Advanced Python

W. H. K. Bester

Scientific Computing 372
Stellenbosch University

Last updated on Wednesday 12th August, 2015 at 23:33.

Contents

1 Object-oriented programming **1**

1.1 Classes and objects 1
1.2 Abstraction 2
1.3 The anatomy of a class 2
1.4 Creating objects 3
1.5 Types in Python 5
1.5.1 Function objects 5
1.5.2 Duck typing 7
1.6 Operator overloading 10
1.7 Methods for operator overloading 13
1.8 Exercises 16

2 Data structures in Python **20**

2.1 List comprehensions 20
2.2 Sets 22
2.2.1 Set creation 23
2.2.2 Set operations 25
2.3 Dictionaries 27
2.3.1 Hashing 27
2.3.2 Dictionary operations 28
2.3.3 Dictionary methods 30

3 Exceptions **32**

3.1 Handling exceptions 32
3.2 Exception objects and handlers 33
3.3 Raising exceptions 36

CHAPTER 1

Object-oriented programming

A *data type* is a set of values and a set of operations defined on those values. For example, when we perform integral arithemetic, we use Python's built-in `int` data type, which allows integers of arbitrary magnitude, and for which Python defines the usual arithmetic operators. The built-in `float` and `complex` types are other examples of numeric data types, as is the `array` type that allows vectorisation in Numpy.

Data types are not limited to numeric types. Just like the definitions of abstract algebraic structures—such as groups, rings, and fields—are valid for any set of objects for which the required axioms hold, data types can be defined for values other than numbers. The string data type `str` in Python, for example, stores and operates on sequences of characters. Some of the operations, such as contatenation and slicing, are available via special syntax, while others like matching and capitalisation are accomplished by invoking string methods.

1.1 Classes and objects

Many modern languages also allow us to define our own data types, in essence allowing us to extend the language, that is, to add functionality. The mechanism through which this is accomplished in Python is called *object orientation*; the paradigm of programming this way is called *object-oriented programming* (OOP). In languages that are object-oriented, we write *classes* to define the allowable values and operations for a new data type. Then, we may create *instances* of these classes, called *objects*, and each object then represents a value of the data type defined by such a class.

When we write a class to create a new data type, we must use existing types, either those built into Python or provided by third-party Python modules, to store values for the new data type. That is to say, we must decide on a representation for the new data type, and this representation can only be built up from what we already have available.

At the most basic level, all data is encoded numerically. When you listen to a digital music file or watch a video file, you are using applications that understand and can translate to output devices the sequence of bits in the file. Such sequences are

typically treated as having mathematical or numeric meaning, for example, specifying colour component values for pixels or the waveforms of sounds.

Fortunately, we do not go into that much detail in this course. We are mostly interested in data types that *aggregate* more elementary types, that is, which model entities that can readily be represented by a set of values of elementary types. For example, a student record on the university systems can be composed of strings (for your name, address, and so on) and numbers (for your marks and, no doubt to somebody's chagrin, your student fees).

1.2 Abstraction

When we use a simplified description of something that captures its essential elements, it is called *abstraction*. The abstractions that we deal with when designing data types concern (i) *state* to represent the values and (ii) *behaviour* to implement the operations. To keep state, classes have *data fields* (or *data members*) that store values and are actually just variables; to add behaviour, classes have *methods* (or *function members*) that can operate on the fields and are actually just functions with a special syntax and an implied parameter.

Note that we do not use the terms *class* and *object* interchangably. A *class* refers to the mechanism of abstraction whereby we define new data types, whereas an *object* refers to a particular data type value, also called an *instance* of the class.

1.3 The anatomy of a class

Figure 1.1 contains Python source code for a class that models a point in two-dimensional Euclidean space. The class defines three methods, all of which has the name `self` as first parameter. Whenever a method (as opposed to a normal function) is called, Python automatically and implicitly passes, as first parameter parameter to that method, a reference to the object on which the method was called. By convention, this object reference is named `self`. The implicit parameter means that a method call always has at least one fewer argument than specified by the method's parameter list.

Note that two of the methods, `__init__` and `__str__`, are named according to the double underscore convention, which indicates that the Python environment attaches special meaning to them.[*] Indeed, the method `__init__` is called whenever a new object is instantiated and is called a *constructor*; its job is to *initialise* the object, that is, to set up the object's initial state. Likewise, `__str__` is called by the built-in function `str`, which is used when a string represention of an object is wanted.

Unlike many other languages, variables in Python need not be defined before they are used: As soon as a value is successfully assigned to a name, that name becomes a variable. Similary, data fields in an object are simply created by assigning to a name.

[*] Since double underscores appear so frequently in Python, and five syllables are a lot to say, it is now customary to call this character sequence "dunder". So, `__init__` is read as "dunder init dunder".

```
import math

class Point:
    """A point in two-dimensional Euclidean space."""

    def __init__(self, x, y):
        """Initialise a new Point with coordinates (x, y)."""
        self.x = x
        self.y = y

    def distance_to(self, p):
        """Return the distance from this Point to Point p."""
        dx = self.x - p.x
        dy = self.y - p.y
        return math.sqrt(dx*dx + dy*dy)

    def __str__(self):
        """x.__str__() <==> str(x)"""
        return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

FIGURE 1.1: A class for a data type `Point` that can store and operate points in two-dimensional Euclidean space.

So, the two statements of `__init__` set up two data fields, `x` and `y`, with their values taken from the corresponding parameters. Note that data fields are qualified with the `self` reference. This is necessary, because an unqualified name will (in most cases) simply be seen as a local variable of the method.

Local variables, such as `dx` and `dy` in the method `distance_to`, function just like local variables in functions. As soon as a method returns, they fall out of scope and cannot be used from whichever function, method, or module called the method.

1.4 Creating objects

To use the class in Figure 1.1, the source code must first be saved in a file, in the same way as we would be a Python module. If we save this class to `point.py`, we can test our new data type as follows.

```
>>> import point
>>> p1 = point.Point(0, 0)
>>> p2 = point.Point(1, 1)
>>> print p2
(1, 1)
>>> p1.distance_to(p2)
1.4142135623730951
```

To create a new object, we simply call the class name like we would a function. We use the parameters for the `__init__` method, remembering that the `self` reference

is passed implicity, so that we do not have to specify it.

In the example, the class name `Point` is qualified by the module name `point`. This is necessary since the class was defined inside that module. If we were to write `from point import Point`, it would not be necessary to qualify the class name. However, the usual terms and conditions for such imports apply, so be careful of naming conflicts. Note also that, as soon as the individual points are instantiated, we do not qualify method calls with anything other than the name of the particular object.

We are now in a position to understand slightly better how Python's class mechanism functions. As soon as the Python interpreter has finished reading a class definition, like the one given in Figure 1.1, a *class object* is created, named by the identifier after the `class` keyword. A class object supports two operations: (i) instantiation and (ii) attribute references[†]. An object, that is, the object that is returned by instantiation of the class object, only understands attribute references.

EXAMPLE 1.4.1 To understand how it all fits together, consider the following interpreter session.

