9, 25]
```

Here, `x` gets fed to the `x**2` expression to be entered into the `x2list` under construction only if `x % 2` evaluates to `True` (i.e., if `x` is *odd*). This is an example of a *filter* (a single `if` conditional expression). If you require a more complex *mapping* of values in the original sequence to values in the constructed list, the `if .. else` expression must appear before the `for` loop:

```
>>> [x**2 if x % 2 else x**3 for x in xlist]
[1, 8, 9, 64, 25, 216]
```

This comprehension squares the odd integers and cubes the even integers in `xlist`.

Of course, the sequence used to construct the list does not have to be another list. For example, strings, tuples and `range` objects are all iterable and can be used in list comprehensions:

```
>>> [x**3 for x in range(1,10)]
[1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> [w.upper() for w in 'abc xyz']
['A', 'B', 'C', ' ', 'X', 'Y', 'Z']
```

Finally, list comprehensions can be nested. For example, the following code flattens a list of lists:

```
>>> vlist = [[1,2,3], [4,5,6], [7,8,9]]
>>> [c for v in vlist for c in v]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Here, the first loop produces the inner lists, one by one, as `v`, and each inner list `v` is iterated over as `c` to be added to the list being created.

Example E4.9 Consider a 3×3 matrix represented by a list of lists:

```
M = [[1,2,3],
     [4,5,6],
     [7,8,9]]
```

Without using list comprehension, the transpose of this matrix could be built up by looping over the rows and columns:

```
MT = [[0,0,0], [0,0,0], [0,0,0]]
for ir in range(3):
    for ic in range(3):
        MT[ic][ir] = M[ir][ic]
```

With one list comprehension, the transpose can be constructed as

```
MT = []
for i in range(3):
    MT.append([row[i] for row in M])
```

where rows of the transposed matrix are built from the columns (indexed with `i=0,1,2`) of each row in turn from `M`). The outer loop here can be expressed as a list comprehension of its own:

```
MT = [[row[i] for row in M] for i in range(3)]
```

Note, however, that NumPy provides a much easier way to manipulate matrices (see Section 6.6).

4.3.3 `lambda` functions

A `lambda` function in Python is a type of simple *anonymous function*. The executable body of a `lambda` function must be an *expression* and not a *statement*; that is, it may *not* contain, for example, loop blocks, conditionals or `print` statements. `lambda` functions provide limited support for a programming paradigm known as *functional programming*.[6] The simplest application of a `lambda` function differs little from the way a regular function `def` would be used:

```
>>> f = lambda x: x**2 - 3*x + 2
>>> print(f(4.))
6.0
```

The argument is passed to `x` and the result of the function specified in the `lambda` definition after the colon is passed back to the caller. To pass more than one argument to a `lambda` function, pass a tuple (without parentheses):

```
>>> f = lambda x,y: x**2 + 2*x*y + y**2
>>> f(2., 3.)
25.0
```

[6] Functional programming is a style of programming in which computation is achieved through the evaluation of mathematical functions with minimal reference to variables defining the *state* of the program.

In these examples, not too much is gained by using a `lambda` function, and the functions defined are not all that anonymous either (because they've been bound to the variable name `f`). A more useful application is in creating a list of functions, as in the following example.

Example E4.10 Functions are objects (like everything else in Python) and so can be stored in lists. Without using `lambda` we would have to define named functions (using `def`) before constructing the list:

```
def const(x):
    return 1.
def lin(x):
    return x
def square(x):
    return x**2
def cube(x):
    return x**3
flist = [const, lin, square, cube]
```

Then `flist[3](5)` returns 125, since `flist[3]` is the function `cube`, and is called with the argument 5.

The value of using `lambda` expressions as anonymous functions is that if these functions do not need to be named if they are just to be stored in a list and so can be defined as items "inline" with the list construction:

```
>>> flist = [lambda x: 1,
...          lambda x: x,
...          lambda x: x**2,
...          lambda x: x**3]
>>> flist[3](5)   # flist[3] is x**3
125
>>> flist[2](4)   # flist[2] is x**2
16
```

Example E4.11 The `sorted` built-in and `sort` list method can order lists based on the returned value of a function called on each element prior to making comparisons. This function is passed as the `key` argument. For example, sorting a list of strings is case sensitive by default:

```
>>> sorted('Nobody expects the Spanish Inquisition'.split())
['Inquisition', 'Nobody', 'Spanish', 'expects', 'the']
```

We can make the sorting case insensitive, however, by passing each word to the `str.lower` method:

```
>>> sorted('Nobody expects the Spanish Inquisition'.split(), key=str.lower)
['expects', 'Inquisition', 'Nobody', 'Spanish', 'the']
```

(Of course, `key=str.upper` would work just as well.) Note that the list elements themselves are not altered: they are being ordered based on a lowercase version of

themselves. We do not use parentheses here, as in `str.lower()`, because we are passing the *function itself* to the `key` argument, not calling it directly.

It is typical to use `lambda` expressions to provide simple anonymous functions for this purpose. For example, to sort a list of atoms as (element symbol, atomic number) tuples in order of atomic number (the *second* item in each tuple):

```
>>> halogens = [('At', 85), ('Br', 35), ('Cl', 17), ('F', 9), ('I', 53)]
>>> sorted(halogens, key=lambda e: e[1])
[('F', 9), ('Cl', 17), ('Br', 35), ('I', 53), ('At', 85)]
```

Here, the sorting algorithm calls the function specified by `key` on each tuple item to decide where it belongs in the sorted list. Our anonymous function simply returns the second element of each tuple, and so sorting is by atomic number.

4.3.4 The `with` statement

The `with` statement creates a block of code that is executed within a certain *context*. A context is defined by a *context manager* that provides a pair of methods describing how to enter and leave the context. User-defined contexts are generally used only in advanced code and can be quite complex, but a common basic example of a built-in context manager involves file input / output. Here, the context is entered by opening the file. Within the context block, the file is read from or written to, and finally the file is closed on exiting the context. The `file` object is a context manager that is returned by the `open()` method. It defines an exit method which simply closes the file (if it was opened successfully), so that this does not need to be done explicitly. To open a file within a context, use

```
with open('filename') as f:
    # process the file in some way, for example:
    lines = f.readlines()
```

The reason for doing this is that you can be sure that the file will be closed after the `with` block, even if something goes wrong in this block: the context manager handles the code you would otherwise have to write to catch such runtime errors.

4.3.5 Generators

Generators are a powerful feature of the Python language; they allow one to declare a function that behaves like an iterable object. That is, a function that can be used in a `for` loop and that will yield its values, in turn, on demand. This is often more efficient than calculating and storing all of the values that will be iterated over (particularly if there will be a very large number of them). A generator function looks just like a regular Python function, but instead of exiting with a `return` value, it contains a `yield` statement which returns a value each time it is required to by the iteration.

A very simple example should make this clearer. Let's define a generator, `count`, to count to `n`:

```
>>> def count(n):
...     i=0
...     while i < n:
...         i += 1
...         yield i
...
>>> for j in count(5):
...     print(j)
...
1
2
3
4
5
```

Note that we can't simply call our generator like a regular function:

```
>>> count(5)
<generator object count at 0x102d8e6e0>
```

The generator `count` is expecting to be called as part of an loop (here, the `for` loop) and on each iteration it `yield`s its result and stores its state (the value of `i` reached) until the loop next calls upon it.

In fact, we have been using generators already because the familiar `range` built-in function is, in Python 3, a type of generator object.

There is a *generator comprehension* syntax similar to list comprehension (use round brackets instead of square brackets):

```
>>> squares = (x**2 for x in range(5))
>>> for square in squares:
...     print(square)
...
0
1
4
9
16
```

However, once we have "exhausted" our generator comprehension defined in this way, we cannot iterate over it again without redefining it. If we try:

```
>>> for square in squares:
...     print(square)
...
>>>
```

we get nothing as we have already reached the end of the `squares` generator.

To obtain a list or tuple of a generator's values, simply pass it to `list` or `tuple`, as shown in the following example.

Example E4.12 This function defines a generator for the *triangular numbers*, $T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n$, for $n = 0, 1, 2, \cdots$: that is, $T_n = 0, 1, 3, 6, 10, \cdots$.

```
>>> def triangular_numbers(n):
...     i, t = 1, 0
```

```
...     while i <= n:
...         yield t
...         t += i
...         i += 1
...
>>> list(triangular_numbers(15))
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105]
```

Note that the statements after the `yield` statement are executed each time `triangular_numbers` resumes. The call to `triangular_numbers(15)` returns an iterator that feeds these numbers into `list` to generate a list of its values.

