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

Chapter 1

Introduction

1.1 Creating and running Pyton programs

Be 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`.

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

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 chracters `#!`, the shell assume 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:

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

or

```
#!/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 ont 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.

```
210624583337114373395836055367340864637790190801098222508621955072
0
"hello"
'world'
```

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

```
'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)`.

```
int("45")
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 `int`s and `str`s, 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.

```

x = "blue"
y = "green"
z = x
```

The syntax is simply `object_reference = value`. The `=` operator is not the same as the variable assignment operater in some other languages. The `=` operater 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 isnt called strong typing in Pythons case because the valid operations can change — for example, if an object reference is re-bound to an object of a different data type.

```

route = 123
print(route, type(route))

route = "North"
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 imuutable while lists are mutable.

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

```

>>> "hello", "world", "mike", "chyson"
("hello", "world", "mike", "chyson")
>>> "one",
("one",)
```

When Python ouptuts 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.

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

One way to create a list is to use square brackets ([]). An empty list is create 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 precedural programming we can function and often pass in data items as arguments.

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

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

```
>>> x = ["zebra", 49, -879, "aardvark", 200]
>>> x.append("more")
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more']

>>> list.append(x, "extra")
>>> x
['zebra', 49, -879, 'aardvark', 200, 'more', 'extra']
```

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

```
>>> a = ["Retention", 3, None]
>>> b = ["Retention", 3, None]
>>> a is b
False
>>> b = a
>>> a is b
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 Pythons comparison operators is that they can be chained. For example:

```

>>> a = 9
>>> 0 <= a <= 10
True
```

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:

```

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

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 (unleass they actually have Boolean operands).

```

>>> five = 5
>>> two = 2
>>> zero = 0
>>> five and two
2
>>> two and five
5
>>> five and zero
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`.

```

>>> nought = 0
>>> five or two
5
>>> two or five
2
>>> zero or five
5
>>> zero or nought
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 precessing 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:

```
>>> 12/3
4.0
```

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

```
>>> a = 5
>>> a
5
>>> a += 8
>>> a
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 `a`'s value only once, so it is potentially faster.

```
>>> name = 'mike'
>>> name + 'chyson'
'mikechyson'
>>> name += ' chyson'
>>> name
'mike chyson'
>>> a = [1, 2, 3]

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

Python overloads the `+` and `+=` operators for both strings and lists, the former mearning 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 is. For example:

```

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:

```

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 imuutable. 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.

```

>>> 14600926                        # decimal
14600926
>>> 0b110111101100101011011110      # binary
14600926
>>> 0o67545336                      # octal
14600926
>>> 0xDECADE                        # hexadecimal
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 Divdes x by y; always produces a `float` (or a `complex` if x or y is `complex`)

x // y Divdes 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 remaninder 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 xs 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 sinle 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 `float`s. Because of their accuracy, `decimal.Decimal` numbers are suitable for financial calculations.

2.3.1 Floating-point numbers

```

import sys

# the samllest difference that the machine can distinguish
# between two floating-point numbers
sys.float_info.epsilon

help(sys.float_info)
```

The `math` module provides many functions that operate on `float`s. 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 $\pm$inf($\pm\infty$)

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 `float`s, one representing the real part and the other the imaginary part of a complex number.

```
>>> z = 1.0 + 2.0j
>>> z.real, z.imag
(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

```
>>> import decimal
>>> a = decimal.Decimal(1234)
>>> b = decimal.Decimal('54321.012345678987654321')
>>> a + b
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.

```
>>> 23 / 1.05
21.904761904761905
>>> print(23 / 1.05)
21.904761904761905
>>> print(decimal.Decimal(23) / decimal.Decimal(1.05))
21.90476190476190383546015179
>>> decimal.Decimal(23) / decimal.Decimal(1.05)
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:

```
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
newline	Escape (i.e., ignore) the newline
\\\\	Backslash (\\)
\\'	Single quote (')
\\"	Double quote (")
\\a	ASCII bell (BEL)
\\b	ASCII backspace (BS)
\\f	ASCII formfeed (FF)
\\n	ASCII linefeed (LF)
\\N{*name*}	Unicode character with the given name
ooo	Character with the given octal value
\\r	ASCII carriage return (CR)
\\t	ASCII tab (TAB)
\\u*hhhh*	Unicode character with the given 16-bit hexadecimal value
\\U*hhhhhhhh*	Unicode character with the given 32-bit hexadecimal value
\\v	ASCII vertical tab (VT)
\\x*hh*	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:

```
import re
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.

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

```
>>>'\N{euro sign}'
''
```

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

```
>>> ord('')
8364
>>> hex(ord(''))
'0x20ac'
>>> '\u20ac'
''
```

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

```
>>> chr(8734)
''
>>> chr(8364)
''
>>> ascii('')
"'\\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

```
s = "Light ray"
```

Figure 2.2 shows all the valid index postions 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.

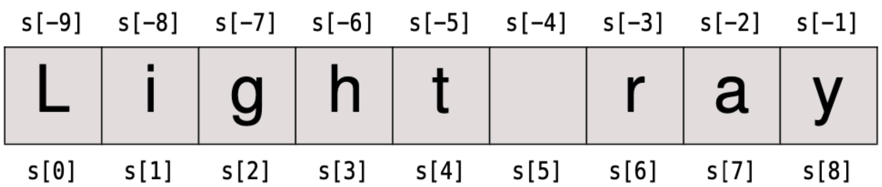

Figure 2.2: String index position

2.4.3 String operators and methods

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

```

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

```

>>> 'just {{{0}}}'.format('brace')
'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.

```

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

```

>>> stock = ["paper", "envelopes", "notepads"]
>>> "We have {0[1]} and {0[2]} in stock".format(stock)
'We have envelopes and notepads in stock'

>>> d = dict(animal="elephant", weight=12000)
>>> "The {0[animal]} weighs {0[weight]}kg".format(d)
'The elephant weighs 12000kg'

>>> "math.pi=={0.pi}".format(math)
'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:

```

>>> element = "Silver"
>>> number = 47
>>> "Element {number} is {element}".format(**locals())
’Element 47 is Silver’
```

2.4.6 conversions

```

>>> decimal.Decimal("3.4084")
Decimal(’3.4084’)
>>> print(decimal.Decimal("3.4084"))
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 ojbect 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 doesnt have a string form and a string is required, Python will use the representational form.

Pythons 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 for
- `a` to for representational form but only using ASCII characters.

```

>>> "{0} {0!s} {0!r} {0!a} {1!r} {1!a}".format(decimal.Decimal(1), "")
"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
```

```

>>> s = "The sword of truth"
>>> '{0}'.format(s)
'The sword of truth'
>>> '{0:25}'.format(s)
'The sword of truth       '
>>> '{0:>25}'.format(s)
'       The sword of truth'
>>> '{0:->25}'.format(s)
'-------The sword of truth'
>>> '{0:.<25}'.format(s)     # the left alignment can not be omitted
'The sword of truth.......'
>>> '{0:.10}'.format(s)
'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
```

```

>>> '{0:0=12}'.format(-1234)
'-00000001234'

>>> "[{0: }] [{1: }]".format(539802, -539802) # space or - sign
'[ 539802] [-539802]'
>>> "[{0:+}] [{1:+}]".format(539802, -539802) # force sign
'[+539802] [-539802]'
>>> "[{0:-}] [{1:-}]".format(539802, -539802) # - sign if needed
'[539802] [-539802]'

>>> "{0:b} {0:o} {0:x} {0:X}".format(123)
'1111011 173 7b 7B'
>>> "{0:#b} {0:#o} {0:#x} {0:#X}".format(123)
'0b1111011 0o173 0x7b 0X7B'

>>> '{0:,}'.format(1234567890)
'1,234,567,890'
```

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.

```

>>> import locale
>>> x = 1234567890
>>> locale.setlocale(locale.LC_ALL, 'C')
'C'
>>> '{:n}'.format(x)
'1234567890'
>>> locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')
'en_US.UTF-8'
>>> '{:n}'.format(x)
'1,234,567,890'
>>> locale.setlocale(locale.LC_ALL, 'de_DE.UTF-8')
'de_DE.UTF-8'
>>> '{:n}'.format(x)
'1234567890'
```

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” formthis is the same as f unless the number is very large, in which case it is the same as e
- G is is almost the same as g, but uses either f or E
- % for percentage

```

>>> '{0:12.2e}'.format(math.pi)
'    3.14e+00'
>>> '{0:12.2f}'.format(math.pi)
'        3.14'
>>> '{:,.6f}'.format(1234567890.1234567890)
'1,234,567,890.123457'
>>> "{:,.4f}".format(3.59284e6-8.984327843e6j)
'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 doesnt 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 attributeif 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 size of a binary operator
- on the right-hand size of a unary statement

other cases with parentheses.

```

a, b = (1, 2)  # left of binary operator
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:

```

a, b = (b, a) # or a, b = b, a
# the parentheses here are for code style

for x, y in ((3, 4), (5, 12), (28, -45)):
    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.