```
>>> from point import Point
>>> p = Point
>>> type(p)
<type 'classobj'>
>>> p1 = Point(0, 0)
>>> type(p1)
<type 'instance'>
>>> p2 = p(1, 1)
>>> type(p2)
<type 'instance'>
>>> Point.distance_to(p1, p2)
1.4142135623730951
>>> p1.distance_to(p2)
1.4142135623730951
>>> p2.y
1
```

The first assignment assigns the class object itself to the identifier p; therefore `type` reports `p` as a "`classobj`". To trigger instantiation, we must use function notation on the class name. The parentheses in the assignment to `p1` causes the class object to call its `__init__` method, returning a new class instance. This is why `type` reports `p1` as an "`instance`". Since `p` is an *alias* to the class object, the assignment to `p2` also creates a new instance. This is not entirely surprising, since we have seen how the statement `import numpy as np` allows us to write `np` as module name instead of `numpy`.

The class object also supports attribute references. The name `distance_to` is a function attribute of the class object `Point`, and can be called directly. But Python

[†] An *attribute* is any name after a dot. For example, in the case of `point.Point`, the class name `Point` is an attribute of the module name `point`.

also allows access to `distance_to` as a method attribute of the object instance. In the latter case, like in the statement `p1.distance_to(p2)`, the object instance is passed as the first argument of `distance_to`. Finally, the constructor associates `x` and `y` attributes with the object instance via the `self` reference, which explains why we can access `p2`'s y-coordinate as `p2.y`. ■

1.5 Types in Python

We now pause for a slight detour to two language features Python does not share with older languages such as C and Java. Before doing so, however, we first briefly consider the broader context.

In general, each programming language enforces some kind *type system*. This does not only dictate which data types are built into the language and how aggregeda data types can be constructed, but also how the types are managed by the language run-time system. In particular, a type system determines how values are represented in memory as bits (zeroes and ones), how different data types can be mixed (possibly in the same expression), and what kind of type information is necessary when a program is compiled or run.

Typically, we distinguished between *static typing* and *dynamic typing*. Languagues such as C and Java are statically typed. In them, we must explicitly write in a program what kind of type each variable has, that is, what kind of values we want to store in this variable. This ensures that types can be checked when a program is compiled, with some (in the case of Java) or no (in the case of C) run-time checks on the data type.

Python, on the other hand, is dynamically typed. When we create a create a variable by assignment of a value, Python itself determines the type of the value, which then automatically becomes the type of the variable. When necessary, Python also enforces the types at run time. For example, the interpreter will not allow us to use the + operator where one operand is a string and the other is a number.

Although there is some dispute about the term, the previous example is evidence that Python is also *strongly typed*. In languages with "weaker" typing, the results of statements like EX `x = "1" + 1` are determined by extra semantic rules for the language. For example, the previous statement, in JavaScript, would be equivalent to EX `x = "1" + "1"`, so that `x` stores the string "`11`". But in the language Perl, the original statement would be equivalent to EX `x = 1 + 1`, so that `x` stores the number 2.

1.5.1 Function objects

Unlike in, say, C and Java, functions in Python are *first-class* objects. This means that they can be (i) assigned to variables, (ii) passed as arguments to other functions, and (iii) returned as values from other functions. If you have never programmed in language where functions are not first-class objects, this property might seem

```
def twice(f, x): return f(f(x))

def square(x): return x ** 2

def cube(x): return x ** 3

print twice(square, 3)
print twice(cube, 3)
```

FIGURE 1.2: An illustration of passing a function as argument.

innocuous. In Python, however, this allows *higher-order functions*, and in particular, the *functional programming* as programming paradigm. Functional programming is a difficult beast for the uninitiated to master, and an introduction to this style of programming is not the intention here. Still, we should look at function objects, because they are useful for some common tasks.

EXAMPLE 1.5.1 Consider the Python code in Figure 1.2. When saved to a file and run from the command line, this code prints the values 81 and 19 683. The function `twice` treats its first parameter as a function, and the value it returns is this function applied twice to the `x` parameter. When the function `square` and the integer 3 is passed to `twice`, mathematically speaking, it returns $f(f(3)) = f(3^2) = (3^2)^2 = 3^4 = 81$. Similarly, when passed the function cube and the integer 3, the result is $f(f(3)) = f(3^3) = (3^3)^3 = 3^9 = 19\,683$. ■

The astute reader might at this point be wondering about the data type of `twice`'s result: Is it a function or an integer? The `type` function reports it is an integer. But is it possible to do something akin to function composition, and then to return a function? The short answer is: Yes, it is possible. We have to use a definition like `def twice(f): return lambda x : f(f(x))`. A call to this new version of `twice` results in a function as return value. Therefore, we can call it as `twice(square)(3)`.

Exactly what a lambda expression is, and how it functions precisely, is outside the scope of this document. Suffice it to say that the `lambda` keyword creates an *anonymous function*, that is, the resulting function object is not bound to a name during creation. A lambda also does not use a `return` statement; instead, it includes an expression that is "returned". Note the result of a lambda expression is a function object, and not the evaluation of this function object on some argument values.

EXAMPLE 1.5.2 Lambda expressions are useful in various contexts. Consider the following example, taken from the *Python Sorting Mini-HOWTO* [4]:

```
>>> students = [ ('john', 'A', 15),
...              ('jane', 'B', 12),
...              ('dave', 'B', 10)  ]
>>> sorted(students)
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"

parrot(1000)                                       # 1 positional argument
parrot(voltage=1000)                               # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')          # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)          # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')      # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

FIGURE 1.3: A Python example that illustrates default argument values and keyword arguments [5, §4.7.2].

```
>>> sorted(students, key=lambda student: student[2])
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Here, `students` is a list of tuples, where each tuple gives the name, grade, and age of a particular student. The built-in function `sorted`, left to its own devices, uses the first element of each tuple as sort key. This behaviour can, however, be changed by using the *keyword argument* `key`, which must be a function object (or the name of a function object) that, when called on an element in the collection to be sorted, returns the value that is the sort key for this element. The second call to `sorted` has a lambda expression as key. It "returns" the value at index 2 for each tuple, so that the `sorted` call results in a list where the student tuples are sorted by age. ■

If you have not come across keyword arguments before, refer to the *Python Tutorial* [5, §4.7]. The short version is that Python functions (and therefore, also methods) can have a variable number of parameters. We make a parameter optional by a supplying a default argument.

EXAMPLE 1.5.3 The definition of the function `parrot` in Figure 1.3 illustrates how parameters can be made option by supplying default arguments. The function calls, in turn, illustrate the different ways that we can call `parrot` with respect to specifying arguments. ■

Incidentally, if you have worked in Java before: Although Python supports operator overloading (see §1.6), it does not support method overloading, that is, in Python we cannot have two different functions with the same name unless they are in different name spaces, for example, unless they are in different modules. But, because Python supports keyword arguments, method overloading is actually unnecessary.

1.5.2 Duck typing

Another aspect that distinguishes Python from languages like C and Java is its use of *duck typing*. In Java, the language nudges the programmer to design by interface. An

interface is an agree upon way to interact, and in Java, this means formally specifying the set of methods an object must to have to be considered as being of a particular type. The Java compiler can then ensure statically—that is, at compile time—that all the required methods are available in a particular class.

Python, however, does not at an object's type to determine whether it has the required interface. Instead, we check whether Python object has a particular attribute, and if it does, we simply use or call that attribute. Although there is no canonical definition, this is called *duck typing*. The name is said to have been inspired by a remark from the American poet James Whitcomb Riley:

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

Actually, things are slightly more complex, as they tend to be. There is one aspect of object-oriented programming that we have simply glossed over, and about which we will not have much to say after the current section. One of the original "selling points" of object orientation is that classes, in many cases, naturally suggests a hierarchy; also see Figure 3.1.

EXAMPLE 1.5.4 With some apologies to the biologists, suppose that we want to represent a collection of mammals hierarchically. To be classified as a mammal, an animal needs certain distinguishing features: (i) hair, (ii) three middle ear bones, (iii) mammary glands, and (iv) a neocortex. Humans, whales, and platypi are all examples (instances) of mammals. As mammals they share the characteristics given above, yet as species of mammals, there are some characteristics they do not share with one another. The platypus, for example and just in case you do not know, is the only known mammal that lays eggs instead of giving birth. This all leads us to the idea of drawing a "family tree" for all mammals. At the root of the tree, we collect all the characteristics shared by all mammals. But as we move towards the leaves, we add to each vertex in the tree (that is, each order, family, genus, species, etc.) those characteristics that define the particular vertex. ■

In classical object orientation, this hierarchy works by allowing a particular class to *inherit* from other classes. Some attribute—a variable or a method—that is available in a class is automatically available in any subclass. When we set up a class so that it inherits from another, we say that we *extend* the latter.

This idea is not without trouble, however. Certain things naturally lead to a hierarchical organisation—such as animals, plants, and different vehicles—but other things do not. Think, for example, of rational numbers. Sure, they are numbers, and as such, they must have certain properties that they share with all other things we call numbers. But when we think of the definition of the set of rational numbers, as in Eq. (1.1), a different way of thinking about representing presents itself: We do not think of a rational number as a specific kind of integer. Rather, a rational needs to be defined in terms of two integers. So, if we build a class to represent rational numbers, each instance must include two integers, for the numerator and denominator, respectively; see §1.6.