4.3.6 ◊ map

The built-in function `map` returns an iterator that applies a given function to every item of a provided sequence, yielding the results as a generator would.[7] For example, one way to sum a list of lists is to map the `sum` built-in to it:

```
>>> mylists = [[1,2,3], [10, 20, 30], [25, 75, 100]]
>>> list(map(sum, mylists))
[6, 60, 200]
```

(We have to cast explicitly back to a `list` because `map` returns a generator-like object.) This statement is equivalent to the list comprehension:

```
>>> [sum(l) for l in mylists]
[6, 60, 200]
```

`map` is occasionally useful but has the potential to create very obscure code, and list or generator comprehensions are generally to be preferred. The same applies to the `filter` built-in, which constructs an iterator from the elements of a given sequence for which a provided function returns `True`. In the following example, the odd integers less than 10 are generated: this function returns `x % 2`, and this expression evaluates to 0, equivalent to `False` if `x` is even:

```
>>> list(filter(lambda x: x%2, range(10)))
[1, 3, 5, 7, 9]
```

Again, the list comprehension is more expressive:

```
>>> [x for x in range(10) if x % 2]
[1, 3, 5, 7, 9]
```

4.3.7 Exercises

Questions

Q4.3.1 Rewrite the list of `lambda` functions created in Example E4.10 using a single list comprehension.

[7] Constructs such as `map` are frequently used in functional programming.

Q4.3.2 What does the following code do and how does it work?

```
>>> nmax = 5
>>> x = [1]
>>> for n in range(1,nmax+2):
...     print(x)
...     x = [([0]+x)[i] + (x+[0])[i] for i in range(n+1)]
...
```

Q4.3.3 Consider the lists

```
>>> a = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
>>> b = [4, 2, 6, 1, 5, 0, 3]
```

Predict and explain the output of the following statements:

a. `[a[x] for x in b]`
b. `[a[x] for x in sorted(b)]`
c. `[a[b[x]] for x in b]`
d. `[x for (y,x) in sorted(zip(b,a))]`

Q4.3.4 Dictionaries are unsorted data structures. Write a one-line Python statement returning a list of (*key*, *value*) pairs sorted by key. Assume that all keys have the same data type (why is this important?). Repeat the exercise to produce a list ordered by dictionary *values*.

Q4.3.5 In the television series *The Wire*, drug dealers encrypt telephone numbers with a simple substitution cypher based on the standard layout of the phone keypad. Each digit of the number, with the exception of 5 and 0, is replaced with the corresponding digit on the other side of the 5 key ("jump the five"); 5 and 0 are exchanged. Thus, 555-867-5309 becomes 000-243-0751. Devise a one-line statement to encrypt and decrypt numbers encoded in this way.

Problems

P4.3.1 Use a list comprehension to calculate the *trace* of the matrix `M` (that is, the sum of its diagonal elements). *Hint:* the `sum` built-in function takes an iterable object and sums its values.

P4.3.2 The ROT13 substitution cipher encodes a string by replacing each letter with the letter 13 letters after it in the alphabet (cycling around if necessary). For example, `a` → `n` and `p` → `c`.

a. Given a word expressed as a string of lowercase characters only, use a list comprehension to construct the ROT13-encoded version of that string. *Hint:* Python has a built-in function, `ord`, which converts a character to its Unicode code point (e.g., `ord('a')` returns `97`); another built-in, `chr` is the inverse of `ord` (e.g., `chr(122)` returns `'z'`).

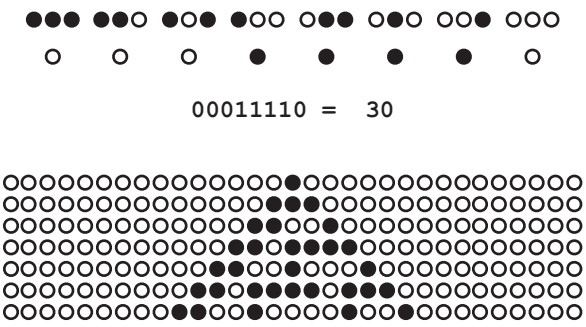

Figure 4.1 Rule 30 of Wolfram's one-dimensional two-state cellular automata and the first seven generations.

b. Extend your list comprehension to encode sentences of words (in lowercase) separated by spaces into a ROT13 sentence (in which the encoded words are also separated by spaces).

P4.3.3 In *A New Kind of Science*,[8] Stephen Wolfram describes a set of simple one-dimensional cellular automata in which each cell can take one of two values: 'on' or 'off'. A row of cells is initialized in some state (e.g., with a single 'on' cell somewhere in the row) and it evolves into a new state according to a rule that determines the subsequent state of a cell ('on' or 'off') from its value and that of its two nearest neighbors. There are $2^3 = 8$ different states for these three "parent" cells taken together and so $2^8 = 256$ different automata rules; that is, the state of cell i in the next generation is determined by the states of cells $i - 1$, i and $i + 1$ in the present generation.

These rules are numbered 0–255 according to the binary number indicated by the eight different outcomes each one specifies for the eight possible parent states. For example, rule 30 produces the outcome (off, off, off, on, on, on, on, off) (or 00011110) from the parent states given in the order shown in Figure 4.1. The evolution of the cells can be illustrated by printing the row corresponding to each generation under its parent as shown in this figure.

Write a program to display the first few rows generated by rule 30 on the command line, starting from a single 'on' cell in the center of a row 80 cells wide. Use an asterisk to indicate an 'on' cell and a space to represent an 'off' cell.

P4.3.4 The file `iban_lengths.txt`, available at scipython.com/ex/add contains two columns of data: a two-letter country code and the length of that country's International Bank Account Number (IBAN):

```
AL 28
AD 24
...
GB 22
```

[8] S. Wofram (2002). *A New Kind of Science*, Wolfram Media.

The code snippet below parses the file into a dictionary of lengths, keyed by the country code:

```
iban_lengths = {}
with open('iban_lengths.txt') as fi:
    for line in fi.readlines():
        fields = line.split()
        iban_lengths[fields[0]] = int(fields[1])
```

Use a `lambda` function and list comprehension to achieve the same goal in (a) two lines, (b) one line.

P4.3.5 The *power set* of a set S, $P(S)$, is the set of all subsets of S, including the empty set and S itself. For example,

$$P(\{1,2,3\}) = \{\{\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

Write a Python program that uses a generator to return the power set of a given set.

Hint: convert your set into an ordered sequence such as a tuple. For each item in this sequence return the power set formed from all subsequent items, inclusive and exclusive of the chosen item. Don't forget to convert the tuples back to sets after you're done.

P4.3.6 The *Brown Corpus* is a collection of 500 samples of (American) English-language text that was compiled in the 1960s for use in the field of computational linguistics. It can be dowloaded from http://nltk.github.com/nltk_data/packages/corpora/brown.zip.

Each sample in the corpus consists of words that have been tagged with their part-of-speech after a forward slash. For example,

```
        The/at football/nn opponent/nn on/in homecoming/nn is/bez ,/, of/in
course/nn ,/, selected/vbn with/in the/at view/nn that/cs
```

Here, `The` has been tagged as an article (`/at`), `football` as a noun (`/nn`) and so on. A full list of the tags is available from the accompanying manual.[9]

Write a program that analyzes the Brown corpus and returns a list of the eight-letter words which feature each possible two-letter combinations *exactly twice*. For example, the two-letter combination *pc* is present in only the words *topcoats* and *upcoming*; *mt* is present only in the words *boomtown* and *undreamt*.

4.4 Operating system services

4.4.1 The `sys` module

The `sys` module provides certain system-specific parameters and functions. Many of them are of interest only to fairly advanced users of less-common Python implementations (the details of how floating point arithmetic is implemented can vary between

[9] This manual is available at www.hit.uib.no/icame/brown/bcm.html though the tags themselves are presented better on the Wikipedia article at http://en.wikipedia.org/wiki/Brown_Corpus.

different systems, for example, but is likely to be the same on all common platforms – see Section 9.1). However, it also provides some that are useful and important: these are described here.

sys.argv

`sys.argv` holds the command line arguments passed to a Python program when it is executed. *It is a list of strings*. The first item, `sys.argv[0]`, is the name of the program itself. This allows for a degree of interactivity without having to read from configuration files or require direct user input, and means that other programs or shell scripts can call a Python program and pass it particular input values or settings. For example, a simple script to square a given number might be written:

```
# square.py
import sys

n = int(sys.argv[1])
print(n, 'squared is', n**2)
```

(Note that it is necessary to convert the input value into an `int`, because it is stored in `sys.argv` as a string.) Running this program from the command line with

```
python square.py 3
```

produces the output

```
3 squared is 9
```

as expected. But because we did not hard-code the value of `n`, the same program can be run with

```
python square.py 4
```

to produce the output

```
4 squared is 16
```

sys.exit

Calling `sys.exit` will cause a program to terminate and exit from Python. This happens "cleanly," so that any commands specified in a `try` statement's `finally` clause are executed first and any open files are closed. The optional argument to `sys.exit` can be any object; if it is an integer, it is passed to the shell which, it is assumed, knows what to do with it.[10] For example, 0 usually denotes "successful" termination of the program and nonzero values indicate some kind of error. Passing no argument or `None` is equivalent to 0. If any other object is specified as an argument to `sys.exit`, it is passed to `stderr`, Python's implementation of the standard error stream. A string, for example, appears as an *error message* on the console (unless redirected elsewhere by the shell).