```

import collections

Fullname = collections.namedtuple('Fullname',
                                  'firstname middlename lastname')
persons = []
persons.append(Fullname('Mike', 'Ming', 'Chyson'))
persons.append(Fullname('Alfred', 'Bernhard', 'Nobel'))
for person in persons:
    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:

```

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

```

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

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

```

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

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

```
leaps = [y for y in range(1900, 1940)
         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 thoughout the ojbect's lifetime, and which can be compared for equality using the `__eq__()` special method. (Special mehtods 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()

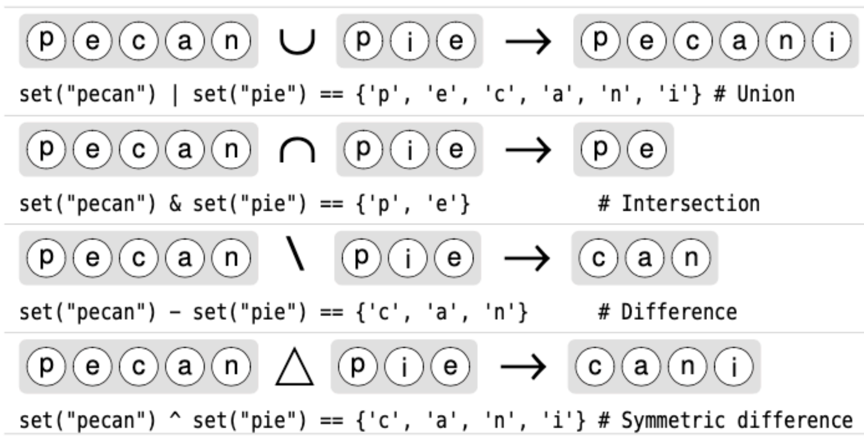

Figure 3.1: Standard set operators

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 used for fast membership test and removring 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 set 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.

```
>>> d = dict()
>>> d
{}
>>> d = {}
>>> d
{}
>>> d = {"hello": 1, "world": 2}
>>> d
{'hello': 1, 'world': 2}
>>> d = dict(hello=1, world=2)
>>> d
{'hello': 1, 'world': 2}
>>> d = dict([("hello", 1), ("world", 2)])
>>> d
{'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 indict d,a new item is inserted with the key k, and with a value of None or of v if v is given

d.update(a) Adds every (key, value) pair from a that isnt 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 dictionarys 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
```

```
>>> d = {}.fromkeys("abcd", 3)
>>> d
{'a': 3, 'b': 3, 'c': 3, 'd': 3}
>>> s = set("abc")
>>> s
{'a', 'c', 'b'}
>>> d.keys() & s
{'a', 'c', 'b'}
>>>
>>> d
{'a': 3, 'b': 3, 'c': 3, 'd': 3}
>>> d.setdefault('a')
3
>>> d
{'a': 3, 'b': 3, 'c': 3, 'd': 3}
>>> d.setdefault('z', 100)
100
>>> d
{'a': 3, 'b': 3, 'c': 3, 'd': 3, 'z': 100}
```

3.3.2 Dictionary comprehensions

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

```
import os

# filename: filesize
d = {name: os.path.getsize(name) for name in os.listdir('.') if os.path.isfile(name)}
print(d)
```

```

# revert dict
inserted_d = {v: k for k, v in d.items()}
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 Pythons 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.

```
>>> words = collections.defaultdict(int)
>>> words
defaultdict(<class 'int'>, {})
>>> words['hello'] += 1
>>> words
defaultdict(<class 'int'>, {'hello': 1})
>>> words['hello'] += 1
>>> words
defaultdict(<class 'int'>, {'hello': 2})
>>>
>>> de = collections.defaultdict(lambda : "Thanks to ")
>>> de
defaultdict(<function <lambda> at 0x7fa999a75a60>, {})
>>> de['Mike'] += 'Mike'
>>> de
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.

```
d = collections.OrderedDict([('z', -4), ('e', 19), ('k', 7)])
for k in d:
    print(k, d[k])

tasks = collections.OrderedDict()
```

```
tasks[8031] = "Backup"
tasks[4027] = "Scan Email"
tasks[5733] = "Build System"
for k in tasks:
    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 evalueates to True

any(i) Returns True if any item in iterable i evalueates 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 oject 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 teh function'sreturn 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 caughted and the loop is terminated.

```
# manner 1
product = 1
for i in [1, 2, 4, 8]:
    product *= i
print(product)


# manner 2
product = 1
i = iter([1, 2, 4, 8])
while True:
    try:
        product *= i
    except StopIteration:
        break
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)`.

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

```
print('{:.^50}'.format('print(x,y)'))
x = [53, 68, ['A', 'B', 'C']]
y = x[:]
print(x, y, sep='\n')

print('{:.^50}'.format('print(x,y)'))
y[1] = 40
x[2][0] = 'Q'
print(x, y, sep='\n')

"""
....................print(x,y)....................
[53, 68, ['A', 'B', 'C']]
[53, 68, ['A', 'B', 'C']]
....................print(x,y)....................
[53, 68, ['Q', 'B', 'C']]
[53, 40, ['Q', 'B', 'C']]
"""
```

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

```
import copy

x = [53, 68, ['A', 'B', 'C']]
y = copy.deepcopy(x)
y[1] = 40
x[2][0] = 'Q'
print('{:.^50}'.format('print(x,y)'))
print(x, y, sep='\n')

"""
....................print(x,y)....................
[53, 68, ['Q', 'B', 'C']]
[53, 40, ['A', 'B', 'C']]
"""
```

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.

```
width = 100 + (10 if margin else 10)
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.

```
i = 0
while i < 100:
    i += 1
else:
    last = i
print(last)  # 100
```

```
def list_find(lst, target):
    """
    Find the first target's index or -1 if not find.

    :param lst:
    :param target:
    :return: index of the target if found or -1 if not found
    """
    index = 0
    while index < len(lst):
        if lst[index] == target:
            break
        index += 1
    else:
        index = -1
    return index
```

for loops

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

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

```
def list_find2(lst, target):
    for index, x in enumerate(lst):
        if x == target:
            break
    else:
        index = -1
    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.

```
def lst_find(lst, target):
    try:
        index = lst.index(target)
    except ValueError:
        index = -1
    return index
```

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

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

```
# remove black lines
def read_data(filename):
    lines = []
    fh = None
    try:
        fh = open(filename)
        for line in fh:
            if line.strip():
                lines.append(line)
    except (IOError, OSError) as err:
        print(err)
        return []
    finally:
        if fh is not None:
            fh.close()
    return lines
```

Rasing 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.

```
def find_word(table, target):
    found = False
    for row, record in enumerate(table):
        for column, field in enumerate(record):
            for index, item in enumerate(field):
                if item == target:
                    found = True
                    break
            if found:
                break
        if found:
            break

    if found:
        print('found at ({}, {}, {})'.format(row, column, index))
    else:
        print('not found')


def find_word2(table, target):
    class FoundException(Exception):
        pass

    try:
        for row, record in enumerate(table):
            for column, field in enumerate(record):
                for index, item in enumerate(field):
                    if item == target:
                        raise FoundException
    except FoundException:
        print('found at ({}, {}, {})'.format(row, column, index))
    else:
        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
+-- ImportWarning
+-- 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
```

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

```


print(my_sum(1, 2, 3)) # 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 doesnt make any difference, but for mutable arguments a subtle trap is lurking.

```
def append_if_even(x, lst=[]):
    if x % 2 == 0:
        lst.append(x)
    return lst


def append_if_even2(x, lst=None):
    lst = [] if lst is None else lst
    if x % 2 == 0:
        lst.append(x)
    return lst


for i in range(3):
    result1 = append_if_even(i)
    result2 = append_if_even2(i)
    print(f'{result1=},{i=}')
    print(f'{result2=},{i=}')

# result1=[0],i=0
# result2=[0],i=0
# result1=[0],i=1
# result2=[],i=1
# result1=[0, 2],i=2
# 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 proprotional 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.

```
def shorten(text, length=25, indicator="..."):
    """Returns text or a truncated copy with the indicator added

    text is any string; length is the maximum length of the returned
    string (including any indicator); indicator is the string added at
    the end to indicate that the text has been shortened

    >>> shorten("Second Variety")
    'Second Variety'
    >>> shorten("Voices from the Street", 17)
    'Voices from th...'
    >>> shorten("Radio Free Albemuth", 10, "*")
    'Radio Fre*'
    """
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    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 arguemnts.

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

print(my_sum(*[1, 2, 3, 4][:3]))    # 6
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.

```
def product(*args):
    result = 1
    for arg in args:
        result *= arg
    return result
```

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

```
def product(*args):
    result = 1
    for arg in args:
        result *= arg
    return result


print(product(*list(range(1, 10))))  # 362880
print(math.factorial(9))             # 362880
```

```
# It is also possible to use * as a parameter in its own right.
# This is used to signify that there can be no positional arguments after the *.
def heron(a, b, c, *, units='square meters'):
    s = (a + b + c) / 2
    area = math.sqrt(s * (s - a) * (s - b) * (s - c))
    return f'{area} {units}'


print(heron(25, 24, 7))
print(heron(41, 9, 40, units="sq. inches"))
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.

```
def print_dict(**kwargs):
    for key in sorted(kwargs):
        print(f'{key:10} : {kwargs[key]}')


print_dict(**{str(i): f'{100 * i:3}%' for i in range(10)})

# 0          :   0%
# 1          : 100%
# 2          : 200%
# 3          : 300%
# 4          : 400%
# 5          : 500%
# 6          : 600%
# 7          : 700%
# 8          : 800%
# 9          : 900%
```

```
def print_args(*args, **kwargs):
    for i, arg in enumerate(args):
        print("positional argument {0} = {1}".format(i, arg))
    for key in kwargs:
        print("keyword argument {0} = {1}".format(key, kwargs[key]))


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.

