

The Author
Mike Chyson

The Big Book of

PYTHON 3

Notebook on programming in python 3

Jan, 15, 2021

Dedication

Notebook on “Programming in Python 3” second edition.

Learn a programming language with:

- Think and summerize.
- Programme.

The code is in <https://github.com/mikechyson/python3>

Contents

Contents	1
1 Introduction	7
1.1 Creating and running Python programs	7
1.2 Python's "Beautiful Heart"	8
1.2.1 Data types	8
1.2.2 Object references	8
1.2.3 Collection data types	9
1.3 Logical operations	10
1.3.1 The identity operator	11
1.3.2 Comparison operators	11
1.3.3 The membership operator	12
1.3.4 Logical operators	12
1.4 Control flow statements	13
1.4.1 The if statement	14
1.4.2 The while statement	14
1.4.3 The for ... in statement	14
1.4.4 Basic exception handling	14
1.5 Arithmetic operators	15
1.6 Input/output	16
1.7 Creating and calling functions	17
2 Data types	19
2.1 Identifiers and keywords	19
2.2 Integral types	20
2.2.1 Integers	20
2.2.2 Booleans	22

2.3	Floating-point types	22
2.3.1	Floating-point numbers	23
2.3.2	Complex numbers	24
2.3.3	Decimal numbers	24
2.4	Strings	24
2.4.1	Comparing strings	26
2.4.2	Slicing and striding strings	26
2.4.3	String operators and methods	27
2.4.4	String formatting with the <code>str.format()</code> method	29
2.4.5	Field names	29
2.4.6	conversions	30
2.4.7	Format specifications	31
3	Collection data types	35
3.1	Sequence types	35
3.1.1	Tuples	36
3.1.2	Named tuples	36
3.1.3	Lists	37
3.1.4	List comprehensions	38
3.2	Set types	39
3.2.1	Sets	39
3.2.2	Set comprehensions	41
3.2.3	Frozen sets	41
3.3	Mapping types	41
3.3.1	Dictionaries	42
3.3.2	Dictionary comprehensions	43
3.3.3	Default dictionaries	44
3.3.4	Ordered dictionaries	44
3.4	Iterating and copying collections	45
3.4.1	Iterators and iterable operations and functions	45
3.4.2	Copying collections	47
4	Control structures and functions	49
4.1	Control structures	49
4.1.1	Conditional branching	49
4.1.2	Looping	50
4.2	Exception handling	51

CONTENTS	3
4.2.1 Catching and raising exceptions	51
4.2.2 Custom exceptions	52
4.3 Costom functions	55
4.3.1 Names and docstrings	57
4.3.2 Argument and parameter unpacking	58
4.3.3 Accessing variables in the global scope	59
4.3.4 Lambda functions	60
4.3.5 Assertions	60
5 Modules	63
5.1 Modules and packages	63
5.1.1 Packages	64
5.1.2 Custom modules	65
5.2 Overview of Pythons standard library	68
5.2.1 Strings	68
5.2.2 Dates and Times	68
5.2.3 Algorithms and collection data types	68
5.2.4 File formats, encodings, and data persistence	69
6 Object-oriented programming	71
6.1 Costom classes	71
6.1.1 Attributes and methods	71
6.1.2 Inheritance and polymorphism	73
6.1.3 Using properties to control attribute access	74
7 File handling	77
8 Advanced programming techniques	79
8.1 Further procedural programming	79
8.1.1 Branching using dictionaries	79
8.1.2 Generator expressions and functions	80
8.1.3 Dynamic code execution and dynamic imports	81
8.1.4 Local and recursive functions	87
8.1.5 Functions and method decorators	87
8.1.6 Function annotations	89
8.2 Further object-oriented programming	91
8.2.1 Controlling attribute access	92
8.2.2 Functors	94

8.2.3	Context manager	95
8.2.4	Descriptors	97
8.2.5	Class decorators	101
8.2.6	Abstract base classes	104
8.2.7	Multiple inheritance	108
8.2.8	Metaclasses	109
8.3	Functional-style programming	112
8.3.1	Partial function application	113
8.3.2	Coroutines	114
9	Debugging, testing, and profiling	117
9.1	Debugging	117
9.1.1	Dealing with syntax errors	117
9.1.2	Dealing with runtime errors	118
9.1.3	Scientific debugging	118
9.2	Unit testing	119
9.3	Profiling	119
10	Processes and threading	123
10.1	Using the multiprocessing module	123
10.2	Using the threading module	126
11	Networking	131
11.1	Console tool	131
11.2	Creating a TCP client	133
11.3	Creating a TCP server	135
11.4	Summary	139
12	Database Programming	141
12.1	DBM databases	141
12.2	SQL databases	141
13	Regular Expressions	143
13.1	Python's regular expression language	143
13.1.1	Character and character classes	144
13.1.2	Quantifiers	145
13.1.3	Grouping and capturing	146
13.1.4	Assertions and flags	147

<i>CONTENTS</i>	5
13.2 The regular expression module	148
14 Introduction to GUI programming	155
14.1 Dialog-style programs	156
14.2 Main-window-style programs	159

Chapter 1

Introduction

1.1 Creating and running Python programs

By default, Python files are assumed to use the UTF-8 character encoding. Python files normally have an extension of `.py`. Python GUI (Graphical User Interface) programs usually have an extension of `.pyw`.

```
1  #!/usr/bin/env python3
2
3  print("Hello ", "world")
```

The first line is a comment. In Python, comments begin with a `#` and continue to the end of the line. The second line is blank. Python ignores blank lines, but they are often useful to humans to break up large blocks of code to make them easier to read. The third line is Python code.

Each statement encountered in a `.py` file is executed in turn, starting with the first one and progressing line by line. Python programs are executed by the Python interpreter, and normally this is done inside a console window.

On Unix, when a program is invoked in the console, the file's first two bytes are read. If these bytes are the ASCII characters `#!`, the shell assumes that the file is to be executed by an interpreter and that the file's first line specifies which interpreter to use. This line is called the **shebang** (shell execute) line, and if the present must be the first line in the file.

The shebang line is commonly written in one of two forms, either:

```
1  #!/usr/bin/python3
```

or

```
1 #!/usr/bin/env python3
```

If written using the first form, the specified interpreter is used. If written using the second form, the first `python3` interpreter found in the shell's current environment is used. The second form is more versatile because it allows for the possibility that the Python 3 interpreter is not located in `/usr/bin`.

1.2 Python's "Beautiful Heart"

1.2.1 Data types

One fundamental thing that any programming language must be able to do is represent items of data.

The size of Python's integers is limited only by machine memory, not by a fixed number of bytes. Strings can be delimited by double or single quotes, as long as the same kind are used at both ends.

```
1 210624583337114373395836055367340864637790190801098222508621955072
2 0
3 "hello "
4 'world '
```

Python uses square brackets (`[]`) to access an item from a sequence such as a string.

```
1 'Hello World'[4]
```

In Python, both `str` and the basic numeric types such as `int` are **immutable**. At first this appears to be a rather strange limitation, but Python's syntax means that this is a nonissue in practice. The only reason for mentioning it is that although we can use square brackets to retrieve the character at a given index position in a string, we cannot use them to set a new character.

To convert a data item from one type to another we can use the syntax `datatype(item)`.

```
1 int('45')
2 str(123)
```

1.2.2 Object references

Once we have some data types, the next thing we need are variables in which to store them. Python doesn't have variables as such, but instead has **object references**. When it comes to immutable objects like `ints` and `strs`,

there is no discernable difference between a variable and an object reference. As for mutable objects, there is a difference, but it rarely matters in practice.

```
1
2 x = "blue"
3 y = "green"
4 z = x
```

The syntax is simply `object_reference = value`. The `=` operator is not the same as the variable assignment operator in some other languages. The `=` operator binds an object reference to an object in memory. If the object reference already exists, it is simply re-bound to refer to the object on the right of the `=` operator; if the object reference does not exist it is created by the `=` operator.

Python uses **dynamic typing**, which means that an object reference can be rebound to refer to a different object at any time. Languages that use strong typing (such as C++ and Java) allow only those operations that are defined for the data types involved to be performed. Python also applies this constraint, but it isn't called strong typing in Python's case because the valid operations can change — for example, if an object reference is re-bound to an object of a different data type.

```
1
2 route = 123
3 print(route, type(route))
4
5 route = "North"
6 print(route, type(route))
```

The `type()` function returns the data type (also known as the “class”) of the data item it is given — this function can be very useful for testing and debugging, but would not normally appear in production code.

1.2.3 Collection data types

To hold entire collections of data items, Python provides several collection data types that can hold items. Python tuples and lists can be used to hold any number of data items of any data types. Tuples are immutable while lists are mutable.

Tuples are created using commas `(,)`, as these examples show:

```
1
2 >>> "hello", "world", "mike", "chyson"
3 ("hello", "world", "mike", "chyson")
4 >>> "one",
5 ("one",)
```

When Python outputs a tuple it encloses it in parentheses. An empty tuple is created by using empty parentheses, (). The comma is also used to separate arguments in function calls, so if we want to pass a tuple literal as an argument we must enclose it in parentheses to avoid confusion.

```
1
2 [1, 2, 3]
3 []
```

One way to create a list is to use square brackets ([]). An empty list is created by using empty brackets, [].

Under the hood, lists and tuples don't store data items at all, but rather object references. When lists and tuples are created (and when items are inserted in the case of lists), they take copies of the object references they are given. In the case of literal items such as integers or strings, an object of the appropriate data type is created in memory and suitably initialized, and then an object reference referring to the object is created, and it is this object reference that is put in the list or tuple.

In procedural programming we can function and often pass in data items as arguments.

```
1
2 >>> len(("one",))
3 1
4 >>> len([1, 2, "hell", 3])
5 4
```

All Python data items are **objects** (also called **instances**) of a particular data type (also called a class).

```
1
2 >>> x = ["zebra", 49, -879, "aardvark", 200]
3 >>> x.append("more")
4 >>> x
5 ['zebra', 49, -879, 'aardvark', 200, 'more']
6
7 >>> list.append(x, "extra")
8 >>> x
9 ['zebra', 49, -879, 'aardvark', 200, 'more', 'extra']
```

Python has conventional functions called like this `function_name(arguments)`; and methods which are called like this `object_name.method_name(arguments)`.

The dot (“access attribute”) operator is used to access an object's attributes.

1.3 Logical operations

Python provides four sets of logical operations.

1.3.1 The identity operator

The `is` operator is a binary operator that returns `True` if its left-hand object reference is referencing to the same object as its right-hand object reference.

```
1 >>> a = ["Retention", 3, None]
2 >>> b = ["Retention", 3, None]
3 >>> a is b
4 False
5 >>> b = a
6 >>> a is b
7 True
```

One benefit of identity comparisons is that they are very fast. This is because the objects referred to do not have to be examined themselves. The `is` operator needs to compare only the memory addresses of the objects — the same address means the same object.

The most common use case for `is` is to compare a data item with the built-in null object, `None`.

The purpose of the identity operator is to see whether two object references refer to the same object, or to see whether an object is `None`. If we want to compare object values we should use a comparison operator instead.

1.3.2 Comparison operators

Python provides the standard set of binary comparison operators:

- `<`
- `<=`
- `==`
- `!=`
- `>=`
- `>`

These operators compare object values, that is, objects that the object references used in the comparison refer to.

In some cases, comparing the identity of two strings or numbers will return `True`, even if each has been assigned separately. This is because some implementations of Python will reuse the same object (since the value is the same and is immutable) for the sake of efficiency. The moral of this is to use `==` and

`!=` when comparing **values**, and to use `is` and `is not` only when comparing with `None` or when we really do want to see if two object references, rather than their values, are the same.

One particularly nice feature of Python's comparison operators is that they can be chained. For example:

```
1 >>> a = 9
2 >>> 0 <= a <= 10
3 True
4
```

This is a nicer way of testing that a given data item is in range than having to do two separate comparisons joined by logical `and`. It also has the additional virtue of evaluating the data item only once (since it appears once only in the expression), something that could make a difference if computing the data item's value is expensive, or if accessing the data item causes side effects.

1.3.3 The membership operator

For data types that are sequences or collections such as strings, lists, and tuples, we can test for membership using the `in` operator, and for nonmembership using the `not in` operator. For example:

```
1 >>> p = (4, "frog", 9, -33, 9, 2)
2 >>> 2 in p
3 True
4 >>> "dog" not in p
5 True
6
```

For lists and tuples, the `in` operator uses a linear search which can be slow for very large collections (tens of thousands of items or more). On the other hand, `in` is very fast when used on a dictionary or a set.

1.3.4 Logical operators

Python provides 3 logical operators:

- `and`
- `or`
- `not`

Both `and` and `or` use short-circuit logic and return the operand that determined the result – they do not return a Boolean (unless they actually have Boolean operands).


```
1
2 >>> five = 5
3 >>> two = 2
4 >>> zero = 0
5 >>> five and two
6 2
7 >>> two and five
8 5
9 >>> five and zero
10 0
```

If the expression occurs in a Boolean context, the result is evaluated as a Boolean, so the preceding expressions would come out as **True**, **True**, and **False**.

```
1
2 >>> nought = 0
3 >>> five or two
4 5
5 >>> two or five
6 2
7 >>> zero or five
8 5
9 >>> zero or nought
10 0
```

The **or** operator is similar; here the results in a Boolean context would be **True**, **True**, **True**, and **False**.

The **not** unary operator evaluates its argument in a Boolean context and always returns a Boolean result.

1.4 Control flow statements

A Boolean expression is anything that can be evaluated to produce a Boolean value (**True** or **False**). In Python, such an expression evaluate to **False** if it is the predefined constant **False**, the special object **None**, an empty sequence or collection, or a numeric data item of value 0; anything else is considered to be **True**.

In Python-speak a block of code, that is, a sequence of one or more statements, is called a **suite**. Because some of Python's syntax requires that a suite be present, Python provides the keyword **pass** which is a statement that does nothing and that can be used where a suite is required but where no preprocessing is necessary.

1.4.1 The if statement

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

Colons are used with `else`, `elif`, and essentially in any other place where a suite is to follow. Unlike most other programming languages, Python uses indentation to signify its block structure.

1.4.2 The while statement

```
while boolean_expression:
    suite
```

1.4.3 The for ... in statement

```
for variable in iterable:
    suite
```

The `variable` is set to refer to each object in the `iterable` in turn.

1.4.4 Basic exception handling

An exception is an object like any other Python object, and when converted to a string, the exception produces a message text. A simple form of the syntax for exception handlers is this:

```
try:
    try_suite
except exceptions1 as variable1:
    exception_suite1
...
except exceptionN as variableN:
    exception_suiteN
```

The `as variable` part is optional.

```
s = input("enter an integer: ")
try:
    i = int(s)
    print("valid integer entered:", i)
except ValueError as err:
    print(err)
```

1.5 Arithmetic operators

Four basic mathematical operations:

- +
- -
- *
- /

In addition, many Python data types can be used with augmented assignment operators such as:

- +=
- -=
- *=
- /=

The `+`, `-`, and `*` operators all behave as expected when both of their operands are integers. Where Python differs from the crowd is when it comes to division:

```
1
2 >>> 12/3
3 4.0
```

The division operator produces a floating-point value, not an integer. If we need an integer result, we can always convert using `int()` or use the truncating division operator `//`.

```
1
2 >>> a = 5
3 >>> a
4 5
5 >>> a += 8
6 >>> a
7 13
```

Comparing to C-like languages, there are two important subtleties, one Python-specific and one to do with augmented operators in any language.

The first point to remember is that the `int` data type is immutable. So, what actually happens behind the scenes when an augmented assignment operator is used on an immutable object is that the operation is performed, and an object holding the result is created; and then the target object reference is re-bound to refer to the result object rather than the object it referred to before.

The second subtlety is that `a operator= b` is not quite the same as `a = a operator b`. The augmented version looks at `a`'s value only once, so it is potentially faster.

```
1
2 >>> name = 'mike'
3 >>> name + 'chyson'
4 'mikechyson'
5 >>> name += 'chyson'
6 >>> name
7 'mike chyson'
8 >>> a = [1, 2, 3]
9
10 >>> a + [4]
11 [1, 2, 3, 4]
12 >>> a += [4]
13 >>> a
14 [1, 2, 3, 4]
```

Python overloads the `+` and `+=` operators for both strings and lists, the former meaning concatenation and the latter meaning append for strings and extend (append another list) for lists.

1.6 Input/output

Rediction:

- > (output)
- < (input)

Function:

- input()
- print()

1.7 Creating and calling functions

```
def function_name(arguments):  
    suite
```

The **arguments** are optional and multiple arguments must be comma-separated. Every Python function has a return value; this defaults to **None** unless we return from the function using the syntax **return value**, in which case **value** is returned.

def is a statement that works in a similar way to the assignment operator. When **def** is executed a function object is created and an object reference with the specified name is created and set to refer to the function object. Since functions are objects, they can be stored in collection data types and passed as arguments to other functions.

Although creating our own functions can be very satisfying, in many cases it is not necessary. This is because Python has a lot of functions built in, and a great many more functions in the modules in its standard library, so what we want may well already be available.

A Python module is just a **.py** file that contains Python code. To access the functionality in a module we must import it. For example:

```
1  
2 import sys
```

To import a module we use the **import** statement followed by the name of the **.py** file, but omitting the extension. Once a module has been imported, we can access any functions, classes, or variables that it contains. For example:

```
1  
2 print(sys.argv)
```

In general, the syntax for using a function from a module is `module_name.function_name(arguments)`. It makes use of the dot (access attribute) operator.

It is conventional to put all the import statements at the beginning of `.py` files, after the shebang line, and after the modules documentation. We recommend importing standard library modules first, then third-party library modules, and finally your own modules.

python template:

```
-----  
|shebang part                               |  
|(like /usr/bin/env python3)                |  
|-----|  
|  
|documentation part                        |  
|-----|  
|  
|import part                               |  
|  -----|  
|  | import standard library              |  
|  | import third-party library          |  
|  | import own library                  |  
|  -----|  
|-----|  
|  
|code                                       |  
|-----|
```

Chapter 2

Data types

2.1 Identifiers and keywords

The name we give to our object references are called **identifiers** or just plain **names**.

A valid Python identifier is a **nonempty** sequence of characters of any length that consists of a “start character” and zero or more “continuation characters”. Such an identifier must obey a couple of rules and ought to follow certain conventions.

Rules:

- The start character can be anything that Unicode considers to be a letter.
- Each continuation character can be any character that is permitted as a start character, or any nonwhitespace character.
- No identifier can have the same name as one of Python’s keywords.

Python has a built-in function called `dir()` that returns a list of object’s attributes.

Conventions:

- Don’t use the name of any of Python’s predefined identifiers for your own identifiers.
- Names that begin and end with two underscores should not be used.

2.2 Integral types

Python provides two built-in integral types, `int` and `bool`. Both integers and Boolean are immutable. When used in Boolean expressions, 0 and `False` are `False` and any other integer and `True` are `True`. When used in numerical expressions `True` evaluates to 1 and `False` to 0.

2.2.1 Integers

Integer literals are written using base 10 by default, but other number bases can be used.

```

1
2 >>> 14600926                # decimal
3 14600926
4 >>> 0b110111101100101011011110  # binary
5 14600926
6 >>> 0o67545336              # octal
7 14600926
8 >>> 0xDECADE                # hexadecimal
9 14600926

```

Binary numbers are written with a leading `0b`, octal numbers with a leading `0o`, and hexadecimal numbers with a leading `0x`. Uppercase letters can also be used.

All the usual mathematical functions and operators can be used with integers. Some of the functionality is provided by functions and other functionality is provided by `int` operators.

Provided by operators:

`x + y`

`x - y`

`x * y`

`x / y` Divides x by y ; always produces a `float` (or a `complex` if x or y is `complex`)

`x // y` Divides x by y ; truncates any fractional part so always produces an `int` result

`x % y`

`x ** y` Same to x^y

`-x` Negates x ;

+x Does nothing; is sometimes used to clarify code

x | y Bitwise OR

x ^ y Bitwise XOR

x & y Bitwise AND

i >> j Shifts **i** left by **j**; like **i * (2 ** j)** without overflow checking

i << j Shifts **i** right by **j**; like **i // (2 ** j)** without overflow checking

~i Inverts **i**'s bits

Provided by functions:

abs(x) Return the absolute value of x

divmod(x, y) Return the quotient and remainder of dividing x by y as a tuple of two **ints**

pow(x, y) Same to x^y

pow(x, y, z) A faster alternative to $(x * y) \% z$

round(x, n) Return x rounded to n integral digits if n is a negative **int** or return x rounded to n decimal places if n is a positive **int**; the returned value has the same type as x ;

bin(i) Return the binary representation of **int** i as a string

hex(i)

int(x) Convert object x to an integer

int(s, base) Convert **str** s to an integer. **base** should be an integer between 2 and 36 inclusive.

oct(i)

All the binary numeric operators (+, -, /, //, %, and **) have augmented assignment versions (+=, -=, /=, //=, %=, and **=) where **x op= y** is logically equivalent to **x = x op y** in the normal case when reading x 's value has no side effects.

All the binary bitwise operators (|, ^, &, <<, and >>) have augmented assignment versions (|=, ^=, &=, <<=, and >>=) where **i op= j** is logically

equivalent to `i = i op j` in the normal case when reading `i`'s value has no side effects.

When an object is created using its data types there are 3 possible use cases:

1. When a data type is called with no arguments an object with a default value is created.
2. When the data type is called with a single argument, if an argument of the same type is given, a new object which is a shallow copy of the original object is created. If argument of a different type is given, a conversion is attempted.
3. If two or more arguments are given — not all types support this, and for those that do the argument types and their meanings vary.

2.2.2 Booleans

There are two built-in Boolean objects: `True` and `False`.

2.3 Floating-point types

Python provides three kinds of floating-point values:

- the built-in `float`
- the built-in `complex`
- the `decimal.Decimal` type from the standard library

All three are immutable.

`decimal.Decimal` perform calculations that are accurate to the level of precision we specify (by default, to 28 decimal places) and can represent periodic numbers like 0.1 exactly; but processing is a lot slower than with `floats`. Because of their accuracy, `decimal.Decimal` numbers are suitable for financial calculations.

2.3.1 Floating-point numbers

```
1
2 import sys
3
4 # the smallest difference that the machine can distinguish
5 # between two floating-point numbers
6 sys.float_info.epsilon
7
8 help(sys.float_info)
```

The `math` module provides many functions that operate on `floats`. The `math` module is very dependent on the underlying math library that Python was compiled against. This means that some error conditions and boundary cases may behave differently on different platforms.

`math.copysign(x, y)` Returns `x` with `y`'s sign

`math.e`

`math.pi`

`math.exp(x)`

`math.factorial(x)` Returns $x!$

`math.floor(x)`

`math.ceil(x)`

`math.hypot(x, y)` Returns $\sqrt{x^2 + y^2}$

`math.isinf(x)` Returns `True` if `float x` is $\pm\infty$

`math.isnan(x)` Returns `True` if `float x` is nan (“not a number”)

`math.log(x, b)` Returns $\log_b x$

`math.log10(x)` Returns $\log_{10} x$

`math.sqrt(x)`

2.3.2 Complex numbers

The `complex` data type is an immutable type that holds a pair of `floats`, one representing the real part and the other the imaginary part of a complex number.

```

1
2 >>> z = 1.0 + 2.0j
3 >>> z.real, z.imag
4 (1.0, 2.0)

```

The functions in the `math` module do not work with complex numbers while `cmath` module does.

2.3.3 Decimal numbers

```

1
2 >>> import decimal
3 >>> a = decimal.Decimal(1234)
4 >>> b = decimal.Decimal('54321.012345678987654321')
5 >>> a + b
6 Decimal('55555.012345678987654321')

```

Numbers of type `decimal.Decimal` work within the scope of a **context**; the context is a collection of settings that affect how `decimal.Decimal` behave.

```

1
2 >>> 23 / 1.05
3 21.904761904761905
4 >>> print(23 / 1.05)
5 21.904761904761905
6 >>> print(decimal.Decimal(23) / decimal.Decimal(1.05))
7 21.90476190476190383546015179
8 >>> decimal.Decimal(23) / decimal.Decimal(1.05)
9 Decimal('21.90476190476190383546015179')

```

When we call `print()` on the result of `decimal.Decimal(23) / decimal.Decimal(1.05)` the bare number is printed — this output is in **string form**. If we simply enter the expression we get a `decimal.Decimal` output — this output is in **representational form**. All Python objects have two output forms. String form is designed to be human-readable. Representational form is designed to produce output that if fed to a Python interpreter would (when possible) reproduce the represented object.

2.4 Strings

Strings are represented by the immutable `str` data type which holds a sequence of Unicode characters.

triple quoted string:

```

1
2 text = '''hello world'''

```

Python uses newline as its **statement terminator**, except inside parentheses `()`, square brackets `[]`, braces `{}`, or triple quoted strings.

All of Python's escape sequences are shown in Table 2.1.

Escape	Meaning
<code>\newline</code>	Escape (i.e., ignore) the newline
<code>\\</code>	Backslash (<code>\</code>)
<code>\'</code>	Single quote (<code>'</code>)
<code>\"</code>	Double quote (<code>"</code>)
<code>\a</code>	ASCII bell (BEL)
<code>\b</code>	ASCII backspace (BS)
<code>\f</code>	ASCII formfeed (FF)
<code>\n</code>	ASCII linefeed (LF)
<code>\N{name}</code>	Unicode character with the given name
<code>\ooo</code>	Character with the given octal value
<code>\r</code>	ASCII carriage return (CR)
<code>\t</code>	ASCII tab (TAB)
<code>\uhhhh</code>	Unicode character with the given 16-bit hexadecimal value
<code>\Uhhhhhhhh</code>	Unicode character with the given 32-bit hexadecimal value
<code>\v</code>	ASCII vertical tab (VT)
<code>\xhh</code>	Character with the given 8-bit hexadecimal value

Figure 2.1: Python's string escapes

In some situations — for example, when writing regular expressions — we need to create strings with lots of literal backslashes. This can be inconvenient since each one must be escaped:

```

1
2 import re
3 phone1 = re.compile("^((?:[()\\d+])?\\s*\\d+(?:-\\d+)?)$")

```

The solution is to use **raw** strings. These are quoted or triple quoted strings whose first quote is preceded by the letter `r`. Inside such strings all characters are taken to be literals, so no escaping is necessary.

```

1
2 phone2 = re.compile(r"^(?:[() \d+])?\s*\d+(?:-\d+)?$")

```

```

1
2 >>> '\N{euro sign}'
3 ''

```

If we want to know the Unicode code point for a particular character in a string, we can use the built-in `ord()` function:

```

1
2 >>> ord('')
3 8364
4 >>> hex(ord(''))
5 '0x20ac'
6 >>> '\u20ac'
7 ''

```

we can convert any integer that represents a valid code point into the corresponding Unicode character using the built-in `chr()` function:

```

1
2 >>> chr(8734)
3 ''
4 >>> chr(8364)
5 ''
6 >>> ascii('')
7 "'\\u20ac'"

```

2.4.1 Comparing strings

Strings support the usual comparison operators `<`, `<=`, `==`, `!=`, `>`, and `>=`. These operators compare strings byte by byte in memory.

2.4.2 Slicing and striding strings

```

1
2 s = "Light ray"

```

Figure 2.2 shows all the valid index positions for string `s`.

The slice operator has three syntaxes:

```

seq[start]
seq[start:end]
seq[start:end:step]

```

Using `+` to concatenate and `+=` to append is not particularly efficient when String many strings are involved. For joining lots of strings it is usually best to use the `str.join()` method.