[10] At least if it is in the range 0–127; undefined results could be produced for values outside this range.

Example E4.13 A common way to help users with scripts that take command line arguments is to issue a usage message if they get it wrong, as in the following code example.

Listing 4.3 Issuing a usage message for a script taking command line arguments

```
# square.py
import sys

try:
    n = int(sys.argv[1])
except (IndexError, ValueError):
    sys.exit('Please enter an integer, <n>, on the command line.\nUsage: '
             'python {:s} <n>'.format(sys.argv[0]))
print(n, 'squared is', n**2)
```

The error message here is reported and the program exits if no command line argument was specified (and hence indexing `sys.argv[1]` raises an `IndexError`) or the command line argument string does not evaluate to an integer (in which case the `int` cast will raise a `ValueError`).

```
$ python eg4-usage.py hello
Please enter an integer, <n>, on the command line.
Usage: python eg4-usage.py <n>

$ python eg4-usage.py 5
5 squared is 25
```

4.4.2 The `os` module

The `os` module provides various operating system interfaces in a platform-independent way. Its many functions and parameters are described in full in the official documentation,[11] but some of the more important are described in this section.

Process information

The Python *process* is the particular instance of the Python application that is executing your program (or providing a Python shell for interactive use). The `os` module provides a number of functions for retrieving information about the context in which the Python process is running. For example, `os.uname()` returns information about the operating system running Python and the network name of the machine running the process.

One function is of particular use: `os.getenv(`*`key`*`)` returns the value of the environment variable *`key`* if it exists (or `None` of it doesn't). Many environment variables are system specific, but commonly include

[11] http://docs.python.org/3/library/os.html.

- `HOME`: the path to the user's home directory,
- `PWD`: the current working directory,
- `USER`: the current user's username and
- `PATH`: the system path environment variable.

For example, on my system:

```
>>> os.getenv('HOME')
'/Users/christian'
```

File system commands

It is often useful to be able to navigate the system directory tree and manipulate files and directories from within a Python program. The `os` module provides the functions listed in Table 4.4 to do just this. There are, of course, inherent dangers: your Python program can do anything that your user can, including renaming and deleting files.

Pathname manipulations

The `os.path` module provides a number of useful functions for manipulating pathnames. The version of this library installed with Python will be the one appropriate for the operating system that it runs on (e.g., on a Windows machine, path name components are separated by the backslash character, '`\`', whereas on Unix and Linux systems, the (forward) slash character, '`/`' is used.

Common usage of the `os.path` module's functions are to find the filename from a path (`basename`), test to see if a file or directory exists (`exists`), join strings together to make a path (`join`), split a filename into a 'root' and an 'extension' (`splitext`) and to find the time of last modification to a file (`getmtime`). Such common applications are described briefly in Table 4.5.

Table 4.4 `os` module: file system commands

Function	Description
`os.listdir(`*`path='.'`*`)`	List the entries in the directory given by *`path`* (or the current working directory if this is not specified).
`os.remove(`*`path`*`)`	Delete the file *`path`* (raises an `OSError` if *`path`* is a directory; use `os.rmdir` instead).
`os.rename(`*`old_name, new_name`*`)`	Rename the file or directory *`old_name`* to *`new_name`*. If a file with the name *`new_name`* already exists, *it will be overwritten* (subject to user-permissions).
`os.rmdir(`*`path`*`)`	Delete the directory *`path`*. If the directory is not empty, an `OSError` is raised.
`os.mkdir(`*`path`*`)`	Create the directory named *`path`*.
`os.system(`*`command`*`)`	Execute *`command`* in a subshell. If the command generates any output, it is redirected to the interpreter standard output stream, `stdout`.

Table 4.5 `os.path` module: common pathname manipulations

Function	Description
`os.path.basename(`*`path`*`)`	Return the basename of the pathname *`path`* giving a relative or absolute path to the file: this usually means the filename.
`os.path.dirname(`*`path`*`)`	Return the directory of the pathname *`path`*.
`os.path.exists(`*`path`*`)`	Return `True` if the directory or file *`path`* exists, and `False` otherwise.
`os.path.getmtime(`*`path`*`)`	Return the time of last modification of *`path`*.
`os.path.getsize(`*`path`*`)`	Return the size of *`path`* in bytes.
`os.path.join(`*`path1, path2, ...`*`)`	Return a pathname formed by joining the path components *`path1`*, *`path2`*, etc. with the directory separator appropriate to the operating system being used.
`os.path.split(`*`path`*`)`	Split *`path`* into a directory and a filename, returned as a tuple (equivalent to calling `dirname` and `basename`) respectively.
`os.path.splitext(`*`path`*`)`	Split *`path`* into a 'root' and an 'extension' (returned as a tuple pair).

Some examples referring to a file `/home/brian/test.py`:

```
>>> os.path.basename('/home/brian/test.py')
'test.py'                    # Just the filename

>>> os.path.dirname('/home/brian/test.py')
'/home/brian'                # Just the directory

>>> os.path.split('/home/brian/test.py')
('/home/brian', 'test.py')   # Directory and filename in a tuple

>>> os.path.splitext('/home/brian/test.py')
('/home/brian/test', '.py') # File path stem and extension in a tuple

>>> os.path.join(os.getenv('HOME'), 'test.py')
'/home/brian/test.py'        # Join directories and/or filename

>>> os.path.exists('/home/brian/test.py')
False                        # File does not exist!
```

Trying to call some of these functions on a path that does not exist will cause a `FileNotFoundError` exception to be raised (which could be caught within a `try ... except` clause, of course).

Example E4.14 Suppose you have a directory of data files identified by filenames containing a date in the form `data-DD-Mon-YY.txt` where `DD` is the two-digit day number, `Mon` is the three-letter month abbreviation and `YY` is the last two digits of the year, for example `'02-Feb-10'`. The following program converts the filenames into the form `data-YYYY-MM-DD.txt` so that an alphanumeric ordering of the filenames puts them in chronological order.

Listing 4.4 Renaming data files by date

```
# eg4-osmodule.py
import os
import sys

months = ['jan', 'feb', 'mar', 'apr', 'may', 'jun',
          'jul', 'aug', 'sep', 'oct', 'nov', 'dec']

dir_name = sys.argv[1]
for filename in os.listdir(dir_name):
    # filename is expected to be in the form 'data-DD-MMM-YY.txt'
    d, month, y = int(filename[5:7]), filename[8:11], int(filename[12:14])
❶   m = months.index(month.lower())+1

    newname = 'data-20{:02d}-{:02d}-{:02d}.txt'.format(y, m, d)
    newpath = os.path.join(dir_name, newname)
    oldpath = os.path.join(dir_name, filename)
    print(oldpath, '->', newpath)
    os.rename(oldpath, newpath)
```

❶ We get the month number from the index of corresponding abbreviated month name in the list `months`, adding 1 because Python list indexes start at 0.

For example, given a directory `testdir` containing the following files:

```
data-02-Feb-10.txt
data-10-Oct-14.txt
data-22-Jun-04.txt
data-31-Dec-06.txt
```

the command `python eg4-osmodule.py testdir` produces the output

```
testdir/data-02-Feb-10.txt -> testdir/data-2010-02-02.txt
testdir/data-10-Oct-14.txt -> testdir/data-2014-10-10.txt
testdir/data-22-Jun-04.txt -> testdir/data-2004-06-22.txt
testdir/data-31-Dec-06.txt -> testdir/data-2006-12-31.txt
```

See also Problem 4.4.4 and the `datetime` module (Section 4.5.3).

4.4.3 Exercises

Problems

P4.4.1 Modify the hailstone sequence generator of Exercise P2.5.7 to generate the hailstone sequence starting at any positive integer that the user provides on the command line (use `sys.argv`). Handle the case where the user forgets to provide n or provides an invalid value for n gracefully.

P4.4.2 The *Haversine* formula gives the shortest (great-circle) distance, d, between two points on a sphere of radius R from their longitudes (λ_1, λ_2) and latitudes (ϕ_1, ϕ_2):

$$d = 2r \arcsin\left(\sqrt{\text{haversin}(\phi_2 - \phi_1) + \cos\phi_1 \cos\phi_2 \text{haversin}(\lambda_2 - \lambda_1)}\right),$$

where the *haversine* function of an angle is defined by

$$\text{haversin}(\alpha) = \sin^2\left(\frac{\alpha}{2}\right).$$

Write a program to calculate the shortest distance in km between two points on the surface of the Earth (considered as a a sphere of radius 6378.1 km) given as two command line arguments, each of which is a comma-separated pair of latitude, longitude values in degrees. For example, the distance between Paris and Rome is given by executing:

```
python greatcircle.py 48.9,2.4 41.9,12.5
1107 km
```

P4.4.3 Write a Python program to create a directory, `test`, in the user's home directory and to populate it with 20 Scalable Vector Graphics (SVG) files depicting a small, filled red circle inside a large, black, unfilled circle. For example,

```
<?xml version="1.0" encoding="utf-8"?>
                <svg xmlns="http://www.w3.org/2000/svg"
                     xmlns:xlink="http://www.w3.org/1999/xlink"
                     width="500" height="500" style="background: #ffffff">
<circle cx="250.0" cy="250.0" r="200" style="stroke: black; stroke-width: 2px;
                                                              fill: none;"/>
<circle cx="430.0" cy="250.0" r="20" style="stroke: red; fill: red;"/>
</svg>
```

Each file should move the red circle around the inside rim of the larger circle so that the 20 files together could form an animation.