```
class Mammal:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return self.name

class Dog(Mammal):
    def bark(self):
        print "Woof, woof!"

class Duck(Mammal):
    def quack(self):
        print "Quack, quack!"

class Human(Mammal):
    def quack(self):
        print "'Quack, quack,' I say."
    def talk(self):
        print "I am sesquipedalianly loquacious."

mammals = [Dog('Pluto'), Duck('Donald'), Human('Sheldon')]
for mammal in mammals:
    if hasattr(mammal, 'quack'):
        print mammal, 'must be a duck:',
        mammal.quack()
    else:
        print mammal, 'is not a duck.'
```

FIGURE 1.4: Python code that illustrates duck typing and class inheritance.

This way of thinking about object orientation is called *composition*. Instead of inheriting from other classes, an instance of a class is composed of instances of other classes. Which way, inheritance or composition, is better, is open to some debate. However, some computer scientists [1] have argued strongly that inheritance creates more problems than it solves, and that composition is, in software engineering terms, a safer and better option.[‡]

EXAMPLE 1.5.5 Consider the Python code in Figure 1.4. If we save this to a file and run it from the command line, the following is produced as output:

```
Pluto is not a duck.
Donald must be a duck: Quack, quack!
Sheldon must be a duck: 'Quack, quack,' I say.
```

So, let's try to understand what is going on.

[‡] As a colleague of mine used to remark: "When I take over this company, I am going to create a new department called *Problem Creating and Solution Solving*, because somebody needs to creates problems for all these 'solutions' the consultants keep selling us."

First we define a class called `Mammal`. For our purposes—and don't tell the biologists!—a mammal is something that we can give a name. So, the constructor for `Mammal` has a `name` parameter. The `__repr__` method allows the Python interpreter to display a suitable string when we write something like `print mammal`, where `mammal` is an instance of `Mammal`.

Next, note that each of the class definitions for `Dog`, `Duck`, and `Human` has `Mammal` included in parentheses. This is how we tell Python that that particular class extends the class of which the name appears in the parentheses. It means that each of `Dog`, `Duck`, and `Mammal` has (i) a constructor that takes a name as parameter, and (ii) that this name will be output by the Python interpreter when an instance is printed out. We don't need to duplicate this functionality in the subclasses, since they inherit this behaviour from the superclass: Each `Dog`, `Duck`, and `Human` automatically has the same `__init__` and `__repr__` method as `Mammal`.

The output of the code in Figure 1.4 can be understood in terms of duck typing. The function `hasattr` returns `True` if its first parameter, any Python object, contains the attribute by the name of that given as the second (string) parameter. Since Pluto, as a `Dog` instance, does not include a `quack` attribute, Python determines it is not a duck. But Sheldon has a `quack` method, so even though he is not a `Duck` instance, Python identifies and treats him as a duck. ■

Python does provide a way of checking whether a given object is an instance of a particular class. If we change the `if` condition in Figure 1.4 to `isinstance(mammal, Duck)`, Python will display:

```
Pluto is not a duck.
Donald must be a duck: Quack, quack!
Sheldon is not a duck.
```

Doing this, however, is counter to the Python way of doing: If an object is capable of pretending to be duck, why not treat it as such? In essence, Python trusts that each class will do what is right. Languages like Java, on the other hand, enforces an object's type (as opposed to its interface) rigorously. Whichever is best is, of course, a question of philosophy.

1.6 Operator overloading

We have already seen that Python allows *operator overloading*: The meaning of an operator depends of the type of its operands. For example, the + operator performs addition on numeric operands, but concatenation on string operands.

Python's object orientation mechanism allows us to overload operators for the new data types that we create as classes. Doing so is a double-edged sword: On the one hand, it allow us to write expressions for the new data type in a natural and concise way, especially if our types are mathematical by nature; on the other hand, it is easy to make mistakes that will crash a program.

```
def gcd(a, b):
    """Return the greatest common divisor of a and b."""
    while b: a, b = b, a%b
    return a

class Rational:
    """A rational number."""

    def __init__(self, numerator, denominator):
       """Initialise a new Rational with the specified numerator and
        denominator."""
        self.a = int(numerator)
        self.b = int(denominator)

        if self.b == 0:
            raise ValueError('zero denominator')

        g = gcd(self.a, self.b)
        self.a /= g
        self.b /= g

    def __str__(self):
        """x.__str__() <==> str(x)"""
        return str(self.a) + '/' + str(self.b)

    def __add__(self, y):
        """x.__add__(y) <==> x+y"""
        return Rational(self.a*y.b + self.b*y.a, self.b*y.b)

    def __float__(self):
        """x.__float__() <==> float(x)"""
        return float(self.a)/float(self.b)