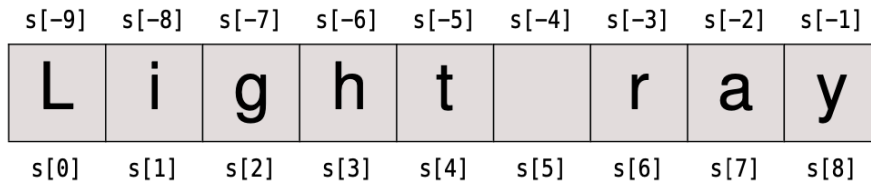


Figure 2.2: String index position

2.4.3 String operators and methods

Since strings are immutable sequences, all the functionality that can be used with immutable sequences can be used with strings.

- membership (in)
- concatenation (+)
- appedning (+=)
- replication (*)
- augmented assignment replication (*=)

There are some common string methods:

s.capitalize()

s.lower()

s.title()

s.upper()

s.swapcase()

s.center(width, char)

s.ljust(width, char)

s.rjust(width, char)

s.count(t, start, end)

s.encode(encoding, err)

`s.startswith(x, start, end)`

`s.endswith(x, start, end)`

`s.expandtabs(size)`

`s.find(t, start, end)`

`s.index(t, start, end)`

`s.format(...)`

`s.isalnum()`

`s.isalpha()`

`s.isdecimal()`

`s.isdigit()`

`s.isidentifier()`

`s.islower()`

`s.istitle()`

`s.isupper()`

`s.isnumeric()`

`s.isprintable()`

`s.isspace()`

`s.join(seq)`

`s.partition(t)`

`s.replace(t, u, n)`

`s.split(t, n)`

`s.splitlines(f)`

`s.strip(chars)`

`s.maketrans()`

`s.translate()`

`s.zfill(w)`

2.4.4 String formatting with the `str.format()` method

The `str.format()` method returns a new string with the replacement fields in its string replaced with its arguments suitably formatted.

```
1
2 >>> "{0} {1} {2}".format("Hello", "world", "mike")
3 'Hello world mike'
```

Each replacement field is identified by a field name in braces. If the field name is a simple integer, it is taken to be the index position of one of the arguments passed to `str.format()`.

If we need to include braces inside format strings, we can do so by doubling them up.

```
1
2 >>> 'just {{0}}'.format('brace')
3 'just {brace}'
```

The replacement field can have any of the following general syntaxes:

```
{field_name}
{field_name!conversion}
{field_name:format_specification}
{field_name!conversion:format_specification}
```

2.4.5 Field names

A field name can be either an integer corresponding to one of the `str.format()` methods arguments, or the name of one of the methods keyword arguments.

```
1
2 >>> '{who} solve a leetcode problem every {0} days'.format(1, who='Mike')
3 'Mike solve a leetcode problem every 1 days'
```

Notice that in an argument list, keyword arguments always come after positional arguments.

If the arguments are collections data types like lists or dictionaries, or have attributes, we can access the part using `[]` or `.` notation.

```
1
2 >>> stock = ["paper", "envelopes", "notepads"]
3 >>> "We have {0[1]} and {0[2]} in stock".format(stock)
4 'We have envelopes and notepads in stock'
5
6 >>> d = dict(animal="elephant", weight=12000)
7 >>> "The {0[animal]} weighs {0[weight]} kg".format(d)
8 'The elephant weighs 12000kg'
9
10 >>> "math.pi=={0.pi}".format(math)
11 'math.pi==3.14159265359'
```

The local variables that are currently in scope are available from the built-in `locals()` function. This function returns a dictionary whose keys are local variable names and whose values are references to the variables' values. We can use **mapping unpacking** to feed this dictionary into the `str.format()` method. The mapping unpacking operator is `**` and it can be applied to a mapping (such as dictionary) to produce a key-value list suitable for passing to a function. For example:

```

1
2 >>> element = "Silver"
3 >>> number = 47
4 >>> "Element {number} is {element}".format(**locals())
5 'Element 47 is Silver'
```

2.4.6 conversions

```

1
2 >>> decimal.Decimal("3.4084")
3 Decimal('3.4084')
4 >>> print(decimal.Decimal("3.4084"))
5 3.4084
```

The first is in representational form. The purpose of this form is to provide a string which if interpreted by Python would re-create the object it represents. Not all objects can provide a reproducing representation, in which case they provide a string enclosed in angle brackets. For example `"module 'sys' (built-in)>".`

The second is in its string form. This form is aimed at human readers, so the concern is to show something that makes sense to people. If a data type doesn't have a string form and a string is required, Python will use the representational form.

Python's built-in data types know about `str.format()`, and when passed as an argument to this method they return a suitable string to display themselves. In addition, it is possible to override the data types normal behavior and force it to provide either its string or its representational form. This is done by adding a conversion specifier to the field. Currently there are three such specifiers:

- **s** to force string form,
- **r** to force representational form
- **a** to force representational form but only using ASCII characters.

```

1
2 >>> "{0} {0!s} {0!r} {0!a} {1!r} {1!a}".format(decimal.Decimal(1), "")
3 "1 1 Decimal('1') Decimal('1') '' '\\u4f60\\u597d'"

```

2.4.7 Format specifications

Specification for string

For strings, the things that we can control are:

- the fill character,
- the alignment within the field, and
- the minimum and
- maximum field widths.

```
: fill      align      min_width  .max_width
      < for left
      > for right
      ^ for center
```

```
1
2 >>> s = "The sword of truth"
3 >>> '{0}'.format(s)
4 'The sword of truth'
5 >>> '{0:25}'.format(s)
6 'The sword of truth'
7 >>> '{0:>25}'.format(s)
8 '      The sword of truth'
9 >>> '{0:->25}'.format(s)
10 '-----The sword of truth'
11 >>> '{0:.<25}'.format(s) # the left alignment can not be omitted
12 'The sword of truth.....'
13 >>> '{0:.10}'.format(s)
14 'The sword'
```

Specification for integer

For integers, the format specification allows us to control:

- the fill character,
- the alignment within the field,
- the sign,
- whether to use a nonlocale-aware comma separator to group digits,
- the minimum field width, and
- the number base.

:	fill	alignment	sign	#	width	,	type
	=	pad between	+ force sign;	prifix		use	b,c,d
		sign and	- sign if	ints		commas	n,o,x,
		digits	needed;	with		for	X
		for numbers	" " space or	0b, 0o,		grouping	
			- as	or 0x			
			appropriate				

```

1 >>> '{0:0=12}'.format(-1234)
2 '-00000001234'
3
4 >>> ' [{0: } ] [ {1: } ]'.format(539802, -539802) # space or - sign
5 '[ 539802] [-539802]'
6 >>> ' [{0:+} ] [ {1:+} ]'.format(539802, -539802) # force sign
7 '[+539802] [-539802]'
8 >>> ' [{0:-} ] [ {1:-} ]'.format(539802, -539802) # - sign if needed
9 '[539802] [-539802]'
10
11 >>> '{0:b} {0:o} {0:x} {0:X}'.format(123)
12 '1111011 173 7b 7B'
13 >>> '{0:#b} {0:#o} {0:#x} {0:#X}'.format(123)
14 '0b1111011 0o173 0x7b 0X7B'
15
16 >>> '{0:,}'.format(1234567890)
17 '1,234,567,890'
18

```

The last format character `n` has the same effect as `d` when given an integer. What makes `n` special is that it respects the current locale and will use locale-specific decimal separator and grouping separator in the output it produces. The default locale is called the C locale, and for this the decimal and grouping characters are a period and an empty string.

```

1 >>> import locale
2 >>> x = 1234567890
3 >>> locale.setlocale(locale.LC_ALL, 'C')
4 'C'
5 >>> '{:n}'.format(x)
6 '1234567890'
7 >>> locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')
8 'en_US.UTF-8'
9 >>> '{:n}'.format(x)
10 '1,234,567,890'
11 >>> locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
12 'de_DE.UTF-8'
13 >>> '{:n}'.format(x)
14 '1234567890'
15

```

Specification for floating

For floating-point numbers, the format specification gives us control over:

- the fill character,

- the alignment within the field,
- the sign,
- whether to use a non-locale aware comma separator to group digits,
- the minimum field width,
- the number of digits after the decimal place, and
- whether to present the number in standard or exponential form, or as a percentage.

```

: fill alignment sign width , .precision      type
                                number of      e,E,f,
                                decimal places  g,G,n,
                                                %

```

- e for exponential form with lowercase e
- E for exponential form with lowercase E
- f for standard floating-point form
- g for “general” form this is the same as f unless the number is very large, in which case it is the same as e
- G is almost the same as g, but uses either f or E
- % for percentage

```

1
2 >>> '{0:12.2e}'.format(math.pi)
3 '      3.14e+00'
4 >>> '{0:12.2f}'.format(math.pi)
5 '      3.14'
6 >>> '{:,.6f}'.format(1234567890.1234567890)
7 '1,234,567,890.123457'
8 >>> '{:,.4f}'.format(3.59284e6-8.984327843e6j)
9 '3,592,840.0000-8,984,327.8430j'

```

Character encodings

Unicode assigns every character to an integer — called a **code point** in Unicode-speak. Nowadays, Unicode is usually stored both on disk and in memory using UTF-8, UTF-16, or UTF-32. The first of these, UTF-8, is backward compatible with 7-bit ASCII since its first 128 code points are represented by single-byte values that are the same as the 7-bit ASCII character values. To represent all the other Unicode characters, UTF-8 uses two, three, or more bytes per character.

A lot of other software, such as Java, uses UCS-2 (which in modern form is the same as UTF-16). This representation uses two or four bytes per character, with the most common characters represented by two bytes. The UTF-32 representation (also called UCS-4) uses four bytes per character. Using UTF-16 or UTF-32 for storing Unicode in files or for sending over a network connection has a potential pitfall: If the data is sent as integers then the endianness matters. One solution to this is to precede the data with a byte order mark so that readers can adapt accordingly. This problem doesn't arise with UTF-8, which is another reason why it is so popular.

Python represents Unicode using either UCS-2 (UTF-16) format, or UCS-4 (UTF-32) format. In fact, when using UCS-2, Python uses a slightly simplified version that always uses two bytes per character and so can only represent code points up to 0xFFFF. When using UCS-4, Python can represent all the Unicode code points. The maximum code point is stored in the read-only `sys.maxunicode` attribute; if its value is 65535, then Python was compiled to use UCS-2; if larger, then Python is using UCS-4.

Chapter 3

Collection data types

3.1 Sequence types

A **sequence** type is one that support:

- the membership operator (`in`)
- the size function (`len()`)
- slices (`[]`)
- and is iterable.

Python provides five built-in sequence types:

- `bytearray`
- `bytes`
- `list`
- `str`
- `tuple`

When iterated, all of these sequences provide their items in order.

3.1.1 Tuples

Tuples are immutable. Tuples are able to hold any items of any data type, including collection types such as tuples and lists, since what they really hold are object references.

Tuples provide just two methods, `t.count(x)` and `t.index(x)`.

tuple coding style: omit parentheses:

- tuples on the left-hand side of a binary operator
- on the right-hand side of a unary statement

other cases with parentheses.

```
1
2 a, b = (1, 2) # left of binary operator
3 del a, b     # right of unary operator
```

When we have a sequences on the right-hand side of an assignment, and we have a tuple on the left-hand side, we say that the right-hand side has been **unpacked**. Sequence unpacking can be used to swap values, for example:

```
1
2 a, b = (b, a) # or a, b = b, a
3 # the parentheses here are for code style
4
5 for x, y in ((3, 4), (5, 12), (28, -45)):
6     print(math.hypot(x, y))
```

3.1.2 Named tuples

A named tuple behaves just like a plain tuple, and has the same performance characteristics. What it adds is the ability to refer to items in the tuple by name as well as by index position.

```
1
2 import collections
3
4 Fullname = collections.namedtuple('Fullname',
5                                   'firstname middlename lastname')
6
7 persons = []
8 persons.append(Fullname('Mike', 'Ming', 'Chyson'))
9 persons.append(Fullname('Alfred', 'Bernhard', 'Nobel'))
10 for person in persons:
11     print('{firstname} {middlename} {lastname}'.format(**person._asdict()))
```


3.1.3 Lists

List are mutable. Since all the items in a list are really object references, lists can hold items of any data type, including collection types such as lists and tuples.

Although we can use the slice operator to access items in a list, in some situations we want to take two or more pieces of a list in one go. This can be done by sequence unpacking. Any iterable (lists, tuples, etc.) can be unpacked using the sequence unpacking operator, an asterisk or star (*). When used with two or more variables on the left-hand side of an assignment, one of which is preceded by *, items are assigned to the variables, with all those left over assigned to the starred variable. Here are some examples:

```
1 >>> a = list(range(10))
2
3 >>> first, *last = a
4 >>> print(first, last)
5 0 [1, 2, 3, 4, 5, 6, 7, 8, 9]
6 >>>
7 >>> first, *middle, last = a
8 >>> print(first, middle, last)
9 0 [1, 2, 3, 4, 5, 6, 7, 8] 9
10 >>>
11 >>> *first, last = a
12 >>> print(first, last)
13 [0, 1, 2, 3, 4, 5, 6, 7, 8] 9
```

List methods:

list.append(x)

list.count(x)

list.extend(m)

list += m

list.index(x, start, end)

list.insert(i, x)

list.pop() Returns and removes the rightmost item of list

list.pop(i)

list.remove(x) Removes the leftmost occurrence of item x from list

list.reverse() Reverses list in-place

list.sort(...) Sorts list in-place

Individual items can be replaced in a list by assigning to a particular index position. Entire slices can be replaced by assigning an iterable to a slice. The slice and the iterable don't have to be the same length. In all cases, the slice's items are removed the the iterable's items are inserted.

```

1 >>> nums = [0, 1, 2, 3, 4, 5, 6]
2 >>> nums[0] = 100
3 >>> nums
4 [100, 1, 2, 3, 4, 5, 6]
5 >>> nums[2:2] = [200] # same to nums.insert(2, 200)
6 >>> nums
7 [100, 1, 200, 2, 3, 4, 5, 6]
8 >>> nums[2:4] = [10, 11, 12, 13, 14]
9 >>> nums
10 [100, 1, 10, 11, 12, 13, 14, 3, 4, 5, 6]
11

```

In lists, striding allows us to access every n-th item which can often be useful. For example:

```

1 >>> x = list(range(1, 11))
2 >>> x
3 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4 >>> x[1::2] = [0] * len(x[1::2])
5 >>> x
6 [1, 0, 3, 0, 5, 0, 7, 0, 9, 0]
7

```

```

1 >>> x = list(range(-5, 5))
2 >>> x
3 [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
4 >>> x.sort(key=lambda x: x**2)
5 >>> x
6 [0, -1, 1, -2, 2, -3, 3, -4, 4, -5]

```

For inserting items, lists perform best when items are added or removed at the end (`list.append()`, `list.pop()`). The worst performance occurs when we search for items in a list, for example, using `list.remove()` or `list.index()`, or using `in` for membership testing. If fast searching or membership testing is required, a `set` or a `dict` may be a more suitable collection choice. Alternatively, lists can provide fast searching if they are kept in order by sorting them and using a binary search (provided by the `bisect` module), to find items.

3.1.4 List comprehensions

A **list comprehension** is an expression and a loop with optional condition enclosed in brackets where the loop is used to generate items for the list, and where the condition can filter out unwanted items.

```
[expression for item in iterable]
```

```
[expression for item in iterable if condition]
```

```
1 leaps = [y for y in range(1900, 1940)
2           if (y % 4) == 0 and y % 100 != 0) or (y % 400 == 0)]
```

If the generated list is very large, it may be more efficient to generate each item as it is needed rather than produce the whole list at once. This can be achieved by using a generator rather than a list comprehension.

3.2 Set types

A **set** is a collection data type that supports:

- the membership operator (**in**),
- the size function (**len()**),
- and is iterable.

Python provides two built-in set types:

- the mutable **set** type
- the immutable **frozenset**

When iterated, set types provide their items in an arbitrary order.

Only **hashable** objects may be added to a set. Hashable objects are objects which have a **__hash__()** special method whose return value is always the same throughout the object's lifetime, and which can be compared for equality using the **__eq__()** special method. (Special methods are methods whose name begins and ends with two underscores)

3.2.1 Sets

A **set** is an unordered collection of zero or more object references that refer to hashable objects. Sets are mutable. Sets always contain unique items — adding duplicate items is safe but pointless.

Set methods:

s.add(x)

s.clear()

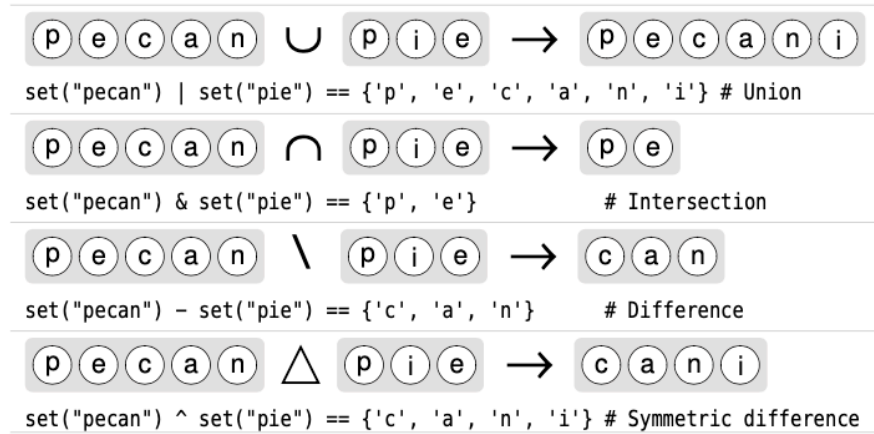


Figure 3.1: Standard set operators

s.copy()**s.difference(t)** Same to `s - t`**s.difference_update(t)** Same to `s -= t`**s.discard** Removes item `x` from set `s` if it is in `s`**s.remove()** Removes item `x` from set `s`, or raises a `KeyError` exception if `x` is not in `s`**s.intersection(t)** Same to `s & t`**s.intersection_update(t)** Same to `s &= t`**s.isdisjoint(t)** Returns `True` if sets `s` and `t` have no items in common**s.issubset(t)** Same to `s <= t`**s.issuperset(t)** Same to `s >= t`**s.pop()** Returns and remove a random item from set `s`, or raises a `KeyError` exception if `s` is empty**s.symmetric_difference(t)** Same to `s ^ t`**s.symmetric_difference_update(t)** Same to `s ^= t`

s.union(t) Same to `s | t`

Sets are used for fast membership test and removing duplicated items.

3.2.2 Set comprehensions

```
{expression for item in iterable}  
{expression for item in iterable if condition}
```

3.2.3 Frozen sets

A frozen set is a set that, once created, cannot be changed. Since frozen sets are immutable, sets and frozen sets can contain frozen sets.

3.3 Mapping types

A **mapping** type is one that supports:

- the membership operator (`in`)
- the size function (`len()`)
- is iterable

Mappings are collection of key-value items and provide methods for accessing items and their keys and values.

When iterated, unordered mapping types provide their items in an arbitrary order.

There are one built-in mapping types and two standard library's mapping types:

- `dict`
- `collections.defaultdict`
- `collections.OrderedDict`

3.3.1 Dictionaries

A **dict** is an unordered collection of zero or more keyvalue pairs whose keys are object references that refer to hashable objects, and whose values are object references referring to objects of any type. Dictionaries are mutable.

```

1 >>> d = dict()
2 >>> d
3 {}
4 >>> d = {}
5 >>> d
6 {}
7 >>> d = {"hello": 1, "world": 2}
8 >>> d
9 {'hello': 1, 'world': 2}
10 >>> d = dict(hello=1, world=2)
11 >>> d
12 {'hello': 1, 'world': 2}
13 >>> d = dict([('hello', 1), ('world', 2)])
14 >>> d
15 {'hello': 1, 'world': 2}

```

Dictionary methods:

d.clear()

d.copy()

d.fromkeys(s, v) Returns a dict whose keys are the items in sequence s and whose values are None or v if v is given

d.get(k) Returns key k's associated value or None if k isn't in dict d

d.get(k, v) Returns key k's associated value, or v if k isn't in dict d

d.items()

d.keys()

d.values()

d.pop(k)

d.pop(k, v)

d.popitem() Returns and removes an arbitrary (key, value) pair from dict d

d.setdefault(k, v) The same as the dict.get() method, except that if the key is not in dict d, a new item is inserted with the key k, and with a value of None or v if v is given

d.update(a) Adds every (key, value) pair from a that isn't in dict d to d, and for every key that is in both d and a, replaces the corresponding value in d with the one in a – a can be a dictionary, an iterable of (key, value) pairs, or keyword arguments

The `dict.items()`, `dict.keys()`, and `dict.values()` methods all return dictionary views. A dictionary view is effectively a read-only iterable object that appears to hold the dictionary's items or keys or values, depending on the view we have asked for.

In general, we can simply treat views as iterables. However, two things make a view different from a normal iterable. One is that if the dictionary the view refers to is changed, the view reflects the change. The other is that key and item views support some set-like operations. Given dictionary view `v` and `set` or dictionary view `x`, the supported operations are:

```
v & x # intersection
v | x # union
v - x # difference
v ^ x # symmetric difference
```

```
1 >>> d = {}.fromkeys('abcd', 3)
2 >>> d
3 {'a': 3, 'b': 3, 'c': 3, 'd': 3}
4 >>> s = set('abc')
5 >>> s
6 {'a', 'c', 'b'}
7 >>> d.keys() & s
8 {'a', 'c', 'b'}
9 >>>
10 >>> d
11 {'a': 3, 'b': 3, 'c': 3, 'd': 3}
12 >>> d.setdefault('a')
13 3
14 >>> d
15 {'a': 3, 'b': 3, 'c': 3, 'd': 3}
16 >>> d.setdefault('z', 100)
17 100
18 >>> d
19 {'a': 3, 'b': 3, 'c': 3, 'd': 3, 'z': 100}
```

3.3.2 Dictionary comprehensions

```
1 {keyexpression: valueexpression for key, value in iterable}
2 {keyexpression: valueexpression for key, value in iterable if condition}
```

```
1 import os
2
3 # filename: filesize
4 d = {name: os.path.getsize(name) for name in os.listdir('.') if os.path.isfile(name)}
5 print(d)
```

```

6
7 # revert dict
8 inserted_d = {v: k for k, v in d.items()}
9 print(inserted_d)

```

3.3.3 Default dictionaries

Default dictionaries are dictionaries — they have all the operators and methods that dictionaries provide. What makes default dictionaries different from plain dictionaries is the way they handle missing keys.

When a default dictionary is created, we can pass in a **factory function**. A factory function is a function that, when called, returns an object of a particular type. All of Python's built-in data types can be used as factory functions. The factory function passed to a default dictionary is used to create default values for missing keys.

Note that the **name** of a function is an object reference to the function — so when we want to pass functions as parameters, we just pass the name. When we use a function with parentheses, the parentheses tell Python that the function should be called.

```

1 >>> words = collections.defaultdict(int)
2 >>> words
3 defaultdict(<class 'int'>, {})
4 >>> words['hello'] += 1
5 >>> words
6 defaultdict(<class 'int'>, {'hello': 1})
7 >>> words['hello'] += 1
8 >>> words
9 defaultdict(<class 'int'>, {'hello': 2})
10 >>>
11 >>> de = collections.defaultdict(lambda : 'Thanks to ')
12 >>> de
13 defaultdict(<function <lambda> at 0x7fa999a75a60>, {})
14 >>> de['Mike'] += 'Mike'
15 >>> de
16 defaultdict(<function <lambda> at 0x7fa999a75a60>, {'Mike': 'Thanks to Mike'})

```

3.3.4 Ordered dictionaries

The ordered dictionaries type is `collections.OrderedDict`. Ordered dictionaries store their items in the order in which they were inserted. If we change an item's value, the order is not changed.

```

1 d = collections.OrderedDict([('z', -4), ('e', 19), ('k', 7)])
2 for k in d:
3     print(k, d[k])
4
5 tasks = collections.OrderedDict()

```



```

6 tasks[8031] = "Backup"
7 tasks[4027] = "Scan Email"
8 tasks[5733] = "Build System"
9 for k in tasks:
10     print(k, tasks[k])

```

If we want to move an item to the end, we must delete it and then reinsert it. We can also call `popitem()` to remove and return the last keyvalue item in the ordered dictionary; or we can call `popitem(last=False)`, in which case the first item will be removed and returned.

3.4 Iterating and copying collections

3.4.1 Iterators and iterable operations and functions

An **iterable** data type is one that can return each of its items one at a time. Any object that has an `__iter__()` method, or any sequence (i.e. an object that has a `__getitem__()` method taking integer arguments starting from 0) is an iterable and can be provide an **iterator**. An iterator is an object that provides a `__next__()` method which returns each successive item in turn, and raises a `StopIteration` exception when there are no more items.

The operators and functions that can be used with iterables:

s + t Returns a sequence that is the concatenation of sequences s and t

s * t Returns a sequences that is int n concatenation of sequences s

x in i Returns True if item x is in iterable i

all(i) Returns True if every item in iterable i evaluateates to True

any(i) Returns True if any item in iterable i evaluateates to True

enumerate(i, start) Normally used in for ... in loops to provide a sequence of (index, item) tuples with indexes starting at 0 or start

len(x)

max(i, key) Returns the biggest item in iterable i or the item with the biggest `key(item)` value if a key function is given

min(i, key) Returns the smallest item in iterable i or the item with the smallest `key(item)` value if a key function is given

range(start, stop, step) Returns an integer iterator.

reversed(i) Returns an iterator that returns the items from iterator *i* in reverse order

sorted(i, key, reverse) Return a list of the items from iterator *i* in sorted order; *key* is used to provide DSU (Decorate, Sort, Undecorate) sorting. If *reverse* is *True* the sorting is done in reverse order.

sum(i, start) Returns the sum of the items in iterable *i* plus *start* (which defaults to 0)

zip(i1, ..., iN) Returns an iterator of tuples using the iterators *i1* to *iN*

The order in which items are returned depends on the underlying iterable. In the case of lists and tuples, items are normally returned in sequential order starting from the first item (index position 0), but some iterators return the items in an arbitrary order — for example, dictionary and set iterators.

The built-in `iter()` function has two quite different behaviors.

- When given a collection data type or a sequence it returns an iterator for the object it is passed — or raise a `TypeError` if the object cannot be iterable.
- When given a callable (a function or method) and a sentinel value, the function passed in is called once at each iteration, returning the function's return value each time, or raising a `StopIteration` exception if the return value equals the sentinel.

When we use a `for item in iterable` loop, Python in effect calls `iter(iterable)` to get an iterator. This iterator's `__next__()` method is then called at each loop iteration to get the next item, and when the `StopIteration` exception is raised, it is caught and the loop is terminated.

```

1  # manner 1
2  product = 1
3  for i in [1, 2, 4, 8]:
4      product *= i
5  print(product)
6
7
8  # manner 2
9  product = 1
10 i = iter([1, 2, 4, 8])
11 while True:
12     try:
13         product *= i
14     except StopIteration:
15         break
16 print(product)

```

Any (finite) iterable, `i`, can be converted into a tuple by calling `tuple(i)`, or can be converted into a list by calling `list(i)`.

```

1 >>> x = []
2 >>> for t in zip(range(-10, 0, 1), range(0, 10, 2), range(1, 10, 2)):
3 ...     x += t
4 ...
5 >>> x
6 [-10, 0, 1, -9, 2, 3, -8, 4, 5, -7, 6, 7, -6, 8, 9]
7 >>> sorted(x)
8 [-10, -9, -8, -7, -6, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
9 >>> sorted(x, reverse=True)
10 [9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -6, -7, -8, -9, -10]
11 >>> sorted(x, key=abs)
12 [0, 1, 2, 3, 4, 5, 6, -6, -7, 7, -8, 8, -9, 9, -10]
```

A function's name is an object reference to the function; it is the parentheses that follow the name that tell Python to call the function.