One way to achieve this is to use the free ImageMagick software (www.imagemagick.org/). Ensure the SVG files are named `fig00.svg`, `fig01.svg`, etc. and issue the following command from your operating system's command line:

```
convert -delay 5 -loop 0 fig*.svg animation.gif
```

to produce an animated GIF image.

P4.4.4 Modify the program of Example E4.14 to catch the following errors and handle them gracefully:

- User does not provide a directory name on the command line (issue a usage message);
- The directory does not exist;
- The name of a file in the directory does not have the correct format;
- The filename is in the correct format but the month abbreviation is not recognized.

Your program should terminate in the first two cases and skip the file in the second two.

4.5 Modules and packages

As we have seen, Python is quite a modular language and has functionality beyond the core programming essentials (the built-in methods and data structures we have

encountered so far) that is made available to a program through the `import` statement. This statement makes reference to *modules* that are ordinary Python files containing definitions and statements. Upon encountering the line

```
import <module>
```

the Python interpreter executes the statements in the file `<module>.py` and enters the module name `<module>` into the current namespace, so that the attributes it defines are available with the "dotted syntax": `<module>.<attribute>`.

Defining your own module is as simple as placing code within a file `<module>.py`, which is somewhere the Python interpreter can find it (for small projects, usually just the same directory as the program doing the importing). Note that because of the syntax of the `import` statement, you should avoid naming your module anything that isn't a valid Python identifier (see Section 2.2.3). For example, the filename `<module>.py` should not contain a hyphen or start with a digit. Do not give your module the same name as any built-in modules (such as `math` or `random`) because these get priority when Python imports.

A Python *package* is simply a structured arrangement of modules within a directory on the file system. Packages are the natural way to organize and distribute larger Python projects. To make a package, the module files are placed in a directory, along with a file named `__init__.py`. This file is run when the package is imported and may perform some initialization and its own imports. It may be an empty file (zero bytes long) if no special initialization is required, but it must exist for the directory to be considered by Python to be a package.

For example, the NumPy package (see Chapter 6) exists as the following directory (some files and directories have been omitted for clarity):

```
numpy/
    __init__.py
    core/
    fft/
        __init__.py
        fftpack.py
        info.py
        ...
    linalg/
        __init__.py
        linalg.py
        info.py
        ...
    polynomial/
        __init__.py
        chebyshev.py
        hermite.py
        legendre.py
        ...
    random/
    version.py
    ...
```

Table 4.6 Python modules and packages

Module / Package	Description
`os`, `sys`	Operating system services, as described in Section 4.4
`math`, `cmath`	Mathematical functions, as introduced in Section 2.2.2
`random`	Random number generator (see Section 4.5.1)
`collections`	Data types for containers that extend the functionality of dictionaries, tuples, etc.
`itertools`	Tools for efficient iterators that extend the functionality of simple Python loops
`glob`	Unix-style pathname pattern expansion
`datetime`	Parsing and manipulating dates and times (see Section 4.5.3)
`fractions`	Rational number arithmetic
`re`	Regular expressions
`argparse`	Parser for command line options and arguments
`urllib`	URL (including web pages) opening, reading and parsing (see Section 4.5.2)
* Django (`django`)	A popular web application framework
* `pyparsing`	Lexical parser for simple grammars
`pdb`	The Python debugger
`logging`	Python's built-in logging module
`xml`, `lxml`	XML parsers
* VPython (`visual`)	Three-dimensional visualization
`unittest`	Unit testing framework for systematically testing and validating individual units of code (see Section 9.3.4)
* NumPy (`numpy`)	Numerical and scientific computing (described in detail in Chapter 6)
* SciPy (`scipy`)	Scientific computing algorithms (described in detail in Chapter 8)
* `matplotlib`, `pylab`	Plotting (see Chapters 3 and 7)
* SymPy (`sympy`)	Symbolic computation (computer algebra)
* `pandas`	Data manipulation and analysis with table-like data structures
* `scikit-learn`	Machine learning
* Beautiful Soup (`beautifulsoup`)	HTML parser, with handling of malformed documents

Thus, for example, `polynomial` is a subpackage of the `numpy` package containing several modules, including `legendre`, which may be imported as

```
import numpy.polynomial.legendre
```

To avoid having to use this full dotted syntax in actually referring to its attributes, it is convenient to use

```
from numpy.polynomial import legendre
```

Table 4.6 lists some of the major, freely available Python modules and packages for general programming applications as well as for numerical and scientific work. Some are installed with the core Python distribution (the *Standard Library*);[12] where

[12] A complete list of the components of the Standard Library is at https://docs.python.org/3/library/index.html.

indicated; others can be downloaded and installed separately. Before implementing your own algorithm, check that it isn't included in an existing Python package.

4.5.1 The `random` module

For simulations, modeling and some numerical algorithms it is often necessary to generate random numbers from some distribution. The topic of random-number generation is a complex and interesting one, but the important aspect for our purposes is that, in common with most other languages, Python implements a *pseudorandom number generator* (PRNG). This is an algorithm that generates a sequence of numbers that approximates the properties of "truly" random numbers. Such sequences are determined by an originating *seed* state and are always the same following the same seed: in this sense they are deterministic. This can be a good thing (so that a calculation involving random numbers can be reproduced) or a bad thing (e.g., if used for cryptography, where the random sequence must be kept secret). Any PRNG will yield a sequence that eventually repeats, and a good generator will have a long period. The PRNG implemented by Python is the *Mersenne Twister*, a well-respected and much-studied algorithm with a period of $2^{19937} - 1$ (a number with more than 6,000 digits in base 10).

Generating random numbers

The random number generator can be seeded with any *hashable* object (e.g., an *immutable* object such as an integer). When the module is first imported, it is seeded with a representation of the current system time (unless the operating system provides a better source of a random seed). The PRNG can be reseeded at any time with a call to `random.seed`.

The basic random number method is `random.random`. It generates a random number selected from the uniform distribution in the semi-open interval [0, 1) – that is, including 0 but not including 1.

```
>>> import random
>>> random.random()      # PRNG seeded 'randomly'
0.5204514767709216
>>> random.seed(42)      # Seed the PRNG with a fixed value
>>> random.random()
0.6394267984578837
>>> random.random()
0.025010755222666936
...
>>> random.seed(42)      # Reseed with the same value as before ...
>>> random.random()
0.6394267984578837       # ... and the sequence repeats.
>>> random.random()
0.025010755222666936
```

Calling `random.seed()` with no argument reseeds the PRNG with a 'random' value as when the `random` module is first imported.

To select a random floating point number, N, from a given range, $a \leq N \leq b$, use `random.uniform(a, b)`:

```
>>> random.uniform(-2., 2.)
-0.899882726523523
>>> random.uniform(-2., 2.)
-1.107157047404709
```

The `random` module has several methods for drawing random numbers from nonuniform distributions – see the documentation[13] – here we mention the most important of them.

To return a number from the normal distribution with mean `mu` and standard deviation `sigma`, use `random.normalvariate(mu, sigma)`:

```
>>> random.normalvariate(100, 15)
118.82178896586194
>>> random.normalvariate(100, 15)
97.92911405885782
```

To select a random *integer*, N, in a given range, $a \leq N \leq b$, use `random.randint(a, b)` method:

```
>>> random.randint(5, 10)
7
>>> random.randint(5, 10)
10
```

Random sequences

Sometimes you may wish to select an item at random from a sequence such as a `list`. This is what the method `random.choice` does:

```
>>> seq = [10, 5, 2, 'ni', -3.4]
>>> random.choice(seq)
-3.4
>>> random.choice(seq)
'ni'
```

Another method, `random.shuffle`, randomly shuffles (permutes) the items of the sequence *in place*:

```
>>> random.shuffle(seq)
>>> seq
[10, -3.4, 2, 'ni', 5]
```

Note that because the random permutation is made in place, the sequence must be mutable: you can't, for example, shuffle `tuples`.

Finally, to draw a `list` of k unique elements from a sequence or set (without replacement) `population`, there is `random.sample(population, k)`:

```
>>> raffle_numbers = range(1, 100001)
>>> winners = random.sample(raffle_numbers, 5)
>>> winners
[89734, 42505, 7332, 30022, 4208]
```

[13] https://docs.python.org/3/library/random.html.