    def __mul__(self, y):
        """x.__mul__(y) <==> x*y"""
        return Rational(self.a*y.a, self.b*y.b)
```

FIGURE 1.5: A class for a data type `Rational` that can store and operate on rational numbers.

Figure 1.5 contains Python source code for a class that defines a data type for rational numbers. That is, we can represent numbers in $\mathbb{Q}$, which is defined as

$$\mathbb{Q} = \left\{ \frac{a}{b} \;\middle|\; a, b \in \mathbb{Z}, b \neq 0 \right\}. \tag{1.1}$$

For two rational numbers p and q, where $q = a/b$, $p = c/d$, and $a, b, c, d \in \mathbb{Z}$, we have well-defined notions of addition and multiplication:

$$p + q = \frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}, \text{ and} \tag{1.2}$$

$$pq = \left(\frac{a}{b}\right)\left(\frac{c}{d}\right) = \frac{ac}{bd}. \tag{1.3}$$

Eqs. (1.2) and (1.3) can now be used to overload Python's `+` and `*` operators so that they can add and multiply, respectively, two `Rational` values.

The `+` operator is overloaded by the `__add__` method, and the `*` operator is overloaded by the `__mul__` method. In Figure 1.5, we used Eq. (1.2) to implement addition in `__add__`, and Eq. (1.3) to implement multiplication in `__mul__`. Note that each of these methods (i) expects the parameter `y`, the second operand of the operator, to be another `Rational` object, and (ii) returns a new `Rational` object, created by calling the constructor of the `Rational` class. Of course, we first could have assigned the expressions for Eqs. (1.2) and (1.3) to local variables, and then have passed along these variables as arguments to the `Rational` constructor, but doing so is no more clear, and some conciseness is lost.

The docstrings for these functions also suggest what happens when the Python interpreter encounters an expression with an overloaded operator and the new data type as operands: If `p` is a `Rational` object and `q` is any other object, the Python interpreter converts the expression `p*q` into the method call `p.__mul__(q)`, which, in turn, is seen by the class as a call to `__mul__` with `p` and `q` as arguments. So, in this case, the parameter `self` refers to `p` and `y` refers to `q`, which are then used to complete the multiplication and return a new `Rational` instance. The case for addition is similar.

Note that the constructor causes the numerator and denominator to be stored in lowest terms, which is to say that for the rational $p = a/b$, we have a and b relatively prime, or formulated equivalently, $\gcd(a, b) = 1$. This is guaranteed by setting

$$a = \frac{a}{\gcd(a', b')} \quad \text{and} \quad b = \frac{b}{\gcd(a', b')},$$

for a' and b' the values of the numerator and denominator, respectively, as passed to the constructor.[§] Doing this in the constructor ensures that when a client instantiates a `Rational` object with parameters that are not lowest terms, we do store the rational

[§]Was it necessary first to assign the result of the gcd function to variable? In which other ways could we have written it? What does the definition of the greatest common divisor allow?

Method call	**Binary operation**
`x.__add__(y)`	`x + y`
`x.__and__(y)`	`x & y`
`x.__div__(y)`	`x / y`
`x.__divmod__(y)`	`divmod(x, y)`
`x.__floordiv__(y)`	`x // y`
`x.__lshift__(y)`	`x << y`
`x.__mod__(y)`	`x % y`
`x.__mul__(y)`	`x * y`
`x.__or__(y)`	`x | y`
`x.__pow__(y[, z])`	`pow(x, y[, z])`
`x.__rshift__(y)`	`x >> y`
`x.__sub__(y)`	`x - y`
`x.__xor__(y)`	`x ^ y`

TABLE 1.1: Methods to implement binary arithmetic operations.

value in lowest terms. Also, it obviates the necessity for having duplicate code in methods like `__add__` and `__mul__` that might pass parameters not in lowest terms.

To compute the greatest denominator, the function `gcd` is used. This function, although in the same module as the class `Rational` is not part of the class definition. Therefore, it is not a method to be called on a `Rational` object, but a normal function with two (implicitly) integral parameters. It is quite common for such helper functions to be present in the same module as a class definition … as is it possible to have more than one class definition in the same module! Note that helper functions are available to *clients*—other code that uses the module—as part of the module.[5]

To conclude the example, consider the `__float__` method. Just like `__str__` is used by the `str` function to return a string representation for an object, `__float__` is used by the `float` function to turn an object into a floating-point number. Here, we simply convert the numerator and denominator into `float`s, then we divide the former by the latter and return the result.

1.7 Methods for operator overloading

The `+` and `*` operators are not the only ones that can be overloaded. Indeed, even some Python keywords like `in` can be overloaded for new classes. Table 1.1 gives the binary operations that can be overload to emulate numeric types, and Table 1.2 gives the unary operators and type conversion operations.

We have not considered what is to happen when an operator is used with different

[5] The `Rational` constructor ensures that its parameters are integers by calling the `int` function. So, the function `gcd` does not bother to check its parameters. However, if it might be used by something else than the class, we might think of instituting such checks.

Method call	Operation
x.__abs__()	abs(x)
x.__invert__()	~x
x.__neg__()	-x
x.__pos__()	+x
x.__complex__()	complex(x)
x.__float__()	float(x)
x.__int__()	int(x)
x.__long__()	long(x)

TABLE 1.2: Methods to implement unary arithmetic and type conversion operations; the latter should return values of the appropriate types.

Method call	Operation	Reflection
x.__eq__(y)	x == y	y.__ne__(x)
x.__ge__(y)	x >= y	y.__le__(x)
x.__gt__(y)	x > y	y.__lt__(x)
x.__le__(y)	x <= y	y.__ge__(x)
x.__lt__(y)	x < y	y.__gt__(x)
x.__ne__(y)	x != y	y.__eq__(x)

TABLE 1.3: Methods to overload the comparison operators.

Method call	Description
`obj.__len__(self)`	For use by the `len` function.
`obj.__getitem__(self, key)`	Evaluate `self[key]`.
`obj.__setitem__(self, key, value)`	Assign `value` to `self[key]`, raising a `TypeError` for keys of inappropriate type, an `IndexError` for keys out of range, and a `KeyError` if the key for a mapping type is missing.
`obj.__delitem__(self, key)`	Delete `self[key]`, with exceptions like the previous method.
`obj.__contains__(self, item)`	To test membership.

TABLE 1.4: Methods to emulate container types.

operand types. This is where some of the finer details of operator overloading comes into play. In some cases, different operand types should not be allowed (for example, Python prevents strings and numeric types from being mixed with `+`), but in other cases it might make sense (such as when we divide a `float` by an `int` and get a `float` value in return). Suffice it so say that Python provides *reflected operand methods* for such cases, where the left operand type does not know about the right operand type. These methods correspond to those in Table 1.1, but their names have an *r* after the first double undersore, and they are used to swap the effect on the operands. For example, if in the expression `x - y`, `x` is an `int` and `y` is one of our `Rationals`, `x` will as a built-in type will not know how to operate on our type of `y`. Then we can add a method `__rsub__` to `Rational` so that `y.__rsub__(x)` can be called when `x.__sub__(y)` does not know what to do. Of course, we must decide what value would be sensible to return.

The comparison (or relational) operators are given in Table 1.3. There are no reflected methods for use with different types. Instead, if the left operand type does not know about the right, the method from the reflection column in Table 1.3 is used.[‖] No implied relationships exists between the operators; if `x == y` evaluates to `True`, it does not automatically mean `x != y` is `False`. It is up to the programmer to define the operators so that they behave as expected.

A number of other methods for overloading behaviour exist. Some of interest are given in Table 1.4, which is by no mean exhaustive, but serves as an indication of what is possible. *Remember:* When you implement any of the overload methods, there is always a first parameter for the reference to `self`. This reference is explicity shown in Table 1.4, but is assumed in the preceding tables.

[‖]Note that the last column in Table 1.3 gives the *reflection* and not the *converse* of the first column. For example, the reflection of $\geq$ is $\leq$, whereas its converse is $<$.

Further reading The following is optional reading, and definitely not for the faint of heart. Unless you have done at least second-year modules in Computer Science, much of the information at these links will probably be incomprehensible. Python's class mechanism is described at `https://docs.python.org/2/tutorial/classes.html`. Python's data model is specified in full at `https://docs.python.org/2/reference/datamodel.html`.

1.8 Exercises

For each of the following exercises, make sure that you write appropriate test cases with which to test your work. More involved exercises are indicated with an asterisk.

EXERCISE 1.1 Create the data type `Charge`, which models data for charged particles, as a Python class that supports the interface and operations of Table 1.5.

TABLE 1.5: The interface and operations for the class `Charge`.

Operations	Description
`q1 = Charge(q, x, y)`	Construct a new `Charge` object with charge `q` (in Coulombs) at the point with coordinates `x` and `y`.
`str(q1)`	Return the string representation "`q @ (x, y)`" for the `Charge` object `q1`, where `q` is the charge (in Coulombs), and `x` and `y` are its x and y coordinates, respectively.
`q1.potential_at(x, y)`	Return the electric potential at the point with coordinates `x` and `y` due to the charge `q1`.

According to Coulomb's law, the electric potential at a point due to a charge is given by

$$V = kq/r, \tag{1.4}$$

where q is the charge value, r is the distance of the point to the charge, and $k = 8.99 \times 10^9\ \text{Nm}^2/\text{C}^2$.

Adapted from Sedgewick and Wayne [6]. ■

EXERCISE 1.2 Colour is the sensation produced when light of different wavelengths falls on the eye. Different formats are used to represent colours on computers. The primary format for digital devices is known as the RGB format; it specifies the levels of red (R), green (G), and blue (B) as integers in the range $[0, 255]$. Create the data type `RGBColour`, which models a colour in the RGB format, as a Python class that supports the interface and operations of Table 1.6.

The property known as *monochrome luminance* or *effective brightness* is important for, among other things, rendering high-quality graphics on electronic displays and converting colour to black-and-white (i.e., grayscale) for monochrome displays or printing. The following standard formula for the luminance Y is derived from the eye's sensitivity to red, green, and blue:

$$Y = 0.299r + 0.587g + 0.114b, \tag{1.5}$$

TABLE 1.6: The interface and operations for the class `RGBColour`.

Operations	Description
`c = RGBColour(r, g, b)`	Construct a new `RGBColour` object with the specified level `r` of red, `g` of green, and `b` of blue; if any parameter is out of range, raise a `ValueError`.
`c.red()`	Return the red level of the `RGBColour` object `c`.
`c.green()`	Return the green level of the `RGBColour` object `c`.
`c.blue()`	Return the blue level of the `RGBColour` object `c`.
`c.luminance()`	Return the effective brightness of the `RGBColour` object `c`.
`c1 == c2`	Return `True` if the `RGBColour` objects `c1` and `c2` have equal red, green, and blue levels; return `False` otherwise.

where r is the red level, g is the green level, and b is the blue level. Since the coefficients sum to 1, and each level is an integer in the range $[0, 255]$, the luminance is a real number in the range $[0, 255]$. Adapted from Sedgewick and Wayne [6]. ■

EXERCISE 1.3 Whereas the RGB colour model of Exercise 1.2 is well suited to the representation of colour for electronic devices, colour for printed matter is often represented in the CMYK format, which specifies the levels of cyan (C), magenta (M), yellow (Y), and black (K) as real numbers in the range $[0, 1]$. Create a data type `CMYKColour`, which models colour in the CMYK format, as a Python class that supports the interface and operations of Table 1.7. ■

TABLE 1.7: The interface and operations for the class `CMYKColour`.

Operations	Description
`d = CMYKColour(c, m, y, k)`	Construct a new `CMYKColour` object with the specified level `c` of cyan, `m` of magenta, `y` of yellow, and `k` of black; if any parameter is out of range, raise a `ValueError`.
`d.cyan()`	Return the cyan level for the `CMYKColour` object `d`.
`d.magenta()`	Return the magenta level for the `CMYKColour` object `d`.
`d.yellow()`	Return the yellow level for the `CMYKColour` object `d`.
`d.black()`	Return the black level for the `CMYKColour` object `d`.
`d1 == d2`	Return `True` if the `CMYKColour` objects `d1` and `d2` have equal cyan, magenta, yellow, and black levels; return `False` otherwise.

EXERCISE 1.4 Add the functionality specified in Table 1.8 to the `RGBColour` class of Exercise 1.2.

The RGB format is used to specify the colours used in Scalable Vector Graphics (SVG), Hypertext Markup Language (HTML), and Cascading Stylesheets (CSS). However, in these languages and formats, RGB colours are written as *hexadecimal triplet* strings. Since each colour level is an integer in the range $[0, 255]$, each level

TABLE 1.8: Additional methods for the class `RGBColour`.

Operations	Description
`c.as_hex()`	Return a string that is the hexadecimal representation of the `RGBColour` object `c`.
`c.to_cmyk()`	Return a new `CMYKColour` object that gives the equivalent colour of the `RGBColour` object `c`.

can be represented as a two-digit hexadecimal number.** For example, the official maroon colour of Stellenbosch University, used as spot colour in this text, has the RGB value $(140, 15, 45)$, which—since $140_{10} = 8C_{16}$, $15_{10} = F_{16}$, and $45_{10} = 2D_{16}$—is the hex triplet string "`8c0f2d`". Note that single-digit hexadecimal components are padded with a leading zero in the output.

RGB can easily be converted to CMYK: If all the RGB levels are 0, then $C = M = Y = 0$, and $K = 1$; otherwise:

$$w = \max(r/255, g/255, b/255), \tag{1.6}$$
$$c = (w - (r/255))/w, \tag{1.7}$$
$$m = (w - (g/255))/w, \tag{1.8}$$
$$y = (w - (b/255))/w, \tag{1.9}$$
$$k = 1 - w, \tag{1.10}$$

where r, g, and b are the levels of red, green, and blue, respectively, in the RGB format, w is the whiteness level, and c, m, y, and k are the levels of cyan, magenta, yellow, and black, respectively, in the CMYK format. ■

EXERCISE 1.5 Although Python has a built-in `complex` data type for complex numbers, create your own `Complex` class, and provide support for the relevant operators. Use the following definitions:

$$(a + bi) + (c + di) = (a + c) + (b + d)i, \tag{1.11}$$
$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i, \tag{1.12}$$
$$|a + bi| = \sqrt{a^2 + b^2}, \tag{1.13}$$
$$\Re(a + bi) = a, \tag{1.14}$$
$$\Im(a + bi) = b. \tag{1.15}$$

The first three definitions must be implemented by operator overloading, but the last two require methods, which you should name `re` and `im`, respectively. ■

EXERCISE 1.6* Complete the example of Figure 1.5 by adding support for the relevant missing operators from Tables 1.1, 1.2, and 1.3. Ensure that your `Rational`

**For more information on the hexadecimal notation and why it is found so often in computer programs, refer to `http://en.wikipedia.org/wiki/Hexadecimal`.

class is compatible with Python's built-in `int` and `float` types: Since `int` models the integers $\mathbb{Z}$, `float` models the real numbers $\mathbb{R}$, and

$$\mathbb{Z} \in \mathbb{Q} \in \mathbb{R}, \tag{1.16}$$

use appropriate *automatic widening conversions* for when your `Rational` class is use with these built-in data types. That is, when a binary operator takes a `Rational` object as one operand, and an `int` or `float` value as the other, the result must be a value in the larger set according to Eq. (1.16). Refer to the Python data model document at `https://docs.python.org/2/reference/datamodel.html`. ■

CHAPTER 2

Data structures in Python

2.1 List comprehensions

In the mathematical sciences, we often define sets with the *set-builder notation*, a way of describing a set by giving the properties its members must satisfy. We write

$$S = \{x \in \mathcal{U} \mid P(x)\}, \tag{2.1}$$

where $\mathcal{U}$ is the so-called *universe of discourse*, and $P(x)$ is some predicate; then S consists of those in elements in $\mathcal{U}$ for which $P(x)$ holds. Forming sets this is also sometimes called *set comprehension*.

In Python, *list comprehensions* allow concise definitions of lists, in a way similar to set comprehension. A list comprehension consists of an expression followed by a `for` clause, followed in turn by zero or more `for` or `if` clauses. The result of a list comprehension is the list of values that results from evaluating the expression in the context given by the `for` and `if` that follow it. In a sense, the `for` clause specifies the universe of discourse, and the `if` clause specifies the predicate that must hold for elements selected from the universe.

EXAMPLE 2.1.1 In the following code snippet, we compute the list of squares of the first five positive integers.