Python's sort algorithm is an adaptive stable mergesort that is both fast and smart, and it is especially well optimized for partially sorted lists. The “adaptive” part means that the sort algorithm adapts to circumstances — for example, taking advantage of partially sorted data. The “stable” part means that the items that sort equally are not moved in relation to each other. When sorting collections of integers, strings, or other simple types their “less than” operator (`<`) is used. Python can sort collections that contain collections, working recursively to any depth.

Lists can be sorted in-place using the `list.sort()` method, which takes the same optional arguments as `sorted()`.

3.4.2 Copying collections

Since Python uses **object references**, when we use the assignment operator (`+`), no copying takes place. If the right-hand operand is a literal such as a string or a number, the left-hand operand is set to be an object reference that refers to the in-memory object that holds the literal's value. If the right-hand operand is an object reference, the left-hand operand is set to be an object reference that refers to the same object as the right-hand operand. One consequence of this is that assignment is very efficient.

For sequences, when we take a slice, the slice is always an independent copy of the items copied. For dictionaries and sets, copying can be achieved using `dict.copy()` and `set.copy()`. In addition, the `copy` module provides the `copy.copy()` function that returns a copy of the object it is given. Another

way to copy the built-in collection types is to use the type as a function with the collection to be copied as its argument.

Note, though, that all of these copying techniques are **shallow** — that is, only object references are copied and not the object themselves. For immutable data types like numbers and strings this has the same effect as copying, but for mutable data types such as nested collections this means that the object they refer to are referred to both by the original collection and by the copied collection.

```

1      print('{:.^50}'.format('print(x,y)'))
2      x = [53, 68, ['A', 'B', 'C']]
3      y = x[:]
4      print(x, y, sep='\n')
5
6      print('{:.^50}'.format('print(x,y)'))
7      y[1] = 40
8      x[2][0] = 'Q'
9      print(x, y, sep='\n')
10
11     """
12     ..... print(x,y) .....
13     [53, 68, ['A', 'B', 'C']]
14     [53, 68, ['A', 'B', 'C']]
15     ..... print(x,y) .....
16     [53, 68, ['Q', 'B', 'C']]
17     [53, 40, ['Q', 'B', 'C']]
18     """

```

If we really need independent copies of arbitrarily nested collections, we can deep-copy:

```

1      import copy
2
3      x = [53, 68, ['A', 'B', 'C']]
4      y = copy.deepcopy(x)
5      y[1] = 40
6      x[2][0] = 'Q'
7      print('{:.^50}'.format('print(x,y)'))
8      print(x, y, sep='\n')
9
10     """
11     ..... print(x,y) .....
12     [53, 68, ['Q', 'B', 'C']]
13     [53, 40, ['A', 'B', 'C']]
14     """

```

Chapter 4

Control structures and functions

4.1 Control structures

4.1.1 Conditional branching

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

conditional expression:

```
expression1 if boolean_expression else expression2
```

One common programming pattern is to set a variable to a default value, and then change the value if necessary.

```
1 width = 100 + (10 if margin else 10)
2 print('{ } file { } '.format(count if count != 0 else 'no', 's' if count != 1 else ''))
```

4.1.2 Looping

while loops

```
while boolean_expression:
    while_suite
else:
    else_suite
```

As long as the `boolean_expression` is `True`, the `while` block's suite is executed. If the `boolean_expression` is or becomes `False`, the loop terminates, and if the optional `else` clause is present, its suite is executed. If the loop does not terminate normally, any optional `else` clause's suite is skipped. That is, if the loop is broken out of due to a `break` statement, or a `return` statement, or if an exception is raised, the `else` clause's suite is not executed.

```
1 i = 0
2 while i < 100:
3     i += 1
4 else:
5     last = i
6 print(last) # 100
```

```
1 def list_find(lst, target):
2     """
3     Find the first target's index or -1 if not found.
4
5     :param lst:
6     :param target:
7     :return: index of the target if found or -1 if not found
8     """
9     index = 0
10    while index < len(lst):
11        if lst[index] == target:
12            break
13        index += 1
14    else:
15        index = -1
16    return index
```

for loops

```
for expression in iterable:
    for_suite
else:
    else_suite
```

The rule to run `else_suite` is same for while loop.

```
1 def list_find2(lst, target):
2     for index, x in enumerate(lst):
3         if x == target:
4             break
5     else:
6         index = -1
7     return index
```

4.2 Exception handling

4.2.1 Catching and raising exceptions

```
try:
    try_suite
except exception_group1 as variable1:
    except_suite1
...
except exception_groupN as variableN:
    except_suiteN
else:
    else_suite
finally:
    finally_suite
```

There must be at least one **except** block, but both the **else** and the **finally** blocks are optional. The **else** blocks suite is executed when the **try** blocks suite has finished normally — but it is not executed if an exception occurs. If there is a **finally** block, it is always executed at the end.

Each **except** clauses exception group can be a single exception or a parenthesized tuple of exceptions.

If an exception occurs in the **try** blocks suite, each **except** clause is tried in turn. If the exception matches an exception group, the corresponding suite is executed. To match an exception group, the exception must be of the same type as the (or one of the) exception types listed in the group, or the same type as the (or one of the) groups exception types subclasses.

```
1 def list_find(lst, target):
2     try:
3         index = lst.index(target)
4     except ValueError:
5         index = -1
6     return index
```

Python offers a simpler `try...finally` block:

```
try:
    try_suite
finally:
    finally_suite
```

```
1 # remove black lines
2 def read_data(filename):
3     lines = []
4     fh = None
5     try:
6         fh = open(filename)
7         for line in fh:
8             if line.strip():
9                 lines.append(line)
10    except (IOError, OSError) as err:
11        print(err)
12        return []
13    finally:
14        if fh is not None:
15            fh.close()
16    return lines
```

Raising exceptions

Exceptions provide a useful means of changing the flow of control.

There are three syntaxes for raising exceptions:

```
raise exception(args)
raise exception(args) from original_exception
raise
```

If we give the exception some text as its argument, this text will be output if the exception is printed when it is caught. When the third syntax is used, `raise` will reraise the currently active exception — and if there isn't one it will raise a `TypeError`.

4.2.2 Custom exceptions

Custom exceptions are custom data types (classes).

```
class exceptionName(baseException): pass
```

The base class should be `Exception` or a class that inherits from `Exception`. One use of custom exceptions is to break out of deeply nested loops.


```

1  def find_word(table, target):
2      found = False
3      for row, record in enumerate(table):
4          for column, field in enumerate(record):
5              for index, item in enumerate(field):
6                  if item == target:
7                      found = True
8                      break
9              if found:
10                 break
11         if found:
12             break
13
14     if found:
15         print('found at ({}, {}, {})'.format(row, column, index))
16     else:
17         print('not found')
18
19
20 def find_word2(table, target):
21     class FoundException(Exception):
22         pass
23
24     try:
25         for row, record in enumerate(table):
26             for column, field in enumerate(record):
27                 for index, item in enumerate(field):
28                     if item == target:
29                         raise FoundException
30     except FoundException:
31         print('found at ({}, {}, {})'.format(row, column, index))
32     else:
33         print('not found')

```

BaseException

```

+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |   +-- ModuleNotFoundError
    +-- LookupError

```

```
|    +--- IndexError
|    +--- KeyError
+--- MemoryError
+--- NameError
|    +--- UnboundLocalError
+--- OSError
|    +--- BlockingIOError
|    +--- ChildProcessError
|    +--- ConnectionError
|    |    +--- BrokenPipeError
|    |    +--- ConnectionAbortedError
|    |    +--- ConnectionRefusedError
|    |    +--- ConnectionResetError
|    +--- FileExistsError
|    +--- FileNotFoundError
|    +--- InterruptedError
|    +--- IsADirectoryError
|    +--- NotADirectoryError
|    +--- PermissionError
|    +--- ProcessLookupError
|    +--- TimeoutError
+--- ReferenceError
+--- RuntimeError
|    +--- NotImplementedError
|    +--- RecursionError
+--- SyntaxError
|    +--- IndentationError
|    +--- TabError
+--- SystemError
+--- TypeError
+--- ValueError
|    +--- UnicodeError
|    |    +--- UnicodeDecodeError
|    |    +--- UnicodeEncodeError
|    |    +--- UnicodeTranslateError
+--- Warning
|    +--- DeprecationWarning
```

```
--- PendingDeprecationWarning
--- RuntimeWarning
--- SyntaxWarning
--- UserWarning
--- FutureWarning
--- ImportError
--- UnicodeWarning
--- BytesWarning
--- ResourceWarning
```

4.3 Costom functions

Functions are a means by which we can package up and parameterize functionality. Four kinds of functions can be created in Python:

- global functions
- local functions
- lambda functions
- methods

Global objects (including functions) are accessible to any code in the same module (i.e., the same .py file) in which the object is created. Global objects can also be accessed from other modules.

Local functions (also called nested functions) are functions that are defined inside other functions. These functions are visible only to the function where they are defined.

Lambda functions are expressions, so they can be created at their point of use; however, they are much more limited than normal functions.

Methods are functions that are associated with a particular data types and can be used only in conjunction with the data type.

The general syntax for creating a (global or local) function is:

```
def function_name(parameters):
    suite
```

```
1 def my_sum(a, b, c=1):
2     return a + b + c
```

```
3
4
5 print(my_sum(1, 2, 3)) # 6
6 print(my_sum(1, 2))    # 4
```

a, b is called **positional arguments**, because each argument passed is set as the value of the parameter in the corresponding position. **c** is called **keyword arguments**, because each argument is passed by keyword not order.

When default values are given they are created at the time the **def** statement is executed (i.e., when the function is created), not when the function is called. For immutable arguments like numbers and strings this doesn't make any difference, but for mutable arguments a subtle trap is lurking.

```
1 def append_if_even(x, lst=[]):
2     if x % 2 == 0:
3         lst.append(x)
4     return lst
5
6
7 def append_if_even2(x, lst=None):
8     lst = [] if lst is None else lst
9     if x % 2 == 0:
10        lst.append(x)
11    return lst
12
13
14 for i in range(3):
15     result1 = append_if_even(i)
16     result2 = append_if_even2(i)
17     print(f'{result1=},{i=}')
18     print(f'{result2=},{i=}')
19
20 # result1=[0],i=0
21 # result2=[0],i=0
22 # result1=[0],i=1
23 # result2=[],i=1
24 # result1=[0, 2],i=2
25 # result2=[2],i=2
```

This idiom of having a default of **None** and creating a fresh object should be used for dictionaries, lists, sets, and any other mutable data types that we want to use as default arguments.

4.3.1 Names and docstrings

A few rules of good names:

- Use a naming scheme, and use it consistently. For example:
 - `UPPERCASE` for constants
 - `TitleCase` for classes
 - `camelCase` for GUI functions and methods
 - `lowercase` or `lowercase_with_underscores` for everything else
- For all names, avoid abbreviations, unless they are both standardized and widely used.
- Be propotional with variable and parameter names: `x` is a perfectly good name for an x-coordinate and `i` is fine for a loop counter, but in general the name should be long enough to be descriptive.
- Functions and methods should have names that say what they do or what they return, but never how they do it — since that might change.

We can add documentation to any function by using a **docstring** — this is simply a string that comes immediately after the `def` line, and before the functions code proper begins.

```

1 def shorten(text, length=25, indicator="..."):
2     """Returns text or a truncated copy with the indicator added
3
4     text is any string; length is the maximum length of the returned
5     string (including any indicator); indicator is the string added at
6     the end to indicate that the text has been shortened
7
8     >>> shorten('Second Variety')
9     'Second Variety'
10    >>> shorten('Voices from the Street', 17)
11    'Voices from th...'
12    >>> shorten('Radio Free Albemuth', 10, '*')
13    'Radio Fre*'
14    """
15    if len(text) > length:
16        text = text[:length - len(indicator)] + indicator
17    return text

```

It is not unusual for a function or methods documentation to be longer than the function itself. One convention is to make the first line of the docstring a brief one-line description, then have a blank line followed by a full description, and then to reproduce some examples as they would appear if typed in interactively.

4.3.2 Argument and parameter unpacking

We can use sequence unpacking operator(`*`) to supply positional arguments and or mapping unpacking operator(`**`) to keyword arguments.

```

1 def my_sum(a, b, c=1):
2     return a + b + c
3
4 print(my_sum(*[1, 2, 3, 4][:3])) # 6
5 print(my_sum(*[1, 2], **{'c': 3})) # 6

```

We can also use the sequence unpacking operator in a function's parameter list. This is useful when we want to create functions that can take a variable number of positional arguments.

```

1 def product(*args):
2     result = 1
3     for arg in args:
4         result *= arg
5     return result

```

Having the `*` in front means that inside the function the `args` parameter will be a **tuple** with its items set to however many positional arguments are given.

```

1 def product(*args):
2     result = 1
3     for arg in args:
4         result *= arg
5     return result
6
7
8 print(product(*list(range(1, 10)))) # 362880
9 print(math.factorial(9))           # 362880

```

```

1 # It is also possible to use * as a parameter in its own right.
2 # This is used to signify that there can be no positional arguments after the *.
3 def heron(a, b, c, *, units='square meters'):
4     s = (a + b + c) / 2
5     area = math.sqrt(s * (s - a) * (s - b) * (s - c))
6     return f'{area} {units}'
7
8
9 print(heron(25, 24, 7))
10 print(heron(41, 9, 40, units='sq. inches'))
11 print(heron(25, 24, 7, 'sq. inches')) # TypeError

```

We can also use the mapping unpacking operator with parameters. This allows us to create functions that will accept as many keyword arguments as are given.

```

1 def print_dict(**kwargs):
2     for key in sorted(kwargs):
3         print(f'{key:10} : {kwargs[key]}')
4
5
6 print_dict(**{str(i): f'{100 * i:3}%' for i in range(10)})
7
8 # 0          : 0%
9 # 1          : 100%
10 # 2          : 200%
11 # 3          : 300%
12 # 4          : 400%
13 # 5          : 500%
14 # 6          : 600%
15 # 7          : 700%
16 # 8          : 800%
17 # 9          : 900%
```

```

1 def print_args(*args, **kwargs):
2     for i, arg in enumerate(args):
3         print("positional argument {0} = {1}".format(i, arg))
4     for key in kwargs:
5         print("keyword argument {0} = {1}".format(key, kwargs[key]))
6
7
8 print_args(*list(range(10)), **locals())
```

4.3.3 Accessing variables in the global scope

There are two ways to create a global variable:

- Object defined in `.py` level is global variables.
- variables defined with `global` keyword.

Others are local variables.

```

1 AUTHOR = 'Mike' # global
2
3
4 def say_hello(): # global
5     global language # global
6     language = 'fr'
7     text = 'hello' # local
8     print(text)
9
10
11 class MyException(Exception): # global
12     pass
13
14
15 say_hello()
16 print(language)
```

4.3.4 Lambda functions

Lambda functions are functions created using the following syntax:

```
lambda parameters: expression
```

The **parameters** are optional, and if supplied they are normally just comma-separated variable names, that is, positional arguments, although the complement argument syntax supported by **def** statements can be used. The **expression** can not contain **branches** or **loops** (although conditional expressions are allowed), and can not have a **return** (or **yield**) statement. The result of a **lambda** expression is an anonymous function. When a lambda function is called it returns the result of computing the **expression** as its result.

```
1 lst = list(range(-3, 3))
2 print(lst)
3 print(sorted(lst, key=lambda x: x ** 2))
4 print(sorted(lst, key=lambda key=None: key ** 2)) # seldom used
5
6 # [-3, -2, -1, 0, 1, 2]
7 # [0, -1, 1, -2, 2, -3]
8 # [0, -1, 1, -2, 2, -3]
```

```
1 s = lambda x: ' ' if x == 1 else 's' # use def instead
2 print(s(1)) #
3 print(s(2)) # s
4
5 p = lambda key='hello': print(key) # use def instead
6 p('world') # world
```

There are two common usage for lambda functions:

- key function
- default value

```
1 sorted(lst, key=lambda x: x ** 2)
2 message_dict = collections.defaultdict(lambda: 'No message available')
```

4.3.5 Assertions

Preconditions and postconditions can be specified using **assert** statements:

```
assert boolean_expression, optional_expression
```

If the **boolean_expression** evaluates to **False** an **AssertionError** exception is raised. If the optional **optional_expression** is given, it is used as the argument to the **AssertionError** exception.


```
1 def product(*args):  
2     assert all(args), "0 argument"  
3     result = 1  
4     for arg in args:  
5         result *= arg  
6     return result
```

Note: Assertions are designed for developers, not end-users. Once a program is ready for public release, we can tell Python not to execute **assert** statements. This can be done with:

- -O option in commandline, `python -O program.py`
- set the `PYTHONOPTIMIZE` environment variable to `O`.

We can use -OO option to strip out both **assert** statements and doc-strings. However, there is no environment variable for setting this option.

Chapter 5

Modules

- Functions allow us to parcel up pieces of code so that they can be reused throughout a program.
- Modules provides a means of collecting sets of functions together so that they can be used by any number of programs.
- Packages group sets of modules because their modules provide related functionality or because they depend on each other.

It is important to be aware of what the library has to offer, since using predefined functionality makes programming much faster than creating everything from scratch.

5.1 Modules and packages

Several syntaxes can be used when importing:

```
import importable
import importable1, importable2, ..., importableN
import importable as preferred_name

from importable import object as preferred_name
from importable import object1, object2, ..., objectN
from importable import *
```

In the last syntax, the `*` means “import everything that is not private”, which in practical terms means either every object in the module is imported except for those whose names begin with a leading underscore, or, if the module has a global `__all__` variable that holds a list of names, that all the objects named in the `__all__` variable are imported.

How does Python know where to look for the modules and packages that are imported?

The built-in `sys` module has a list called `sys.path` that holds a list of the directories that constitutes the **Python path**. The first directory is the directory that contains the program itself, even if the program was invoked from another directory. If the `PYTHONPATH` environment variable is set, the paths specified in it are the next ones in the list. The final paths are those needed to access Python's standard library — these are set when Python is installed.

When we first import a module, if it isn't built-in, Python looks for the module in each path listed in `sys.path` in turn.

Using byte-code compiled files leads to faster start-up times since the interpreter only has to load and run the code, rather than load, compile, (save if possible), and run the code; runtimes are not affected, though. When Python is installed, the standard library modules are usually byte-code compiled as part of the installation process.

5.1.1 Packages

A package is simply a directory that contains a set of modules and a file called `__init__.py`.

In some situations it is convenient to load in all of a **packages** modules using a single statement. To do this we must edit the **packages** `__init__.py` file to contain a statement which specifies which modules we want loaded. This statement must assign a list of module names to the special variable `__all__`.

This syntax can also be applied to a module in which case all the functions, variables, and other objects defined in the module (apart from those whose names begin with a leading underscore) will be imported. If we want to control exactly what is imported, we can define an `__all__` list in the module itself.

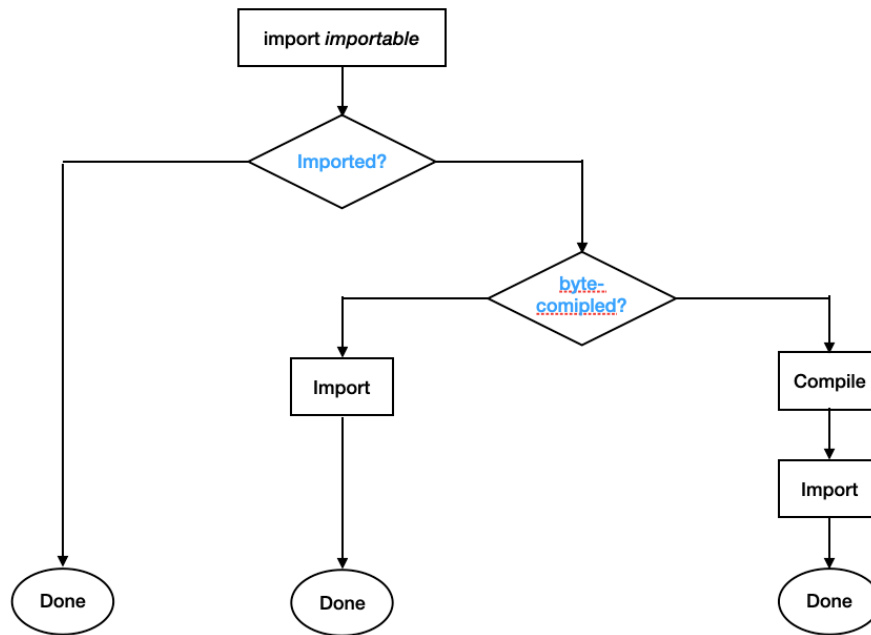


Figure 5.1: Import

5.1.2 Custom modules

The TextUtil module

TextUtil.py:

```

1  #!/usr/bin/env python3
2  # Copyright
3  """
4  This module provides a few string manipulation functions.
5
6  >>> is_balanced('Python (is (not (lisp)))')
7  True
8  >>> shorten('The Crossing', 10)
9  'The Cro...'
10 >>> simplify(' some text with spurious whitespace ')
11 'some text with spurious whitespace'
12 """
13
14 import string
15
16
17 def simplify(text, whitespace=string.whitespace, delete=''):
18     r"""Returns the text with multiple spaces reduced to single spaces
19
20     The whitespace parameter is a string of characters, each of which
21     is considered to be a space.
22     If delete is not empty it should be a string, in which case any
23     characters in the delete string are excluded from the resultant
24     string.
  
```

```

25
26 >>> simplify(" this      and\n that\t too")
27 'this and that too'
28 >>> simplify(" Washington    D.C.\n")
29 'Washington D.C.'
30 >>> simplify(" Washington    D.C.\n", delete=',;:.' )
31 'Washington DC'
32 >>> simplify(" disemvoweled ", delete='aeiou')
33 'dsmvwld'
34 """
35 result = []
36 word = ""
37 for char in text:
38     if char in delete:
39         continue
40     elif char in whitespace:
41         if word:
42             result.append(word)
43             word = ""
44     else:
45         word += char
46 if word:
47     result.append(word)
48 return " ".join(result)
49
50
51 def is_balanced(text, brackets="()[]{}<>"):
52     """Returns True if all the brackets in the text are balanced
53
54     For each pair of brackets, the left and right bracket characters
55     must be different.
56
57     >>> is_balanced("no brackets at all")
58     True
59     >>> is_balanced("<b>bold</b>")
60     True
61     >>> is_balanced("[<b>(some {thing}) goes</b>]")
62     True
63     >>> is_balanced("<b>[not (where {it}) is]</b>")
64     False
65     >>> is_balanced("(not (<tag>(like) (anything)</tag>)")
66     False
67     """
68     counts = {}
69     left_for_right = {}
70     for left, right in zip(brackets[:2], brackets[1:2]):
71         assert left != right, "the bracket characters must differ"
72         counts[left] = 0
73         left_for_right[right] = left
74     for c in text:
75         if c in counts:
76             counts[c] += 1
77         elif c in left_for_right:
78             left = left_for_right[c]
79             if counts[left] == 0:
80                 return False
81             counts[left] -= 1
82     return not any(counts.values())
83
84
85 def shorten(text, length=25, indicator="..."):
86     """Returns text or a truncated copy with the indicator added
87
88     text is any string; length is the maximum length of the returned
89     string (including any indicator); indicator is the string added at
90     the end to indicate that the text has been shortened
91
92     >>> shorten("Second Variety")
93     'Second Variety'
94     >>> shorten("Voices from the Street", 17)
95     'Voices from th...'
96     >>> shorten("Radio Free Albemuth", 10, "**")
97     'Radio Fre*'

```

```

98     """
99     if len(text) > length:
100         text = text[:length - len(indicator)] + indicator
101     return text
102
103
104 if __name__ == '__main__':
105     import doctest
106
107     doctest.testmod()

```

To use the our module:

```

1 import TextUtil
2 text = " a puzzling conundrum "
3 text = TextUtil.simplify(text) # text == 'a puzzling conundrum'

```

If we want our module to be available to a particular program, we just need to put our module in the same directory as the program. If we want our module to be available to all our programs, there are several approaches:

1. put the module in the Python distribution's **site-packages** subdirectory
2. create a directory specifically for the custom modules, and set the **PYTHONPATH** environment variable to this directory
3. put the module in the local site-packages subdirectory (`./local/lib/python3.1/site-packages`)

The second and third approaches have the advantage of keeping our own code separate from the official installation.

Doctest is done by:

```

1 if __name__ == '__main__':
2     import doctest
3
4     doctest.testmod()

```

Whenever a module is imported Python creates a variable for the module called `__name__` and stores the modules name in this variable. A modules name is simply the name of its `.py` file but without the extension. So in this example, when the module is imported `__name__` will have the value `"TextUtil"`, and the if condition will not be met, so the last two lines will not be executed. This means that these last three lines have virtually no cost when the module is imported.

Whenever a `.py` file is run Python creates a variable for the program called `__name__` and sets it to the string `"__main__"`. So if we were to run `TextUtil.py` as though it were a program, Python will set `__name__` to `"__main__"` and the if condition will evaluate to `True` and the last two lines will be executed.

5.2 Overview of Python's standard library

The standard library is the library installed when you install Python. You do not need to install the library separately. Python's standard library is generally described as “batteries included”. The third-party library is the library you should install by yourself.

5.2.1 Strings

```

1 import sys
2 import io
3
4 print('hello ', file=sys.stdout)
5 print('hello ', file=sys.stderr)
6 sys.stdout.write('world ')
7
8 fh = open('text.txt', 'w')
9 print('hello world', file=fh)
10
11 string_io = io.StringIO()
12 sys.stdout = string_io
13
14 for i in range(100):
15     print('hello ' + i)
16
17 sys.stdout = sys.__stdout__ # recover the default sys.stdout
18 print(string_io.getvalue()) # get the value in string io

```

5.2.2 Dates and Times

```

1 import datetime
2 import time
3
4 # current seconds
5 current_second = time.time()
6
7 # current date and time
8 current_time = datetime.datetime.now()
9
10 # current struct time
11 current_struct_time = time.localtime()
12
13 # format from struct time
14 string_time = time.strftime('%Y-%m-%d %H:%M:%S', current_struct_time)
15
16 # struct time from string
17 struct_time = time.strptime('2021-02-04 11:07:09', '%Y-%m-%d %H:%M:%S')
18
19 # seconds from struct time
20 seconds = time.mktime(current_struct_time)

```

5.2.3 Algorithms and collection data types

```

1 import heapq
2
3 heap = []
4 heapq.heappush(heap, (5, 'a'))

```



```

5  heapq.heappush(heap, (2, 'b'))
6  heapq.heappush(heap, (4, 'c'))
7
8  for _ in range(len(heap)):
9      print(heapq.heappop(heap))
10
11 h = [1, 5, 11, 10, 3, 6, 20, 8, 7, 2, 9]
12 heapq.heapify(h)
13 for _ in range(len(h)):
14     print(heapq.heappop(h))

```

5.2.4 File formats, encodings, and data persistence

```

1  import base64
2
3  binary = open('ali.png', 'rb').read()
4  ascii_text = ''
5  for i, c in enumerate(base64.b64encode(binary)):
6      if i and i % 68 == 0:
7          ascii_text += '\\\\n'
8          ascii_text += chr(c)
9  print(ascii_text)
10
11 binary = base64.b64decode(ascii_text)
12 open('ali_copy.png', 'wb').write(binary)

```

```

1  import tarfile
2  import string
3  import sys
4  import os
5
6  BZ2_AVAILABLE = True
7  try:
8      import bz2
9  except ImportError:
10     BZ2_AVAILABLE = False
11
12 # absolute path is not permitted
13 UNTRUSTED_PREFIXES = tuple(['/', '\\'] + [c + ':' for c in string.ascii_letters])
14
15
16 def untar(archive):
17     tar = None
18     try:
19         tar = tarfile.open(archive)
20         for member in tar.getmembers():
21             if member.name.startswith(UNTRUSTED_PREFIXES):
22                 print('untrusted prefix, ignoring', member.name)
23             elif '..' in member.name:
24                 print('suspect path, ignoring', member.name)
25             else:
26                 tar.extract(member)
27                 print('unpacked', member.name)
28     except (tarfile.TarError, EnvironmentError) as err:
29         print(err)
30     finally:
31         if tar is not None:
32             tar.close()
33
34
35 def error(message, exit_status=1):
36     print(message)
37     sys.exit(exit_status)

```


Chapter 6

Object-oriented programming

6.1 Costom classes

6.1.1 Attributes and methods

```
class className:
    suite

class className(base_classes):
    suite
```

Just like `def` statements, `class` is a statement, so we can create classes dynamically if we want to. Class instances are created by calling the class with any necessary arguments.

```
1  #!/usr/bin/env python3
2  # Copyright (c) 2021-02
3
4  import math
5
6
7  class Point:
8      def __init__(self, x=0, y=0):
9          """
10             A 2D cartesian coordinate
11             :param x:
12             :param y:
13             """
14             self.x = x
15             self.y = y
16
```

```

17     def distance_from_origin(self):
18         return math.hypot(self.x, self.y)
19
20     def __eq__(self, other):
21         return self.x == other.x and self.y == other.y
22
23     def __repr__(self):
24         return f'Point({self.x!r}, {self.y!r})'
25
26     def __str__(self):
27         return f'({self.x!r}, {self.y!r})'

```

Python automatically supplies the first argument in method calls – it is an object reference to the object itself. We must include this argument in the parameter list, and by convention the parameter is called `self`. All object attributes (data and method attributes) must be qualified by `self`.