The resulting list is in selection order (the first-indexed element is the first drawn) so that one could, for example, without bias declare ticket number 89734 to be the jackpot winner and the remaining four tickets second-placed winners.

Example E4.15 The *Monty Hall problem* is a famous conundrum in probability which takes the form of a hypothetical game show. The contestant is presented with three doors; behind one is a car and behind each of the other two is a goat. The contestant picks a door and then the game show host opens a different door to reveal a goat. The host knows which door conceals the car. The contestant is then invited to switch to the other closed door or stick with his or her initial choice.

Counterintuitively, the best strategy for winning the car is to switch, as demonstrated by the following simulation.

Listing 4.5 The Monty Hall problem

```
# eg4-montyhall.py
import random

def run_trial(switch_doors, ndoors=3):
    """
    Run a single trial of the Monty Hall problem, with or without switching
    after the game show host reveals a goat behind one of the unchosen doors.
    (switch_doors is True or False). The car is behind door number 1 and the
    game show host knows that. Returns True for a win, otherwise returns False.

    """

    # Pick a random door out of the ndoors available
    chosen_door = random.randint(1, ndoors)
    if switch_doors:
        # Reveal a goat
        revealed_door = 3 if chosen_door==2 else 2
        # Make the switch by choosing any other door than the initially
        # selected one and the one just opened to reveal a goat.
        available_doors = [dnum for dnum in range(1,ndoors+1)
                                if dnum not in (chosen_door, revealed_door)]
        chosen_door = random.choice(available_doors)

    # You win if you picked door number 1
❶   return chosen_door == 1

def run_trials(ntrials, switch_doors, ndoors=3):
    """
    Run ntrials iterations of the Monty Hall problem with ndoors doors, with
    and without switching (switch_doors = True or False). Returns the number
    of trials which resulted in winning the car by picking door number 1.

    """

    nwins = 0
    for i in range(ntrials):
        if run_trial(switch_doors, ndoors):
            nwins += 1
    return nwins
```

```
ndoors, ntrials = 3, 10000
nwins_without_switch = run_trials(ntrials, False, ndoors)
nwins_with_switch = run_trials(ntrials, True, ndoors)

print('Monty Hall Problem with {} doors'.format(ndoors))
print('Proportion of wins without switching: {:.4f}'
            .format(nwins_without_switch/ntrials))
print('Proportion of wins with switching: {:.4f}'
            .format(nwins_with_switch/ntrials))
```

❶ Without loss of generality, we can place the car behind door number 1, leaving the contestant initially to choose any door at random.

To make the code a little more interesting, we have allowed for a variable number of doors in the simulation (but only one car).

```
Monty Hall Problem with 3 doors
Proportion of wins without switching: 0.3334
Proportion of wins with switching: 0.6737
```

4.5.2 ◊ The `urllib` package

The `urllib` package in Python 3 is a set of modules for opening and retrieving the content referred to by uniform resource locators (URLs), typically web addresses accessed with HTTP(S) or FTP. Here we give a very brief introduction to its use.

Opening and reading URLs

To obtain the content at a URL using HTTP you first need to make an HTTP *request* by creating a `Request` object. For example,

```
import urllib.request
req = urllib.request.Request('http://www.wikipedia.org')
```

The `Request` object allows you to pass data (using GET or POST) and other information about the request (metadata passed through the HTTP headers – see later). For a simple request, however, one can simply open the URL immediately as a file-like object with `urlopen()`:

```
response = urllib.request.urlopen(req)
```

It's a good idea to catch the two main types of exception that can arise from this statement. The first type, `URLError`, results if the server doesn't exist or if there is no network connection; the second type, `HTTPError`, occurs when the server returns an error code (such as *404: Page Not Found*). These exceptions are defined in the `urllib.error` module.

```
from urllib.error import URLError, HTTPError
try:
    response = urllib.request.urlopen(req)
except HTTPError as e:
    print('The server returned error code', e.code)
except URLError as e:
```

```
    print('Failed to reach server at {} for the following reason:\n{}'
          .format(url, e.reason))
else:
    # the response came back OK
```

Assuming the `urlopen()` worked, there is often nothing more to do than simply read the content from the response:

```
content = response.read()
```

The content will be returned as a *bytestring*. To decode it into a Python (Unicode) string you need to know how it is encoded. A good resource will include the character set used in the `Content-Type` HTTP header. This can be used as follows:

```
charset = response.headers.get_content_charset()
html = content.decode(charset)
```

where `html` is now a decoded Python Unicode string. If no character set is specified in the headers returned, you may have to guess (e.g., set `charset='utf-8'`).

GET and POST requests

It is often necessary to pass data along with the URL to retrieve content from a server. For example, when submitting an HTML form from a web page, the values corresponding to the entries you have made are encoded and passed to the server according to either the GET or POST protocols.

The `urllib.parse` module allows you to encode data from a Python dictionary into a form suitable for submission to a web server. To take an example from the Wikipedia API using a GET request:

```
>>> url = 'http://wikipedia.org/w/api.php'
>>> data = {'page': 'Monty_Python', 'prop': 'text', 'action': 'parse', 'section': 0}
>>> encoded_data = urllib.parse.urlencode(data)
>>> full_url = url + '?' + encoded_data
>>> full_url
'http://wikipedia.org/w/api.php?page=Monty_Python&prop=text&action=parse
         &section=0'
>>> req = urllib.request.Request(full_url)
>>> response = urllib.request.urlopen(req)
>>> html = response.read().decode('utf-8')
```

To make a POST request, instead of appending the encoded data to the string `<url>?`, pass it to the `Request` constructor directly:

```
req = urllib.request.Request(url, encoded_data)
```

4.5.3 The `datetime` module

Python's `datetime` module provides classes for manipulating dates and times. There are many subtle issues surrounding the handling of such data (time zones,

different calendars, Daylight Saving Time etc.,) and full documentation is available online;[14] here we provide an overview of only the most common uses.

Dates

A `datetime.date` object represents a particular day, month and year in an idealized calendar (the current Gregorian calendar is assumed to be in existence for all dates, past and future). To create a `date` object, pass valid year, month and day numbers explicitly, or call the `date.today` constructor:

```
>>> from datetime import date
>>> birthday = date(2004, 11, 5)    # OK

>>> notadate = date(2005, 2, 29)    # Oops: 2005 wasn't a leap year

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: day is out of range for month

>>> today = date.today()
>>> today
datetime.date(2014, 12, 6)  # (for example)
```

Dates between 1/1/1 and 31/12/9999 are accepted. Parsing dates to and from strings is also supported (see `strptime` and `strftime`).

Some more useful `date` object methods:

```
>>> birthday.isoformat()    # ISO 8601 format: YYYY-MM-DD
'2004-11-05'

>>> birthday.weekday()      # Monday = 0, Tuesday = 1, ..., Sunday = 6
4   # (Friday)

>>> birthday.isoweekday()   # Monday = 1, Tuesday = 2, ..., Sunday = 7
5

>>> birthday.ctime()        # C-standard time output
'Fri Nov  5 00:00:00 2004'
```

`date`s can also can be compared (chronologically):

```
>>> birthday < today
True

>>> today == birthday
False
```

Times

A `datetime.time` object represents a (local) time of day to the nearest microsecond. To create a `time` object, pass the number of hours, minutes, seconds and microseconds (in that order; missing values default to zero).

[14] https://docs.python.org/3/library/datetime.html.

```
>>> from datetime import time
>>> lunchtime = time(hour=13, minute=30)
>>> lunchtime
datetime.time(13, 30)

>>> lunchtime.isoformat()       # ISO 8601 format: HH:MM:SS if no microseconds
'13:30:00'

>>> precise_time = time(4,46,36,501982)
>>> precise_time.isoformat()    # ISO 8601 format: HH:MM:SS.mmmmmm
'04:46:36.501982'

>>> witching_hour = time(24)    # Oops: hour must satisfy 0 <= hour < 24

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: hour must be in 0..23
```

`datetime` objects

A `datetime.datetime` object contains the information from both the `date` and `time` objects: `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`. As well as passing values for these quantities directly to the `datetime` constructor, the methods `today` (returning the current date) and `now` (returning the current date and time) are available:

```
>>> from datetime import datetime   # (a notoriously ugly import)
>>> now = datetime.now()
>>> now
datetime.datetime(2014, 12, 6, 12, 4, 51, 763063)

>>> now.isoformat()
'2014-12-06T12:04:51.763063'

>>> now.ctime()
'Sat Dec  6 12:04:51 2014'
```

Date and time formatting

`date`, `time` and `datetime` objects support a method, `strftime` to output their values as a string formatted according to a syntax set using the format specifiers listed in Table 4.7.

```
>>> birthday.strftime('%A, %d %B %Y')
'Friday, 05 November 2004'

>>> now.strftime('%I:%M:%S on %d/%m/%y')
'12:04:51 on 06/12/14'
```

The reverse process, parsing a string into a `datetime` object is the purpose of the `strptime` method:

```
>>> launch_time = datetime.strptime('09:32:00 July 16, 1969',
                                    '%H:%M:%S %B %d, %Y')
>>> print(launch_time)
1969-07-16 09:32:00
```

Table 4.7 `strftime` and `strptime` format specifiers. Note that many of these are locale-dependent (e.g., on a German-language system, `%A` will yield `Sonntag`, `Montag`, etc.).