```
AUTHOR = 'Mike'  # global


def say_hello():  # global
    global language  # global
    language = 'fr'
    text = 'hello'  # local
    print(text)


class MyException(Exception):  # global
    pass


say_hello()
print(language)
```

4.3.4 Lambda functions

Lambda functions are functions created using the following syntax:

```
lambda parameters: expression
```

The **parameters** are optinal, 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`) statment. 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.

```
lst = list(range(-3, 3))
print(lst)
print(sorted(lst, key=lambda x: x ** 2))
print(sorted(lst, key=lambda key=None: key ** 2))  # seldom used

# [-3, -2, -1, 0, 1, 2]
# [0, -1, 1, -2, 2, -3]
# [0, -1, 1, -2, 2, -3]
```

```
s = lambda x: '' if x == 1 else 's'  # use def instead
print(s(1))  #
print(s(2))  # s

p = lambda key='hello': print(key)  # use def instead
p('world')  # world
```

There are two common usage for lambda functions:

- key function
- default value

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

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.

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

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

- -0 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 docstrings. 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 moduels 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 importint:

```
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 syntex, 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 consistutes 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 Pythons standard library — these are set when Python is installed.
> When we first import a module, if it isnt 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. WhenPythonis 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 **package**s modules using a single statement. To do this we must edit the **package**s `__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 object defined in the module (appart 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.

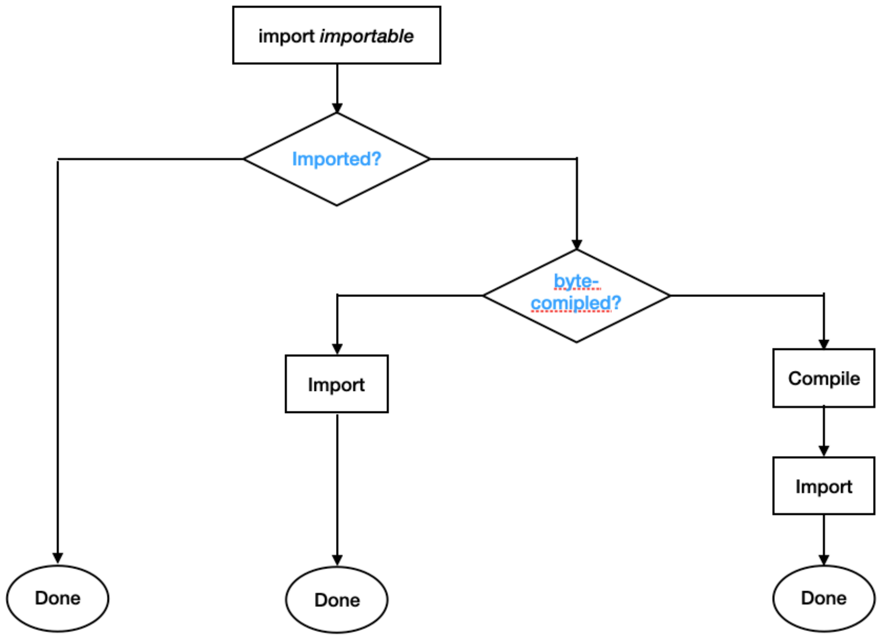

Figure 5.1: Import

5.1.2 Custom modules

The TextUtil module

TextUtil.py:

```
#!/usr/bin/env python3
# Copyright
"""
This module provides a few string manipulation functions.

>>> is_balanced("(Python (is (not (lisp))))")
True
>>> shorten("The Crossing", 10)
'The Cro...'
>>> simplify(" some  text   with spurious   whitespace   ")
'some text with spurious whitespace'
"""

import string


def simplify(text, whitespace=string.whitespace, delete=''):
    r"""Returns the text with multiple spaces reduced to single spaces

    The whitespace parameter is a string of characters, each of which
    is considered to be a space.
    If delete is note empty it should be a string, in which case any
    characters in the delete string are excluded from the resultant
    string.
```

```

    >>> simplify(" this    and\n that\t too")
    'this and that too'
    >>> simplify("  Washington    D.C.\n")
    'Washington D.C.'
    >>> simplify("  Washington   D.C.\n", delete=',;:.')
    'Washington DC'
    >>> simplify(" disemvoweled ", delete="aeiou")
    'dsmvwld'
    """
    result = []
    word = ""
    for char in text:
        if char in delete:
            continue
        elif char in whitespace:
            if word:
                result.append(word)
                word = ""
        else:
            word += char
    if word:
        result.append(word)
        return " ".join(result)


def is_balanced(text, brackets="()[]{}<>"):
    """Returns True if all the brackets in the text are balanced

    For each pair of brackets, the left and right bracket characters
    must be different.

    >>> is_balanced("no brackets at all")
    True
    >>> is_balanced("<b>bold</b>")
    True
    >>> is_balanced("[<b>(some {thing}) goes</b>]")
    True
    >>> is_balanced("<b>[not (where {it}) is}]</b>")
    False
    >>> is_balanced("(not (<tag>(like) (anything)</tag>)")
    False
    """
    counts = {}
    left_for_right = {}
    for left, right in zip(brackets[::2], brackets[1::2]):
        assert left != right, "the bracket characters must differ"
        counts[left] = 0
        left_for_right[right] = left
    for c in text:
        if c in counts:
            counts[c] += 1
        elif c in left_for_right:
            left = left_for_right[c]
            if counts[left] == 0:
                return False
            counts[left] -= 1
    return not any(counts.values())


def shorten(text, length=25, indicator="..."):
    """Returns text or a truncated copy with the indicator added

    text is any string; length is the maximum length of the returned
    string (including any indicator); indicator is the string added at
    the end to indicate that the text has been shortened

    >>> shorten("Second Variety")
    'Second Variety'
    >>> shorten("Voices from the Street", 17)
    'Voices from th...'
    >>> shorten("Radio Free Albemuth", 10, "*")
    'Radio Fre*'
```

```
    """
    if len(text) > length:
        text = text[:length - len(indicator)] + indicator
    return text


if __name__ == '__main__':
    import doctest

    doctest.testmod()
```

To use the our module:

```
import TextUtil
text = " a puzzling conundrum "
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.

Doctesting is done by:

```
if __name__ == '__main__':
    import doctest

    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 Pythons 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

```
import sys
import io

print('hello', file=sys.stdout)
print('hello', file=sys.stderr)
sys.stdout.write('world')

fh = open('text.txt', 'w')
print('hello world', file=fh)

string_io = io.StringIO()
sys.stdout = string_io

for i in range(100):
    print('hello' + i)

sys.stdout = sys.__stdout__  # recover the default sys.stdout
print(string_io.getvalue())  # get the value in string io
```

5.2.2 Dates and Times

```
import datetime
import time

# current seconds
current_second = time.time()

# current date and time
current_time = datetime.datetime.now()

# current struct time
current_struct_time = time.localtime()

# format from struct time
string_time = time.strftime('%Y-%m-%d %H:%M:%S', current_struct_time)

# struct time from string
struct_time = time.strptime('2021-02-04 11:07:09', '%Y-%m-%d %H:%M:%S')

# seconds from struct time
seconds = time.mktime(current_struct_time)
```

5.2.3 Algorithms and collection data types

```
import heapq

heap = []
heapq.heappush(heap, (5, 'a'))
```

```
heapq.heappush(heap, (2, 'b'))
heapq.heappush(heap, (4, 'c'))

for _ in range(len(heap)):
    print(heapq.heappop(heap))

h = [1, 5, 11, 10, 3, 6, 20, 8, 7, 2, 9]
heapq.heapify(h)
for _ in range(len(h)):
    print(heapq.heappop(h))
```

5.2.4 File formats, encodings, and data persistence

```
import base64

binary = open('ali.png', 'rb').read()
ascii_text = ''
for i, c in enumerate(base64.b64encode(binary)):
    if i and i % 68 == 0:
        ascii_text += '\\\n'
    ascii_text += chr(c)
print(ascii_text)

binary = base64.b64decode(ascii_text)
open('ali_copy.png', 'wb').write(binary)
```

```
import tarfile
import string
import sys
import os

BZ2_AVAILABLE = True
try:
    import bz2
except ImportError:
    BZ2_AVAILABLE = False

# absolute path is not permitted
UNTRUSTED_PREFIXES = tuple(['/', '\\'] + [c + ':' for c in string.ascii_letters])


def untar(archive):
    tar = None
    try:
        tar = tarfile.open(archive)
        for member in tar.getmembers():
            if member.name.startswith(UNTRUSTED_PREFIXES):
                print('untrusted prefix, ignoring', member.name)
            elif '..' in member.name:
                print('suspect path, ignoring', member.name)
            else:
                tar.extract(member)
                print('unpacked', member.name)
    except (tarfile.TarError, EnvironmentError) as err:
        print(err)
    finally:
        if tar is not None:
            tar.close()


def error(message, exit_status=1):
    print(message)
    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.