To create an object, two steps are necessary:

1. a raw or uninitialized object must be created (`__new__()`)
2. the object must be initialized, ready for use (`__init__()`)

If we call a method on an object and the object's class does not have an implementation of that method, Python will automatically go through the object's base classes, and their base classes, and so on, until it finds the method – and if the method is not found an `AttributeError` exception is raised.

Calling `super().__init__()` is to call the base class's `__init__()` method. For classes that directly inherit `object` there is no need to do this and we call base class methods only when necessary – for example, when creating classes that are designed to be subclassed, or when creating classes that don't directly inherit `object`.

If we want to avoid inappropriate comparisons, we can this:

```

1     def __eq__(self, other):
2         if not isinstance(other, Point):
3             return NotImplemented
4         return self.x == other.x and self.y == other.y

```

In this case, if `NotImplemented` is returned, Python will then try calling `other.__eq__(self)` to see whether the `other` type supports the comparison with the `Point` type, and if there is no such method or if that method also returns `NotImplemented`, Python will give up and raise a `TypeError` exception. Only the following methods may return `NotImplemented`:

- `__lt__(self, other)`
- `__le__(self, other)`

- `__eq__(self, other)`
- `__ne__(self, other)`
- `__ge__(self, other)`
- `__gt__(self, other)`

Powerful `eval(): (eval(expression))`

```
1 p = Shape.Point(3, 9)
2 repr(p)
# returns: 'Point(3, 9)'
3 q = eval(p.__module__ + "." + repr(p))
4 repr(q)
# returns: 'Point(3, 9)'
```

```
1 a0 = 0
2 a1 = 1
3 a2 = 2
4 a3 = 3
5
6 for i in range(4):
7     print(eval(f'a{i} * {i}'))
```

A more powerful function is `exec()`, it can accept python code not only python expression.

```
1 for i in range(3):
2     exec(f'a{i} = {i}')
3     exec(f'print(a{i})')
4 exec('me = "Mike Chyson"')
5 print(me)
6
7
8 def say_hello():
9     print('hello ')
10
11 say_hello()
```

6.1.2 Inheritance and polymorphism

```
1 class Circle(Point):
2     def __init__(self, radius, x=0, y=0):
3         super().__init__(x, y)
4         self.radius = radius
5
6     def edge_distance_from_origin(self):
7         return abs(self.distance_from_origin() - self.radius)
8
9     def area(self):
10        return math.pi * (self.radius ** 2)
11
12    def circumference(self):
13        return 2 * math.pi * self.radius
14
15    def __eq__(self, other):
16        return self.radius == other.radius and super().__eq__(other)
```

```

17     def __repr__(self):
18         return f'Circle({self.radius!r}, {self.x!r}, {self.y!r})'
19
20
21     def __str__(self):
22         return repr(self)

```

6.1.3 Using properties to control attribute access

A property is an item of object data that is accessed like an instance variable but where the accesses are handled methods behind the scenes.

```

1 class Circle(Point):
2     def __init__(self, radius, x=0, y=0):
3         super().__init__(x, y)
4         self.radius = radius
5
6     @property
7     def edge_distance_from_origin(self):
8         return abs(self.distance_from_origin - self.radius)
9
10    @property
11    def area(self):
12        return math.pi * (self.radius ** 2)
13
14    @property
15    def circumference(self):
16        return 2 * math.pi * self.radius
17
18    @property
19    def radius(self):
20        return self.__radius
21
22    @property.setter
23    def radius(self, radius):
24        assert radius > 0, 'radius must be nonzero and non-negative'
25        self.__radius = radius

```

A decorator is a function that takes a function or method as its argument and returns a “decorated” version, that is, a version of the function or method that is modified in some way. A decorator is indicated by preceding its name with an at symbol (@).

The `property()` decorator function is built-in and takes up to four arguments:

- a getter function
- a setter function
- a deleter function
- a docstring

The effect of using `@property` is the same as calling the `property()` function with just one argument, the getter function. We could have created the `area` property like this:

```
1 def area(self):
2     return math.pi * (self.radius ** 2)
3 area = property(area)
```

We rarely use this syntax, since using a decorator is shorter and clearer.

To turn an attribute into a readable/writable property we must create a private attribute where the data is actually held and supply getter and setter methods.

```
1 @property
2 def radius(self):
3     return self.__radius
4
5 @property.setter
6 def radius(self, radius):
7     assert radius > 0, 'radius must be nonzero and non-negative'
8     self.__radius = radius
```

Every property that is created has a `getter`, `setter`, and `deleter` attribute, so once the `radius` property is created using `@property`, the `radius.getter`, `radius.setter`, and `radius.deleter` attributes become available. The `radius.getter` is set to the getter method by the `@property` decorator. The other two are set up by Python so that they do nothing (so the attribute cannot be written to or deleted), unless they are used as decorators, in which case they in effect replace themselves with the method they are used to decorate.

The `Circle`'s initializer, `Circle.__init__()`, includes the statement `self.radius = radius`; this will call the `radius` property's setter.

Chapter 7

File handling

There are several file formats to choose when you are doing serialization and deserialization, for example:

- binary
- test
- XML

Which is the best file format?

The question is too context-dependent to have a single definitive answer, especially since there are pros and cons for each format and for each way of handling them.

Binary formats are usually very **fast** to save and load and they can be very **compact**. Binary data doesn't need parsing since each data type is stored using its **natural representation**. Binary data is **not human readable or editable**, and without knowing the format in detail it is not possible to create separate tools to work with binary data.

Text formats are **human readable and editable**. Text formats can be **tricky to parse** and it is not always easy to give good error messages if a text files format is broken.

XML formats are **human readable and editable**. XML formats can be processed using separate tools. Parsing XML is straightforward and some parsers have good error reporting. XML parsers can be slow, so reading very large XML files can take a lot more time than reading an equivalent binary or

text file. XML includes metadata and this can make XML more portable than text files.

Some programs use an XML file format for all the data they handle, whereas others use XML as a convenient import/export format. The ability to import and export XML is useful and is always worth considering even if a programs main format is a text or binary format.

Chapter 8

Advanced programming techniques

Normal programming techniques enable us to build “standard Python toolbox” and advanced programming techniques can bring us “deluxe Python toolbox”.

8.1 Further procedural programming

8.1.1 Branching using dictionaries

Functions are objects like everything else in Python, and a function’s name is an object reference that refers to the function. If we write a function’s name without parentheses, Python knows we mean the object reference, and we can pass such object references around like any others.

We can use this fact to produce `if` statements that have lots of `elif` clauses with a single function call.

```
1 # (A)dd (E)dit (L)ist (R)emove (I)mport e(X)port (Q)uit
2 if action == 'a':
3     add_dvd(db)
4 elif action == 'e':
5     edit_dvd(db)
6 elif action == 'l':
7     list_dvds(db)
8 elif action == 'r':
9     remove_dvd(db)
```

```

10 elif action == "i":
11     import_(db)
12 elif action == "x":
13     export(db)
14 elif action == "q":
15     quit(db)
16
17 # The same effect
18 # (A)dd (E)dit (L)ist (R)emove (I)mport e(X)port (Q)uit
19 functions = dict(a=add_dvd, e=edit_dvd, l=list_dvds, r=remove_dvd,
20                 i=import_, x=export, q=quit)
21 functions[action](db)

```

Not only is the code on the bottom much shorter than the code on the top, but also it can scale (have far more dictionary items) without affecting its performance, unlike the upper code whose speed depends on how many `elif`s must be tested to find the appropriate function to call.

```

1 def import_(self, filename, reader=None):
2     extension = os.path.splitext(filename)[1].lower()
3     call = {
4         ('.aix', 'dom'): self.import_xml_dom,
5         ('.aix', 'etree'): self.import_xml_etree,
6         ('.aix', 'sax'): self.import_xml_sax,
7         ('.ait', 'manual'): self.import_text_manual,
8         ('.ait', 'regex'): self.import_text_regex,
9         ('.aib', None): self.import_binary,
10        ('.aip', None): self.import_pickle
11    }
12    result = call[extension, reader](filename)
13    if not result:
14        self.clear()
15    return result

```

8.1.2 Generator expressions and functions

Generator expression:

```

(expression for item in iterable)
(expression for item in iterable if condition)

```

```

1 def items_in_key_order(d):
2     for key in sorted(d):
3         yield key, d[key] # yield
4
5
6 def items_in_key_order2(d):
7     return ((key, d[key]) for key in sorted(d)) # generator expression

```

Both functions return a generator. If we need all the items in one go we can pass the generator returned to `list()` or `tuple()`.

Generators provide a means of performing lazy evaluation, which means that they compute only the values that are actually needed. This can be more efficient than, say, computing a very large list in one go. Some generators produce as many values as we ask for – without any upper limit. For example

```

1 def quarters(next_quarter=0.0):
2     while True:
3         yield next_quarter
4         next_quarter += 0.25

```

This function will return 0.0, 0.25, 0.5, and so on, forever. Here is how we could use the generator:

```

1 result = []
2 for x in quarters():
3     result.append(x)
4     if x >= 1.0:
5         break
6 # [0.0, 0.25, 0.5, 0.75, 1.0]

```

```

1 def quarters(next_quarter=0.0):
2     while True:
3         received = (yield next_quarter)
4         if received is None:
5             next_quarter += 0.25
6         else:
7             next_quarter = received
8
9
10 result = []
11 generator = quarters()
12 while len(result) < 5:
13     x = next(generator)
14     if abs(x - 0.5) < sys.float_info.epsilon:
15         x = generator.send(1.0) # Notice this
16     result.append(x)
17 print(result) # [0.0, 0.25, 1.0, 1.25, 1.5]

```

We create a variable to refer to the generator and call the built-in `next()` function which retrieves the next item from the generator it is given. (The same effect can be achieved by calling the `generators __next__()` special method, in this case, `x = generator.__next__()`.) If the value is equal to 0.5 we send the value 1.0 into the generator (which immediately yields this value back).

8.1.3 Dynamic code execution and dynamic imports

Dynamic code execution

There are two built-in functions for dynamic code execution:

eval() for expression

exec() for code

```

1 import math
2
3 x = eval('2 ** 10')
4 print(x) # 1024
5
6 code = '''
7 def area_of_sphere(r):
8     return 4 * math.pi * r ** 2

```

```

9 '''
10 # context = {}
11 # context['math'] = math
12 # exec(code, context) # define the function area_of_sphere
13 context = globals().copy()
14 exec(code, context)
15
16 area_of_sphere = context['area_of_sphere']
17 sphere = area_of_sphere(5)
18 print(sphere) # 314.1592653589793

```

If `exec()` is called with some code as its only argument there is no way to access any functions or variables that are created as a result of the code being executed. Furthermore, `exec()` cannot access any imported modules or any of the variables, functions, or the objects that are in scope at the point of the call. Both of these problems can be solved by passing a dictionary as the second argument. The dictionary provides a place where object references can be kept for accessing after the `exec()` call has finished. For example, the use of the `context` dictionary means that after the `exec()` call, the dictionary has an object reference to the `area_of_sphere()` function that was created by `exec()`. In this example we needed `exec()` to be able to access the `math` module, so we inserted an item into the context dictionary whose key is the module's name and whose value is an object reference to the corresponding module object. This ensures that inside the `exec()` call, `math.pi` is accessible.

Dynamic importing modules

Python provides three straightforward mechanisms that can be used to create plug-ins, all of which involve importing modules by name at runtime. And once we have dynamically imported additional modules, we can use Python's introspection functions to check the availability of the functionality we want, and to access it as required.

The main function is:

```

1 def main():
2     modules = load_modules()
3     get_file_type_functions = []
4     for module in modules:
5         get_file_type = get_function(module, 'get_file_type')
6         if get_file_type is None:
7             get_file_type_functions.append(get_file_type)
8
9     for file in get_files(sys.argv[1:]):
10        fh = None
11        try:
12            fh = open(file, 'rb')
13            magic = fh.read(1000)
14            for get_file_type in get_file_type_functions:
15                filetype = get_file_type(magic, os.path.splitext(file)[1]) # file extension
16                if filetype is not None:
17                    print(f'{filetype:.<20}{file}')
18                break

```

```

19         else:
20             print('{:.<20}{}'.format('Unknown', file))
21         except EnvironmentError as err:
22             print(err)
23         finally:
24             if fh is not None:
25                 fh.close()

```

The longest and most difficult approach (approach 1):

```

1 def load_modules():
2     modules = []
3     for name in os.listdir(os.path.dirname(__file__) or '.'):
4         if name.endswith('.py') and 'magic' in name.lower():
5             filename = name
6             name = os.path.splitext(name)[0] # remove extension
7             if name.isidentifier() and name not in sys.modules:
8                 fh = None
9                 try:
10                     fh = open(filename)
11                     code = fh.read()
12                     module = type(sys)(name)
13                     sys.modules[name] = module
14                     exec(code, module.__dict__)
15                     modules.append(module)
16                 except (EnvironmentError, SyntaxError) as err:
17                     sys.modules.pop(name, None)
18                     print(err)
19                 finally:
20                     if fh is not None:
21                         fh.close()
22     return modules

```

We begin by iterating over all the files in the program's directory. If this the current directory, `os.path.dirname(__file__)` will return an empty string which would cause `os.listdir()` to raise an exception, so we pass "." if necessary.

The line `module = type(sys)(name)` is quite subtle. When we call `type()` it returns the type object of the object it is given. So if we called `type(1)` we would get `int` back. If we call the type object as a function, we get an object of that type back. For example, we can get the integer 5 in variable `x` by writing `x = 5`, or `x = int(5)`, or `x = type(0)(5)`. In this case we've used `type(sys)` and `sys` is a module, so we get back the module type object, can can be used to create a new module with the given name. Just as with the `int` example where it didn't matter what integer we used to get the `int` type object, it doesn't matter what module we use to get the module type object.

Once we have a new module, we add it to the global list of modules to preven the module from accidentally reimported. This is done before calling `exec()` to more closely mimic the behavior of the `import` statement.

The second way to dynamically load a module at runtime (approch 2) – the code shown here replaces the first approachs `try ... except` block:

```

1         try:
2             exec('import ' + name)

```

```

3         modules.append(sys.modules[name])
4     except SyntaxError as err:
5         print(err)

```

One theoretical problem with this approach is that it is potentially insecure. The name variable could begin with `sys`; and be followed by some destructive code.

The easiest way to dynamically import module and is slightly safer than using `exec()` (approach 3):

```

1         try:
2             module = __import__(name)
3             modules.append(module)
4         except SyntaxError as err:
5             print(err)

```

Having imported the module we need to be able to access the functionality it provides. This can be achieved using Python's built-in introspection functions, `getattr()` and `hasattr()`.

```

1 def get_function(module, function_name):
2     function = get_function.cache.get((module, function_name), None)
3     if function is None:
4         try:
5             function = getattr(module, function_name)
6             if not hasattr(function, '__call__'):
7                 raise AttributeError()
8             get_function.cache[(module, function_name)] = function
9         except AttributeError:
10            function = None
11    return function

```

Ignoring the cache-related code for a moment, what the function does is call `getattr()` on the module object with the name of the function we want. If there is no such attribute an `AttributeError` exception is raised, but if there is such an attribute we use `hasattr()` to check that the attribute itself has the `__call__` attribute – something that all callables (functions and methods) have. If the attribute exists and is callable we can return it to the caller; otherwise, we return `None` to signify that the function isn't available.

If hundreds of files were being processed (e.g. `*.*`), we don't want to go through the lookup process for every module for every file. So immediately after defining the `get_function()` function, we add an attribute to the function, a dictionary called `cache`. (In general, Python allows us to add arbitrary attributes to arbitrary objects.) The first time that `get_function()` is called the cache dictionary is empty, so the `dict.get()` call will return `None`. But each time a suitable function is found it is put in the dictionary with a 2-tuple of the module and function name used as the key and the function itself as the value. So the second and all subsequent times a particular function is requested

the function is immediately returned from the cache and no attribute lookup takes place at all.

The technique used for caching the `get_function()`'s return value for a given set of arguments is called **memorizing**. It can be used for any function that has no **side effects** (does not change any global variables), and that always returns the same result for the same (immutable) arguments.

Dynamic programming and introspection functions:

__import__(...) Imports a module by name

compile(source, file, mode) Returns the code object that results from compiling the **source** text; **file** must be the filename, or “<string>”; **mode** must be “single”, “eval”, or “exec”

delattr(obj, name) Deletes the attribute called **name** from object **obj**

getattr(obj, name, val) Returns the value of the attribute called **name** from object **obj**, or **val** if given and there is no such attribute

hasattr(obj, name) Returns **True** if object **obj** has an attribute called **name**

setattr(obj, name, val) Sets the attribute called **name** to the value **val** for the object **obj**, creating the attribute if necessary

eval(source, globals, locals) Returns the result of evaluating the single expression in **source**; if supplied, **globals** is the global context and **locals** is the local context (as dictionaries)

exec(obj, globals, locals) Evaluates object **obj**, which can be a string or a code object from **compile()**, and returns **None**; if supplied, **globals** is the global context and **locals** is the local context

dir(obj) Returns the list of names in the local scope, or if **obj** is given then **obj**’s names

globals() Returns a dictionary of the current global context

locals() Returns a dictionary of the current local context

type(obj) Returns object **obj**’s type object

vars(obj) Returns object **obj**’s context as a dictionary; or the local context if **obj** is not given

8.1.4 Local and recursive functions

Functions defined inside the definition of an existing function are called **nested functions** or **local functions**.

One common use case for local functions is when we want to use recursion. Recursive functions can be **computationally expensive** because for every recursive call another stack frame is used; however, some algorithms are most naturally expressed using recursion. Most Python implementations have a fixed limit to how many recursive calls can be made. The limit is returned by `sys.getrecursionlimit()` and can be changed by `sys.setrecursionlimit()`, although increasing the limit is most often a sign that the algorithm being used is inappropriate or that the implementation has a bug.

8.1.5 Functions and method decorators

A decorator is a function that takes a function or method as its sole argument and returns a new function or method that incorporates the decorated function or method with some additional functionality added.

```
1 @positive_result
2 def discriminant(a, b, c):
3     return b ** 2 - 4 * a * c
```

Here's the decorator's implementation:

```
1 def positive_result(function):
2     def wrapper(*args, **kwargs):
3         result = function(*args, **kwargs)
4         assert result >= 0, function.__name__ + "() result isn't >= 0"
5         return result
6
7     wrapper.__name__ = function.__name__
8     wrapper.__doc__ = function.__doc__
9     return wrapper
```

Decorator define a new local function (here `wrapper()`) tha calls the original function. The wrapper finishes by returning the result computed by the wrapped function. After creating the wrapper, we set its name and docstring to those of the original function. **This helps with introspection, since we want error messages to mention the name of the original function, not the wrapper.** Finally, we return the wrapper function – it is this function that will be used in place of the original.

Here is slightly cleaner version:

```
1 import functools
2
```

```

3 def positive_result(function):
4     @functools.wraps(function)
5     def wrapper(*args, **kwargs):
6         result = function(*args, **kwargs)
7         assert result >= 0, function.__name__ + "() result isn't >=0"
8         return result
9
10    return wrapper

```

The wrapper itself is wrapped using the `functools` module's `@functools.wraps` decorator, which ensures that the `wrapper()` function has the name and doc-string of the original function.

In some cases it would be useful to be able to parameterize a decorator. (At first sight this does not seem possible since a decorator takes just one argument, a function or method. But there is a neat solution to this. We can call a function with the parameters we want and that returns a decorator which can then decorate the function that follows it.) For example:

```

1 @bounded(0, 100)
2 def percent(amount, total):
3     return (amount / total) * 100

```

Here's the implementation of the `bounded()` function:

```

1 def bounded(minimum, maximum):
2     def decorator(function):
3         @functools.wraps(function)
4         def wrapper(*args, **kwargs):
5             result = function(*args, **kwargs)
6             if result < minimum:
7                 return minimum
8             elif result > maximum:
9                 return maximum
10            return result
11
12    return wrapper
13
14    return decorator

```

Here is a log decorator:

```

1 import logging
2 import functools
3 import os
4 import tempfile
5
6
7 def logged(file):
8     """
9     Log the output of the decorated function into a logged file.
10
11     :param file: If file is None, use temple file, otherwise use the given file
12     :return: decorated function
13     """
14     def decorator(function):
15         if __debug__:
16             logger = logging.getLogger('Logger')
17             logger.setLevel(logging.DEBUG)
18             if file is None:
19                 handler = logging.FileHandler(os.path.join(tempfile.gettempdir(), 'logged.log'))
20             else:
21                 handler = logging.FileHandler(file)

```

```

22     logger.addHandler(handler)
23
24     @functools.wraps(function)
25     def wrapper(*args, **kwargs):
26         # accumulate string
27         log = 'called: ' + function.__name__ + '('
28         log += ', '.join([f'{a!r}' for a in args] + [f'{k!s}={v!r}' for k, v in kwargs.items()])
29         result = exception = None
30         try:
31             result = function(*args, **kwargs)
32             return result
33         except Exception as err:
34             exception = err
35         finally:
36             log += ')' -> ' + str(result)) if exception is None else f'{type(exception)}: {exception}'
37             logger.debug(log)
38             if exception is not None:
39                 raise exception
40
41     return wrapper
42 else:
43     return function
44
45 return decorator
46
47
48 @logged(None)
49 def say_word(word='hello '):
50     return word

```

8.1.6 Function annotations

Functions and methods can be defined with annotations — expressions that can be used in a function’s signature.

General syntax:

```
def functionName(par: exp1, par2: exp2, ..., parN: expN) -> rexp:
    suite
```

Every colon expression part (: expX) is an optional annotation, and so is the arrow return expression part (-> rexp).

If annotations are present they are added to the function’s `__annotations__` dictionary; if they are not present this dictionary is empty. The dictionary’s keys are the parameter names, and the value are the corresponding expressions. The syntax allows us to annotate all, some, or none of the parameters and to annotate the return value or not. Annotations have no special significance to Python. **The only thing that Python does in the face of annotations is to put them in the `__annotations__` dictionary;** any other action is up to us.

```

1 def is_unicode_punctuations(s: str) -> bool:
2     for c in s:
3         # print(unicodedata.category(c))
4
5     # Every Unicode character belongs to a particular category and each category is

```

```

6         # identified by a two-character identifier. All the categories that begin with P are
7         # punctuation characters.
8         if unicodedata.category(c)[0] != 'P':
9             return False
10        return True
11
12
13    print(is_unicode_punctuations.__annotations__)
14    # {'s': <class 'str'>, 'return': <class 'bool'>}

```

If we want to give meaning to annotations, for example, to provide type checking, one approach is to decorate the functions we want the meaning to apply to with a suitable decorator. Here is a very basic type-checking decorator:

```

1  import inspect
2  import functools
3
4  def strictly_typed(function):
5      """
6      # This decorator requires that every argument and the return value must be
7      # annotated with the expected type.
8      # Notice that the checking is done only in debug mode (which is Python's default
9      # mode — controlled by the -O command-line option and the PYTHONOPTIMIZE environment variable).
10
11      :param function:
12      :return: decorated function
13      """
14      annotations = function.__annotations__
15      arg_spec = inspect.getfullargspec(function)
16
17      # assert all type is given
18      assert 'return' in annotations, 'missing type for return value'
19      # arg_spec.args: positional arguments
20      # kwonlyargs: keyword only arguments (kwargs after position delimiter sing(*))
21      for arg in arg_spec.args + arg_spec.kwonlyargs:
22          assert arg in annotations, f'missing type for parameter "{arg}"'
23
24      @functools.wraps(function)
25      def wrapper(*args, **kwargs):
26          # argument check
27          # zip() returns an iterator and dictionary.items() returns a dictionary view
28          # we cannot concatenate them directly, so first we convert them both to lists
29          all_args = list(zip(arg_spec.args, args)) + list(kwargs.items())
30          for name, arg in all_args:
31              assert isinstance(arg, annotations[name]), (
32                  f'expected argument "{name}" of {annotations[name]} got {type(arg)}')
33
34          # result check
35          result = function(*args, **kwargs)
36          assert isinstance(result, annotations['return']), (
37              f'expected return of {annotations["return"]} got {type(result)}')
38
39          return result
40
41      return wrapper

```

This decorator requires that every argument and the return value must be annotated with the expected type. The `inspect` module provides powerful introspection services for objects.

```

1  @strictly_typed
2  def just_type(s1: str, n1: int, *, s2: str = 's2') -> bool:
3      return True
4
5  print(just_type.__annotations__)
6  just_type('s1', 1, s2='2')
7  just_type(1, 2)

```

```
8 # AssertionError: expected argument 's1' of <class 'str'> got <class 'int'>
```

Notice that the checking is done only in debug mode (which is Python's default mode - controlled by the `-O` command-line option and the `PYTHONOPTIMIZE` environment variable).
Why?

8.2 Further object-oriented programming

```
1 class Point:
2     def __init__(self, x=0, y=0):
3         self.x = x
4         self.y = y
5
6
7 class PointFixedAttribute:
8     __slots__ = ('x', 'y')
9
10    def __init__(self, x=0, y=0):
11        self.x = x
12        self.y = y
13
14
15 if __name__ == '__main__':
16     pfa = PointFixedAttribute()
17     p = Point()
18
19     # print(pfa.__dict__) # AttributeError: 'PointFixedAttribute' object has no attribute '__dict__'
20     print(pfa.__slots__) # ('x', 'y')
21     print(p.__dict__) # {'x': 0, 'y': 0}
22
23     # pfa.z = 1 # AttributeError: 'PointFixedAttribute' object has no attribute 'z'
24     p.z = 1
25     print(p.z) # 1
26     print(p.__dict__) # {'x': 0, 'y': 0, 'z': 1}
```

When a class is created without the use of `__slot__`, behind the scenes Python creates a private dictionary called `__dict__` for each **instance**, and this dictionary holds the instance's data attributes. This is why we can add or remove attributes from object.