Specifier	Description
`%a`	Abbreviated weekday (`Sun`, `Mon`, etc.)
`%A`	Full weekday (`Sunday`, `Monday`, etc.)
`%w`	Weekday number (0=Sunday, 1=Monday, ..., 6=Saturday).
`%d`	Zero-padded day of month: 01, 02, 03, ..., 31.
`%b`	Abbreviated month name (`Jan`, `Feb`, etc.)
`%B`	Full month name (`January`, `February`, etc.)
`%m`	Zero-padded month number: 01, 02, ..., 12.
`%y`	Year without century (two-digit, zero-padded): 01, 02, ..., 99.
`%Y`	Year with century (four-digit, zero-padded): 0001, 0002, ... 9999.
`%H`	24-hour clock hour, zero-padded: 00, 01, ..., 23.
`%I`	12-hour clock hour, zero-padded: 00, 01, ..., 12.
`%p`	AM or PM (or locale equivalent).
`%M`	Minutes (two-digit, zero-padded): 00, 01, ..., 59.
`%S`	Seconds (two-digit, zero-padded): 00, 01, ..., 59.
`%f`	Microseconds (six-digit, zero-padded): 000000, 000001, ..., 999999.
`%%`	The literal `%` sign.

```
>>> print(launch_time.strftime('%I:%M %p on %A, %d %b %Y'))
09:32 AM on Wednesday, 16 Jul 1969
```

4.6 ◊ An introduction to object-oriented programming

4.6.1 Object-oriented programming basics

Structured programming styles may be broadly divided into two categories: *procedural* and *object-oriented*. The programs we have looked at so far in this book have been *procedural* in nature: we have written functions (of the sort that would be called procedures or subroutines in other languages) that are called, passed data, and which return values from their calculations. The functions we have defined do not hold their own data or remember their state in between being called, and we haven't modified them after defining them.

An alternative programming paradigm that has gained popularity through the use of languages such as C++ and Java is *object-oriented programming*. In this context, an *object* represents a concept of some sort which holds data about itself (*attributes*) and defines functions (*methods*) for manipulating data. That manipulation may cause a change in the object's state (i.e., it may change some of the object's attributes). An object is created (*instantiated*) from a "blueprint" called a *class*, which dictates its behavior by defining its attributes and methods.

In fact, as we have already pointed out, everything in Python is an object. So, for example, a Python string is an instance of the `str` class. A `str` object possesses its

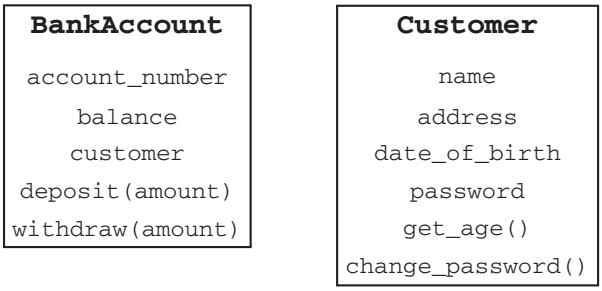

Figure 4.2 Basic classes representing a bank account and a customer.

own data (the sequence of characters making up the string) and provides (*"exposes"*) a number of methods for manipulating that data. For example, the `capitalize` method returns a new string object created from the original string by capitalizing its first letter; the `split` method returns a list of strings by splitting up the original string:

```
>>> a = 'hello, aloha, goodbye, aloha'
>>> a.capitalize()
'Hello, aloha, goodbye, aloha'
>>> a.split(',')
['hello', ' aloha', ' goodbye', ' aloha']
```

Even indexing a string is really to call the method `__getitem__`:

```
>>> b = [10, 20, 30, 40, 50]
>>> b.__getitem__(4)
50
```

That is, `a[4]` is equivalent to `a.__getitem__(4)`.[15]

Part of the popularity of object-oriented programming, at least for larger projects, stems from the way it helps conceptualize the problem that a program aims to solve. It is often possible to break a problem down into units of data and operations that it is appropriate to carry out on that data. For example, a retail bank deals with people who have bank accounts. A natural object-oriented approach to managing a bank would be to define a `BankAccount` class, with attributes such as an account number, balance and owner, and a second, `Customer` class with attributes such as a name, address, and date of birth. The `BankAccount` class might have methods for allowing (or forbidding) transactions depending on its balance and the `Customer` class might have methods for calculating the customer's age from their date of birth for example (see Figure 4.2).

An important aspect of object-oriented programming is *inheritance*. There is often a relationship between objects which takes the form of a hierarchy. Typically, a general type of object is defined by a base class, and then customized classes with more specialized functionality are derived from it. In our bank example, there may be different kinds of bank accounts: savings accounts, current (checking) accounts, etc. Each is derived from a generic base bank account, which might simply define basic attributes such as a balance and an account number. The more specialized bank account classes *inherit*

[15] The double-underscore syntax usually denotes a name with some special meaning to Python.

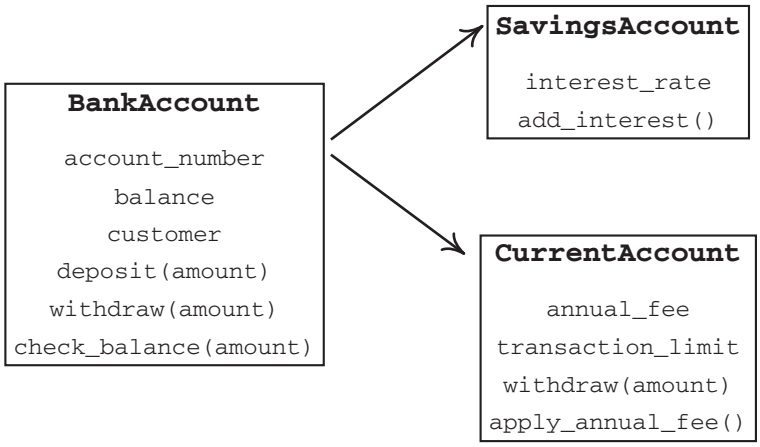

Figure 4.3 Two classes derived from an abstract base class: `SavingsAccount` and `CurrentAccount` *inherit* methods and attributes from `BankAccount` but also customize and extend its functionality.

the properties of the base class but may also customize them by overriding (redefining) one or more methods and may also add their own attributes and methods. This helps structure the program and encourages *code reuse* – there is no need to declare an account number separately for both savings and current accounts because both classes inherit one automatically from the base class. If a base class is not to be instantiated itself, but serves only as a template for the derived classes, it is called an *abstract class*.

In Figure 4.3, the relationship between the base class and two derived subclasses is depicted. The base class, `BaseAccount`, defines some attributes (`account_number`, `balance` and `customer`) and methods (such as `deposit` and `withdraw`) common to all types of account, and these are inherited by the subclasses. The subclass `SavingsAccount` adds an attribute and a method for handling interest payments on the account; the subclass `CurrentAccount` instead adds two attributes describing the annual account fee and transaction withdrawal limit, and overrides the base `withdraw` method, perhaps to check that the transaction limit has not been reached before a withdrawal is allowed.

4.6.2 Defining and using classes in Python

A class is defined using the `class` keyword and indenting the body of statements (attributes and methods) in a block following this declaration. It is conventional to give classes names written in *CamelCase*. It is a good idea to follow the `class` statement with a docstring describing what it is that the class does (see Section 2.7.1). Class methods are defined using the familiar `def` keyword, but the first argument to each

method should be a variable named `self`[16] – this name is used to refer to the object itself when it wants to call its own methods or refer to attributes, as we shall see.

In our example of a bank account, the base class could be defined as follows:

Listing 4.6 The definition of the abstract base class, `BankAccount`

```
# bank_account.py

class BankAccount:
    """ A abstract base class representing a bank account."""
    currency = '$'

    def __ init __(self, customer, account_number, balance=0):
        """
        Initialize the BankAccount class with a customer, account number
        and opening balance (which defaults to 0.)

        """

        self.customer = customer
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount):
        """ Deposit amount into the bank account."""
        if amount > 0:
            self.balance += amount
        else:
            print('Invalid deposit amount:', amount)

    def withdraw(self, amount):
        """
        Withdraw amount from the bank account, ensuring there are sufficient
        funds.

        """
        if amount > 0:
            if amount > self.balance:
                print('Insufficient funds')
            else:
                self.balance -= amount
        else:
            print('Invalid withdrawal amount:', amount)
```

To use this simple class, we can save the code defining it as `bank_account.py` and import it into a new program or the interactive Python shell with

```
from bank_account import BankAccount
```

This new program can now create `BankAccount` objects and manipulate them by calling the methods described earlier.