```python
#!/usr/bin/env python3
# Copyright (c) 2021-02

import math


class Point:
    def __init__(self, x=0, y=0):
        """
        A 2D cartesian coordinate
        :param x:
        :param y:
        """
        self.x = x
        self.y = y

```

```
    def distance_from_origin(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return f'Point({self.x!r}, {self.y!r})'

    def __str__(self):
        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 `AtributeError` 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:

```
    def __eq__(self, other):
        if not isinstance(other, Point):
            return NotImplemented
        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 comparion with the `Point` type, and if there is no such method or if that method alse 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)`

Poweful `eval()`: (`eval(expression)`)

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

```
    a0 = 0
    a1 = 1
    a2 = 2
    a3 = 3

    for i in range(4):
        print(eval(f'a{i} * {i}'))
```

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

```
    for i in range(3):
        exec(f'a{i} = {i}')
        exec(f'print(a{i})')
    exec('me = "Mike Chyson"')
    print(me)

    exec('''
    def say_hello():
        print('hello')
    ''')
    say_hello()
```

6.1.2 Inheritance and plymorphism

```
class Circle(Point):
    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius

    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin() - self.radius)

    def area(self):
        return math.pi * (self.radius ** 2)

    def circumference(self):
        return 2 * math.pi * self.radius

    def __eq__(self, other):
        return self.radius == other.radius and super().__eq__(other)
```

```

    def __repr__(self):
        return f'Circle({self.radius!r}, {self.x!r}, {self.y!r})'

    def __str__(self):
        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.

```
class Circle(Point):
    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius

    @property
    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin - self.radius)

    @property
    def area(self):
        return math.pi * (self.radius ** 2)

    @property
    def circumference(self):
        return 2 * math.pi * self.radius

    @roperty
    def radius(self):
        return self.__radius

    @property.setter
    def radius(self, radius):
        assert radius > 0, 'radius must be nonzero and non-negative'
        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:

```
def area(self):
    return math.pi * (self.radius ** 2)
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.

```
    @property
    def radius(self):
        return self.__radius

    @property.setter
    def radius(self, radius):
        assert radius > 0, 'radius must be nonzero and non-negative'
        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` properity'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 doesnt 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 pragramming 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.

```
# (A)dd (E)dit (L)ist (R)emove (I)mport e(X)port (Q)uit
if action == "a":
    add_dvd(db)
elif action == "e":
    edit_dvd(db)
elif action == "l":
    list_dvds(db)
elif action == "r":
    remove_dvd(db)
```

```
elif action == "i":
    import_(db)
elif action == "x":
    export(db)
elif action == "q":
    quit(db)

# The same effect
# (A)dd (E)dit (L)ist (R)emove (I)mport e(X)port (Q)uit
functions = dict(a=add_dvd, e=edit_dvd, l=list_dvds, r=remove_dvd,
                 i=import_, x=export, q=quit)
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.

```
    def import_(self, filename, reader=None):
        extension = os.path.splitext(filename)[1].lower()
        call = {
            ('.aix', 'dom'): self.import_xml_dom,
            ('.aix', 'etree'): self.import_xml_etree,
            ('.aix', 'sax'): self.import_xml_sax,
            ('.ait', 'manual'): self.import_text_manual,
            ('ait', 'regex'): self.import_text_regex,
            ('.aib', None): self.import_binary,
            ('.aip', None): self.import_pickle
        }
        result = call[extension, reader](filename)
        if not result:
            self.clear()
        return result
```

8.1.2 Generator expressions and functions

Generator expression:

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

```
def items_in_key_order(d):
    for key in sorted(d):
        yield key, d[key]  # yield


def items_in_key_order2(d):
    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()`.

Generator 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 generator produce as many values as we ask for – without any upper limit. For example

```
def quarters(next_quarter=0.0):
    while True:
        yield next_quarter
        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:

```
result = []
for x in quarters():
    result.append(x)
    if x >= 1.0:
        break
# [0.0, 0.25, 0.5, 0.75, 1.0]
```

```
def quarters(next_quarter=0.0):
    while True:
        received = (yield next_quarter)
        if received is None:
            next_quarter += 0.25
        else:
            next_quarter = received


result = []
generator = quarters()
while len(result) < 5:
    x = next(generator)
    if abs(x - 0.5) < sys.float_info.epsilon:
        x = generator.send(1.0)  # Notice this
    result.append(x)
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

```
import math

x = eval('2 ** 10')
print(x)  # 1024

code = '''
def area_of_sphere(r):
    return 4 * math.pi * r ** 2
```

```
'''
# context = {}
# context['math'] = math
# exec(code, context)  # define the function area_of_sphere
context = globals().copy()
exec(code, context)

area_of_sphere = context['area_of_sphere']
sphere = area_of_sphere(5)
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 bing 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 modules 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 Pythons introspection functions to check the availability of the functionality we want, and to access it as required.

The main function is:

```
def main():
    modules = load_modules()
    get_file_type_functions = []
    for module in modules:
        get_file_type = get_function(module, 'get_file_type')
        if get_file_type is None:
            get_file_type_functions.append(get_file_type)

    for file in get_files(sys.argv[1:]):
        fh = None
        try:
            fh = open(file, 'rb')
            magic = fh.read(1000)
            for get_file_type in get_file_type_functions:
                filetype = get_file_type(magic, os.path.splitext(file)[1])  # file extension
                if filetype is not None:
                    print(f'{filetype:.<20}{file}')
                    break
```

```
            else:
                print('{:.<20}{}'.format('Unknown', file))
        except EnvironmentError as err:
            print(err)
        finally:
            if fh is not None:
                fh.close()
```

The longest and most difficult approach (approach 1):

```
def load_modules():
    modules = []
    for name in os.listdir(os.path.dirname(__file__) or '.'):
        if name.endswith('.py') and 'magic' in name.lower():
            filename = name
            name = os.path.splitext(name)[0]  # remove extension
            if name.isidentifier() and name not in sys.modules:
                fh = None
                try:
                    fh = open(filename)
                    code = fh.read()
                    module = type(sys)(name)
                    sys.modules[name] = module
                    exec(code, module.__dict__)
                    modules.append(module)
                except (EnvironmentError, SyntaxError) as err:
                    sys.modules.pop(name, None)
                    print(err)
                finally:
                    if fh is not None:
                        fh.close()
    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 interger 5 in variable `x` by writting `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:

```
                try:
                    exec('import ' + name)
```

```
            modules.append(sys.modules[name])
        except SyntaxError as err:
            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):

```
        try:
            module = __import__(name)
            modules.append(module)
        except SyntaxError as err:
            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()`.

```
def get_function(module, function_name):
    function = get_function.cache.get((module, function_name), None)
    if function is None:
        try:
            function = getattr(module, function_name)
            if not hasattr(function, '__call__'):
                raise AttributeError()
            get_function.cache[(module, function_name)] = function
        except AttributeError:
            function = None
    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 isnt available.

If hundreds of files were being processed (e.g. *.*), we don't want to go throught 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 docorated function or method with some additional functionality added.

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

Here's the decorator's implementation:

```
def positive_result(function):
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result

    wrapper.__name__ = function.__name__
    wrapper.__doc__ = function.__doc__
    return wrapper
```

Docorator 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:

```
import functools

```

```
def positive_result(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >=0"
        return result

    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 docstring 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:

```
@bounded(0, 100)
def percent(amount, total):
    return (amount / total) * 100
```

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

```
def bounded(minimum, maximum):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            result = function(*args, **kwargs)
            if result < minimum:
                return minimum
            elif result > maximum:
                return maximum
            return result

        return wrapper

    return decorator
```

Here is a log decorator:

```
import logging
import functools
import os
import tempfile


def logged(file):
    """
    Log the output of the decorated function into a logged file.

    :param file: If file is None, use temple file, otherwise use the given file
    :return: decorated function
    """
    def decorator(function):
        if __debug__:
            logger = logging.getLogger('Logger')
            logger.setLevel(logging.DEBUG)
            if file is None:
                handler = logging.FileHandler(os.path.join(tempfile.gettempdir(), 'logged.log'))
            else:
                handler = logging.FileHandler(file)
```

```
            logger.addHandler(handler)

            @functools.wraps(function)
            def wrapper(*args, **kwargs):
                # accumulate string
                log = 'called: ' + function.__name__ + '('
                log += ', '.join([f'{a!r}' for a in args] + [f'{k!s}={v!r}' for k, v in kwargs.items()])
                result = exception = None
                try:
                    result = function(*args, **kwargs)
                    return result
                except Exception as err:
                    exception = err
                finally:
                    log += (') -> ' + str(result)) if exception is None else f'{type(exception)}: {exception}'
                    logger.debug(log)
                    if exception is not None:
                        raise exception

            return wrapper
        else:
            return function

    return decorator


@logged(None)
def say_word(word='hello'):
    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 annoation, and so is the arrow return expression part (`-> rexp`).

If annoations 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 annoate all, some, or none of the parameters and to annoate 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.

```
def is_unicode_punctuations(s: str) -> bool:
    for c in s:
        # print(unicodedata.category(c))

        # Every Unicode character belongs to a particular category and each category is
```

```
        # identified by a two-character identifier. All the categories that begin with P are
        # punctuation characters.
        if unicodedata.category(c)[0] != 'P':
            return False
    return True


print(is_unicode_punctuations.__annotations__)
# {'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:

```
import inspect
import functools

def strictly_typed(function):
    """
    # This decorator requires that every argument and the return value must be
    # annotated with the expected type.
    # Notice that the checking is done only in debug mode (which is Pythons default
    # mode - controlled by the -O command-line option and the PYTHONOPTIMIZE environment variable).