```
>>> ints = range(1, 6)
>>> [x*x for x in ints]
[1, 4, 9, 16, 25]
```

The resulting list is equivalent to the list `squares` built by the following code.

```
>>> squares = []
>>> for x in range(1, 6):
...     squares.append(x*x)
...
>>> squares
[1, 4, 9, 16, 25]
```

The use of a list comprehension, however, results in code that is shorter, and we do not have to start by creating an empty list. ■

EXAMPLE 2.1.2 Suppose we want to select, for small x, only those x^2 such that $x^2 > 2x$. Therefore, we add an `if` clause to the list comprehension.

```
>>> [x**2 for x in range(1, 6) if x**2 > 2*x]
[9, 16, 25]
```

In the `for` clause, unsurprisingly, we can use any list, tuple, sequence, or indeed, anything we can iterate over. Equivalent statements, without using a list comprehension, to construct this list are:

```
>>> squares = []
>>> for x in range(1, 6):
...     if x**2 > 2*x:
...         squares.append(x**2)
...
>>> squares
[9, 16, 25]
```

Comparing the list comprehension to its "unwrapped" form, note that we read the `for` and `if` clauses from left to right. ■

The expression in the list comprehension can also evaluate to a tuple, but then the tuple expression must be parenthesised.

EXAMPLE 2.1.3 Let us make a list of tuples (x, x^3) for integer x, where $1 \leq x < 6$.

```
>>> [(x, x**3) for x in range(1, 6)]
[(1, 1), (2, 8), (3, 27), (4, 64), (5, 125)]
```

If the parentheses are left out of the list comprehension, it will cause a `SyntaxError` to be raised. ■

EXAMPLE 2.1.4 It is also possible to have more than one `for` clause in a list comprehension.

```
>>> [(x, y) for x in range(3) for y in range(3) if x != y]
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

The unwrapped version would look as follows.

```
>>> pairs = []
>>> for x in range(3):
...     for y in range(3):
...         if x != y:
...             pairs.append((x, y))
...
>>> pairs
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

Note that we still read the `for` and `if` clauses from left to right. ■

Like lists, list comprehensions can be nested. But be careful: The nested structures inside a list comprehension must be read from right to left.

EXAMPLE 2.1.5 We can use a nested list comprehension to build the transpose of a matrix. Let us first consider what it would look like without using any list comprehensions.

```
>>> M = [[1, 2, 3],
...      [4, 5, 6],
...      [7, 8, 9]]
>>> Mtrans1 = []
>>> for i in range(len(M[0])):
...     Mtrans_row = []
...     for row in M:
...         Mtrans_row.append(row[i])
...     Mtrans1.append(Mtrans_row)
...
>>> Mtrans1
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Note how we approach the loops so that we do not need to setup an emptry matrix first. (What would happen if we tried to loop over `i` and `j` for the rows and columns, respectively, and then attempt assignments of the form `Mtrans[j][i] = M[i][j]`?)

Next, we replace the inner loop with a list comprehension:

```
>>> Mtrans2 = []
>>> for i in range(len(M[0])):
...     Mtrans2.append([row[i] for row in M])
...
>>> Mtrans2
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Finally, we replace the outer loop with a list comprehension:

```
    >>> [[row[i] for row in M] for i in range(len(M[0]))]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

From these three code snippets, the power and conciseness of the list comprehension paradigm should be self-evident. ■

2.2 Sets

In Python, a set is an unordered collection of distinct elements.