If we only need objects where we access the original attributes and don't need to add or remove attributes, we can create classes that don't have a `__dict__`. This is achieved simply by defining a class attribute called `__slot__` whose value is a tuple of **attribute** names. (Here attributes is different from method.) Each object of such a class will have attributes of the specified names and no `__dict__`; no attributes can be added or removed from such classes. These objects consume less memory and are faster than conventional objects, although this is unlikely to make much difference unless large numbers of objects

are created. If we inherit from a class that uses `__slots__` we must declare slots in our subclass, even if empty, such as `__slots__ = ()`; or the memory and speed savings will be lost.

8.2.1 Controlling attribute access

It is sometimes convenient to have a class where attribute values are computed **on the fly** rather than stored. Here's the complete implementation of such a class: (`__getattr__` equal to `.`)

```

1 class Ord:
2     def __getattr__(self, item):
3         # builtin.ord() is used to avoid ord is used by the user
4         # like ord = Ord()
5         return builtins.ord(item)
6
7 ord = Ord()
8 print(ord.a) # 97
9 print(ord.Z) # 90

```

```

1 class Const:
2     def __setattr__(self, key, value):
3         if key in self.__dict__:
4             raise ValueError('cannot change a const attribute')
5         self.__dict__[key] = value
6
7     def __delattr__(self, item):
8         if item in self.__dict__:
9             raise ValueError('cannot delete a const attribute')
10        raise AttributeError(f'{self.__class__.__name__} object has no attribute '{item}')
11
12 const = Const()
13 const.limit = 591
14 print(const.limit)
15 # const.limit = 1 # ValueError: cannot change a const attribute
16 # del const.limit # ValueError: cannot delete a const attribute

```

The class works because we are using the object's `__dict__` which is what the base class `__getattr__()`, `__setattr__()`, and `__delattr__()` method used.

If there are a lot of read-only properties, here is a different solution:

```

1 USE_GETATTR = True
2
3
4 class Image:
5     def __init__(self, width, height, filename="", background="#FFFFFF"):
6         self.filename = filename
7         self.__background = background
8         self.__data = {}
9         self.__width = width
10        self.__height = height
11        self.__colors = {self.__background}
12
13        if USE_GETATTR:
14            def __getattr__(self, name):
15                if name == 'color':
16                    return set(self.__colors)
17                classname = self.__class__.__name__
18                if name in frozenset({'background', 'width', 'height'}):

```


Special Method	Usage	Description
<code>__delattr__(self, name)</code>	<code>del x.n</code>	Deletes object <code>x</code> 's <code>n</code> attribute
<code>__dir__(self)</code>	<code>dir(x)</code>	Returns a list of <code>x</code> 's attribute names
<code>__getattr__(self, name)</code>	<code>v = x.n</code>	Returns the value of object <code>x</code> 's <code>n</code> attribute if it isn't found directly
<code>__getattribute__(self, name)</code>	<code>v = x.n</code>	Returns the value of object <code>x</code> 's <code>n</code> attribute; see text
<code>__setattr__(self, name, value)</code>	<code>x.n = v</code>	Sets object <code>x</code> 's <code>n</code> attribute's value to <code>v</code>

Figure 8.1: Attribute access speical methods

```

19         # image = Image(10, 10)
20         # image.__dict__
21         # {'filename': '',
22         #  '_Image__background': '#FFFFFF',
23         #  '_Image__data': {},
24         #  '_Image__width': 10,
25         #  '_Image__height': 10,
26         #  '_Image__colors': {'#FFFFFF'}}
27         return self.__dict__[f'_{classname}__{name}']
28         raise AttributeError(f'_{classname}' object has no attribute {name}')
29     else:
30         @property
31         def background(self):
32             return self.__background
33
34         @property
35         def width(self):
36             return self.__width
37
38         @property
39         def height(self):
40             return self.__height
41
42         @property
43         def colors(self):
44             return set(self.__colors)

```

If the variable `USE_GETATTR` is true, `__getattr__` is used to create read-only properties, otherwise the property decorator is used to create the read-only properties.

If we attempt to access an object's attribute and the attribute is not found, Python will call the `__getattr__` method with the name of the attribute as a parameter.

There is a subtle difference in the that:

- using `__getattr__()` provides access to the attribute in the instance's class (which may be subclass)

- accessing the attribute directly uses the class the attribute is defined in

Where as the `__getattr__()` method is called last when looking for (non-speical) attribute, the `__getattribute__()` method is called first for every attribute access. Although it can be useful or even essential in some cases to call `__getattribute__()`, reimplementing the `__getattribute__()` method can be tricky.

8.2.2 Functors

In Python, a **function object** is an object reference to any callable, such as a function, a lambda function, or a method. The definition also includes classes, since an object reference to a class is a callable that, when called, returns an object of the given class. In computer science a **functor** is an object that can be called as though it ware a function, so in Python terms a functor is just another kind of function object. Any **class** that has a `__call__()` special method is a functor.

The key benefit that functors offer is that they can maintain some state information.

```

1 class Strip:
2     def __init__(self, characters):
3         self.characters = characters
4
5     def __call__(self, string):
6         return string.strip(self.characters)
7
8 strip_punctuation = Strip(',;:!.? ')
9 print(strip_punctuation('Mike Chyson!')) # Mike Chyson

```

We could achieve the same thing using a plain function or lambda, but if we need to store a bit more state or perform more complex processing, a functor is often the right solution.

A functors ability to capture state by using a class is very versatile and powerful, but sometimes it is more than we really need. Another way to capture state is to use a **closure**. A **closure is a function or method that captures some external state**.

```

1 def make_strip_function(characters):
2     def strip_function(string):
3         return string.strip(characters)
4
5     return strip_function
6
7 strip_punctuation = make_strip_function(',;:!.? ')
8 print(strip_punctuation('Mike Chyson!')) # Mike Chyson

```

If the state are complex, you can you a functor, otherwise you can use a plain function or lambda or closure.

8.2.3 Context manager

Context managers allow us to simplify code by ensuring that certain operations are performed before and after a particular block of code is executed. The behavior is achieved because context managers define two special methods, `__enter__()` and `__exit__()`, that Python treats specially in the scope of a `with` statement. When a context manager is created in a `with` statement its `__enter__()` method is automatically called, and when the context manager goes out of scope after its `with` statement its `__exit__()` method is automatically called.

The syntax for using context managers is:

```
with expression as variable
    suite
```

The **expression** must be or must produce a context manager object; if the optional **as variable** part is specified, the variable is set to refer to the object returned by the context managers `__enter__()` method (and this is often the context manager itself). Because a context manager is guaranteed to execute its “exit” code (even in the face of exceptions), context managers can be used to eliminate the need for `finally` blocks in many situations. **The file objects returned by the built-in `open()` function are context managers.**

A file object is a context manager whose exit code always closes the file if it was opened. The exit code is executed whether or not an exception occurs, but in the latter case, the exception is propagated. This ensures that the file gets closed and we still get the chance to handle any errors. For example:

```
1 # without context manager
2 fh = None
3 try:
4     fh = open(filename)
5     for line in fh:
6         process(line)
7 except EnvironmentError as err:
8     print(err)
9 finally:
10     if fh is not None:
11         fh.close()
12
13 # with context manager
14 try:
```

```

15     with open(filename) as fh:
16         for line in fh:
17             process(line)
18 except EnvironmentError as err:
19     print(err)

```

```

1  try:
2      with open(source) as fin, open(target, 'w') as fout:
3          for line in fin:
4              fout.write(process(line))
5  except EnvironmentError as err:
6      print(err)

```

If we want to create a custom context manager we must create a class that provides two methods: `__enter__()` and `__exit__()`. Whenever a `with` statement is used on an instance of such a class, the `__enter__()` method is called and the return value is used for the `as variable` (or thrown away if there isn't one). When control leaves the scope of the `with` statement the `__exit__()` method is called (with details of an exception if one has occurred passed as arguments).

Suppose we want to perform several operations on a list in an atomic manner. For example, if we have a list of integers and want to append an integer, delete an integer, and change a couple of integers, all as a single operation, we could write code like this:

```

1  try:
2      with AtomicList(items) as atomic:
3          atomic.append(1111)
4          del atomic[3]
5          atomic[8] = 2222
6          atomic[index] = 3333
7  except (AttributeError, IndexError, ValueError) as err:
8      print('no changes applied:', err)

```

Here is the code for the `AtomicList` context manager:

```

1  class AtomicList:
2      def __init__(self, alist, shallow_copy=True):
3          self.original = alist
4          self.shallow_copy = shallow_copy
5
6      def __enter__(self):
7          self.modified = (self.original[:] if self.shallow_copy else copy.deepcopy(self.original))
8          return self.modified
9
10     def __exit__(self, exc_type, exc_val, exc_tb):
11         if exc_type is None: # exception type
12             self.original[:] = self.modified

```

If no exception occurred the `exc_type` ("exception type") will be `None` and we know that we can safely replace the original list's items with the items from the modified list. (We cannot do `self.original = self.modified` because that would just replace one object reference with another and would not affect the original list at all. There is no `return` in `__exit__()`) But if an exception occurred, we do nothing to the original list and the modified list is discarded.

The return value of `__exit__()` is used to indicate whether any exception that occurred should be propagated. A `True` value means that we have handled any exception and so no propagation should occur. Normally we always return `False` or something that evaluates to `False` in a Boolean context to allow any exception that occurred to propagate. By not giving an explicit `return` value, our `__exit__()` returns `None` which evaluates to `False` and correctly causes any exception to propagate.

8.2.4 Descriptors

Descriptors are **classes** which provide access control for the attributes of other **classes**. Any class that implements one or more of the descriptor special methods, `__get__()`, `__set__()`, and `__delete__()`, is called (and can be used as) a descriptor.

The built-in `property()` and `classmethod()` functions are implemented using descriptors. The key to understanding descriptors is that although we create an instance of a descriptor in a class as a class attribute, Python accesses the descriptor through the class's instances.

Lets imagine that we have a class whose instances hold some strings. We want to access the strings in the normal way, for example, as a property, but we also want to get an XML-escaped version of the strings whenever we want. One simple solution would be that whenever a string is set we immediately create an XML-escaped copy. But if we had thousands of strings and only ever read the XML version of a few of them, we would be wasting a lot of processing and memory for nothing. So we will create a descriptor that will provide XML-escaped strings on demand **without storing them**. Here's the client(owner) class, that is, the class uses the discriptor:

```

1 class Product:
2     __slots__ = ('__name', '__description', '__price')
3
4     name_as_xml = XmlShadow('name')
5     description_as_xml = XmlShadow('description')
6
7     def __init__(self, name, description, price):
8         self.__name = name
9         self.__description = description
10        self.__price = price
11
12    @property
13    def name(self):
14        return self.__name
15
16    @property
17    def description(self):
18        return self.__description
19
20    @description.setter

```

```

21     def description(self, description):
22         self.__description = description
23
24     @property
25     def price(self):
26         return self.__price
27
28     @price.setter
29     def price(self, price):
30         self.__price = price
31
32 product = Product("Chisel <3cm>", "Chisel & cap", 45.25)
33 print(product.name, product.name_as_xml, product.description_as_xml, sep='\n')
34 # Chisel <3cm>
35 # Chisel &lt;3cm&gt;
36 # Chisel &amp; cap

```

The `name_as_xml` and `description_as_xml` class attributes are set to be instances of the `XmlShadow` descriptor. Although no `Product` object has a `name_as_xml` attribute or a `description_as_xml` attribute, thanks to the descriptor we can write code like the previous. This works because when we try to access, for example `name_as_xml` attribute, Python finds that the `Product` class has a descriptor with that name, and so uses the descriptor to get the attribute's value.

```

1 from xml.sax.saxutils import escape
2
3 class XmlShadow:
4     def __init__(self, attribute_name):
5         self.attribute_name = attribute_name
6
7     def __get__(self, instance, owner=None):
8         return escape(getattr(instance, self.attribute_name))

```

When the `name_as_xml` or `description_as_xml` attribute is looked up, Python calls the descriptor's `__get__()` method. The `self` argument is the instance of the descriptor, the `instance` argument is the `Product` instance, and the `owner` argument is the owning class (`Product` in this case). We use the `getattr()` function to retrieve the relevant attribute from the product (in this case the relevant property), and return an XML-escaped version of it.

If the use case was that only a small proportion of the products were accessed for their XML strings, but the strings were often long and the same ones were frequently accessed, we could use a cache.

```

1 class CachedXmlShadow:
2     def __init__(self, attribute_name):
3         self.attribute_name = attribute_name
4         self.cache = {}
5
6     def __get__(self, instance, owner=None):
7         xml_text = self.cache.get(id(instance))
8         if xml_text is not None:
9             return xml_text
10        return self.cache.setdefault(id(instance), escape(getattr(instance, self.attribute_name)))

```

We store the unique identity of the instance as key rather than the instance itself because dictionary keys must be hashable, but we don't want to impose that as a requirement on classes that use the `CachedXmlShadow` descriptor. The key is necessary because descriptors are created per class rather than per instance.

Here's an example that use a descriptor to store all of an object's attribute data, with the object not needing to store anything itself.

```

1 class Point:
2     # By setting __slots__ to an empty tuple we ensure that the class cannot store
3     # any data attributes at all.
4     __slots__ = ()
5     x = ExternalStorage('x')
6     y = ExternalStorage('y')
7
8     def __init__(self, x=0, y=0):
9         self.x = x
10        self.y = y

```

By setting `__slots__` to an empty tuple we ensure that the class cannot store any data attributes at all. When `self.x` is assigned to, Python finds that there is a descriptor with the name “x”, and so uses the descriptor's `__set__()` method.

```

1 class ExternalStorage:
2     __slots__ = ('attribute_name',)
3     __storage = {} # class attribute
4
5     def __init__(self, attribute_name):
6         self.attribute_name = attribute_name
7
8     def __set__(self, instance, value):
9         self.__storage[id(instance), self.attribute_name] = value
10
11    def __get__(self, instance, owner):
12        if instance is None:
13            d = {}
14            for k in self.__storage.keys():
15                if self.attribute_name in k:
16                    d[k] = self.__storage[k]
17            return d
18        return self.__storage[id(instance), self.attribute_name]
19
20
21 p = Point(3, 4)
22 print(p.x, p.y) # 3 4
23 p = Point(1, 2)
24 print(p.x, p.y) # 1 2
25 print(Point.x, Point.y, Point, sep='\n')
26 # {(140338432304624, 'x'): 3, (140338432304640, 'x'): 1}
27 # {(140338432304624, 'y'): 4, (140338432304640, 'y'): 2}
28 # <class '__main__.Point'>

```

Although `__storage` is a class attribute, we can access it as `self.__storage`, because Python will look for it as an instance attribute, and not finding it will then look for it as a class attribute.

The implementation of the `__get__()` special method is slightly more sophisticated than before because we provide a means by which all the attribute

values in the `ExternalStorage` instance itself can be accessed.

We create the `Property` descriptor that mimics the behavior of the built-in `property()` function, at least for setters and getters. Here's the class that makes use of it:

```

1 class NameAndExtension:
2     def __init__(self, name, extension):
3         self.__name = name
4         self.extension = extension
5
6     @Property
7     def name(self):
8         return self.__name
9
10    @Property
11    def extension(self):
12        return self.__extension
13
14    @extension.setter
15    def extension(self, extension):
16        self.__extension = extension

```

Here's the `Property` decorator:

```

1 class Property:
2     def __init__(self, getter, setter=None):
3         self.__getter = getter
4         self.__setter = setter
5         self.__name__ = getter.__name__
6
7     def __get__(self, instance, owner=None):
8         if instance is None:
9             return self
10        return self.__getter(instance)
11
12    def __set__(self, instance, value):
13        if self.__setter is None:
14            raise AttributeError(f'"{self.__name__}" is read-only')
15        return self.__setter(instance, value)
16
17    def setter(self, setter):
18        self.__setter = setter
19        return self.__setter

```

The class's initializer takes one or two **functions** as arguments. If it is used as a decorator, it will get just the decorated function and this becomes the getter, while the setter is set to `None`. We use the getter's name as the property's name. So for each property, we have a getter, possibly a setter, and a name.

When a property is accessed we return the result of calling the getter function where we have passed the instance as its first parameter. At first sight, `self.__getter()` looks like a method call, but it is not. In fact, `self.__getter` is an attribute, one that happens to hold an object reference to a method that was passed. So what happens is that first we retrieve the attribute (`self.__getter`), and then we call it as a function (`()`). And because it is called as a function rather than a method we must pass in the relevant `self` object explicitly ourselves.

And in the case of a descriptor the `self` object is called `instance`.

The `setter()` method is called when the interpreter reaches, for example, `@extension.setter`, with the function it decorates as its `setter` argument. It stores the setter method it has been given (which can now be used in the `__set__()` method), and returns the setter, since decorator should return the function or method they decorate.

8.2.5 Class decorators

Class decorators takes a class object, and should return a class – normally a modified version of the class they decorate.

Example: delegate

Here's an example without class decorator:

```

1  __identity = lambda x: x
2
3
4  class SortedList:
5      def __init__(self, sequence=None, key=None):
6          self.__key = key or __identity
7          assert hasattr(self.__key, "__call__")
8          if sequence is None:
9              self.__list = []
10             elif isinstance(sequence, SortedList) and
11                 sequence.key == self.__key):
12                 self.__list = sequence.__list[:]
13             else:
14                 self.__list = sorted(list(sequence), key=self.__key)
15
16     def pop(self, index=-1):
17         return self.__list.pop(index)
18
19     def __delitem__(self, index):
20         del self.__list[index]
21
22     def __getitem__(self, index):
23         return self.__list[index]
24
25     def __setitem__(self, index, value):
26         raise TypeError("use add() to insert a value and rely on "
27                         "the list to put it in the right place")
28
29     def __iter__(self):
30         return iter(self.__list)
31
32     def __reversed__(self):
33         return reversed(self.__list)
34
35     def __len__(self):
36         return len(self.__list)
37
38     def __str__(self):
39         return str(self.__list)

```

Here's the decorated version:

```

1  @delegate("__list", ("pop", "__delitem__", "__getitem__",
2                      "__iter__", "__reversed__", "__len__", "__str__"))

```

```

3 class SortedList:
4
5     def __init__(self, sequence=None, key=None):
6         self.__key = key or _identity
7         assert hasattr(self.__key, "__call__")
8         if sequence is None:
9             self.__list = []
10        elif (isinstance(sequence, SortedList) and
11              sequence.key == self.__key):
12            self.__list = sequence.__list[:]
13        else:
14            self.__list = sorted(list(sequence), key=self.__key)

```

Here's class decorator:

```

1 def delegate(attribute_name, method_names):
2     def decorator(cls):
3         nonlocal attribute_name
4         if attribute_name.startswith('__'):
5             attribute_name = '_' + cls.__name__ + attribute_name
6         for name in method_names:
7             setattr(cls, name, eval("lambda self, *a, **kw: "
8                                     f"self.{attribute_name}.{name}(*a, **kw)"))
9         return cls
10
11    return decorator

```

We could not use a plain decorator because we want to pass arguments to the decorator, so we have instead created a function that takes our arguments and then returns a class decorator. The decorator itself takes a single argument, a class (just as a function decorator takes a single function or method as its argument).

We must use `nonlocal` so that the nested function uses the `attribute_name` from the outer scope rather than attempting to use one from its own scope. And we must be able to correct the attribute name if necessary to take account of the name mangling of private attributes. The decorator's behavior is quite simple: It iterates over all the method names that the `delegate()` function has been given, and for each one creates a new method which it sets as an attribute on the class with the given method name.

We have used `eval()` to create each of the delegated methods since it can be used to execute a single statement, and a `lambda` statement produces a method or function. For example, the code executed to produce the `pop()` method is:

```

1 lambda self, *a, **kw: self._SortedList__list.pop(*a, **kw)

```

Example: complete comparisons

In the following example, only `__lt__()` special method is supplied, and the other comparison method is created by the class decorator.

```

1 @complete_comparisons
2 class FuzzyBool:

```

```

3  def __init__(self, value=0.0):
4      self.__value = value if 0.0 <= value <= 1.0 else 0.0
5
6  def __lt__(self, other):
7      return self.__value < other.__value

```

Here's the decorator:

```

1  def complete_comparisons(cls):
2      assert cls.__lt__ is not object.__lt__, (
3          f'{cls.__name__} must define < and ideally =='
4      )
5      if cls.__eq__ is object.__eq__:
6          cls.__eq__ = lambda self, other: (
7              not (cls.__lt__(self, other) or cls.__lt__(other, self))
8          )
9      cls.__ne__ = lambda self, other: not cls.__eq__(self, other)
10     cls.__gt__ = lambda self, other: cls.__lt__(other, self)
11     cls.__le__ = lambda self, other: not cls.__lt__(other, self)
12     cls.__ge__ = lambda self, other: not cls.__lt__(self, other)

```

Given a class that defines only $<$ (or $<$ and $==$), the decorator produces the missing comparison operators by using the following logical equivalences:

$$\begin{aligned}
 x = y &\iff \neg (x < y \vee y < x) \\
 x \neq y &\iff \neg (x = y) \\
 x > y &\iff y < x \\
 x \leq y &\iff \neg (y < x) \\
 x \geq y &\iff \neg (x < y)
 \end{aligned}$$

Figure 8.2: Logical equivalences

In fact, Python automatically produces $>$ if $<$ is supplied, $!=$ if $==$ is supplied, and $>=$ if $<=$ is supplied, so it is sufficient to just implement the three operators $<$, $<=$, and $==$ and to leave Python to infer the others. However, using the class decorator reduces the minimum that we must implement to just $<$. This is convenient, and also ensures that all the comparison operators use the same consistent logic.

One problem that the decorator faces is that class `object` from which every other class is ultimately derived defines all six comparison operators, all of which raise a `TypeError` exception if used. So we need to know whether `<` and `==` have been reimplemented (and are therefore usable). This can easily be done by comparing the relevant special methods in the class being decorated with those in `object`.

Using class decorators is probably the simplest and most direct way of changing classes. Another approach is to use metaclasses.

8.2.6 Abstract base classes

An abstract base class (ABC) is a class that **cannot be used to create objects**. Instead, the purpose of such classes is **to define interface**, that is, to in effect list the methods and properites that classes that inherit the abstract base class must provide. This is useful because we can use an abstract base class as a kind of **promise** – a promise that any derived class will provide the methods and properites that the abstract base class specifies.

Abstract base classes are classes that have at least one abstract method or property.

All ABCs must have a metaclass of `abc.ABCMeta` (from the `abc` module), or from one of its subclasses.

Example: Appliance

```
1 import abc
2
3
4 class Appliance(metaclass=abc.ABCMeta):
5     @abc.abstractmethod
6     def __init__(self, model, price):
7         self.__model = model
8         self.price = price
9
10    def get_price(self):
11        return self.__price
12
13    def set_price(self, price):
14        self.__price = price
15
16    price = abc.abstractproperty(get_price, set_price)
17
18    @property
```

```

19     def model(self):
20         return self.__model

```

We have set the class's metaclass to be `abc.ABCMeta` since this is a requirement for ABCs. We have made `__init__()` an abstract method to ensure that it is reimplemented, and we have also provided an implementation which we expect (but can't force) inheritors to call. To make an abstract readable/writable property we cannot use decorator syntax; also we have not used private names for the getter and setter since doing so would be inconvenient for subclasses.

The `price` property is abstract (so we cannot use the `@property` decorator), and is readable/writable data as a property. We initialize the `price` in the `__init__()` method rather than setting the private data directly – this ensures that the setter is called (and may potentially do validation or other work, although it doesn't in this particular example).

The `model` property is not abstract, so subclasses don't need to reimplement it, and we can make it a property using the `@property` decorator.

Here is an example subclass:

```

1 class Cooker(Appliance):
2     def __init__(self, model, price, fuel):
3         super().__init__(model, price)
4         self.fuel = fuel
5
6     price = property(lambda self: super().price,
7                     lambda self, price: super().set_price(price))

```

Example: TextFilter

```

1 import abc
2
3
4 class TextFilter(metaclass=abc.ABCMeta):
5     @abc.abstractmethod
6     def is_transformer(self):
7         raise NotImplementedError()
8
9     @abc.abstractmethod
10    def __call__(self):
11        raise NotImplementedError()

```

The `TextFilter` ABC provides no functionality at all; it exists purely to define an interface. Since the abstract property and method have no implementations we don't want subclasses to call them, so instead of using an innocuous pass statement we raise an exception if they are used.

Here are some subclasses:

```

1 class CharCounter(TextFilter):
2     @property
3     def is_transformer(self):
4         return False

```

```

5
6     def __call__(self, text, chars):
7         count = 0
8         for c in text:
9             if c in chars:
10                 count += 1
11         return count
12
13
14 class RunLengthEncoder(TextFilter):
15     @property
16     def is_transformer(self):
17         return True
18
19     def __call__(self, utf8_string):
20         byte = None
21         count = 0
22         binary = bytearray()
23         for b in utf8_string.encode("utf8"):
24             if byte is None:
25                 if b == 0:
26                     binary.extend((0, 1, 0))
27                 else:
28                     byte = b
29                     count = 1
30             else:
31                 if byte == b:
32                     count += 1
33                     if count == 255:
34                         binary.extend((0, count, b))
35                         byte = None
36                         count = 0
37                 else:
38                     if count == 1:
39                         binary.append(byte)
40                     elif count == 2:
41                         binary.extend((byte, byte))
42                     elif count > 2:
43                         binary.extend((0, count, byte))
44                     if b == 0:
45                         binary.extend((0, 1, 0))
46                         byte = None
47                         count = 0
48                     else:
49                         byte = b
50                         count = 1
51             if count == 1:
52                 binary.append(byte)
53             elif count == 2:
54                 binary.extend((byte, byte))
55             elif count > 2:
56                 binary.extend((0, count, byte))
57         return bytes(binary)
58
59
60 class RunLengthDecoder(TextFilter):
61     @property
62     def is_transformer(self):
63         return True
64
65     def __call__(self, rle_bytes):
66         binary = bytearray()
67         length = None
68         for b in rle_bytes:
69             if length == 0:
70                 length = b
71             elif length is not None:
72                 binary.extend([b for x in range(length)])
73                 length = None
74             elif b == 0:
75                 length = 0
76             else:
77                 binary.append(b)

```

```

78         length = None
79         if length:
80             binary.extend([b for x in range(length)])
81         return binary.decode("utf8")
82
83
84 if __name__ == '__main__':
85     vowel_counter = CharCounter()
86     count = vowel_counter('dog fish and cat fish', 'aeiou')
87     print(count) # 5
88
89     print('=' * 100)
90     text = 'Mack Chyson ====='
91     encoder = RunLengthEncoder()
92     encoded_text = encoder(text)
93     print(encoded_text) # b'Mack Chyson \x00\x17='
94     decoder = RunLengthDecoder()
95     original_text = decoder(encoded_text)
96     print(original_text)
97     # Mack Chyson =====

```

Example: Abstract

```

1  class Undo(metaclass=abc.ABCMeta):
2      @abc.abstractmethod
3      def __init__(self):
4          self.__undos = []
5
6      @abc.abstractmethod
7      def can_undo(self):
8          return bool(self.__undos)
9
10     @abc.abstractmethod
11     def undo(self):
12         assert self.__undos, 'nothing left to undo'
13         self.__undos.pop()(self)
14
15     def add_undo(self, undo):
16         self.__undos.append(undo)
17
18     def clear(self):
19         self.__undos = []

```

The `self.__undos` list is expected to hold object references to methods. Each method must cause the corresponding action to be undone if it is called. So to perform an undo we pop the last undo method off the `self.__undos` list, and then call the method as a function, passing `self` as an argument. (We must pass `self` because the method is being called as a function not at a method.)