    :param function:
    :return: decorated function
    """
    annotations = function.__annotations__
    arg_spec = inspect.getfullargspec(function)

    # assert all type is given
    assert 'return' in annotations, 'missing type for return value'
    # arg_spec.args: positional arguments
    # kwonlyargs: keyword only arguments (kwargs after position delimiter sing(*))
    for arg in arg_spec.args + arg_spec.kwonlyargs:
        assert arg in annotations, f'missing type for parameter "{arg}"'

    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        # argument check
        # zip() returns an iterator and dictionary.times() returns a dictionary view
        # we cannot concatenate them directly, so first we convert them both to lists.
        all_args = list(zip(arg_spec.args, args)) + list(kwargs.items())
        for name, arg in all_args:
            assert isinstance(arg, annotations[name]), (
                f'expected argument "{name}" of {annotations[name]} got {type(arg)}')

        # result check
        result = function(*args, **kwargs)
        assert isinstance(result, annotations['return']), (
            f'expected return of {annotations["return"]} got {type(result)}')

        return result

    return wrapper
```

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

```
@strictly_typed
def just_type(s1: str, n1: int, *, s2: str = 's2') -> bool:
    return True

print(just_type.__annotations__)
just_type('s1', 1, s2='2')
just_type(1, 2)
```

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

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

8.2 Further object-oriented programming

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y


class PointFixedAttribute:
    __slots__ = ("x", "y")

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y


if __name__ == '__main__':
    pfa = PointFixedAttribute()
    p = Point()

    # print(pfa.__dict__)  # AttributeError: 'PointFixedAttribute' object has no attribute '__dict__'
    print(pfa.__slots__)  # ('x', 'y')
    print(p.__dict__)  # {'x': 0, 'y': 0}

    # pfa.z = 1  # AttributeError: 'PointFixedAttribute' object has no attribute 'z'
    p.z = 1
    print(p.z)  # 1
    print(p.__dict__)  # {'x': 0, 'y': 0, 'z': 1}
```

When a class is created without the use of `__slot__`, behind the scences Python creates a private dictionary called `__dict__` for each **instance**, and this dictionary holds the instances'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 convetional 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

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

```
class Ord:
    def __getattr__(self, item):
        # builtin.ord() is used to avoid ord is used my the user
        # like ord = Ord()
        return builtins.ord(item)

ord = Ord()
print(ord.a)  # 97
print(ord.Z)  # 90
```

```
class Const:
    def __setattr__(self, key, value):
        if key in self.__dict__:
            raise ValueError('cannot change a const attribute')
        self.__dict__[key] = value

    def __delattr__(self, item):
        if item in self.__dict__:
            raise ValueError('cannot delete a const attribute')
        raise AttributeError(f"'{self.__class__.__name__}' object has no attribute '{item}'")

const = Const()
const.limit = 591
print(const.limit)
# const.limit = 1  # ValueError: cannot change a const attribute
# del const.limit  # ValueError: cannot delete a const attribute
```

The class work 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 properites, here is a different solution:

```
USE_GETATTR = True


class Image:
    def __init__(self, width, height, filename="", background="#FFFFFF"):
        self.filename = filename
        self.__background = background
        self.__data = {}
        self.__width = width
        self.__height = height
        self.__colors = {self.__background}

    if USE_GETATTR:
        def __getattr__(self, name):
            if name == 'color':
                return set(self.__colors)
            classname = self.__class__.__name__
            if name in frozenset({'background', 'width', 'height'}):
```

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

Figure 8.1: Attribute access speical methods

```
                # image = Image(10, 10)
                # image.__dict__
                # {'filename': '',
                #  '_Image__background': '#FFFFFF',
                #  '_Image__data': {},
                #  '_Image__width': 10,
                #  '_Image__height': 10,
                #  '_Image__colors': {'#FFFFFF'}}
                return self.__dict__[f'_{classname}__{name}']
            raise AttributeError(f"'{classname}' object has no attribute {name}")
    else:
        @property
        def background(self):
            return self.__background

        @property
        def width(self):
            return self.__width

        @property
        def height(self):
            return self.__height

        @property
        def colors(self):
            return set(self.__colors)
```

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

If we attemt 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.

```
class Strip:
    def __init__(self, characters):
        self.characters = characters

    def __call__(self, string):
        return string.strip(self.characters)

strip_punctuation = Strip(',;:.!?')
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.**

```
def make_strip_function(characters):
    def strip_function(string):
        return string.strip(characters)

    return strip_function

strip_punctuation = make_strip_function(',;:.!?')
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:

```
# without context manager
fh = None
try:
    fh = open(filename)
    for line in fh:
        process(line)
except EnvironmentError as err:
    print(err)
finally:
    if fh is not None:
        fh.close()

# with context manager
try:
```

```
    with open(filename) as fh:
        for line in fh:
            process(line)
except EnvironmentError as err:
    print(err)
```

```
try:
    with open(source) as fin, open(target, 'w') as fout:
        for line in fin:
            fout.write(process(line))
except EnvironmentError as err:
    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 isnt 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:

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

Here is the code for the AtomicList context manager:

```
class AtomicList:
    def __init__(self, alist, shallow_copy=True):
        self.original = alist
        self.shallow_copy = shallow_copy

    def __enter__(self):
        self.modified = (self.original[:] if self.shallow_copy else copy.deepcopy(self.original))
        return self.modified

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is None:  # exception type
            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 lists 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:

```
class Product:
    __slots__ = ('__name', '__description', '__price')

    name_as_xml = XmlShadow('name')
    description_as_xml = XmlShadow('description')

    def __init__(self, name, description, price):
        self.__name = name
        self.__description = description
        self.__price = price

    @property
    def name(self):
        return self.__name

    @property
    def description(self):
        return self.__description

    @description.setter
```

```
    def description(self, description):
        self.__description = description

    @property
    def price(self):
        return self.__price

    @price.setter
    def price(self, price):
        self.__price = price

product = Product("Chisel <3cm>", "Chisel & cap", 45.25)
print(product.name, product.name_as_xml, product.description_as_xml, sep='\n')
# Chisel <3cm>
# Chisel &lt;3cm&gt;
# 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 work 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.

```
from xml.sax.saxutils import escape

class XmlShadow:
    def __init__(self, attribute_name):
        self.attribute_name = attribute_name

    def __get__(self, instance, owner=None):
        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.

```
class CachedXmlShadow:
    def __init__(self, attribute_name):
        self.attribute_name = attribute_name
        self.cache = {}

    def __get__(self, instance, owner=None):
        xml_text = self.cache.get(id(instance))
        if xml_text is not None:
            return xml_text
        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 attrbute data, with the object not needing to store anything itself.

```
class Point:
    # By setting __slots__ to an empty tuple we ensure that the class cannot store
    # any data attributes at all.
    __slots__ = ()
    x = ExternalStorage('x')
    y = ExternalStorage('y')

    def __init__(self, x=0, y=0):
        self.x = x
        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.

```
class ExternalStorage:
    __slots__ = ('attribute_name',)
    __storage = {}  # class attribute

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name

    def __set__(self, instance, value):
        self.__storage[id(instance), self.attribute_name] = value

    def __get__(self, instance, owner):
        if instance is None:
            d = {}
            for k in self.__storage.keys():
                if self.attribute_name in k:
                    d[k] = self.__storage[k]
            return d
        return self.__storage[id(instance), self.attribute_name]


p = Point(3, 4)
print(p.x, p.y)  # 3 4
p = Point(1, 2)
print(p.x, p.y)  # 1 2
print(Point.x, Point.y, Point, sep='\n')
# {(140338432304624, 'x'): 3, (140338432304640, 'x'): 1}
# {(140338432304624, 'y'): 4, (140338432304640, 'y'): 2}
# <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:

```
class NameAndExtension:
    def __init__(self, name, extension):
        self.__name = name
        self.extension = extension

    @Property
    def name(self):
        return self.__name

    @Property
    def extension(self):
        return self.__extension

    @extension.setter
    def extension(self, extension):
        self.__extension = extension
```

Here's the `Property` decorator:

```
class Property:
    def __init__(self, getter, setter=None):
        self.__getter = getter
        self.__setter = setter
        self.__name__ = getter.__name__

    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.__getter(instance)

    def __set__(self, instance, value):
        if self.__setter is None:
            raise AttributeError(f"'{self.__name__}' is read-only")
        return self.__setter(instance, value)

    def setter(self, setter):
        self.__setter = setter
        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, possiblely 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 is is not. In face, `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 is 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:

```
_identity = lambda x: x


class SortedList:
    def __init__(self, sequence=None, key=None):
        self.__key = key or _identity
        assert hasattr(self.__key, "__call__")
        if sequence is None:
            self.__list = []
        elif (isinstance(sequence, SortedList) and
              sequence.key == self.__key):
            self.__list = sequence.__list[:]
        else:
            self.__list = sorted(list(sequence), key=self.__key)

    def pop(self, index=-1):
        return self.__list.pop(index)

    def __delitem__(self, index):
        del self.__list[index]

    def __getitem__(self, index):
        return self.__list[index]

    def __setitem__(self, index, value):
        raise TypeError("use add() to insert a value and rely on "
                        "the list to put it in the right place")

    def __iter__(self):
        return iter(self.__list)

    def __reversed__(self):
        return reversed(self.__list)

    def __len__(self):
        return len(self.__list)

    def __str__(self):
        return str(self.__list)
```

Here's the decorated version:

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

```
class SortedList:

    def __init__(self, sequence=None, key=None):
        self.__key = key or _identity
        assert hasattr(self.__key, "__call__")
        if sequence is None:
            self.__list = []
        elif (isinstance(sequence, SortedList) and
              sequence.key == self.__key):
            self.__list = sequence.__list[:]
        else:
            self.__list = sorted(list(sequence), key=self.__key)
```