- It is *unordered* in that there is no notion of a first element, a last element, or in fact, an ith element in a particular set. Compare this to a list, where we often index numerically into the list.
- The elements are *distinct* in that there can be no duplicate elements in a set, that is, any particular element can appear at most once in a given set.

2.2.1 Set creation

A set is created with the `set` constructor. This constructor takes a optional iterable argument that specifies the contents of the set.

EXAMPLE 2.2.1 To create an empty set, simply use the constructor without any parameters (but with parentheses).

```
>>> set()
set([])
```

Note how a set is evaluated and displayed when entered as an expression. ■

EXAMPLE 2.2.2 We can also construct a set with initial values, which we supply as a list argument.

```
>>> set([2, 3, 5, 7, 11])
set([11, 2, 3, 5, 7])
>>> set(2, 3, 5, 7, 11)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: set expected at most 1 arguments, got 5
```

Simply specifying the values, separated by commas, as if they were separate arguments, results in an error. ■

EXAMPLE 2.2.3 Since there can be no duplicate elements, the following two constructor statements create the same set.

```
>>> set([2, 3, 4])
set([2, 3, 4])
>>> set([2, 3, 4, 3, 3, 4])
set([2, 3, 4])
```

Remember that the order in which the elements are displayed does not imply any order on the elements of the set. ■

The optional iterable argument can be anything like a list or a sequence over which Python can iterate with a `for` loop. This implies we can use a string as argument to a set constructor, in which case we get a set containing the (unique) individual characters in the string.

EXAMPLE 2.2.4 To construct a set over a string, simply pass the string as argument to the `set` constructor:

```
>>> set("Science Experiment")
set([' ', 'c', 'e', 'i', 'm', 'n', 'p', 'S', 'r', 't', 'x', 'E'])
```

Note that the space is stored explicity, there are no duplicates, and the usual lexicographic order is not used when the set is displayed. ■

We can also use *generator expressions*, which generalise list comprehensions and generators.* They have the same syntax as a list comprehension, but do not create entire lists at once. Therefore, they should be more space efficient than list comprehensions, especially in contexts where data must be passed as a throwaway, iterable argument, but having a list is not required.

EXAMPLE 2.2.5 It is possible to construct a set with a list comprehension:

```
>>> set([x**3 for x in range(-75, 75, 19)])
set([-175616, 8000, -50653, 1, 195112, -421875, 59319, -5832])
```

To turn the argument into a generator expression, simply remove the brackets around the list comprehension:

```
>>> set(x**3 for x in range(-75, 75, 19))
set([-175616, 8000, -50653, 1, 195112, -421875, 59319, -5832])
```

This generation expression requires less memory than the equivalent list comprehension. ■

Creating a list that is not going to be used again is not only a waste of memory, but often a waste of time as well. For example, when we use the `range` function to iterate over a list by index, `range` first explicitly constructs and populates a list (of indices) before iteration begins. In this case, it is better to use the special `xrange` iterator object. Although it is used in the same way as `range`, the `xrange` object is typically much more efficient.

EXAMPLE 2.2.6 We can use Python's `timeit` module [3, §26.6] from the command line to compare the running times of setting up `range` as opposed to `xrange`. The `-m timeit` command-line option tells Python to run `timeit` module, and the text between the single quotes is the code to run.

```
whkbester@open:~$ python -m timeit 'range(10000000)'
10 loops, best of 3: 148 msec per loop
whkbester@open:~$ python -m timeit 'xrange(10000000)'
1000000 loops, best of 3: 0.219 usec per loop
```

The exact timing is, of course, dependent on the computer system on which the examples are run. Note, however, that `range` runs roughly 640 000 times longer than `xrange`! Even if we use `range` and `xrange` in a loop, we see a difference:

```
whkbester@open:~$ python -m timeit 'for i in range(10000000): pass'
10 loops, best of 3: 332 msec per loop
whkbester@open:~$ python -m timeit 'for i in xrange(10000000): pass'
10 loops, best of 3: 211 msec per loop
```

Although, now `range` runs only one and half times as long as `xrange`, using `xrange` still translates to significant time savings, especially if your program contains many loops. ■

---

*We do not cover generators in this course. Suffice it to say that are written like functions, but provide *lazy evaluation*. That is, they only evaluate to a result when that value is needed. For more information, refer to the online Python Tutorial [5], §9.10 and §9.11.

2.2.2 Set operations

Like the other data types we have met in Python—integers, floating-point numbers, strings, lists, and tuples—sets are objects. This means they "own" methods. In mathematics, certain operations are defined on sets. As should become apparent now, sets in Python correspond to sets in mathematics.[†]

Mathematical set operations such as intersection and union are implemented as set methods in Python. Since they support update operations, whereby the set can be changed, sets in Python are not immutable. The basic set operations are described in Table 2.1 and illustrated in the following Python interpreter session listing.

```
>>> lowints = set(range(10))
>>> lowprimes = set([2, 3, 5, 7])
>>> lowevens = set([x for x in lowints if x > 0 and x % 2 == 0])
>>> lowints.remove(0); lowints
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> lowints.add(10); lowints
set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> lowprimes.difference(lowevens)
set([3, 5, 7])
>>> lowprimes.symmetric_difference(lowevens)
set([3, 4, 5, 6, 7, 8])
>>> lowprimes.intersection(lowevens)
set([2])
>>> lowprimes.union(lowevens)
set([2, 3, 4, 5, 6, 7, 8])
>>> lowprimes.issubset(lowints)
True
>>> lowevens.issuperset(lowprimes)
False
```

Many of the operations performed by the methods of Table 2.1 can also by written with overloaded arithmetic and comparison operators. These "shortcuts" are given in Table 2.2; since they are binary operators, the ones given are for exactly those operations that are performed on two sets.

EXAMPLE 2.2.7 The following shows a quick way for testing set membership.

```
>>> lowprimes = set([2, 3, 5, 7])
>>> 2 in lowprimes
True
>>> 4 in lowprimes
False
```

As an exercise, see for which other data types you can use the `in` keyword. ■

Finally, we can iterate over the elements in a set with a `for` statement. However, note that sets do not have predictable iteration order as do sequences like lists. That

[†] A Python set can only store immutable objects. The reason for this is explained in the section on dictionaries.

TABLE 2.1: Methods for basic set operations.

Method use	Description
`S.add(x)`	Adds an element x to the set S
`S.clear()`	Removes all elements from the set S.
`S.difference(T)`	Returns a new set with all the elements that are in the set S but not in the set T.
`S.intersection(T)`	Returns a new set with the elements that are in both the sets S and T.
`S.issubset(T)`	Returns whether the set S is a subset of the set T or not.
`S.issuperset(T)`	Returns whether the set S is a superset of the set T or not.
`S.remove(x)`	Removes the element x from the set S.
`S.symmetric_difference(T)`	Returns a new set with the elements that are either in the set S or the set T, but not in both.
`S.union(T)`	Returns a new set with the elements that are in the set S or the set T (or both).

TABLE 2.2: Binary operators for basic set operations.

Method	Binary operation
`S.difference(T)`	`S - T`
`S.intersection(T)`	`S & T`
`S.issubset(T)`	`S <= T`
`S.issuperset(T)`	`S >= T`
`S.union(T)`	`S | T`
`S.symmetric_difference(T)`	`S ^ T`

is, they are not necessarily encountered in the order they were added to the set, by value, alphabetically, or indeed, any other order.

```
>>> s = set(['one', 'two', 'three', 'four'])
>>> for word in s: print word,
...
four three two one
```

2.3 Dictionaries

In mathematics, we often find the notion of a mapping, where an element of one set is uniquely mapped to an element of another (or possibly, the same) set. From this perspective, a Python list is simply a mapping from a subset of the integers to another, arbitrary set: We use the index, an integer in the valid range for that list, as a key to get the value at that index.