[16] Actually, it could be named anything, but `self` is almost universally used.

Instantiating the object

An *instance* of a class is created with the syntax `object = ClassName(args)`. You may want to require that an object instantiated from a class should initialize itself in some way (perhaps by setting attributes with appropriate values) – such initialization is carried out by the special method `__init__` which receives any arguments, `args`, specified in this statement.

In our example, an account is opened by creating a `BankAccount` object, passing the name of the account owner (customer), an account number and, optionally, an opening balance (which defaults to 0 if not provided):

```
my_account = BankAccount('Joe Bloggs', 21457288)
```

We will replace the string `customer` with a `Customer` object in Example E4.16.

Methods and attributes

The class defines two methods: one for depositing a (positive) amount of money and one for withdrawing money (if the amount to be withdrawn is both positive and not greater than the account balance).

The `BankAccount` class possesses two different kinds of attribute: `self.customer`, `self.account_number` and `self.balance` are *instance variables*: they can take different values for different objects created from the `BankAccount` class. Conversely, the variable `currency` is a *class variable*: this variable is defined inside the class but outside any of its methods and is shared by all instances of the class.

Both attributes and methods are accessed using the `object.attr` notation. For example,

```
>>> my_account.account_number     # access an attribute of my_account
21457288
>>> my_account.deposit(64)        # call a method of my_account
>>> my_account.balance
64
```

Let's add a third method, for printing the balance of the account. This must be defined inside the `class` block:

```
    def check_balance(self):
        """ Print a statement of the account balance. """
        print('The balance of account number {:d} is {:s}{:f.2}'
                .format(self.account_number, self.currency, self.balance))
```

Example E4.16 We now define the `Customer` class described in class diagram of Figure 4.2: an instance of this class will become the `customer` attribute of the `BankAccount` class. Note that it was possible to instantiate a `BankAccount` object by passing a string literal as `customer`. This is a consequence of Python's dynamic typing: no check is automatically made that the object passed as an argument to the class constructor is of any particular type.

The following code defines a `Customer` class and should be saved to a file called `customer.py`:

```
from datetime import datetime
class Customer:
    """ A class representing a bank customer. """

    def __ init __(self, name, address, date_of_birth):
        self.name = name
        self.address = address
        self.date_of_birth = datetime.strptime(date_of_birth, '%Y-%m-%d')
        self.password = '1234'

    def get_age(self):
        """ Calculates and returns the customer's age. """
        today = datetime.today()
        try:
            birthday = self.date_of_birth.replace(year=today.year)
        except ValueError:
            # birthday is 29 Feb but today's year is not a leap year
            birthday = self.date_of_birth.replace(year=today.year,
                                                  day=self.date_of_birth.day - 1)
        if birthday > today:
            return today.year - self.date_of_birth.year - 1
        return today.year - self.date_of_birth.year
```

Then we can pass `Customer` objects to our `BankAccount` constructor:

```
>>> from bank_account import BankAccount
>>> from customer import Customer
>>>
>>> customer1 = Customer('Helen Smith', '76 The Warren, Blandings, Sussex',
                         '1976-02-29')
>>> account1 = BankAccount(customer1, 21457288, 1000)
>>> account1.customer.get_age()
39
>>> print(account1.customer.address)
76 The Warren, Blandings, Sussex
```

4.6.3 Class inheritance in Python

A subclass may be derived from one or more other base classes with the syntax:

```
class SubClass(BaseClass1, BaseClass2, ...):
```

We will now define the two derived classes (or *subclasses*) illustrated in Figure 4.3 from the base `BankAccount` class. They can be defined in the same file that defines `BankAccount` or in a different Python file which imports `BankAccount`.

```
class SavingsAccount(BankAccount):
    """ A class representing a savings account. """

    def __ init __(self, customer, account_number, interest_rate, balance=0):
        """ Initialize the savings account. """
❶       self.interest_rate = interest_rate
```

```
❷          super().__ init __(customer, account_number, balance)

        def add_interest(self):
            """ Add interest to the account at the rate self.interest_rate. """

            self.balance *= (1. + self.interest_rate / 100)
```

❶ The `SavingsAccount` class adds a new attribute, `interest_rate`, and a new method, `add_interest` to its base class, and overrides the `__ init __` method to allow `interest_rate` to be set when a `SavingsAccount` is instantiated.
❷ Note that the new `__ init __` method calls the base class's `__ init __` method in order to set the other attributes: the built-in function `super` allows us to refer to the parent base class.[17] Our new `SavingsAccount` might be used as follows:

```
>>> my_savings = SavingsAccount('Matthew Walsh', 41522887, 5.5, 1000)
>>> my_savings.check_balance()
The balance of account number 41522887 is $1000
>>> my_savings.add_interest()
>>> my_savings.check_balance()
The balance of account number 41522887 is $1055.00
```

The second subclass, `CurrentAccount`, has a similar structure:

```
class CurrentAccount(BankAccount):
    """ A class representing a current (checking) account. """
    def __ init __(self, customer, account_number, annual_fee,
                   transaction_limit, balance=0):
        """ Initialize the current account. """

        self.annual_fee = annual_fee
        self.transaction_limit = transaction_limit
        super().__ init __(customer, account_number, balance)

    def withdraw(self, amount):
        """
        Withdraw amount if sufficient funds exist in the account and amount
        is less than the single transaction limit.

        """
        if amount <= 0:
            print('Invalid withdrawal amount:', amount)
            return

        if amount > self.balance:
            print('Insufficient funds')
            return

        if amount > self.transaction_limit:
            print('{0:s}{1:.2f} exceeds the single transaction limit of'
                  ' {0:s}{2:.2f}'.format(self.currency, amount,
                                         self.transaction_limit))
            return
```

[17] The built-in function `super()` called in this way creates a "proxy" object that delegates method calls to the parent class (in this case, `BankAccount`).

```
        self.balance -= amount

    def apply_annual_fee(self):
        """ Deduct the annual fee from the account balance. """

        self.balance = max(0., self.balance - self.annual_fee)
```

Note what happens if we call `withdraw` on a `CurrentAccount` object:

```
>>> my_current = CurrentAccount('Alison Wicks', 78300991, 20., 200.)
>>> my_current.withdraw(220)
Insufficient Funds

>>> my_current.deposit(750)
>>> my_current.check_balance()
The balance of account number 78300991 is $750.00

>>> my_current.withdraw(220)
$220.00 exceeds the transaction limit of $200.00
```

The `withdraw` method called is that of the `CurrentAccount` class, as this method overrides that of the same name in the base class, `BankAccount`.

Example E4.17 A simple model of a polymer in solution treats it as a sequence of randomly oriented segments; that is, one for which there is no correlation between the orientation of one segment and any other (this is the so-called *random-flight* model).

We will define a class, `Polymer`, to describe such a polymer, in which the segment positions are held in a list of `(x,y,z)` tuples. A `Polymer` object will be initialized with the values `N` and `a`, the number of segments and the segment length respectively. The initialization method calls a `make_polymer` method to populate the segment positions list.

The `Polymer` object will also calculate the end-to-end distance, R, and will implement a method `calc_Rg` to calculate and return the polymer's *radius of gyration*, defined as

$$R_{\mathrm{g}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\mathbf{r}_i - \mathbf{r}_{\mathrm{CM}})^2}$$

Listing 4.7 Polymer class

```
# polymer.py

import math
import random

class Polymer:
    """ A class representing a random-flight polymer in solution. """

    def __ init __(self, N, a):
        """
        Initialize a Polymer object with N segments, each of length a.
```

```
        """

        self.N, self.a = N, a
        # self.xyz holds the segment position vectors as tuples
        self.xyz = [(None, None, None)] * N
        # End-to-end vector
        self.R = None
        # Make our polymer by assigning segment positions
        self.make_polymer()

    def make_polymer(self):
        """
        Calculate the segment positions, center of mass and end-to-end
        distance for a random-flight polymer.

        """

        # Start our polymer off at the origin, (0,0,0).
        self.xyz[0] = x, y, z = cx, cy, cz = 0., 0., 0.
        for i in range(1, self.N):
❶           # Pick a random orientation for the next segment.
            theta = math.acos(2 * random.random() - 1)
            phi = random.random() * 2. * math.pi
            # Add on the corresponding displacement vector for this segment.
            x += self.a * math.sin(theta) * math.cos(phi)
            y += self.a * math.sin(theta) * math.sin(phi)
            z += self.a * math.cos(theta)
            # Store it, and update or center of mass sum.
            self.xyz[i] = x, y, z
            cx, cy, cz = cx + x, cy + y, cz + z
❷       # Calculate the position of the center of mass.
        cx, cy, cz = cx / self.N, cy / self.N, cz / self.N
        # The end-to-end vector is the position of the last
        # segment, since we started at the origin.
        self.R = x, y, z

        # Finally, re-center our polymer on the center of mass.
        for i in range(self.N):
            self.xyz[i] = self.xyz[i][0]-cx,self.xyz[i][1]-cy,self.xyz[i][2]-cz
    def calc_Rg(self):
        """
        Calculates and returns the radius of gyration, Rg. The polymer
        segment positions are already given relative to the center of
        mass, so this is just the rms position of the segments.

        """

        self.Rg = 0.
        for x,y,z in self.xyz:
            self.Rg += x**2 + y**2 + z**2
        self.Rg = math.sqrt(self.Rg / self.N)
        return self.Rg
```

❶ One way to pick the location of the next segment is to pick a random point on the surface of the unit sphere and use the corresponding pair of angles in the spherical polar coordinate system, θ and ϕ (where $0 \leq \theta < \pi$ and $0 \leq \phi < 2\pi$) to set the displacement

from the previous segment's position as

$$\Delta x = a \sin\theta \cos\phi$$
$$\Delta y = a \sin\theta \sin\phi$$
$$\Delta z = a \cos\theta$$

❷ We calculate the position of the polymer's center of mass, $\mathbf{r}_{CM}$, and then shift the origin of the polymer's segment coordinates so that they are measured relative to this point (that is, the segment coordinates have their origin at the polymer center of mass).