Here's `Stack` class:

```

1  class Stack(Undo):
2      def __init__(self):
3          super().__init__()
4          self.__stack = []
5
6      @property
7      def can_undo(self):
8          return super().can_undo
9
10     def undo(self):
11         super().undo()
12
13     def push(self, item):

```

```

14     self.__stack.append(item)
15     self.add_undo(lambda self: self.__stack.pop())
16
17     def pop(self):
18         item = self.__stack.pop()
19         self.add_undo(lambda self: self.__stack.append(item))
20         return item
21
22     def top(self):
23         assert self.__stack, 'Stack is empty'
24         return self.__stack[-1]
25
26     def __str__(self):
27         return str(self.__stack)

```

8.2.7 Multiple inheritance

Multiple inheritance is where one class inherits from two or more other classes. One problem is that multiple inheritance can lead to the same class being inherited more than once and this means that the version of a method that is called depends on the method resolution order, which potentially makes classes that use multiple inheritance somewhat fragile.

Multiple inheritance can generally be avoided by:

- Using single inheritance and setting a metaclass if we want to support an additional API
- Using multiple inheritance with one concrete class and one or more abstract base classes for additional APIs
- Using single inheritance and aggregate instances of other classes

Nonetheless, in some cases, multiple inheritance can provide a very convenient solution. For example, suppose we want to create a new version of the `Stack` class from the previous subsection, but want the class to support loading and saving using a pickle. We might well want to add the loading and saving functionality to several classes, so we will implement it in a class of its own:

```

1 class LoadSave:
2     def __init__(self, filename, *attribute_names):
3         self.filename = filename
4         self.__attribute_names = []
5         for name in attribute_names:
6             if name.startswith('__'):
7                 name = '_' + self.__class__.__name__ + name
8                 self.__attribute_names.append(name)
9
10    def save(self):
11        with open(self.filename, 'wb') as fh:
12            data = []
13            for name in self.__attribute_names:
14                data.append(getattr(self, name))
15            pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)

```



```

16
17     def load(self):
18         with open(self.filename, 'rb') as fh:
19             data = pickle.load(fh)
20             for name, value in zip(self.__attribute_names, data):
21                 setattr(self, name, value)

```

```

1 class FileStack(Undo, LoadSave):
2     def __init__(self, filename):
3         Undo.__init__(self)
4         LoadSave.__init__(self, filename, '__stack')
5         self.__stack = []
6
7     def load(self):
8         super().load()
9         Undo.clear(self)

```

Instead of using `super()` in the `__init__()` method we must specify the base classes that we initialize since `super()` cannot guess out intentions.

Multiple inheritance can be convenient and works well when the inheritance classes have no overlapping APIs.

8.2.8 Metaclasses

A metaclass is to a class what a class is to an instance; that is, a metaclass is used to create classes, just as classes are used to create instances. We can ask whether a class object inherits another class using `issubclass()`.

Register

The simplest use of metaclasses is to make custom classes fit into Python's standard ABC hierarchy.

```

1 import collections
2
3
4 class SortedList:
5     pass
6
7
8 collections.abc.Sequence.register(SortedList)

```

Registering a class like this makes it a **virtual subclass**. A virtual subclass reports that it is a subclass of the class or classes it is registered with, but does not inherit any data or methods from any of the classes it is registered with. Registering a class like this provides a promise that the class provides the API of the classes it is registered with, but does not provide any guarantee that it will honor its promise.

Guarantee

One use of metaclasses is to provide both a promise and a guarantee about a class's API. Another use is to modify a class in some way (like a class decorator does). And metaclasses can be used for both purposes at the same time.

Suppose we want to create a group of classes that all provide `load()` and `save()` methods. We can do this by creating a class that when used as a metaclass, checks that these methods are present:

```

1 # API guarantee
2 class LoadableSavable(type):
3     def __init__(cls, classname, bases, dictionary):
4         super().__init__(classname, bases, dictionary)
5
6         assert (
7             hasattr(cls, 'load') and
8             isinstance(getattr(cls, 'load'), collections.abc.Callable)
9         ), "class '" + classname + "' must provide a load() method"
10        assert (
11            hasattr(cls, 'save') and
12            isinstance(getattr(cls, 'save'), collections.abc.Callable)
13        ), "class '" + classname + "' must provide a save() method"
```

Classes that are to serve as metaclasses must inherit from the ultimate metaclass base class, `type`, or one of its subclasses.

Once the class has been created, the metaclass is initialized by calling its `__init__()` method. The arguments given to `__init__()` are `cls`, the class that just been created; `classname`, the class's name; `bases`, a list of the class's base classes (excluding `object`, and therefore possibly empty); and `dictionary` that holds the attributes that became class attributes when the `cls` class was created, unless we intervened in a reimplementaion of the metaclass's `__new__()` method.

```

1 # class Bad(metaclass=LoadableSavable):
2 #     def some_method(self): pass
3 # AssertionError: class 'Bad' must provide a load() method
4
5 class Good(metaclass=LoadableSavable):
6     def load(self): pass
7     def save(self): pass
8
9
10 g = Good()
```

Modify

We can use metaclasses to change the classes use them. If the change involves the name, base classes, or dictionary of the class being created, the we need to reimplement the metaclass's `__new__()` method; but for other changes, such as adding methods or data attributes, reimplementing `__init__()` is sufficient, although this can also be done in `__new__()`.

This example is just used to show the modification function. Normally the following code is not good. Suppose we did not use property decorators before, but use a simple naming convention to identify properties. Here, the class has methods of the form `get_name()` and `set_name()`, we would expect the class to have a private `__name` property accessed using `instance.name` for getting and setting. Here is the class:

```

1 class Product(metaclass=AutoSlotProperties):
2     def __init__(self, barcode, description):
3         self.__barcode = barcode
4         self.description = description
5
6     def get_barcode(self):
7         return self.__barcode
8
9     def get_description(self):
10        return self.__description
11
12    def set_description(self, description):
13        if description is None or len(description) < 3:
14            self.__description = '<Invalid Description>'
15        else:
16            self.__description = description

```

This can be done using a metaclass.

```

1 # modifying properties
2 class AutoSlotProperties(type):
3     def __new__(mcl, classname, bases, dictionary):
4         slots = list(dictionary.get('__slots__', [])) # get slots
5         for getter_name in [key for key in dictionary if key.startswith('get_')]:
6             if isinstance(dictionary[getter_name], collections.abc.Callable):
7                 name = getter_name[4:]
8                 slots.append('__' + name) # alter slots
9                 getter = dictionary.pop(getter_name)
10                setter_name = 'set_' + name
11                setter = dictionary.get(setter_name, None)
12                if setter is not None and isinstance(setter, collections.abc.Callable):
13                    del dictionary[setter_name]
14                    dictionary[name] = property(getter, setter) # convert to property
15                dictionary['__slots__'] = tuple(slots)
16            return super().__new__(mcl, classname, bases, dictionary)
17
18
19
20 product = Product('111', '8mm Stapler')
21 print(product.barcode, product.description) # 111 8mm Stapler
22 product.description = '8mm Stapler (long)'
23 print(product.barcode, product.description) # 111 8mm Stapler (long)
24
25 print(product.__slots__) # ('__barcode', '__description')
26
27 for i in dir(product):
28     print(i)
29 # __Product__barcode
30 # __Product__description
31 # __class__
32 # __delattr__
33 # __dir__
34 # __doc__
35 # __eq__
36 # __format__
37 # __ge__
38 # __getattr__
39 # __gt__
40 # __hash__
41 # __init__

```

```
42 # __init_subclass__
43 # __le__
44 # __lt__
45 # __module__
46 # __ne__
47 # __new__
48 # __reduce__
49 # __reduce_ex__
50 # __repr__
51 # __setattr__
52 # __sizeof__
53 # __slots__
54 # __str__
55 # __subclasshook__
56 # barcode
57 # description
```

8.3 Functional-style programming

Functional-style programming is an approach to programming where computations are built up from combining functions:

- that don't modify their arguments and
- that don't refer to or change the program's state, and
- that provide their results as return values.

Three concepts that are strongly associated with functional programming are

1. mapping
2. filtering
3. reducing

Mapping involves taking a function and an iterable and producing a new iterable (or a list) where each item is the result of calling the function on the corresponding item in the original iterable. This is supported by the built-in `map()` function, for example:

```
1 list(map(lambda x: x ** 2, [1, 2, 3, 4]))
2 Out[3]: [1, 4, 9, 16]
```

Filtering involves taking a function and an iterable and producing a new iterable where each item is from the original iterable – providing the function returns `True` when called on the item. The built-in `filter()` function supports this:

```

1 list(filter(lambda x: x > 0, [1, -2, 3, -4]))
2 Out[5]: [1, 3]

```

Reducing involves taking a function and an iterable and producing a single result value. The way this works is that the function is called on the iterables first two values, then on the computed result and the third value, then on the computed result and the fourth value, and so on, until all the values have been used. The `functools` module's `functools.reduce()` function supports this.

```

1 import functools
2 functools.reduce(lambda x, y: x * y, [1, 2, 3, 4])
3 Out[7]: 24
4 import operator
5 functools.reduce(operator.mul, [1, 2, 3, 4])
6 Out[9]: 24

```

The `operator` module has functions for all of Python's operators specifically to make functional-style programming easier.

Python provides some built-in reducing functions:

all() given an iterable, returns **True** if all the iterables items return **True** when **bool()** is applied to them;

any() returns **True** if any of the iterables items is **True**;

max() returns the largest item in the iterable;

min() returns the smallest item in the iterable;

sum() returns the sum of the iterables items.

8.3.1 Partial function application

Partial function application is the creation of a function from an existing function and some arguments to produce a new function that does what the original function did, but with some arguments fixed so that callers don't have to pass them. Here's a very simple example:

```

1 enumerate1 = functools.partial(enumerate, start=1)
2 for lino, line in enumerate1(lines):
3     process_line(lino, line)

```

Using partial function application can simplify our code, especially when we want to call the same functions with the same arguments again and again. For example:

```

1 reader = functools.partial(open, mode='rt', encoding='utf8')
2 writer = functools.partial(open, mode='wt', encoding='utf8')

```

```

1 Conv2D_ = functools.partial(Conv2D, kernel_size=(3, 3), activation='relu', padding='same')
2
3 h = Conv2D_(256)(input_layer)
4 h = Conv2D_(64)(h)
5
6 # The same full code is:
7 # h = Conv2D(256, (3, 3), activation='relu', padding='same')(input_layer)
8 # h = Conv2D(64, (3, 3), activation='relu', padding='same')(h)

```

8.3.2 Coroutines

Coroutines are functions whose processing can be suspended and resumed at specific points.

In Python, a coroutine is a function that takes its input from a **yield** expression. It may also send results to a receiver function (which itself must be a coroutine). Whenever a coroutine reaches a **yield** expression it suspends waiting for data; and once it receives data, it resumes execution from that point.

Composing pipelines

A pipeline is simply the composition of one or more functions where data items are sent to the first function, which then either discards the item (filters it out) or passes it on to the next function (either as is or transformed in some way). The second function receives the item from the first function and repeats the process, discarding or passing on the item (possibly transformed in a different way) to the next function, and so on. Items that reach the end are then output in some way.

One benefit of using pipelines is that we can read data items incrementally, often one at a time, and have to give the pipeline only enough data items to fill it (usually one or a few items per component). This can lead to significant **memory savings** compared with, say, reading an entire data set into memory and then processing it all in one go.

Here is an example – a file matcher that reads all the filenames given on the command line (including those in the directories given on the command line, recursively), and that output the absolute paths of those files that meet certain criteria.

```

1 import os
2 import sys
3 import functools
4
5

```

```

6 def coroutine(function):
7     @functools.wraps(function)
8     def wrapper(*args, **kwargs):
9         generator = function(*args, **kwargs)
10        next(generator)
11        return generator
12
13    return wrapper
14
15
16 @coroutine
17 def reporter():
18     while True:
19         filename = (yield)
20         print(filename)
21
22
23 @coroutine
24 def get_files(receiver):
25     """
26     A wrapper of os.walk()
27     :param receiver:
28     :return:
29     """
30     while True:
31         path = (yield)
32         if os.path.isfile(path):
33             receiver.send(os.path.abspath(path))
34         else:
35             for root, dirs, files in os.walk(path):
36                 for filename in files:
37                     receiver.send(os.path.abspath(os.path.join(root, filename)))
38
39
40 @coroutine
41 def suffix_matcher(receiver, suffixes):
42     while True:
43         filename = (yield)
44         if filename.endswith(suffixes):
45             receiver.send(filename)
46
47
48 @coroutine
49 def size_matcher(receiver, minimum=None, maximum=None):
50     while True:
51         filename = (yield)
52         size = os.path.getsize(filename)
53         if ((minimum is None or size >= minimum) and
54             (maximum is None or size <= maximum)):
55             receiver.send(filename)
56
57
58 if __name__ == '__main__':
59     # notice the order in coroutine
60     pipes = []
61     pipes.append(reporter)
62     pipes.append(size_matcher(pipes[-1], minimum=1024))
63     pipes.append(suffix_matcher(pipes[-1], (".png", ".jpg", ".jpeg", ".py")))
64     pipes.append(get_files(pipes[-1]))
65     pipeline = pipes[-1]
66     # Equal to
67     # pipeline = get_files(suffix_matcher(size_matcher(reporter(), minimum=1024), (".png", ".jpg", ".jpeg", ".py")))
68
69     try:
70         for file in sys.argv[1:]:
71             print(file)
72             pipeline.send(file)
73             # pipeline.py
74             # /Users/mike/PycharmProjects/python3/c8_advanced_programming_techniques/functional_programming/pipeline.py
75             # partial.py
76             # /Users/mike/PycharmProjects/python3/c8_advanced_programming_techniques/functional_programming/partial.py
77             # __init__.py
78     finally:

```

```
79     for pipe in pipes:  
80         pipe.close()
```

The `@coroutine` decorator takes a coroutine function, and calls the built-in `next()` function on it – this causes the function to be executed up to the first `yield` expression, ready to receive data.

Chapter 9

Debugging, testing, and profiling

Writing programs is a mixture of art, craft, and science, and because it is done by **humans**, mistakes are made.

Mistakes fall into several categories:

syntax error The program can not run.

logical error The program runs, but some aspect of its behavior is not what we intended or expected.

poor performance This is almost always due to a poor choice of algorithm or data structure or both.

9.1 Debugging

9.1.1 Dealing with syntax errors

```
1 for i in range(10)
2     print(1)
3
4 #   File "/Users/mike/PycharmProjects/python3/c9_debugging_testing_profiling/t.py", line 10
5 #       for i in range(10)
6 #           ^
7 # SyntaxError: invalid syntax
```

If we try to run a program that has a syntax error, Python will stop execution and print the filename, line number, and offending line, with a caret(^) underneath indicating exactly where the error was detected.

```

1 try:
2     if True:
3         print('')
4 except Exception as err:
5     print(err)
6
7 # File "/Users/mike/PycharmProjects/python3/c9_debugging_testing_profiling/t.py", line 21
8 #     except Exception as err:
9 #         ^
10 # SyntaxError: invalid syntax

```

There is no syntax error in the line indicated, so both the line number and the carets position are wrong. We have omited a parenthese, but Python didn't realize this until it reach the `except` keyword on the following line.

9.1.2 Dealing with runtime errors

```

1 def div():
2     1 / 0
3
4
5 if __name__ == '__main__':
6     div()
7
8 # Traceback (most recent call last):
9 #   File "/Users/mike/PycharmProjects/python3/c9_debugging_testing_profiling/e3.py", line 17, in <module>
10 #       div()
11 #   File "/Users/mike/PycharmProjects/python3/c9_debugging_testing_profiling/e3.py", line 13, in div
12 #       1 / 0
13 # ZeroDivisionError: division by zero

```

If an unhandled exception occurs at runtime, Python will stop executing our program and print a traceback. Tracebacks should be read from their last line back toward their first line. The last line specifies the unhandled exception that occurred. Above the line, the filename, line number, and function name, followed by the line that caused the exception, are shown.

9.1.3 Scientific debugging

To be able to kill a bug we must be able to do the following.

1. Reproduce the bug.
2. Locate the bug.
3. Fix the bug.
4. Test the fix.

Reproducing the bug is sometimes easy – it always occurs on every run; and sometimes hard – it occurs intermittently. In either case we should try

to reduce the bugs dependencies, that is, find the smallest input and the least amount of processing that can still produce the bug.

The scientific method of finding and fixing the bug has three steps:

1. Think up an explanation – a hypothesis – that reasonably accounts for the bug.
2. Create an experiment to test the hypothesis.
3. Run the experiment.

9.2 Unit testing

A key point of TDD (Test Driven Development) is that when we want to add a feature, we **first** write a test for it.

Python's standard library provides two unit testing modules, `doctest` and `unittest`. Creating doctests is straightforward: We write the tests in the module, function, class, and methods docstrings, and for modules, we simply add three lines at the end of the module:

```
1 if __name__ == "__main__":
2     import doctest
3     doctest.testmod()
```

To exercise the program's doctests there are two approaches:

1. Import the `doctest` module and then run the program – for example, at the console, `python -m doctest yourprogram.py`.
2. Create a separate test program using the `unittest` module.

9.3 Profiling

There are some programming habits that are good for performance:

- Prefer tuples to lists when read-only sequence is needed.
- Use generators rather than creating large tuples or lists to iterate over.
- Use Python's built-in data structures – `dicts`, `lists`, and `tuples` – rather than custom data structures implemented in Python, since the built-in ones are all very highly optimized.

- When creating large strings out of lots of small strings, instead of concatenating the small strings, accumulate them all in a list, and join the list of strings into a single string at the end.
- If an object (including a function or method) is accessed a large number of times using attribute access (e.g., when accessing a function in a module), or from a data structure, it may be better to create and use a local variable that refers to the object to provide faster access.

The `cProfile` module (or the `profile` module) can be used to compare the performance of functions and methods. And it also shows precisely what is being called and how long each call takes.

```

1 import cProfile
2 import math
3
4
5 def log(x, y):
6     return math.log(x, y)
7
8
9 code = """
10 for i in range(10000):
11     log(10, 2)
12 """
13 cProfile.run(code)
```

20003 function calls in 0.006 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.002	0.002	0.006	0.006	<string>:2(<module>)
10000	0.002	0.000	0.004	0.000	cprofile_.py:5(log)
1	0.000	0.000	0.006	0.006	{built-in method builtins.exec}
10000	0.002	0.000	0.002	0.000	{built-in method math.log}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler'}

The `ncalls` (“number of calls”) column lists the number of calls to the specified function. The `tottime` (“total time”) column lists the total time spent in the function, but excluding time spent inside functions called by the function. The first `percall` column lists the average time of each call to the function (`tottime // ncalls`). The `cumtime` (“cumulative time”) column lists the time spent in the function and includes the time spent inside functions called by the function. The second `percall` column lists the average time of each call to the function, including functions called by it.

The `cProfile` module allows us to profile code without instrumenting it. The command line to use is `python -m cProfile program_or_module.py`.

`MyModule.py`:

```

1 import math
2
3
4 def log(x, y):
5     return math.log(x, y)
6
7
8 for i in range(10000):
9     log(10, 2)

```

We can save the complement profile data and analyze it using the `pstats` module.

```

1
2 (base) mike@Mikes-MacBook-Pro c9_debugging_testing_profiling % python -m cProfile -o profile.dat MyModule.py
3 (base) mike@Mikes-MacBook-Pro c9_debugging_testing_profiling % python -m pstats profile.dat
4 Welcome to the profile statistics browser.
5 % read profile.dat
6 profile.dat% callers log
7     Random listing order was used
8     List reduced from 65 to 2 due to restriction <'log'>
9
10 Function                                was called by...
11                                     ncalls  tottime  cumtime
12 {built-in method math.log} <-      10000    0.001    0.001  MyModule.py:4(log)
13 MyModule.py:4(log)           <-      10000    0.002    0.003  MyModule.py:1(<module>)
14
15
16 profile.dat% callees log
17     Random listing order was used
18     List reduced from 65 to 2 due to restriction <'log'>
19
20 Function                                called...
21                                     ncalls  tottime  cumtime
22 {built-in method math.log} ->          10000    0.001    0.001  {built-in method math.log}
23 MyModule.py:4(log)           ->          10000    0.001    0.001  {built-in method math.log}
24
25
26 profile.dat% quit
27 Goodbye.

```


Chapter 10

Processes and threading

With the advent of **multicore** processors, it is more tempting and more practical than ever before to spread the processing load so as to get the most out of all the available cores. There are two main approaches to spreading the workload:

- multiple processes
- multiple threads

	advantage	disadvantage
multiple processes	each process runs independently	communication and data sharing can be inconvenient
multiple threads	can communicate simply by data sharing	more complex than single-threaded program

Table 10.1: multiple processes and multiple threads

10.1 Using the multiprocessing module

```
1  #!/usr/bin/env python3
2  """
3  @project: python3
4  @file: grepword_p
5  @author: mike
6  @time: 2021/2/22
```

```

7
8 @function:
9 Searches for a word specified on the command line in the files listed after the word.
10 This the parent program.
11 The corresponding child program is grepword_p_child.py.
12 """
13 import os
14 import sys
15 import subprocess
16 import optparse
17
18
19 def main():
20     child = os.path.join(os.path.dirname(__file__), 'grepword_p_child.py')
21     opts, word, args = parse_options()
22     filelist = get_files(args, opts.recurse)
23     files_per_process = len(filelist) // opts.count
24     # Usually the number of files wont be an exact multiple of the number of processes,
25     # so we increase the number of files the first process is given by the remainder.
26     start, end = 0, files_per_process + (len(filelist) % opts.count)
27     number = 1
28
29     pipes = []
30     while start < len(filelist):
31         command = [sys.executable, child]
32         if opts.debug:
33             command.append(str(number))
34         pipe = subprocess.Popen(command, stdin=subprocess.PIPE)
35         pipes.append(pipe)
36         pipe.stdin.write(word.encode('utf8') + b'\n')
37         for filename in filelist[start:end]:
38             pipe.stdin.write(filename.encode('utf8') + b'\n')
39         pipe.stdin.close()
40         number += 1
41         start, end = end, end + files_per_process
42
43     while pipes:
44         pipe = pipes.pop()
45         pipe.wait()
46
47
48 def parse_options():
49     parser = optparse.OptionParser(
50         usage=("usage: %prog [options] word name1 "
51              "[name2 [... nameN]]\n\n"
52              "names are filenames or paths; paths only "
53              "make sense with the -r option set"))
54     parser.add_option("-p", "--processes", dest="count", default=7,
55                      type="int",
56                      help=("the number of child processes to use (1..20) "
57                            "[default %default]"))
58     parser.add_option("-r", "--recurse", dest="recurse",
59                      default=False, action="store_true",
60                      help="recurse into subdirectories")
61     parser.add_option("-d", "--debug", dest="debug", default=False,
62                      action="store_true")
63     opts, args = parser.parse_args()
64     if len(args) == 0:
65         parser.error("a word and at least one path must be specified")
66     elif len(args) == 1:
67         parser.error("at least one path must be specified")
68     if (not opts.recurse and
69         not any([os.path.isfile(arg) for arg in args])):
70         parser.error("at least one file must be specified; or use -r")
71     if not (1 <= opts.count <= 20):
72         parser.error("process count must be 1..20")
73     return opts, args[0], args[1:]
74
75
76 def get_files(args, recurse):
77     filelist = []
78     for path in args:
79         if os.path.isfile(path):

```



```

80         filelist.append(path)
81     elif recurse:
82         for root, dirs, files in os.walk(path):
83             for filename in files:
84                 filelist.append(os.path.join(root, filename))
85     return filelist
86
87
88 main()

```

The `number` variable (line 22) is used purely for debugging so that we can see which process produce each line of output. For each `start:end` slice of the `filelist` we specify the Python interpreter (conveniently available in `sys.executable`) (line 26).

Once all the processes have started we wait for each child process to finish. This is not essential, but on Unix-like systems it ensures that we are returned to the console prompt when all the processes are done (otherwise, we must press Enter when they are all finished). Another benefit of waiting is that if we interrupt the program (e.g., by pressing Ctrl+C), all the processes that are still running will be interrupted and will terminate with an uncaught `KeyboardInterrupt` exception – if we did not wait the main program would finish (and therefore not be interruptible), and the child processes would continue (unless killed by a kill program or a task manager).

```

1  #!/usr/bin/env python3
2  """
3  @project: python3
4  @file: grepword_p_child
5  @author: mike
6  @time: 2021/2/22
7
8  @function:
9  """
10 import sys
11
12 coding = 'utf8'
13 BLOCK_SIZE = 8000
14 number = f'{sys.argv[1]}' if len(sys.argv) == 2 else ''
15 stdin = sys.stdin.buffer.read()
16 lines = stdin.decode(coding, 'ignore').splitlines()
17 word = lines[0].rstrip()
18
19 for filename in lines[1:]:
20     filename = filename.rstrip()
21     previous = ''
22     try:
23         with open(filename, 'rb') as fh:
24             while True:
25                 current = fh.read(BLOCK_SIZE)
26                 if not current:
27                     break
28                 current = current.decode(coding, 'ignore')
29                 if word in current or word in previous[-len(word):] + current[:len(word)]:
30                     print(f'{number}{filename}')
31                     break
32                 if len(current) != BLOCK_SIZE:
33                     break
34                 previous = current
35     except EnvironmentError as err:
36         print(f'{number}{err}')

```

It is possible that some of the files might be very large and this could be a problem, especially if there are 20 child processes running concurrently, all reading big files. We handle this by reading each file in blocks, keeping the previous block read to ensure that we don't miss cases when the only occurrence of the search word happens to fall across two blocks.

10.2 Using the threading module

Setting up two or more separate threads of execution in Python is quite straightforward. The complexity arises when we want to separate threads to share data.

One common solution is to use some kind of locking mechanism.

Every Python program has at least one thread, the main thread. To create multiple threads we must import the `threading` module and use that to create as many additional threads as want. There are two ways to create threads:

1. We can call `threading.Thread()` and pass it a callable object
2. We can subclass the `threading.Thread` class.

```

1  #!/usr/bin/env python3
2  """
3  @project: python3
4  @file: grepword_t
5  @author: mike
6  @time: 2021/2/22
7
8  @function:
9  """
10
11 import queue
12 import os
13 import threading
14
15 BLOCK_SIZE = 8000
16 from grepword_p import parse_options, get_files
17
18
19 def main():
20     opts, word, args = parse_options()
21     filelist = get_files(args, opts.recurse)
22     work_queue = queue.Queue()
23     for i in range(opts.count):
24         number = f'{i + 1}: ' if opts.debug else ''
25         worker = Worker(work_queue, word, number)
26         worker.daemon = True
27         worker.start()
28     for filename in filelist:
29         work_queue.put(filename)
30     work_queue.join()
31
32
33 class Worker(threading.Thread):

```

```

34     def __init__(self, work_queue, word, number):
35         super().__init__()
36         self.work_queue = work_queue
37         self.word = word
38         self.number = number
39
40     def run(self) -> None:
41         while True:
42             try:
43                 filename = self.work_queue.get()
44                 self.process(filename)
45             finally:
46                 self.work_queue.task_done()
47
48     def process(self, filename):
49         previous = ""
50         try:
51             with open(filename, "rb") as fh:
52                 while True:
53                     current = fh.read(BLOCK_SIZE)
54                     if not current:
55                         break
56                     current = current.decode('utf8', 'ignore')
57                     if (self.word in current or
58                         self.word in previous[-len(self.word):] +
59                         current[:len(self.word)]):
60                         print("{0}{1}".format(self.number, filename))
61                         break
62                     if len(current) != BLOCK_SIZE:
63                         break
64                     previous = current
65         except EnvironmentError as err:
66             print("{0}{1}".format(self.number, err))

```

The program will not terminate while it has any threads running. This is a problem because once the worker threads have done their work, although they have finished they are technically still running. The solution is to turn the threads into daemons. The effect of this is that the program will terminate as soon as the program has no non-daemon threads running. The main thread is not a daemon, so once the main thread finishes, the program will cleanly terminate each daemon thread and then terminate itself. Of course, this can now create the opposite problem – once the threads are up and running we must ensure that the main thread does not finish until the work is done. This is achieved by calling `queue.Queue.join()` – this method blocks until the queue is empty.