Here's class decorater:

```
def delegate(attribute_name, method_names):
    def decorator(cls):
        nonlocal attribute_name
        if attribute_name.startswith('__'):
            attribute_name = '_' + cls.__name__ + attribute_name
        for name in method_names:
            setattr(cls, name, eval("lambda self, *a, **kw: "
                                    f"self.{attribute_name}.{name}(*a, **kw)"))
        return cls

    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 decorators 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:

```
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.

```
@complete_comparisons
class FuzzyBool:
```

```
    def __init__(self, value=0.0):
        self.__value = value if 0.0 <= value <= 1.0 else 0.0

    def __lt__(self, other):
        return self.__value < other.__value
```

Here's the decorator:

```
def complete_comparisons(cls):
    assert cls.__lt__ is not object.__lt__, (
        f'{cls.__name__} must define < and ideally =='
    )
    if cls.__eq__ is object.__eq__:
        cls.__eq__ = lambda self, other: (
            not (cls.__lt__(self, other) or cls.__lt__(other, self))
        )
    cls.__ne__ = lambda self, other: not cls.__eq__(self, other)
    cls.__gt__ = lambda self, other: cls.__lt__(other, self)
    cls.__le__ = lambda self, other: not cls.__lt__(other, self)
    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:

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

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

```
import abc


class Appliance(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def __init__(self, model, price):
        self.__model = model
        self.price = price

    def get_price(self):
        return self.__price

    def set_price(self, price):
        self.__price = price

    price = abc.abstractproperty(get_price, set_price)

    @property
```

```
    def model(self):
        return self.__model
```

We have set the classs 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 cant 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/wriable data as a property. We initialize the **property** 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 doesnt in this particular example).

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

Here is an example subclass:

```
class Cooker(Appliance):
    def __init__(self, model, price, fuel):
        super().__init__(model, price)
        self.fuel = fuel

    price = property(lambda self: super().price,
                     lambda self, price: super().set_price(price))
```

Example: TextFilter

```
import abc


class TextFilter(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def is_tranformer(self):
        raise NotImplementedError()

    @abc.abstractmethod
    def __call__(self):
        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 dont 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:

```
class CharCounter(TextFilter):
    @property
    def is_tranformer(self):
        return False
```

```

    def __call__(self, text, chars):
        count = 0
        for c in text:
            if c in chars:
                count += 1
        return count


class RunLengthEncoder(TextFilter):
    @property
    def is_tranformer(self):
        return True

    def __call__(self, utf8_string):
        byte = None
        count = 0
        binary = bytearray()
        for b in utf8_string.encode("utf8"):
            if byte is None:
                if b == 0:
                    binary.extend((0, 1, 0))
                else:
                    byte = b
                    count = 1
            else:
                if byte == b:
                    count += 1
                    if count == 255:
                        binary.extend((0, count, b))
                        byte = None
                        count = 0
                else:
                    if count == 1:
                        binary.append(byte)
                    elif count == 2:
                        binary.extend((byte, byte))
                    elif count > 2:
                        binary.extend((0, count, byte))
                    if b == 0:
                        binary.extend((0, 1, 0))
                        byte = None
                        count = 0
                    else:
                        byte = b
                        count = 1
        if count == 1:
            binary.append(byte)
        elif count == 2:
            binary.extend((byte, byte))
        elif count > 2:
            binary.extend((0, count, byte))
        return bytes(binary)


class RunLengthDecoder(TextFilter):
    @property
    def is_tranformer(self):
        return True

    def __call__(self, rle_bytes):
        binary = bytearray()
        length = None
        for b in rle_bytes:
            if length == 0:
                length = b
            elif length is not None:
                binary.extend([b for x in range(length)])
                length = None
            elif b == 0:
                length = 0
            else:
                binary.append(b)
```

```
                length = None
        if length:
            binary.extend([b for x in range(length)])
        return binary.decode("utf8")


if __name__ == '__main__':
    vowel_counter = CharCounter()
    count = vowel_counter('dog fish and cat fish', 'aeiou')
    print(count)  # 5

    print('=' * 100)
    text = 'Mack Chyson ======================='
    encoder = RunLengthEncoder()
    encoded_text = encoder(text)
    print(encoded_text)  # b'Mack Chyson \x00\x17='
    decoder = RunLengthDecoder()
    original_text = decoder(encoded_text)
    print(original_text)
    # Mack Chyson =======================
```

Example: Abstract

```
class Undo(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def __init__(self):
        self.__undos = []

    @abc.abstractmethod
    def can_undo(self):
        return bool(self.__undos)

    @abc.abstractmethod
    def undo(self):
        assert self.__undos, 'nothing left to undo'
        self.__undos.pop()(self)

    def add_undo(self, undo):
        self.__undos.append(undo)

    def clear(self):
        self.__undos = []
```

The `self.__undos` list is exprected 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:

```
class Stack(Undo):
    def __init__(self):
        super().__init__()
        self.__stack = []

    @property
    def can_undo(self):
        return super().can_undo

    def undo(self):
        super().undo()

    def push(self, item):
```

```
        self.__stack.append(item)
        self.add_undo(lambda self: self.__stack.pop())

    def pop(self):
        item = self.__stack.pop()
        self.add_undo(lambda self: self.__stack.append(item))
        return item

    def top(self):
        assert self.__stack, 'Stack is empty'
        return self.__stack[-1]

    def __str__(self):
        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 ba avoided by:

- Using single inheritance and setting a metaclass if we want to support an additional API
- Using mutiple inheritance with one concrete class and one or more abstract base classes for addtional 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:

```
class LoadSave:
    def __init__(self, filename, *attribute_names):
        self.filename = filename
        self.__attribute_names = []
        for name in attribute_names:
            if name.startswith('__'):
                name = '_' + self.__class__.__name__ + name
            self.__attribute_names.append(name)

    def save(self):
        with open(self.filename, 'wb') as fh:
            data = []
            for name in self.__attribute_names:
                data.append(getattr(self, name))
            pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)
```

```

    def load(self):
        with open(self.filename, 'rb') as fh:
            data = pickle.load(fh)
            for name, value in zip(self.___attribute_names, data):
                setattr(self, name, value)
```

```
class FileStack(Undo, LoadSave):
    def ___init___(self, filename):
        Undo.___init___(self)
        LoadSave.___init___(self, filename, '___stack')
        self.___stack = []

    def load(self):
        super().load()
        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 intensions.

> 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.

```
import collections


class SortedList:
    pass


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 it registed 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:

```
# API guarantee
class LoadableSavable(type):
    def __init__(cls, classname, bases, dictionary):
        super().__init__(classname, bases, dictionary)

        assert (
            hasattr(cls, 'load') and
            isinstance(getattr(cls, 'load')), collections.abc.Callable
        ), "class '" + classname + "' must provide a load() method"
        assert (
                hasattr(cls, 'save') and
                isinstance(getattr(cls, 'save'), collections.abc.Callable)
        ), "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 reimplementation of the metaclass's `__new__()` method.

```
# class Bad(metaclass=LoadableSavable):
#     def some_method(self): pass
# AssertionError: class 'Bad' must provide a load() method

class Good(metaclass=LoadableSavable):
    def load(self): pass
    def save(self): pass


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; buf for other changes, such as adding methods or data attributes, reimplementing `__init__()` is sufficient, although this can also be done in `__new__()`.

This exmaple is just used to show the modification function. Normall the following code is not good. Suppose we did not use property decorators before, but use a simple naming convetion ti 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:

```
class Product(metaclass=AutoSlotProperties):
    def __init__(self, barcode, description):
        self.__barcode = barcode
        self.description = description

    def get_barcode(self):
        return self.__barcode

    def get_description(self):
        return self.__description

    def set_description(self, description):
        if description is None or len(description) < 3:
            self.__description = '<Invalid Description>'
        else:
            self.__description = description
```

This can be done using a metaclass.

```
# modifying properties
class AutoSlotProperties(type):
    def __new__(mcl, classname, bases, dictionary):
        slots = list(dictionary.get('__slot__', []))  # get slots
        for getter_name in [key for key in dictionary if key.startswith('get_')]:
            if isinstance(dictionary[getter_name], collections.abc.Callable):
                name = getter_name[4:]
                slots.append('__' + name)  # alter slots
                getter = dictionary.pop(getter_name)
                setter_name = 'set_' + name
                setter = dictionary.get(setter_name, None)
                if setter is not None and isinstance(setter, collections.abc.Callable):
                    del dictionary[setter_name]
                dictionary[name] = property(getter, setter)  # convert to property
        dictionary['__slots__'] = tuple(slots)
        return super().__new__(mcl, classname, bases, dictionary)



product = Product('111', '8mm Stapler')
print(product.barcode, product.description)  # 111 8mm Stapler
product.description = '8mm Stapler (long)'
print(product.barcode, product.description)  # 111 8mm Stapler (long)

print(product.__slots__)  # ('__barcode', '__description')

for i in dir(product):
    print(i)
# _Product__barcode
# _Product__description
# __class__
# __delattr__
# __dir__
# __doc__
# __eq__
# __format__
# __ge__
# __getattribute__
# __gt__
# __hash__
# __init__
```

```
# __init_subclass__
# __le__
# __lt__
# __module__
# __ne__
# __new__
# __reduce__
# __reduce_ex__
# __repr__
# __setattr__
# __sizeof__
# __slots__
# __str__
# __subclasshook__
# barcode
# description
```

8.3 Functional-style programming

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

- tha 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:

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

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

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

The `operator` module has functions for all of Pythons 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 dont have to pass them. Heres a very simple example:

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

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