We can test the `Polymer` class by importing it in the Python shell:

```
>>> from polymer import Polymer
>>> polymer = Polymer(1000, 0.5)   # A polymer with 1000 segments of length 0.5
>>> polymer.R           # End-to-end vector
(5.631332375722011, 9.408046667059947, -1.3047608473668109)
>>> polymer.calc_Rg()   # Radius of gyration
5.183761585363432
```

Let's now compare the distribition of the end-to-end distances with the theoretically predicted probability density function:

$$P(R) = 4\pi R^2 \left(\frac{3}{2\pi\langle r^2\rangle}\right)^{3/2} \exp\left(-\frac{3R^2}{2\langle r^2\rangle}\right),$$

where the mean square position of the segments is $\langle r^2\rangle = Na^2$

Listing 4.8 The distribution of random flight polymers

```
# eg4-c-ii-polymer-a.py
# Compare the observed distribution of end-to-end distances for Np random-
# flight polymers with the predicted probability distribution function.

import pylab
from polymer import Polymer
pi = pylab.pi

# Calculate R for Np polymers
Np = 3000
# Each polymer consists of N segments of length a
N, a = 1000, 1.
R = [None] * Np
for i in range(Np):
    polymer = Polymer(N, a)
    Rx, Ry, Rz = polymer.R
    R[i] = pylab.sqrt(Rx**2 + Ry**2 + Rz**2)
    # Output a progress indicator every 100 polymers
    if not (i+1) % 100:
        print(i+1, '/', Np)

# Plot the distribution of Rx as a normalized histogram
# using 50 bins
pylab.hist(R, 50, normed=1)

# Plot the theoretical probability distribution, Pr, as a function of r
r = pylab.linspace(0,200,1000)
```

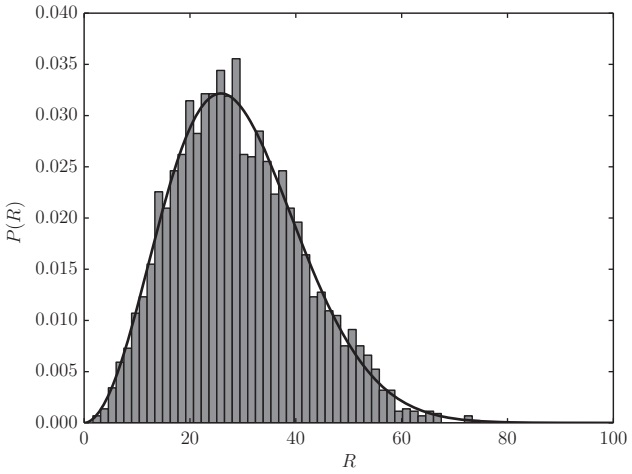

Figure 4.4 Distribution of the end-to-end distances, R, of random flight-polymers with $N = 1,000, a = 1$.

```
msr = N * a**2
Pr = 4.*pi*r**2 * (2 * pi * msr / 3)**-1.5 * pylab.exp(-3*r**2 / 2 / msr)
pylab.plot(r, Pr, lw=2, c='r')
pylab.xlabel('R')
pylab.ylabel('P(R)')
pylab.show()
```

The earlier mentioned program produces a plot that typically looks like Figure 4.4, suggesting agreement with theory.

4.6.4 Exercises

Problems

P4.6.1 a. Modify the base `BankAccount` class to verify that the account number passed to its `__init__` constructor conforms to the Luhn algorithm described in Exercise P2.5.3.

b. Modify the `CurrentAccount` class to implement a free overdraft. The limit should be set in the `__init__` constructor; withdrawals should be allowed to within the limit.

P4.6.2 Add a method, `save_svg` to the `Polymer` class of Example E4.17 to save an image of its polymer as an SVG file. Refer to Exercise P4.4.3 for a template of an SVG file.

P4.6.3 Write a program to create an image of a constellation using the data from the Yale Bright Star Catalog (http://tdc-www.harvard.edu/catalogs/bsc5.html).

Create a class, `Star`, to represent a star with attributes for its name, magnitude and position in the sky, parsed from the file `bsc5.dat` which forms part of the catalog. Implement a method for this class which converts the star's position on the celestial sphere as (Right Ascension: α, Declination: δ) to a point in a plane, (x, y), for example using the orthographic projection about a central point (α_0, δ_0):

$$\begin{aligned}\Delta\alpha &= \alpha - \alpha_0\\ x &= \cos\delta \sin\Delta\alpha\\ y &= \sin\delta\cos\delta_0 - \cos\delta\cos\Delta\alpha\sin\delta_0\end{aligned}$$

Suitably scaled projected, star positions can be output to an SVG image as `circles` (with a larger radius for brighter stars). For example, the line

```
<circle cx="200" cy="150" r="5" stroke="none" fill="#ffffff"/>
```

represents a white circle of radius 5 pixels, center on the canvas at (200, 150).

Hint: you will need to convert the right ascension from (hr, min, sec) and the declination from (deg, min, sec) to radians. Use the data corresponding to "equinox J2000, epoch 2000.0" in each line of `bsc5.dat`. Let the user select the constellation from the command line using its three-letter abbreviation (e.g., 'Ori' for Orion): this is given as part of the star name in the catalog. Don't forget that star magnitudes are *smaller* for *brighter* stars. If you are using the orthographic projection suggested, choose (α_0, δ_0) to be the mean of (α, δ) for stars in the constellation.

P4.6.4 Design and implement a class, `Experiment`, to read in and store a simple series of (x, y) data as `pylab` (i.e., NumPy) arrays from a text file. Include in your class methods for transforming the data series by some simple function (e.g., $x' = \ln x$, $y' = 1/y$) and to perform a linear leastsquares regression on the transformed data (returning the gradient and intercept of the best-fit line, $y'_{\text{fit}} = mx' + c$). NumPy provides methods for performing linear regression (see Section 6.5.3), but for this exercise the following equations can be implemented directly:

$$\begin{aligned}m &= \frac{\overline{xy} - \bar{x}\bar{y}}{\overline{x^2} - \bar{x}^2}\\ c &= \bar{y} - m\bar{x}\end{aligned}$$

where the bar notation, $\bar{\cdot}$, denotes the arithmetic mean of the quantity under it. (*Hint:* use `pylab.mean(arr)` to return the mean of array `arr`.)

Chloroacetic acid is an important compound in the synthetic production of phamaceuticals, pesticides and fuels. At high concentration under strong alkaline conditions its hydrolysis may be considered as the following reaction:

$$\mathrm{ClCH_2COO^-} + \mathrm{OH^-} \rightleftharpoons \mathrm{HOCH_2COO^-} + \mathrm{Cl^-}.$$

Data giving the concentration of $\mathrm{ClCH_2COO^-}$, c (in M), as a function of time, t (in s), are provided for this reaction carried out in excess alkalai at five different temperatures in the data files `caa-`T`.txt` (T = `40`, `50`, `60`, `70`, `80` in °C): these may be obtained

from scipython.com/ex/ade. The reaction is known to be second order and so obeys the integrated rate law

$$\frac{1}{c} = \frac{1}{c_0} + kt$$

where k is the effective rate constant and c_0 the initial ($t = 0$) concentration of chloroacetic acid.

Use your `Experiment` class to interpret these data by linear regression of $1/c$ against t, determining $m(\equiv k)$ for each temperature. Then, for each value of k, determine the activation energy of the reaction through a second linear regression of $\ln k$ against $1/T$ in accordance with the Arrhenius law:

$$k = Ae^{-E_a/RT} \quad \Rightarrow \quad \ln k = \ln A - \frac{E_a}{RT},$$

where $R = 8.314\ \mathrm{J\,K^{-1}\,mol^{-1}}$ is the gas constant. Note: the temperature must be in Kelvin.