We have made the `run()` method infinite loop. This is common for daemon threads. Once we have a file we process it, and afterward we must tell the queue that we have done that particular job – calling `queue.Queue.task_done()` is essential to the correct working of `queue.Queue.join()`.

```

1  #!/usr/bin/env python3
2  """
3  @project: python3
4  @file: findduplicates_t
5  @author: mike
6  @time: 2021/2/23
7
8  @function:
9  The program iterates over all the files in the current directory (or the specified path),
10 recursively going into subdirectories. It compares the lengths of all the files with the

```

```

11 same name, and for those files that have the same name and the same size it then uses
12 the MD5 (Message Digest) algorithm to check whether the files are the same, reporting
13 any that are.
14 """
15
16 import collections
17 import os
18 import queue
19 import threading
20 import hashlib
21 import optparse
22
23
24 def main():
25     # parse commandline arguments
26     opts, path = parse_options()
27     # prepare the data
28     data = collections.defaultdict(list)
29     for root, dirs, files in os.walk(path):
30         for filename in files:
31             fullname = os.path.join(root, filename)
32             try:
33                 key = (os.path.getsize(fullname), filename)
34             except EnvironmentError:
35                 continue
36
37             if key[0] == 0:
38                 continue
39
40             data[key].append(fullname)
41
42     # Create the worker threads
43     work_queue = queue.PriorityQueue()
44     results_queue = queue.Queue()
45     # Reduce the duplicate computation of the same file
46     md5_from_filename = {}
47     for i in range(opts.count):
48         number = f'{i + 1}': ' if opts.debug else ''
49         worker = Worker(work_queue, md5_from_filename, results_queue, number)
50         worker.daemon = True
51         worker.start()
52
53     # Create the result thread
54     result_thread = threading.Thread(target=lambda: print_results(results_queue))
55     result_thread.daemon = True
56     result_thread.start()
57
58     for size, filename in sorted(data):
59         names = data[size, filename]
60         if len(names) > 1:
61             work_queue.put((size, names))
62         # Blocks until all items in the Queue have been gotten and processed.
63         work_queue.join()
64         results_queue.join()
65
66
67 def print_results(results_queue):
68     while True:
69         try:
70             results = results_queue.get()
71             if results:
72                 print(results)
73         finally:
74             results_queue.task_done()
75
76
77 class Worker(threading.Thread):
78     # class attribute
79     Md5_lock = threading.Lock()
80
81     def __init__(self, work_queue, md5_from_filename, results_queue, number):
82         super().__init__()
83         self.work_queue = work_queue

```

```

84         self.md5_from_filename = md5_from_filename
85         self.results_queue = results_queue
86         self.number = number
87
88     def run(self):
89         while True:
90             try:
91                 size, names = self.work_queue.get()
92                 self.process(size, names)
93             finally:
94                 self.work_queue.task_done()
95
96     def process(self, size, filenames):
97         md5s = collections.defaultdict(set)
98         for filename in filenames:
99             with self.Md5_lock:
100                 md5 = self.md5_from_filename.get(filename, None)
101                 if md5 is not None:
102                     md5s[md5].add(filename)
103                 else:
104                     try:
105                         md5 = hashlib.md5()
106                         with open(filename, 'rb') as fh:
107                             md5.update(fh.read())
108                         md5 = md5.digest()
109                         md5s[md5].add(filename)
110                         with self.Md5_lock:
111                             self.md5_from_filename[filename] = md5
112                     except EnvironmentError:
113                         continue
114
115         for filenames in md5s.values():
116             if len(filenames) == 1:
117                 continue
118             self.results_queue.put(
119                 "{0} Duplicate files ({1:n} bytes): {2}\n".format(self.number, size, "\n\t".join(sorted(filenames)))
120             )
121
122
123 def parse_options():
124     parser = optparse.OptionParser(
125         usage=("usage: %prog [options] [path]\n"
126              "outputs a list of duplicate files in path "
127              "using the MD5 algorithm\n"
128              "ignores zero-length files\n"
129              "path defaults to ."))
130     parser.add_option("-t", "--threads", dest="count", default=7,
131                      type="int",
132                      help=("the number of threads to use (1..20) "
133                            "[default %default]"))
134     parser.add_option("-v", "--verbose", dest="verbose",
135                      default=False, action="store_true")
136     parser.add_option("-d", "--debug", dest="debug", default=False,
137                      action="store_true")
138     opts, args = parser.parse_args()
139     if not (1 <= opts.count <= 20):
140         parser.error("thread count must be 1..20")
141     return opts, args[0] if args else "."
142
143
144 main()

```

Whether we access the `md5_from_filename` dictionary to read it or to write it, we put the access in the context of a lock (line 79). Instances of the `threading.Lock()` class are context managers that acquire the lock on entry and release the lock on exit. The `with` statements will block if another thread has the `Md5_lock`, until the lock is released.

Chapter 11

Networking

Networking allows computer programs to **communicate** with each other, even if they are running on different machines.

11.1 Console tool

```
1  #!/usr/bin/env python3
2  """
3  @project: python3
4  @file: Console
5  @author: mike
6  @time: 2021/2/23
7
8  @function:
9  """
10 import sys
11 import datetime
12
13
14 class _RangeError(Exception):
15     pass
16
17
18 def get_string(message, name="string", default=None,
19               minimum_length=0, maximum_length=80,
20               force_lower=False):
21     message += ": " if default is None else f" [{default}]: "
22     while True:
23         try:
24             line = input(message)
25             if not line:
26                 if default is not None:
27                     return default
28                 if minimum_length == 0:
29                     return ""
30             else:
31                 raise ValueError(f'{name} may not be empty')
32             if not (minimum_length <= len(line) <= maximum_length):
33                 raise ValueError("{0} must have at least {1} and "
34                                   "at most {2} characters".format(
35                                       name, minimum_length, maximum_length))
36             return line if not force_lower else line.lower()
```

```

37         except ValueError as err:
38             print("ERROR", err)
39
40
41     def get_integer(message, name="integer", default=None, minimum=None,
42                     maximum=None, allow_zero=True):
43         message += ": " if default is None else f" [{default}]: "
44         while True:
45             try:
46                 line = input(message)
47                 if not line and default is not None:
48                     return default
49                 x = int(line)
50                 if x == 0:
51                     if allow_zero:
52                         return x
53                     else:
54                         raise _RangeError(f"{name} may not be 0")
55                 if ((minimum is not None and minimum > x) or
56                     (maximum is not None and maximum < x)):
57                     raise _RangeError("{0} must be between {1} and {2} "
58                                         "inclusive {3}".format(name, minimum, maximum,
59                                                                 (" (or 0)" if allow_zero else "")))
60                 return x
61             except _RangeError as err:
62                 print("ERROR", err)
63             except ValueError as err:
64                 print("ERROR {0} must be an integer".format(name))
65
66
67     def get_float(message, name="float", default=None, minimum=None,
68                  maximum=None, allow_zero=True):
69         message += ": " if default is None else f" [{default}]: "
70         while True:
71             try:
72                 line = input(message)
73                 if not line and default is not None:
74                     return default
75                 x = float(line)
76                 if abs(x) < sys.float_info.epsilon:
77                     if allow_zero:
78                         return x
79                     else:
80                         raise _RangeError(f"{name} may not be 0.0")
81                 if ((minimum is not None and minimum > x) or
82                     (maximum is not None and maximum < x)):
83                     raise _RangeError("{0} must be between {1} and {2} "
84                                         "inclusive {3}".format(name, minimum, maximum,
85                                                                 (" (or 0.0)" if allow_zero else "")))
86                 return x
87             except _RangeError as err:
88                 print("ERROR", err)
89             except ValueError as err:
90                 print("ERROR {0} must be a float".format(name))
91
92
93     def get_bool(message, default=None):
94         yes = frozenset({"l", "y", "yes", "t", "true", "ok"})
95         message += " (y/yes/n/no)"
96         message += ": " if default is None else f" [{default}]: "
97         line = input(message)
98         if not line and default is not None:
99             return default in yes
100         return line.lower() in yes
101
102
103     def get_date(message, default=None, format="%y-%m-%d"):
104         # message should include the format in human-readable form, e.g.
105         # for %y-%m-%d, "YY-MM-DD".
106         message += ": " if default is None else f" [{default}]: "
107         while True:
108             try:
109                 line = input(message)

```



```

110         if not line and default is not None:
111             return default
112         return datetime.datetime.strptime(line, format)
113     except ValueError as err:
114         print("ERROR", err)
115
116
117 def get_menu_choice(message, valid, default=None, force_lower=False):
118     message += ": " if default is None else " [{0}]: ".format(default)
119     while True:
120         line = input(message)
121         if not line and default is not None:
122             return default
123         if line not in valid:
124             print("ERROR only {0} are valid choices".format(
125                 ", ".join(["{0}".format(x)
126                             for x in sorted(valid)])))
127         else:
128             return line if not force_lower else line.lower()

```

11.2 Creating a TCP client

```

1  #!/usr/bin/env python3
2  """
3  @project: python3
4  @file: car_registration
5  @author: mike
6  @time: 2021/2/23
7
8  @function:
9  """
10 import sys
11 import Console
12 import collections
13 import struct
14 import pickle
15 import socket
16
17 Address = ['localhost', 9653]
18 CarTuple = collections.namedtuple("CarTuple", "seats mileage owner")
19
20
21 def main():
22     if len(sys.argv) > 1:
23         Address[0] = sys.argv[1]
24     call = dict(c=get_car_details,
25               m=change_mileage,
26               o=change_owner,
27               n=new_registration,
28               s=stop_server,
29               q=quit_)
30     menu = '(C)ar Edit (M)ileage Edit (O)wner Edit (N)ew car (S)top server (Q)uit '
31     valid = frozenset('cmnsq')
32     previous_license = None
33     while True:
34         action = Console.get_menu_choice(menu, valid, 'c', True)
35         previous_license = call[action](previous_license)
36
37
38 def get_car_details(previous_license):
39     license, car = retrieve_car_details(previous_license)
40     if car is not None:
41         print('License: {0}\nSeats: {seats}\nMileage: {mileage}\n'
42               'Owner: {owner}'.format(license, **car._asdict()))
43     return license
44
45

```

```

46 def retrieve_car_details(previous_license):
47     license = Console.get_string('License', 'license', previous_license)
48     if not license:
49         return previous_license, None
50     license = license.upper()
51     ok, *data = handle_request('GET_CAR_DETAILS', license)
52     if not ok:
53         print(data[0])
54         return previous_license, None
55     return license, CarTuple(*data)
56
57
58 def change_mileage(previous_license):
59     license, car = retrieve_car_details(previous_license)
60     if car is None:
61         return previous_license
62     mileage = Console.get_integer('Mileage', 'mileage', car.mileage, 0)
63     if mileage == 0:
64         return license
65     ok, *data = handle_request('CHANGE_MILEAGE', license, mileage)
66     if not ok:
67         print(data[0])
68     else:
69         print('Mileage successfully changed')
70     return license
71
72
73 def change_owner(previous_license):
74     license, car = retrieve_car_details(previous_license)
75     if car is None:
76         return previous_license
77     owner = Console.get_string('Owner', 'owner', car.owner)
78     if not owner:
79         return license
80     ok, *data = handle_request('CHANGE_OWNER', license, owner)
81     if not ok:
82         print(data[0])
83     else:
84         print('Owner successfully changed')
85     return license
86
87
88 def new_registration(previous_license):
89     license = Console.get_string('License', 'license')
90     if not license:
91         return previous_license
92     license = license.upper()
93     seats = Console.get_integer('Seats', 'seats', 4, 0)
94     if not (1 < seats < 10):
95         return previous_license
96     mileage = Console.get_integer('Mileage', 'mileage', 0, 0)
97     owner = Console.get_string('Owner', 'owner')
98     if not owner:
99         return previous_license
100
101     ok, *data = handle_request('NEW_REGISTRATION', license, seats, mileage, owner)
102     if not ok:
103         print(data[0])
104     else:
105         print(f'Car {license} successfully registered')
106     return license
107
108
109 def quit_(*ignore):
110     sys.exit()
111
112
113 def stop_server(*ignore):
114     handle_request('SHUTDOWN', wait_for_reply=False)
115     sys.exit()
116
117
118 def handle_request(*items, wait_for_reply=True):

```

```

119 SizeStruct = struct.Struct('!I')
120 data = pickle.dumps(items, 3) # 3 is protocol version
121
122 try:
123     with SocketManager(tuple(Address)) as sock:
124         sock.sendall(SizeStruct.pack(len(data)))
125         sock.sendall(data)
126
127         if not wait_for_reply:
128             return
129
130         size_data = sock.recv(SizeStruct.size)
131         size = SizeStruct.unpack(size_data)[0]
132         result = bytearray()
133         while True:
134             data = sock.recv(4000)
135             if not data:
136                 break
137             result.extend(data)
138             if len(result) >= size:
139                 break
140         return pickle.loads(result)
141 except socket.error as err:
142     print(f'{err}: is the server running?')
143     sys.exit(1)
144
145
146 class SocketManager:
147     def __init__(self, address):
148         self.address = address
149
150     def __enter__(self):
151         # AF_INET: address family ipv4
152         # SOCK_STREAM: TCP
153         self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
154         self.sock.connect(self.address)
155         return self.sock
156
157     def __exit__(self, *ignore):
158         self.sock.close()
159
160
161 main()

```

If we choose to quit the program we do a clean termination by calling `sys.exit()`. Every menu function is called with the previous license, but we don't care about the argument in this particular case. We cannot write `def quit()`: because that would create a function that expects no arguments and so when the function was called with the previous license a `TypeError` exception would be raised saying that no arguments were expected but that one was given. So instead we specify a parameter of `*ignore` which can take any number of positional arguments.

11.3 Creating a TCP server

```

1  #!/usr/bin/env python3
2  """
3  @project: python3
4  @file: car_registration_server
5  @author: mike

```

```

6 @time: 2021/2/23
7
8 @function:
9 """
10 import os
11 import pickle
12 import sys
13 import gzip
14 import socketserver
15 import threading
16 import struct
17 import copy
18 import random
19
20
21 class Car:
22     def __init__(self, seats, mileage, owner):
23         self.__seats = seats
24         self.mileage = mileage
25         self.owner = owner
26
27     @property
28     def seats(self):
29         return self.__seats
30
31     @property
32     def mileage(self):
33         return self.__mileage
34
35     @mileage.setter
36     def mileage(self, mileage):
37         self.__mileage = mileage
38
39     @property
40     def owner(self):
41         return self.__owner
42
43     @owner.setter
44     def owner(self, owner):
45         self.__owner = owner
46
47     def __str__(self):
48         return f'{self.seats}, {self.mileage}, {self.owner}'
49
50
51 class Finish(Exception):
52     pass
53
54
55 def main():
56     filename = os.path.join(os.path.dirname(__file__), 'car_registration.dat')
57     cars = load(filename)
58     print(f'Loaded {len(cars)} car registrations')
59     RequestHandler.Cars = cars # set Cars attribute into RequestHandler
60     server = None
61     try:
62         server = CarRegistrationServer(('', 9653), RequestHandler)
63         server.serve_forever()
64     except Exception as err:
65         print('ERROR', err)
66     finally:
67         if server is not None:
68             server.shutdown()
69             save(filename, cars)
70             print(f'Save {len(cars)} car registrations')
71
72
73 def load(filename):
74     if not os.path.exists(filename):
75         # Generate fake data
76         cars = {}
77         owners = []
78         for forename, surname in zip(

```

```

79         ('Warisha', 'Elysha', 'Liona',
80          'Kassandra', 'Simone', 'Halima', 'Liona', 'Zack',
81          'Josiah', 'Sam', 'Braedon', 'Eleni'),
82         ('Chandler', 'Drennan', 'Stead', 'Doole', 'Reneau',
83          'Dent', 'Sheckles', 'Dent', 'Reddihough', 'Dodwell',
84          'Conner', 'Abson')):
85     owners.append(forename + ' ' + surname)
86     for license in (
87         '1H1890C', 'PHV449', 'ABK3035', '215 MZN',
88         '6DQX521', '174-WWA', '9999991', 'DA 4020', '303 LNM',
89         'BEQ 0549', '1A US923', 'A37 4791', '393 TUT', '458 ARW',
90         '024 HYR', 'SKM 648', '1253 QA', '4EB S80', 'BYC 6654',
91         'SRK-423', '3DB 09J', '3C-5772F', 'PYJ 996', '768-VHN',
92         '262 2636', 'WYZ-94L', '326-PKF', 'EJB-3105', 'XXN-5911',
93         'HVP 283', 'EKW 6345', '069 DSM', 'GZB-6052', 'HGD-498',
94         '833-132', '1XG 831', '831-THB', 'HMR-299', 'A04 4HE',
95         'ERG 827', 'XVT-2416', '306-XXL', '530-NBE', '2-4JHJ'):
96         mileage = random.randint(0, 100000)
97         seats = random.choice((2, 4, 5, 6, 7))
98         owner = random.choice(owners)
99         cars[license] = Car(seats, mileage, owner)
100     return cars
101
102     try:
103         with gzip.open(filename, 'rb') as fh:
104             cars = pickle.load(fh)
105             print(cars)
106             return cars
107     except (EnvironmentError, pickle.UnpicklingError) as err:
108         print(f'server cannot load data: {err}')
109         sys.exit(1)
110
111
112 def save(filename, cars):
113     try:
114         with gzip.open(filename, 'wb') as fh:
115             pickle.dump(cars, fh, 3)
116     except (EnvironmentError, pickle.UnpicklingError) as err:
117         print(f'server failed to save data: {err}')
118         sys.exit(1)
119
120
121 class CarRegistrationServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
122     pass
123
124
125 class RequestHandler(socketserver.StreamRequestHandler):
126     CarsLock = threading.Lock()
127     CallLock = threading.Lock()
128
129     Call = dict(
130         GET_CAR_DETAILS=lambda self, *args: self.get_car_details(*args),
131         CHANGE_MILEAGE=lambda self, *args: self.change_mileage(*args),
132         CHANGE_OWNER=lambda self, *args: self.change_owner(*args),
133         NEW_REGISTRATION=lambda self, *args: self.new_registration(*args),
134         SHUTDOWN=lambda self, *args: self.shutdown(*args)
135     )
136
137     def handle(self) -> None:
138         SizeStruct = struct.Struct('!I')
139         size_data = self.rfile.read(SizeStruct.size)
140         size = SizeStruct.unpack(size_data)[0]
141         data = pickle.loads(self.rfile.read(size))
142
143         try:
144             with RequestHandler.CallLock:
145                 function = self.Call[data[0]]
146                 reply = function(self, *data[1:])
147             except Finish:
148                 return
149             data = pickle.dumps(reply, 3)
150             self.wfile.write(SizeStruct.pack(len(data)))
151             self.wfile.write(data)

```

```

152
153     def shutdown(self, *ignore):
154         self.server.shutdown()
155         raise Finish()
156
157     def get_car_details(self, license):
158         with RequestHandler.CarsLock:
159             car = copy.copy(self.Cars.get(license, None))
160             if car is not None:
161                 return True, car.seats, car.mileage, car.owner
162             return False, 'This license is not registered'
163
164     def change_mileage(self, license, mileage):
165         if mileage < 0:
166             return False, 'Cannot set a negative mileage'
167         with RequestHandler.CarsLock:
168             car = self.Cars.get(license, None)
169             if car is not None:
170                 if car.mileage < mileage:
171                     car.mileage = mileage
172                     return True, None
173                 return False, 'Cannot wind the odometer back'
174             return False, 'This license is not registered'
175
176     def change_owner(self, license, owner):
177         with RequestHandler.CarsLock:
178             car = self.Cars.get(license, None)
179             if car is not None:
180                 car.owner = owner
181                 return True, None
182             return False, 'This license is not registered'
183
184     def new_registration(self, license, seats, mileage, owner):
185         if not license:
186             return False, 'Cannot set an empty license'
187         if seats not in {2, 4, 5, 6, 7, 8, 9}:
188             return False, 'Cannot register car with invalid seats'
189         if mileage < 0:
190             return False, 'Cannot set a negative mileage'
191         if not owner:
192             return False, 'Cannot set an empty owner'
193
194         with RequestHandler.CarsLock:
195             if license not in self.Cars:
196                 self.Cars[license] = Car(seats, mileage, owner)
197                 return True, None
198             return False, 'Cannot register duplicate license'
199
200
201     main()

```

Since the code for creating servers often follows the same design, rather than having to use the low-level `socket` module, we can use the high-level `socketserver` module which takes care of all the housekeeping for us. All we have to do is provide a request handler class with a `handle()` method which is used to read requests and write replies.

Our request handler class needs to be able to access the `cars` dictionary, but we cannot pass the dictionary to an instance because the server creates the instances for us — one to handle each request. So we set the dictionary to the `RequestHandler.Cars` class variable where it is accessible to all instances.

Note that the `socketserver` mixin class we used must always be inherited first. This is to ensure that the mixin class methods are used in preference to

the second class methods for those methods that are provided by both, since Python looks for methods in the base classes in the order in which the base classes are specified, and uses the first suitable method it finds.

The `RequestHandler.Cars` dictionary is a class variable that was added in the `main()` function; it holds all the registration data. Adding additional attributes to objects (such as classes and instances) can be done outside the class (in this case in the `main()` function) without formality (as long as the object has a `__dict__`), and can be very convenient.

The `Call` dictionary is another class variable. We cannot use the methods directly because there is no `self` available at the class level. The solution we have used is to provide wrapper functions that will get `self` when they are called, and which in turn call the appropriate method with the given `self` and any other arguments.

11.4 Summary

Creating network clients and servers can be quite straightforward in Python thanks to the standard library's networking modules, and the `struct` and `pickle` modules.

Chapter 12

Database Programming

There are two commonly used database:

1. RDBMS (Relational Database Management System). These systems use tables (spreadsheet-like grids) with rows equating to records and columns equating to fields. The tables and the data they hold are created and manipulated using statements written in SQL (Structured Query Language).
2. DBM (Database Manager). It stores any number of key-value items.

12.1 DBM databases

The `shelve` module provides a wrapper around a DBM that allows us to interact with the DBM as though it were a dictionary, providing that we use only string keys and pickable values. Behind the scenes the `shelve` module converts the keys and values to and from `bytes` objects.

https://github.com/mikechyson/python3/blob/master/c12_database/dvds_dbm.py

12.2 SQL databases

To make it as easy as possible to switch between database backends, PEP 249 (Python Database API Specification v2.0) provides an API specification called DB-API 2.0 that database interfaces ought to honor. There are two major objects specified by the API, the connection object and the cursor object.

Syntax	Description
<code>db.close()</code>	Closes the connection.
<code>db.commit()</code>	Commits any pending transaction to the database; does nothing for databases that don't support transactions.
<code>db.cursor()</code>	Returns a database cursor object through which queries can be executed.
<code>db.rollback()</code>	Rolls back any pending transaction to the state that existed before the transaction began; does nothing for databases that don't support transactions.

Table 12.1: DB-API 2.0 Connection Object Methods

Syntax	Description
<code>c.arraysize</code>	The (readable/writable) number of rows that <code>fetchall()</code> will return if no size is specified
<code>c.fetchmany(size)</code>	Returns a sequence of rows (each row itself being a sequence); <code>size</code> default to <code>c.arraysize</code>
<code>c.fetchall()</code>	Returns a sequence of all the rows that have not yet been fetched
<code>c.fetchone()</code>	Returns the next row of the query result set as a sequence, or <code>None</code> when the results are exhausted.
<code>c.description</code>	A read-only sequence of 7-tuples (<code>name</code> , <code>type_code</code> , <code>display_size</code> , <code>internal_size</code> , <code>precision</code> , <code>scale</code> , <code>null_ok</code>), describing each successive column of cursor <code>c</code>
<code>c.execute(sql, params)</code>	Executes the SQL query in string <code>sql</code> , replacing each placeholder with the corresponding parameter from the <code>params</code> sequence of mapping if given
<code>c.execute(sql, seqofparams)</code>	Executes the SQL query once for each item in the <code>seq_of_params</code> sequence of sequences or mappings; this method should not be used for operations that create result sets (such as <code>SELECT</code> statements)
<code>c.close()</code>	Closes the cursor, <code>c</code> ; this is done automatically when the cursor goes out of scope

Table 12.2: DB-API 2.0 Cursor Object Attributes and Methods

https://github.com/mikechyson/python3/blob/master/c12_database/dvds_sql.py

Chapter 13

Regular Expressions

A regular expression is a compact notation for representing a collection of strings. What makes regular expressions so powerful is that a **single** regular expression can represent an **unlimited** number of strings.

Regexes (REGular EXpression) are used for five main purposes:

Parsing

Searching

Searching and replacing

Splitting strings

Validation

The core of regular expression is **matching**. All the above five purposes is based on the matching.

13.1 Python's regular expression language

bold font for regular expression; underlining for matching; **shading** for capture;

13.1.1 Character and character classes

The simplest expressions are just literal characters, such as **a** or **5**, and if no quantifier is explicitly given it is taken to be “match one occurrence”.

For example, the regex **mike** consists of four expressions, each implicitly quantified to match once, so it matches one *m* followed by one *i* followed by one *k* followed by one *e*, and hence matches the string mike and mikechyson.

Although most characters can be used as literals, some are “special characters” – these are symbols in the regex language and so must be escaped by preceding them with a backslash(\) to use as literals. The special characters are `\. ^ $? + * { } [] () | .`. Most of Python’s standard string escapes can also be used within regexes, for example `\n` for newline and `\t` for tab, as well as hexadecimal escapes for characters using the `\xHH`, `\uHHHH` and `\UHHHHHHHH` syntaxes.

In many cases, rather than matching one particular character we want to match any one of a set of characters. This can be achieved by using a **character class** – one or more characters enclosed in square brackets. (This has nothing to do with a Python class, and is simply the regex term for “set of characters”.) A character class is an expression, and like any other expression, if not explicitly quantified it matches exactly one character (which can be any of the characters in the character class).

For example, the regex **r[ea]d** matches both red and rad, but not read.

For convenience we can specify a range of character using a hyphen, so the regex **[0-9]** matches a digit. It is possible to negate the meaning of a character class by following the opening bracket with a caret, so **[^0-9]** matches any character that is not a digit.

Note that inside a character class, apart from `\`, the special characters lose their special meaning, although in the case of `^` it acquires a new meaning (negation) if it is the first character in the character class, and otherwise is simply a literal caret. Also, `-` signifies a character range unless it is the first character, in which case it is a literal hyphen.

Since some sets of characters are required so frequently, several have shorthand forms, as shown in table 13.1. With one exception the shorthands can be used inside character sets, so for example, the regex **[\dA-Fa-f]** matches any hexadecimal digit. The exception is `.` which is a shorthand outside a character class but matches a literal `.` inside a character class.