```
Conv2D_ = functools.partial(Conv2D, kernel_size=(3, 3), activation='relu', padding='same')

h = Conv2D_(256)(input_layer)
h = Conv2D_(64)(h)

# The same full code is:
# h = Conv2D(256, (3, 3), activation='relu', padding='same')(input_layer)
# 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.

```
import os
import sys
import functools


```

```python
def coroutine(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        generator = function(*args, **kwargs)
        next(generator)
        return generator

    return wrapper


@coroutine
def reporter():
    while True:
        filename = (yield)
        print(filename)


@coroutine
def get_files(receiver):
    """
    A wrapper of os.walk()
    :param receiver:
    :return:
    """
    while True:
        path = (yield)
        if os.path.isfile(path):
            receiver.send(os.path.abspath(path))
        else:
            for root, dirs, files in os.walk(path):
                for filename in files:
                    receiver.send(os.path.abspath(os.path.join(root, filename)))


@coroutine
def suffix_matcher(receiver, suffixes):
    while True:
        filename = (yield)
        if filename.endswith(suffixes):
            receiver.send(filename)


@coroutine
def size_matcher(receiver, minimum=None, maximum=None):
    while True:
        filename = (yield)
        size = os.path.getsize(filename)
        if ((minimum is None or size >= minimum) and
                (maximum is None or size <= maximum)):
            receiver.send(filename)


if __name__ == '__main__':
    # notice the order in coroutine
    pipes = []
    pipes.append(reporter)
    pipes.append(size_matcher(pipes[-1], minimum=1024))
    pipes.append(suffix_matcher(pipes[-1], (".png", ".jpg", ".jpeg", ".py")))
    pipes.append(get_files(pipes[-1]))
    pipeline = pipes[-1]
    # Equal to
    # pipeline = get_files(suffix_matcher(size_matcher(reporter(), minimum=1024), (".png", ".jpg", ".jpeg", ".py")))

    try:
        for file in sys.argv[1:]:
            print(file)
            pipeline.send(file)
            # pipeline.py
            # /Users/mike/PycharmProjects/python3/c8_advanced_programming_techniques/functional_programming/pipeline.py
            # partial.py
            # /Users/mike/PycharmProjects/python3/c8_advanced_programming_techniques/functional_programming/partial.py
            # __init__.py
    finally:
```

```
        for pipe in pipes:
            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.

Mistackes fall into sevaral 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

```
for i in range(10)
    print(1)

#   File "/Users/mike/PycharmProjects/python3/c9_debugging_testing_profiling/t.py", line 10
#     for i in range(10)
#                      ^
# 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(ˆ) underneath indicating exactly where the error was detected.

```
try:
    if True:
        print(''
except Exception as err:
    print(err)

#   File "/Users/mike/PycharmProjects/python3/c9_debugging_testing_profiling/t.py", line 21
#     except Exception as err:
#     ^
# 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 omite 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

```
def div():
    1 / 0


if __name__ == '__main__':
    div()

# Traceback (most recent call last):
#   File "/Users/mike/PycharmProjects/python3/c9_debugging_testing_profiling/e3.py", line 17, in <module>
#     div()
#   File "/Users/mike/PycharmProjects/python3/c9_debugging_testing_profiling/e3.py", line 13, in div
#     1 / 0
# 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 **frist** 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:

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

To exercise the program's doctests there 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 Pythons 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 sued to compare the performance of functions and methods. And it also shows precisely what is being called and how long each call takes.

```
import cProfile
import math


def log(x, y):
    return math.log(x, y)


code = """
for i in range(10000):
    log(10, 2)
"""
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’ o
```

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 all 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`:

```python
import math


def log(x, y):
    return math.log(x, y)


for i in range(10000):
    log(10, 2)
```

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

```

(base) mike@Mikes-MacBook-Pro c9_debugging_testing_profiling % python -m cProfile -o profile.dat MyModule.py
(base) mike@Mikes-MacBook-Pro c9_debugging_testing_profiling % python -m pstats
Welcome to the profile statistics browser.
% read profile.dat
profile.dat% callers log
   Random listing order was used
   List reduced from 65 to 2 due to restriction <'log'>

Function                      was called by...
                                  ncalls  tottime  cumtime
{built-in method math.log}  <-   10000    0.001    0.001  MyModule.py:4(log)
MyModule.py:4(log)          <-   10000    0.002    0.003  MyModule.py:1(<module>)


profile.dat% callees log
   Random listing order was used
   List reduced from 65 to 2 due to restriction <'log'>

Function                      called...
                                  ncalls  tottime  cumtime
{built-in method math.log}  ->
MyModule.py:4(log)          ->   10000    0.001    0.001  {built-in method math.log}


profile.dat% quit
Goodbye.
```

Chapter 10

Processes and threading

With the advant 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 avaiable 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

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

```

@function:
Searches for a word specified on the command line in the files listed after the word.
This the parent program.
The corresponding child program is grepword_p_child.py.
"""
import os
import sys
import subprocess
import optparse


def main():
    child = os.path.join(os.path.dirname(__file__), 'grepword_p_child.py')
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    files_per_process = len(filelist) // opts.count
    # Usually the number of files wont be an exact multiple of the number of processes,
    # so we increase the number of files the first process is given by the remainder.
    start, end = 0, files_per_process + (len(filelist) % opts.count)
    number = 1

    pipes = []
    while start < len(filelist):
        command = [sys.executable, child]
        if opts.debug:
            command.append(str(number))
        pipe = subprocess.Popen(command, stdin=subprocess.PIPE)
        pipes.append(pipe)
        pipe.stdin.write(word.encode('utf8') + b'\n')
        for filename in filelist[start:end]:
            pipe.stdin.write(filename.encode('utf8') + b'\n')
        pipe.stdin.close()
        number += 1
        start, end = end, end + files_per_process

    while pipes:
        pipe = pipes.pop()
        pipe.wait()


def parse_options():
    parser = optparse.OptionParser(
        usage=("usage: %prog [options] word name1 "
               "[name2 [... nameN]]\n\n"
               "names are filenames or paths; paths only "
               "make sense with the -r option set"))
    parser.add_option("-p", "--processes", dest="count", default=7,
                      type="int",
                      help=("the number of child processes to use (1..20) "
                            "[default %default]"))
    parser.add_option("-r", "--recurse", dest="recurse",
                      default=False, action="store_true",
                      help="recurse into subdirectories")
    parser.add_option("-d", "--debug", dest="debug", default=False,
                      action="store_true")
    opts, args = parser.parse_args()
    if len(args) == 0:
        parser.error("a word and at least one path must be specified")
    elif len(args) == 1:
        parser.error("at least one path must be specified")
    if (not opts.recurse and
            not any([os.path.isfile(arg) for arg in args])):
        parser.error("at least one file must be specified; or use -r")
    if not (1 <= opts.count <= 20):
        parser.error("process count must be 1..20")
    return opts, args[0], args[1:]


def get_files(args, recurse):
    filelist = []
    for path in args:
        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).

```
#!/usr/bin/env python3
"""
@project: python3
@file: grepword_p_child
@author: mike
@time: 2021/2/22

@function:
"""
import sys

coding = 'utf8'
BLOCK_SIZE = 8000
number = f'{sys.argv[1]}' if len(sys.argv) == 2 else ''
stdin = sys.stdin.buffer.read()
lines = stdin.decode(coding, 'ignore').splitlines()
word = lines[0].rstrip()

for filename in lines[1:]:
    filename = filename.rstrip()
    previous = ''
    try:
        with open(filename, 'rb') as fh:
            while True:
                current = fh.read(BLOCK_SIZE)
                if not current:
                    break
                current = current.decode(coding, 'ignore')
                if word in current or word in previous[-len(word):] + current[:len(word)]:
                    print(f'{number}{filename}')
                    break
                if len(current) != BLOCK_SIZE:
                    break
                previous = current
    except EnvironmentError as err:
        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 dont 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.

```
#!/usr/bin/env python3
"""
@project: python3
@file: grepword_t
@author: mike
@time: 2021/2/22

@function:
"""

import queue
import os
import threading

BLOCK_SIZE = 8000
from grepword_p import parse_options, get_files


def main():
    opts, word, args = parse_options()
    filelist = get_files(args, opts.recurse)
    work_queue = queue.Queue()
    for i in range(opts.count):
        number = f'{i + 1}: ' if opts.debug else ''
        worker = Worker(work_queue, word, number)
        worker.daemon = True
        worker.start()
    for filename in filelist:
        work_queue.put(filename)
    work_queue.join()


class Worker(threading.Thread):
```

```
    def __init__(self, work_queue, word, number):
        super().__init__()
        self.work_queue = work_queue
        self.word = word
        self.number = number

    def run(self) -> None:
        while True:
            try:
                filename = self.work_queue.get()
                self.process(filename)
            finally:
                self.work_queue.task_done()

    def process(self, filename):
        previous = ""
        try:
            with open(filename, "rb") as fh:
                while True:
                    current = fh.read(BLOCK_SIZE)
                    if not current:
                        break
                    current = current.decode("utf8", "ignore")
                    if (self.word in current or
                            self.word in previous[-len(self.word):] +
                            current[:len(self.word)]):
                        print("{0}{1}".format(self.number, filename))
                        break
                    if len(current) != BLOCK_SIZE:
                        break
                    previous = current
        except EnvironmentError as err:
            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 nondaemon 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 dees 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()` emthod 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 ensential to the correct working of `queue.Queue.join()`.