Dictionaries—also called *maps*, *associative arrays*, or *hashes*—allow us to use data types other than integers as indices into a list. That is, we use them as keys to get to a uniquely determined value. This partially motivates calling this kind of data structure a "dictionary": Given a word, a dictionary gives us its definition.[‡] Before looking at the details, we first consider how it is possible to use a non-integer value as a key to some other value.

2.3.1 Hashing

Python has a built-in function called `hash` with the following description in the help system:

> hash(object) → integer
>
> Return a hash value for the object. Two objects with the same value have the same hash value. The reverse is not necessarily true, but likely.

So, Python provides the ability of turning any value into an integer.[§] Furthermore, by definition, two values that are equal have the same hash value. Although two different values may hash to the same value, hash functions are typically constructed so that values that differ only in a small part definitely have different hash values. For example, consider the hash values for the following strings.

```
>>> hash("aluminium")
608505218
>>> hash("aluminum")
-2106332058
```

[‡]However, the terminology *is* somewhat confusing, for many words in a dictionary have more than one definition associated with them. In this case, we can reason that each word is associated with a unique *set* of definitions.

[§]Of course, we are sweeping a lot of detail under the carpet. Actually, `hash` can turn any *object* into an integer, calling upon the object's `__hash__()` method. For those of you familiar with Java, think of `hashCode()`.

The hash function allows Python to use a data structure called a *hash table*. A hash table works like an indexed list, but instead of storing a single value at any particular index, a collection of values can be stored at an index. When an object is inserted into a hash table, its hash value is computed and mapped to a table index by some modulo operation. Then, the value is added to the collection at that index. When testing membership, all that is required is computing the hash value and index again, and then looking through the (relatively small) collection of values at that index.

This idea underlies how Python handles the sets of the previous section. It also explains why only immutable values may be stored in a set. Allowing mutable values would create problems when the value is changed and this change leads to a change in its hash value. For such a value, where must Python look in the hash table? Therefore, it is also an error to call `hash` on a mutable object.

```
>>> hash((1, 2, 3))
-378539185
>>> hash([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

To implement a dictionary, Python also uses hashing, but instead of just storing values, it stores key–value pairs. The hash value of the key determines where Python looks for the value in the hash table.

2.3.2 Dictionary operations

In Python, a dictionary is an unordered, mutable collection of key–value pairs. For each pair, the value is associated with the key, which can then be used to locate, update, or remove the value. For the reasons given above, each key must be an immutable object; the values, on the other hand, may be mutable. We also say that the keys form a set: Any particular key can appear at most once in any particular dictionary.

We can create a dictionary by enclosing key–value pairs inside two braces, where a key is separated from its value by a colon. In the following example, we associate birth dates with surnames.

```
>>> scientists = {'Turing': 1912, 'Darwin': 1809}
>>> scientists
{'Turing': 1912, 'Darwin': 1809}
```

An empty dictionary would simply be written `{}`. To get the value associated with a key, write the key in brackets after the dictionary name. Using a key that is not in a particular dictionary is an error, just like an out-of-range index in a list. We continue the previous example.

```
>>> scientists['Darwin']
```

```
1809
>>> scientists['Newton']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Newton'
```

Updating dictionaries and testing for membership also happens in a familiar way. To test whether the key `k` is in the dictionary `dict`, use the expression `k in dict`. To assign a value to key, use a bracket assignment expression, just like a list. If the key is already in the dictionary, the new value is associated with the key, and the old one discarded; if the key is new, a new key–value pair is created in the dictionary. The `del` operator removes a key–value pair. Again, we continue the previous examples.

```
>>> if 'Darwin' in scientists:
...     print 'Darwin born in', scientists['Darwin']
...
Darwin born in 1809
>>> if 'Newton' in scientists: print 'oops!'
...
>>> scientists
{'Turing': 1912, 'Darwin': 1809}
>>> scientists['Newton'] = 1641
>>> scientists
{'Turing': 1912, 'Darwin': 1809, 'Newton': 1641}
>>> scientists['Newton'] = 1642
>>> scientists
{'Turing': 1912, 'Darwin': 1809, 'Newton': 1642}
>>> del scientists['Turing']
>>> scientists
{'Darwin': 1809, 'Newton': 1642}
```

Dictionaries are collections, and therefore we can loop over them with `for` loops, similarly to lists. However, where lists have predictable iteration order and each iteration assigns the next list value to the iteration variable, dictionaries iterate over the keys in arbitrary order.

```
>>> halogens = {'F': 'Fluorine', 'Cl': 'Chlorine', 'Br': 'Bromine',
...             'I': 'Iodine', 'At': 'Astatine'}
>>> for h in halogens:
...     print h + ':', halogens[h]
...
I: Iodine
Cl: Chlorine
At: Astatine
Br: Bromine
F: Fluorine
```

Method	Description
`D.clear()`	Removes all key–value pairs from the dictionary D.
`D.get(k, v)`	Returns the value associated with the key k in the dictionary D, or returns the default value v if k is not present; the parameter v is optional, and if it is not specified, it defaults to `None`.
`D.keys()`	Returns the keys of the dictionary D as a list; the entries are guaranteed to be unique.
`D.items()`	Returns a list of key–value pairs in the dictionary D as tuples.
`D.iteritems()`	Returns an iteration object for key–value pairs in the dictionary D.
`D.values()`	Returns a list of the values in the dictionary D; the entries may or may not be unique.
`D.update(E)`	Adds the key–value pairs in the dictionary D to the dictionary E; if a key k already exists in D, it is associated with $E[k]$.

TABLE 2.3: Methods for basic dictionary operations.

2.3.3 Dictionary methods

Like the other collections we have encountered, dictionaries are objects and own methods, the most important of which are given in Table 2.3. Note that there are two methods that allow us to iterate over the key–value pairs in a dictionary, `items()` and `iteritems()`. The difference between the two is this: `items()` creates an actual list of key–value pairs, which is inefficient for large dictionaries, whereas `iteritems()` returns the key–value pairs one by one, as if "on demand".

To understand the usefulness of the `get`, consider an example where we want to count the occurrences of different strings in a file. Assuming that there is only one string per line in the open input file `infile`, without `get` we would have to iterate thus:

```
for line in infile:
    word = line.strip()
    if name in count:
        count[name] = count[name] + 1
    else:
        count[name] = 1
```

Because the `get` method allows us to specify a default value, the above simply becomes:

```
for line in infile:
    word = line.strip()
    count[name] = count.get(name, 0) + 1
```

The `sorted` function is useful when we want to iterate over the keys in alphabetical order. Take the `halogens` dictionary from a previous example, and iterate alphabetically over the keys as follows.

```
for h in sorted(halogens):
    print h + ':', halogens[h]
```

One last thing: Sometimes we want to invert the dictionary so that the values become the keys with which the existing keys are associated as values. Remember that although the keys are unique, the values do not have to be. So, when inverting a dictionary, it is necessary to use a collection such as a list to store the values for each key in the inverted dictionary.

CHAPTER 3

Exceptions

Writing programs free of errors is a difficult enterprise. If we imagine, for a moment, a program that is completely self-contained, then the errors it contains are entirely our own and must be corrected at the source. But how feasible is this abstraction? Even a program that generates a very long sequence of pseudorandom numbers, which could theoretically run forever without human intervention, is not completely sealed off: It still runs in a system of which the hardware might fail or where other programs can misbehave and interfere; or even more pedestrian problems may cause failure, such as when too much memory is consumed or when a disk is written full.

So, programming defensively is an often overlooked fact of writing software. We must cater not only for own fallibility—we must also antipicate errors when a program interacts with people or data sources, and we must write programs that are able to handle gracefully erroneous, or sometimes even malicious, input.

Older languages mostly signal error conditions with specially distinguished values; the integer −1, and values akin to Python's `False` and `None` are typically used for this purpose. Often, the use of values in a "special" sense is difficult to understand, or even worse, conflicts with the use of those values in a "normal" sense. For example, if a function returns −1 when attempting division by zero during some computation on arbitrary input values, how is this to be distinguished from the computation actually resulting in −1?

Newer languages, such as Python, use *exceptions* to signal error conditions. When an error occurs, the programmer or the system *raises* an exception that must be handled outside of the normal program flow, else the program terminates. Separating error-handling code from normal code allows us to deal with an error where it occurs—or someplace else. Also, the exception object that is created contains information on what went wrong, and where; this is useful both for debugging and system logging.

3.1 Handling exceptions

A piece of code can react to an exception by providing an *exception handler*. This is written by putting the code that might raise an exception inside a `try` block, which

is followed by an `exception` block that provides the *exception handler* to react to the error.