Symbol	Meaning
.	Matches any character except newline; or any character at all with the <code>re.DOTALL</code> flag; or inside a character class matches a literal .
\d	Matches a Unicode digit; or <code>[0-9]</code> with the <code>re.ASCII</code> flag
\D	Matches a Unicode nondigit; or <code>[^0-9]</code> with the <code>re.ASCII</code> flag
\s	Matches a Unicode whitespace; or <code>[\t\n\r\f\v]</code> with the <code>re.ASCII</code> flag
\S	Matches a Unicode nonwhitespace; or <code>[^\t\n\r\f\v]</code> with the <code>re.ASCII</code> flag
\w	Matches a Unicode “word” character; or <code>[a-zA-Z0-9_]</code> with the <code>re.ASCII</code> flag
\W	Matches a Unicode non-“word” character; or <code>[^a-zA-Z0-9_]</code> with the <code>re.ASCII</code> flag

Table 13.1: Character class shorthands

13.1.2 Quantifiers

A quantifier has the form `m,n` where `m` and `n` are the minimum and maximum times the expression the quantifier applies to must match. For example, both `e1,1e1,1` and `e2,2` match `feel`, but neither matches `felt`.

Writing a quantifier after every expression would soon become tedious. Fortunately, the regex language supports several convenient shorthands. If only one number is given in the quantifier it is taken to be both the minimum and the maximum, so `e2` is the same as `e2,2`. If no quantifier is explicitly given, it is assumed to be one.

Syntax	Meaning
<code>e?</code> or <code>e{0,1}</code>	Greedily match zero or one occurrence of expression <code>e</code>
<code>e+</code> or <code>e{1,}</code>	Greedily match one or more occurrences of expression <code>e</code>
<code>e*</code> or <code>e{0,}</code>	Greedily match zero or more occurrences of expression <code>e</code>
<code>e{m}</code>	Match exactly <code>m</code> occurrences of expression <code>e</code>
<code>e{m,}</code>	Greedily match at least <code>m</code> occurrences of expression <code>e</code>
<code>e{,n}</code>	Greedily match at most <code>n</code> occurrences of expression <code>e</code>
<code>e{m,n}</code>	Greedily match at least <code>m</code> and at most <code>n</code> occurrences of expression <code>e</code>

Table 13.2: Regular Expression Quantifiers

By default, all quantifiers are *greedy* – they match as many characters as

then can. We can make any quantifiers nongreedy (also called *minimal*) by following it with a `?` symbol. The question mark has two different meanings – on its own it is a shorthand for the `{0,1}` quantifier, and when it follows a quantifier it tells the quantifier to be nongreedy.)

13.1.3 Grouping and capturing

In practical applications we often need regexes that can match any one of two or more alternatives, and we often need to capture the match or some part of the match for further processing. Also, we sometimes want a quantifier to apply to several expressions. All of these can be achieved by grouping with `()`, and in the case of alternatives using alternation with `|`. For example, the regex `air(craft|plane)|jet` will match any text that containing “aircraft” or “airplane” or “jet”.

Parentheses serve two different purpose – to group expressions and to capture the text that matches an expression. We use the term *group* to refer to a grouped expression whether it captures or not, and *capture* and *capture group* to refer to a captured group. For example, The regex `(aircraft|airplane|jet)` would not only match any of the three expressions, but would also capture whichever one was matched for later reference.

We can switch off the capturing effect by following on opening parenthesis with `?:`. For example, `(?:aircraft|airplane|jet)`.

A grouped expression is an expression and so can be quantified. Like any other expression the quantity is assumed to be one unless explicitly given.

For example, the regex `(\w+)=(.+)` will match `topic= physical geography` with the two captures shown shaded. The regex `[\t]*(\+)[\t]*=[\t]*(.+)` will match `topic = physical geography`. The later regex keep the whitespace matching parts outside the capturing parentheses, and to allow for lines that have no whitespace at all.

Captures can be referred to using *backreferences*, that is, by referring back to an earlier capture group¹. One syntax for backreferences inside regexes themselves is `|i` where *i* is the capture number. Captures are numbered starting from one and increasing by one going from left to right as each new (capturing) left parenthesis is encountered. For example, to simplistically match duplicated words we can use the regex `(\w+)\s+\1` which matches a “word”, then at least one whitespace, and then the same word as was captured. (Capture num-

¹Backreferences cannot be used inside character classes, that is, inside `[]`.

ber 0 is created automatically without the need for parentheses; it holds the entire match.)

In long or complicated regexes it is often more convenient to use names rather than numbers for captures. This can also make maintenance easier since adding or removing capturing parentheses may change the numbers but won't affect names. To name a capture we follow the opening parenthesis with `?P<name>`. For example, `(?P<key>\w+)= (?P<value>.+)` has two captures called “key” and “value”. The syntax for backreferences to named captures inside a regex is `(?=name)`. For example, `(?P<word>\w+)\s+(?P=word)` matches duplicate words using a capture called “word”.

13.1.4 Assertions and flags

One problem that affects many of the regexes is that they can match more or different text than we intended. For example, the regex `aircraft|airplane|jet` will match “waterjet” and “jetski” as well as “jet”. This kind of problem can be solved by using assertions. An assertion does not match any text, but instead says something about the text at the point where the assertion occurs.

Some assertions are shown in Table 13.3

Symbol	Meaning
<code>^</code>	Matches at the start; also matches after each newline with the <code>re.MULTILINE</code> flag
<code>\$</code>	Matches at the end; also matches before each newline with the <code>re.MULTILINE</code> flag
<code>\A</code>	Matches at the start
<code>\b</code>	Matches at a “word” boundary
<code>\B</code>	Matches at a non-“word” boundary
<code>\Z</code>	Matches at the end
<code>(?=e)</code>	Matches if the expression <code>e</code> matches at this assertion but does not advance over it – called lookahead or positive lookahead
<code>(?!e)</code>	Matches if the expression <code>e</code> does not match at this assertion and does not advance over it – called negative lookahead
<code>(?<=e)</code>	Matches if the expression <code>e</code> matches immediately before this assertion – called positive lookbehind
<code>(?<!e)</code>	Matches if the expression <code>e</code> does not match immediately before this assertion – called negative lookbehind

Table 13.3: Regular Expression Assertions

The key value regex is as follows:

```
1 r'''^[ \t]*(?P<key>\w+)[ \t]*=[ \t]*(?P<value>[^\n]+)(?<![ \t])'''
```

It looks quite complicated. One way to make it more maintainable is to include comments in it. This can be done by adding inline comments using the syntax `(?#the comment)`, but in practice comments like this can easily make the regex even more difficult to read. A much nicer solution is to use the `re.VERBOSE` flag - this allows us to freely use whitespace and normal Python comments in regexes, with the one constraint that if we need to match whitespace we must either use `\s` or a character class such as `[\t]`. Heres the `key=value` regex with comments:

```
1 r'''
2 ^[ \t]*           # start of line and optional leading whitespace
3 (?P<key>\w+)      # the key text
4 [ \t]*=[ \t]*    # the equals with optional surrounding whitespace
5 (?P<value>[^\n]+) # the value text
6 (?<![ \t])       # negative lookahead to avoid trailing whitespace
7 '''
```

Suppose we want to extract the filenames referred to by the `src` attribute in HTML `img` tags.

```
1 r'''
2 <img\s+          # start of the tag
3 [^>]*?         # any attributes that precede the src
4 src=            # start of the src attribute
5 (?
6     (?P<quote>["'])          # opening quote
7     (?P<qimage>[^\1>]+?)    # image filename
8     (?P=quote)              # closing quote matching the opening quote
9 |
10    (?P<uimage>[^\s">]+)    # unquoted image filename
11 )
12 [^>]*?         # any attribute that follow the src
13 >
14 '''
```

Note that to refer to the matching quote inside the character class we have to use a numbered backreference `\1`, instead of `(?=quote)`, since only numbered backreference work inside character classes.

13.2 The regular expression module

The `re` module provides two ways of working with regexes:

1. Use the functions listed in Figure 13.1
2. Compile the regex and call compiled regex methods in Figure 13.2

For example (search):

Syntax	Description
<code>re.compile(r, f)</code>	Returns compiled regex <code>r</code> with its flags set to <code>f</code> if specified. (The flags are described in Table 13.5.)
<code>re.escape(s)</code>	Returns string <code>s</code> with all nonalphanumeric characters backslash-escaped—therefore, the returned string has no special regex characters
<code>re.findall(r, s, f)</code>	Returns all nonoverlapping matches of regex <code>r</code> in string <code>s</code> (influenced by the flags <code>f</code> if given). If the regex has captures, each match is returned as a tuple of captures.
<code>re.finditer(r, s, f)</code>	Returns a match object for each nonoverlapping match of regex <code>r</code> in string <code>s</code> (influenced by the flags <code>f</code> if given)
<code>re.match(r, s, f)</code>	Returns a match object if the regex <code>r</code> matches at the start of string <code>s</code> (influenced by the flags <code>f</code> if given); otherwise, returns <code>None</code>
<code>re.search(r, s, f)</code>	Returns a match object if the regex <code>r</code> matches anywhere in string <code>s</code> (influenced by the flags <code>f</code> if given); otherwise, returns <code>None</code>
<code>re.split(r, s, m, f)</code>	Returns the list of strings that results from splitting string <code>s</code> on every occurrence of regex <code>r</code> doing up to <code>m</code> splits (or as many as possible if no <code>m</code> is given, and for Python 3.1 influenced by flags <code>f</code> if given). If the regex has captures, these are included in the list between the parts they split.
<code>re.sub(r, x, s, m, f)</code>	Returns a copy of string <code>s</code> with every (or up to <code>m</code> if given, and for Python 3.1 influenced by flags <code>f</code> if given) match of regex <code>r</code> replaced with <code>x</code> —this can be a string or a function; see text
<code>re.subn(r, x, s, m, f)</code>	The same as <code>re.sub()</code> except that it returns a 2-tuple of the resultant string and the number of substitutions that were made

Figure 13.1: The Regular Expression Modules Functions

```

1 import re
2
3 # manner 1
4 text = '#COC0AB'
5 match = re.search(r'#[\dA-Fa-f]{6}\b', text)
6
7 # manner 2
8 color_re = re.compile(r'#[\dA-Fa-f]{6}\b')
9 match = color_re.search(text)
10
11 # flag
12 match = re.search(r'#[\dA-F]{6}\b', text, re.IGNORECASE)
13 match = re.search(r'(?i)#[\dA-F]{6}\b', text)

```

The methods provided by match objects are listed in Figure 13.4

Syntax	Description
<code>rx.findall(s, start, end)</code>	Returns all nonoverlapping matches of the regex in string <i>s</i> (or in the <i>start:end</i> slice of <i>s</i>). If the regex has captures, each match is returned as a tuple of captures.
<code>rx.finditer(s, start, end)</code>	Returns a match object for each nonoverlapping match in string <i>s</i> (or in the <i>start:end</i> slice of <i>s</i>)
<code>rx.flags</code>	The flags that were set when the regex was compiled
<code>rx.groupindex</code>	A dictionary whose keys are capture group names and whose values are group numbers; empty if no names are used
<code>rx.match(s, start, end)</code>	Returns a match object if the regex matches at the start of string <i>s</i> (or at the start of the <i>start:end</i> slice of <i>s</i>); otherwise, returns <code>None</code>
<code>rx.pattern</code>	The string from which the regex was compiled
<code>rx.search(s, start, end)</code>	Returns a match object if the regex matches anywhere in string <i>s</i> (or in the <i>start:end</i> slice of <i>s</i>); otherwise, returns <code>None</code>
<code>rx.split(s, m)</code>	Returns the list of strings that results from splitting string <i>s</i> on every occurrence of the regex doing up to <i>m</i> splits (or as many as possible if no <i>m</i> is given). If the regex has captures, these are included in the list between the parts they split.
<code>rx.sub(x, s, m)</code>	Returns a copy of string <i>s</i> with every (or up to <i>m</i> if given) match replaced with <i>x</i> —this can be a string or a function; see text
<code>rx.subn(x, s, m)</code>	The same as <code>re.sub()</code> except that it returns a 2-tuple of the resultant string and the number of substitutions that were made

Figure 13.2: Regular Expression Object Methods

Example (check duplicate words):

```

1 text = "one and and two let's say"
2 double_word_re = re.compile(r"\b(?:P<word>\w+)\s+(?P=word)(?!\\w)", re.IGNORECASE)
3 double_word_re = re.compile(r"\b(?:P<word>\w+)\s+(?P=word)\b", re.IGNORECASE) # same to the above
4 for match in double_word_re.finditer(text):
5     print(f'{match.group("word")} is duplicated')
6 # and is duplicated

```

Example (extract image filenames):

```

1 text = '''
2 
3 
4 '''
5 image_re_text = r'''
6 <img\s+                               # start of the tag

```

Flag	Meaning
re.A or re.ASCII	Makes <code>\b</code> , <code>\B</code> , <code>\s</code> , <code>\S</code> , <code>\w</code> , and <code>\W</code> assume that strings are ASCII; the default is for these character class short-hands to depend on the Unicode specification
re.I or re.IGNORECASE	Makes the regex match case-insensitively
re.M or re.MULTILINE	Makes <code>^</code> match at the start and after each newline and <code>\$</code> match before each newline and at the end
re.S or re.DOTALL	Makes <code>.</code> match every character including newlines
re.X or re.VERBOSE	Allows whitespace and comments to be included

Figure 13.3: The Regular Expression Module's Flags

```

7  [^>]*?           # any attributes that precede the src
8  src=             # start of the src attribute
9  (?
10     (?P<quote>[''])           # opening quote
11     (?P<qimage>[^\1]+)?      # image filename
12     (?P=quote)              # closing quote matching the opening quote
13 |
14     (?P<uimage>[^\s'>]+)     # unquoted image filename
15 )
16 [^>]*?           # any attribute that follow the src
17 >
18 '''
19 image_re = re.compile(image_re_text, re.IGNORECASE | re.VERBOSE)
20 image_files = []
21 for match in image_re.finditer(text):
22     image_files.append(match.group('qimage') or match.group('uimage'))
23 for image_file in image_files:
24     print(image_file)
25 # /images/stickman.gif
26 # https://www.w3schools.com/images/lamp.jpg

```

Example (convert html to text):

```

1  def html2text(html_text):
2      def char_from_entity(match):
3          code = html.entities.name2codepoint.get(match.group(1), 0xFFFD)
4          return chr(code)
5
6      # (?s) math . include newline
7      text = re.sub(r'(?s)<!--.*?-->', '', html_text) # HTML comments
8      text = re.sub(r'<[Pp][^>]*?>', '\n\n', text) # opening paragraph tags
9      text = re.sub(r'<[^>]*?>', '', text) # any tag
10     text = re.sub(r'&#(\d+);', lambda m: chr(int(m.group(1))), text) # &#165; for ë
11     text = re.sub(r'&([A-Za-z]+);', char_from_entity, text) # named entities
12     text = re.sub(r'\n(?:[ \xA0\t]+\n)+', '\n', text) # lines that contain only whitespace
13     # Replace sequences of two or more newlines with exactly two newlines
14     text = re.sub(r'\n\n+', '\n\n', text.strip())
15     return text

```

Example (switch name order in fullname):

```

1  import re
2
3  # from Forename Middlename1 ... MiddlenameN Surname
4  # to Surname,ForenameMiddlename1...MiddlenameN
5  names = ['Mike Ming Chyson', 'Maël Ming Li']

```

Syntax	Description
<code>m.end(g)</code>	Returns the end position of the match in the text for group <i>g</i> if given (or for group 0, the whole match); returns -1 if the group did not participate in the match
<code>m.endpos</code>	The search's end position (the end of the text or the <i>end</i> given to <code>match()</code> or <code>search()</code>)
<code>m.expand(s)</code>	Returns string <i>s</i> with capture markers (<code>\1</code> , <code>\2</code> , <code>\g<name></code> , and similar) replaced by the corresponding captures
<code>m.group(g, ...)</code>	Returns the numbered or named capture group <i>g</i> ; if more than one is given a tuple of corresponding capture groups is returned (the whole match is group 0)
<code>m.groupdict(default)</code>	Returns a dictionary of all the named capture groups with the names as keys and the captures as values; if a <i>default</i> is given this is the value used for capture groups that did not participate in the match
<code>m.groups(default)</code>	Returns a tuple of all the capture groups starting from 1; if a <i>default</i> is given this is the value used for capture groups that did not participate in the match
<code>m.lastgroup</code>	The name of the highest numbered capturing group that matched or <code>None</code> if there isn't one or if no names are used
<code>m.lastindex</code>	The number of the highest capturing group that matched or <code>None</code> if there isn't one
<code>m.pos</code>	The start position to look from (the start of the text or the <i>start</i> given to <code>match()</code> or <code>search()</code>)
<code>m.re</code>	The regex object which produced this match object
<code>m.span(g)</code>	Returns the start and end positions of the match in the text for group <i>g</i> if given (or for group 0, the whole match); returns (-1, -1) if the group did not participate in the match
<code>m.start(g)</code>	Returns the start position of the match in the text for group <i>g</i> if given (or for group 0, the whole match); returns -1 if the group did not participate in the match
<code>m.string</code>	The string that was passed to <code>match()</code> or <code>search()</code>

Figure 13.4: Match Object Attribute and Methods

```

6 new_names = []
7 for name in names:
8     # name = re.sub(r"(\w+(?:\s+\w+)*)\s+(\w+)", r"\2, \1", name)
9     name = re.sub(r"(?P<forenames>\w+(?:\s+\w+)*)"
10                r"\s+(?P<surname>\w+)",
11                r"\g<surname>, \g<forenames>",
12                name)
13     new_names.append(name)
14 for name in new_names:
15     print(name)

```

```
16 # Chyson, Mike Ming
17 # Li, Maël Ming
```

Example (detect encoding):

```
1 import re
2
3 binary = b''
4
5 re_binary = r'''
6 # A lookbehind assertion that says that the
7 # match cannot be preceded by a hyphen or a word character.
8 (?<![\w])
9 (?:(?:en)?coding|charset) # encoding|coding|charset
10 (?:([' '])?([\w]+)?(1)\1)
11 |: \s*([\w]+)
12 '''
13 match = re.search(re_binary, binary, re.IGNORECASE | re.VERBOSE)
14 encoding = match.group(match.lastindex) if match else b'utf8'
```

Example (split text based on whitespace):

```
1 text = 'hello world'
2 re.split(r'\s+', text)
3 # same to
4 text.split()
```


Chapter 14

Introduction to GUI programming

Python has no native support for GUI (Graphical User Interface) programming, but this isn't a problem since many GUI libraries written in other languages can be used by Python programmers. This is possible because many GUI libraries have Python **wrappers** or **bindings** – these are packages and modules that are imported and used like any other Python packages and modules but which access functionality that is in non-Python libraries under the hood.

Python's standard library includes Tcl/Tk – Tcl is an almost syntax-free scripting language and Tk is a GUI library written in Tcl and C. Python's `tkinter` module provides Python binding for the Tk GUI library. Tk has three advantages compared with other GUI libraries:

- It is installed as standard with Python, so it is always available.
- It is small.
- It comes with IDLE.

For developing GUI programs that must run on any or all Python desktop platform, using only a standard Python installation with no additional libraries, there is just one choice: Tk.

14.1 Dialog-style programs

In most object-oriented programs, a custom class is used to represent a single main window or dialog, with most of the widgets it contains being instances of standard widgets, such as buttons or checkboxes, supplied by the library. Like most cross-platform GUI libraries, Tk doesn't really make distinction between a window and a widget – a window is simply a widget that has no widget parent. Widgets that don't have a widget parent (windows) are automatically supplied with a frame and window decorations (such as a title bar and close button). In addition to distinguishing between widgets and windows (also called top-level widgets), the parentchild relationships help ensure that widgets are deleted in the right order and that child widgets are automatically deleted when their parent is deleted.

The interface is shown in Figure 14.1.

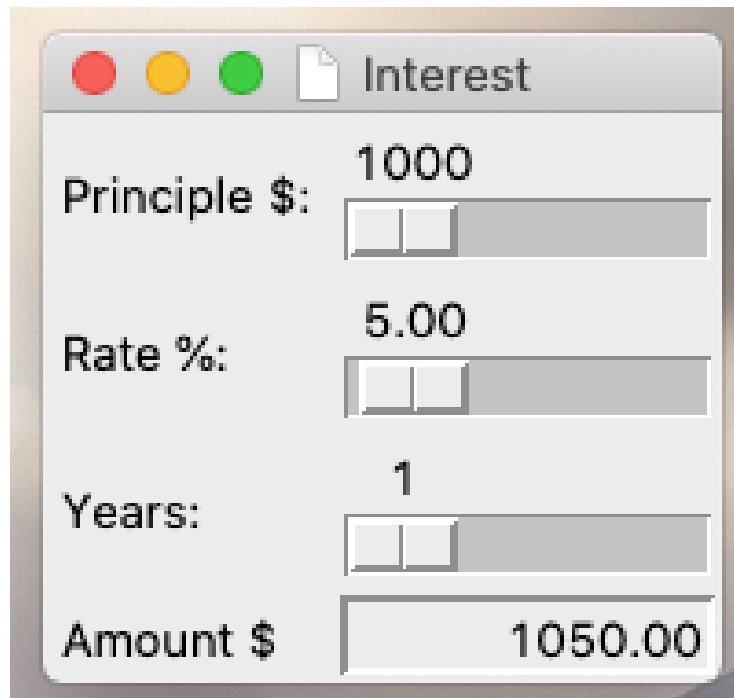


Figure 14.1: The interest program

The corresponding code is shown below:

```
1 #!/usr/bin/env python3
2 """
```



```

3  @project: python3
4  @file: interest
5  @author: mike
6  @time: 2021/2/23
7
8  @function:
9  """
10 import tkinter
11 import os
12 import sys
13
14
15 class MainWindow(tkinter.Frame):
16     def __init__(self, parent):
17         super().__init__(parent)
18         self.parent = parent
19         # Lays out the frame using the grid layout manager
20         self.grid(row=0, column=0)
21
22         self.principal = tkinter.DoubleVar()
23         self.principal.set(1000.0)
24         self.rate = tkinter.DoubleVar()
25         self.rate.set(5.0)
26         self.years = tkinter.IntVar()
27         self.amount = tkinter.StringVar()
28
29         principal_label = tkinter.Label(self, text='Principle $:',
30                                         anchor=tkinter.W,
31                                         underline=0)
32         principal_scale = tkinter.Scale(self, variable=self.principal,
33                                         command=self.updateUi,
34                                         from_=100,
35                                         to=10000000,
36                                         resolution=100,
37                                         orient=tkinter.HORIZONTAL)
38         rate_label = tkinter.Label(self, text='Rate %:',
39                                    underline=0,
40                                    anchor=tkinter.W)
41         rate_scale = tkinter.Scale(self, variable=self.rate,
42                                    command=self.updateUi,
43                                    from_=1,
44                                    to=100,
45                                    resolution=0.25,
46                                    digits=5,
47                                    orient=tkinter.HORIZONTAL)
48         year_label = tkinter.Label(self, text='Years:',
49                                    underline=0,
50                                    anchor=tkinter.W)
51         year_scale = tkinter.Scale(self, variable=self.years,
52                                    command=self.updateUi,
53                                    from_=1,
54                                    to=50,
55                                    orient=tkinter.HORIZONTAL)
56         amount_label = tkinter.Label(self, text='Amount $', anchor=tkinter.W)
57         actual_amount_label = tkinter.Label(self, textvariable=self.amount,
58                                             relief=tkinter.SUNKEN,
59                                             anchor=tkinter.E)
60
61         principal_label.grid(row=0, column=0, padx=2, pady=2, sticky=tkinter.W)
62         principal_scale.grid(row=0, column=1, padx=2, pady=2, sticky=tkinter.EW)
63         rate_label.grid(row=1, column=0, padx=2, pady=2, sticky=tkinter.W)
64         rate_scale.grid(row=1, column=1, padx=2, pady=2, sticky=tkinter.EW)
65         year_label.grid(row=2, column=0, padx=2, pady=2, sticky=tkinter.W)
66         year_scale.grid(row=2, column=1, padx=2, pady=2, sticky=tkinter.EW)
67         amount_label.grid(row=3, column=0, padx=2, pady=2, sticky=tkinter.W)
68         actual_amount_label.grid(row=3, column=1, padx=2, pady=2, sticky=tkinter.EW)
69
70         principal_scale.focus_set()
71         self.updateUi()
72         parent.bind('<Alt-p>', lambda *ignore: principal_scale.focus_set())
73         parent.bind('<Alt-r>', lambda *ignore: rate_scale.focus_set())
74         parent.bind('<Alt-y>', lambda *ignore: year_scale.focus_set())
75         parent.bind('<Control-q>', self.quit)

```

```

76     parent.bind('<Escape>', self.quit)
77
78     def updateUi(self, *ignore):
79         amount = self.principal.get() * (
80             (1 + (self.rate.get() / 100.0)) ** self.years.get()
81         )
82         self.amount.set(f'{amount:.2f}')
83
84     def quit(self, event=None):
85         self.parent.destroy()
86
87
88 application = tkinter.Tk()
89 path = os.path.join(os.path.dirname(__file__), 'images/')
90 if sys.platform.startswith('win'):
91     icon = path + 'interest.ico'
92 else:
93     icon = '@' + path + 'interest.xbm'
94 application.iconbitmap(icon)
95 application.title('Interest')
96 window = MainWindow(application)
97 application.protocol('WM_DELETE_WINDOW', window.quit)
98 application.mainloop()

```

Rather than using absolute positions and sizes, widgets are laid out inside other widgets using layout managers. The call to `grid()` lays out the frame using the grid layout manager. Every widget that is shown must be laid out, even top-level ones.

(Line 22-27) Tk allows us to create variables that are associated with widgets. If a variable's value is changed programmatically, the change is reflected in its associated widget, and similarly, if the user changes the value in the widget the associated variable's value is changed.

(Line 29-59) This part of the initializer is where we create the widgets. The `tkinter.Label` widget is used to display read-only text to the user. Like all widgets it is created with a parent, and then keyword arguments are used to set various other aspects of the widget's behavior and appearance. We have set the `principalLabels` text appropriately, and set its anchor to `tkinter.W`, which means that the label's text is aligned west (left). The `underline` parameter is used to specify which character in the label should be underlined to indicate a keyboard accelerator (e.g., Alt+P). (A keyboard accelerator is a key sequence of the form **Alt+letter** where **letter** is an underlined letter and which results in the keyboard focus being switched to the widget associated with the accelerator, most commonly the widget to the right or below the label that has the accelerator.)

(Line 72-77) We set up a few key bindings.

To give the program an icon on Windows we use an `.ico` file and pass the name of the file (with its full path) to the `iconbitmap()` method. But for Unix platforms we must provide a bitmap. Tk has several built-in bitmaps, so to

distinguish one that comes from the file system we must precede its name with an @ symbol.

14.2 Main-window-style programs

https://github.com/mikechyson/python3/blob/master/c15_gui/bookmarks_tk.pyw

Chapter 15

Environment

15.1 Repo

To set up a repo: Create a file `~/.config/pip/pip.conf` and write the following content into it:

```
1 [global]
2 index-url = http://mirrors.aliyun.com/pypi/simple/
3 [install]
4 trusted-host = mirrors.aliyun.com
```

You can alter the repo to your preferred one.

15.2 Freeze and install packages

To generate the requirement file:

```
1 pip freeze > requirement.txt
```

To install the packages from the requirement file:

```
1 pip install -r requirement.txt
```