```
>>> try:
...     r = 1.0/2.0
...     print 'The reciprocal of 2.0 is', r
...     r = 1.0/0.0
...     print 'The reciprocal of 0.0 is', r
... except:
...     print 'Error: The reciprocal is undefined'
...
The reciprocal of 2.0 is 0.5
Error: The reciprocal is undefined
```

The following happens when Python executes this `try–except` statement: (i) The statements in the `try` clause are executed up to where the division by zero, which raises an exception, is attempted; (ii) then, skipping the rest of the clause, execution jumps to `except` clause; and (iii) when these statements have finished, execution resumes normally *after* `except` clause—except if a statement in this `except` block causes program control to leave the container block of `try–except` statement, for example, by using the `return` statement, or by raising an exception.

3.2 Exception objects and handlers

The previous example illustrates only a small subset of the behaviour possible with the `try` statement. Here, we sketch the complete picture. Remember that an exception is *raised* (implicitly or explicitly) at the point an error is detected; it may be *handled* by the surrounding code block or by any other code block that directly or indirectly called the code block in which the error occurred.

The `try` statement may have more than one `except` clause, each one giving a different *exception handler*; at most one handler will be executed. Each `except` keyword, except the last, must be followed by an exception name or a tuple of exception names to be handled by the following handler. The first handler that matches the type of the exception raised will be executed.

```
>>> values = [-1, 0, 1]
>>> for i in xrange(4):
...     try:
...         r = 1.0 / values[i]
...         print 'reciprocal of', values[i], 'at', i, 'is', r
...     except IndexError:
...         print 'index', i, 'out of range'
...     except ArithmeticError:
...         print 'cannot compute reciprocal of', values[i]
...
reciprocal of -1 at 0 is -1.0
cannot compute reciprocal of 0
```

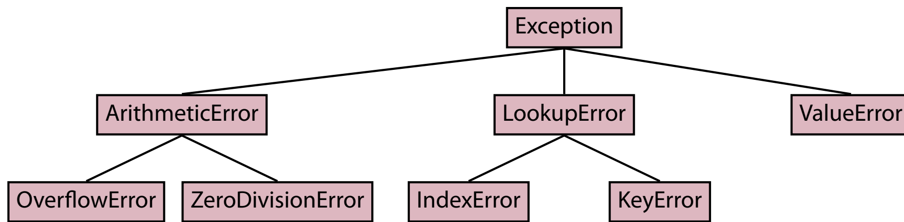

FIGURE 3.1: The Python exception hierarchy for some commonly used exceptions.

```
reciprocal of 1 at 2 is 1.0
index 3 out of range
```

It is also possible to assign to a variable the exception that was caught by the current handler. This allows us to get, amongst other things, standard error messages from the Python system. Note that a comma may be used in the place of the `as` keyword, which is what we do in the following example.

```
>>> values = [-1, 0, 1]
>>> for i in xrange(4):
...     try:
...         r = 1.0 / values[i]
...         print 'reciprocal of', values[i], 'at', i, 'is', r
...     except IndexError as e:
...         print 'index error:', e
...     except ArithmeticError as e:
...         print 'arithmetic error:', e
...
reciprocal of -1 at 0 is -1.0
arithmetic error: float division
reciprocal of 1 at 2 is 1.0
index error: list index out of range
```

To understand when an exception matches a handler, note that Python exceptions are structured hierarchically. Figure 3.1 gives a tree with some of the most common exceptions. The tree, read from the leaves to the root (bottom-up), defines an IS-A-SUBCLASS-OF relation. For example, `OverflowError` is a subclass of `ArithmeticError`, which, in turn, is a subclass of `Exception`. The relation is transitive, so that a `OverflowError`, for example, is also a subclass of `Exception`.* An `except` clause matches an exception object if the exception object is the exception specified by the clause or a subclass thereof.

The last `except` clause may omit the exception name, and then serves as a wildcard that catches any exception. This is, essentially, what we did in the code example of §3.1. Note, however, that since such a wildcard `except` clause may hide a programming problem, its use is almost always a bad idea.

---

* For the algebraically inclined, the exception hierarchy is a partial order. The relation it defines is also reflexive and antisymmetric. Reflexivity, in particular, implies that any class is a subclass of itself.

The last `except` clause, whether with or without specified exception name, may be followed by an (optional) `else` clause, which will only be executed if no exception was raised in the `try` clause. In Python, it is usual to make the `try` block as small as possible, so that we do not inadvertently catch errors we did not want to catch. Therefore, we can add statements that should follow those in the `try` block, but only makes sense if those in the `try` block completed without error.

```
>>> def invert(x):
...     try:
...         r = 1.0 / x
...     except ArithmeticError as e:
...         print 'arithmetic error:', e
...     else:
...         print 'reciprocal of', x, 'is', r
...
>>> invert(7)
reciprocal of 7 is 0.142857142857
>>> invert(0)
arithmetic error: float division
```

An optional `finally` clause may follow the last `except`, or if it is present, `else` clause, and is executed whether or not an exception was raised in the `try` clause. Even when an exception is raised in the `try` clause but subsequently not caught by an `except` clause, the `finally` clause is executed.

```
>>> def divide(x, y):
...     try:
...         q = x / y
...     except ZeroDivisionError as e:
...         print 'zero division error:', e
...     else:
...         print 'quotient is', q
...     finally:
...         print 'finally....'
...
>>> divide(1., 3.)
quotient is 0.333333333333
finally....
>>> divide(3., 0.)
zero division error: float division
finally....
>>> divide("1.", "3.")
finally....
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Note that the last call to `divide` resulted in an error because the `/` operator is not overloaded for strings; still, the `finally` clause was executed.

3.3 Raising exceptions

Until now, we have waited for Python statements to cause exceptions. But this does not have to be the case: We can raise exceptions ourselves using the `raise` keyword. It is quite elementary:

```
>>> raise NameError('Goodbye, cruel world!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: Goodbye, cruel world!
```

The sole argument to the `raise` keyword must either be an exception object or the name of a subclass of `Exception`; for common classes in the exception hierarchy, refer to Figure 3.1. Note that it is good style—and good manners—to include an error message to explain what went wrong.

```
>>> def divide(x, y):
...     if y == 0:
...         raise ValueError('divisor is zero')
...     else:
...         return x / y
...
>>> for i in xrange(-1, 2):
...     try:
...         print divide(1, i)
...     except ValueError as e:
...         print 'caught ValueError:', e
...
-1
caught ValueError: divisor is zero
1
```

Since the `raise` keyword also accepts an exception object as argument, we can use `raise` on a variable to which an exception has been assigned. This might, for example, be useful when we want to catch an exception to do something locally, but then to "re-raise" it so that invoking context—the part of the program that originally called the code causing the exception—is also aware of the problem. Such an example now follows; also note that the `finally` clause is executed.

```
>>> try:
...     r = 1.0 / 0.0
... except ArithmeticError as e:
...     print 'ArithmeticError:', e
...     raise e
... finally:
...     print 'Executing finally clause'
...
ArithmeticError: float division
Executing finally clause
```

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
ZeroDivisionError: float division
```

Bibliography

[1] Bloch, J. *Effective Java*, second ed. Addison-Wesley, Upper Saddle River, NJ, 2008.

[2] McKinney, W. *Python for Data Analysis*. O'Reilly, Sebastopol, CA, 2012.

[3] The Python standard library. `https://docs.python.org/2/library/`. Version 2.7.6.

[4] The Python sorting mini-howto. `https://wiki.python.org/moin/HowTo/Sorting`.

[5] The Python tutorial. `https://docs.python.org/2/tutorial/`. Version 2.7.6.

[6] Sedgewick, R., and Wayne, K. *Introduction to Programming in Java: An Interdisciplinary Approach*. Pearson, Boston, MA, 2008.

[7] Sedgewick, R., and Wayne, K. *Algorithms*, fourth ed. Addison-Wesley, Upper Saddle River, NJ, 2011.