```
#!/usr/bin/env python3
"""
@project: python3
@file: findduplicates_t
@author: mike
@time: 2021/2/23

@function:
The program iterates over all the files in the current directory (or the specified path),
recursively going into subdirectories. It compares the lengths of all the files with the
```

```
same name, and for those files that have the same name and the same size it then uses
the MD5 (Message Digest) algorithm to check whether the files are the same, reporting
any that are.
"""

import collections
import os
import queue
import threading
import hashlib
import optparse


def main():
    # parse commandline arguments
    opts, path = parse_options()
    # prepare the data
    data = collections.defaultdict(list)
    for root, dirs, files in os.walk(path):
        for filename in files:
            fullname = os.path.join(root, filename)
            try:
                key = (os.path.getsize(fullname), filename)
            except EnvironmentError:
                continue

            if key[0] == 0:
                continue

            data[key].append(fullname)

    # Create the worker threads
    work_queue = queue.PriorityQueue()
    results_queue = queue.Queue()
    # Reduce the duplicate computation of the same file
    md5_from_filename = {}
    for i in range(opts.count):
        number = f'{i + 1}: ' if opts.debug else ''
        worker = Worker(work_queue, md5_from_filename, results_queue, number)
        worker.daemon = True
        worker.start()

    # Create the result thread
    result_thread = threading.Thread(target=lambda: print_results(results_queue))
    result_thread.daemon = True
    result_thread.start()

    for size, filename in sorted(data):
        names = data[size, filename]
        if len(names) > 1:
            work_queue.put((size, names))
        # Blocks until all items in the Queue have been gotten and processed.
        work_queue.join()
        results_queue.join()


def print_results(results_queue):
    while True:
        try:
            results = results_queue.get()
            if results:
                print(results)
        finally:
            results_queue.task_done()


class Worker(threading.Thread):
    # class attribute
    Md5_lock = threading.Lock()

    def __init__(self, work_queue, md5_from_filename, results_queue, number):
        super().__init__()
        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): \n\t{2}".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

```
#!/usr/bin/env python3
"""
@project: python3
@file: Console
@author: mike
@time: 2021/2/23

@function:
"""
import sys
import datetime


class _RangeError(Exception):
    pass


def get_string(message, name="string", default=None,
               minimum_length=0, maximum_length=80,
               force_lower=False):
    message += ": " if default is None else f" [{default}]: "
    while True:
        try:
            line = input(message)
            if not line:
                if default is not None:
                    return default
                if minimum_length == 0:
                    return ""
                else:
                    raise ValueError(f"{name} may not be empty")
            if not (minimum_length <= len(line) <= maximum_length):
                raise ValueError("{0} must have at least {1} and "
                                 "at most {2} characters".format(
                    name, minimum_length, maximum_length))
            return line if not force_lower else line.lower()
```

```
        except ValueError as err:
            print("ERROR", err)


def get_integer(message, name="integer", default=None, minimum=None,
                maximum=None, allow_zero=True):
    message += ": " if default is None else f" [{default}]: "
    while True:
        try:
            line = input(message)
            if not line and default is not None:
                return default
            x = int(line)
            if x == 0:
                if allow_zero:
                    return x
                else:
                    raise _RangeError(f"{name} may not be 0")
            if ((minimum is not None and minimum > x) or
                    (maximum is not None and maximum < x)):
                raise _RangeError("{0} must be between {1} and {2} "
                                  "inclusive{3}".format(name, minimum, maximum,
                                                        (" (or 0)" if allow_zero else "")))
            return x
        except _RangeError as err:
            print("ERROR", err)
        except ValueError as err:
            print("ERROR {0} must be an integer".format(name))


def get_float(message, name="float", default=None, minimum=None,
              maximum=None, allow_zero=True):
    message += ": " if default is None else f" [{default}]: "
    while True:
        try:
            line = input(message)
            if not line and default is not None:
                return default
            x = float(line)
            if abs(x) < sys.float_info.epsilon:
                if allow_zero:
                    return x
                else:
                    raise _RangeError(f"{name} may not be 0.0")
            if ((minimum is not None and minimum > x) or
                    (maximum is not None and maximum < x)):
                raise _RangeError("{0} must be between {1} and {2} "
                                  "inclusive{3}".format(name, minimum, maximum,
                                                        (" (or 0.0)" if allow_zero else "")))
            return x
        except _RangeError as err:
            print("ERROR", err)
        except ValueError as err:
            print("ERROR {0} must be a float".format(name))


def get_bool(message, default=None):
    yes = frozenset({"1", "y", "yes", "t", "true", "ok"})
    message += " (y/yes/n/no)"
    message += ": " if default is None else f" [{default}]: "
    line = input(message)
    if not line and default is not None:
        return default in yes
    return line.lower() in yes


def get_date(message, default=None, format="%y-%m-%d"):
    # message should include the format in human-readable form, e.g.
    # for %y-%m-%d, "YY-MM-DD".
    message += ": " if default is None else f" [{default}]: "
    while True:
        try:
            line = input(message)
```

```
            if not line and default is not None:
                return default
            return datetime.datetime.strptime(line, format)
        except ValueError as err:
            print("ERROR", err)


def get_menu_choice(message, valid, default=None, force_lower=False):
    message += ": " if default is None else " [{0}]: ".format(default)
    while True:
        line = input(message)
        if not line and default is not None:
            return default
        if line not in valid:
            print("ERROR only {0} are valid choices".format(
                ", ".join(["'{0}'".format(x)
                           for x in sorted(valid)])))
        else:
            return line if not force_lower else line.lower()
```

11.2 Creating a TCP client

```
#!/usr/bin/env python3
"""
@project: python3
@file: car_registration
@author: mike
@time: 2021/2/23

@function:
"""
import sys
import Console
import collections
import struct
import pickle
import socket

Address = ['localhost', 9653]
CarTuple = collections.namedtuple("CarTuple", "seats mileage owner")


def main():
    if len(sys.argv) > 1:
        Address[0] = sys.argv[1]
    call = dict(c=get_car_details,
                m=change_mileage,
                o=change_owner,
                n=new_registration,
                s=stop_server,
                q=quit_)
    menu = '(C)ar Edit (M)ileage Edit (O)wner Edit (N)ew car (S)top server (Q)uit'
    valid = frozenset('cmonsq')
    previous_license = None
    while True:
        action = Console.get_menu_choice(menu, valid, 'c', True)
        previous_license = call[action](previous_license)


def get_car_details(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is not None:
        print('License: {0}\nSeats:    {seats}\nMileage: {mileage}\n'
              'Owner:    {owner}'.format(license, **car._asdict()))
        return license


```

```
def retrieve_car_details(previous_license):
    license = Console.get_string('License', 'license', previous_license)
    if not license:
        return previous_license, None
    license = license.upper()
    ok, *data = handle_request('GET_CAR_DETAILS', license)
    if not ok:
        print(data[0])
        return previous_license, None
    return license, CarTuple(*data)


def change_mileage(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is None:
        return previous_license
    mileage = Console.get_integer('Mileage', 'mileage', car.mileage, 0)
    if mileage == 0:
        return license
    ok, *data = handle_request('CHANGE_MILEAGE', license, mileage)
    if not ok:
        print(data[0])
    else:
        print('Mileage successfully changed')
    return license


def change_owner(previous_license):
    license, car = retrieve_car_details(previous_license)
    if car is None:
        return previous_license
    owner = Console.get_string('Owner', 'owner', car.owner)
    if not owner:
        return license
    ok, *data = handle_request('CHANGE_OWNER', license, owner)
    if not ok:
        print(data[0])
    else:
        print('Owner successfully changed')
    return license


def new_registration(previous_license):
    license = Console.get_string('License', 'license')
    if not license:
        return previous_license
    license = license.upper()
    seats = Console.get_integer('Seats', 'seats', 4, 0)
    if not (1 < seats < 10):
        return previous_license
    mileage = Console.get_integer('Mileage', 'mileage', 0, 0)
    owner = Console.get_string('Owner', 'owner')
    if not owner:
        return previous_license

    ok, *data = handle_request('NEW_REGISTRATION', license, seats, mileage, owner)
    if not ok:
        print(data[0])
    else:
        print(f'Car {license} successfully registered')
    return license


def quit_(*ignore):
    sys.exit()


def stop_server(*ignore):
    handle_request('SHUTDOWN', wait_for_reply=False)
    sys.exit()


def handle_request(*items, wait_for_reply=True):
```

```
    SizeStruct = struct.Struct('!I')
    data = pickle.dumps(items, 3)  # 3 is protocol version

    try:
        with SocketManager(tuple(Address)) as sock:
            sock.sendall(SizeStruct.pack(len(data)))
            sock.sendall(data)

            if not wait_for_reply:
                return

            size_data = sock.recv(SizeStruct.size)
            size = SizeStruct.unpack(size_data)[0]
            result = bytearray()
            while True:
                data = sock.recv(4000)
                if not data:
                    break
                result.extend(data)
                if len(result) >= size:
                    break
        return pickle.loads(result)
    except socket.error as err:
        print(f'{err}: is the server running?')
        sys.exit(1)


class SocketManager:
    def __init__(self, address):
        self.address = address

    def __enter__(self):
        # AF_INET: address family ipv4
        # SOCK_STREAM: TCP
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, *ignore):
        self.sock.close()


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 dont 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

```
#!/usr/bin/env python3
"""
@project: python3
@file: car_registration_server
@author: mike
```

```
@time: 2021/2/23

@function:
"""
import os
import pickle
import sys
import gzip
import socketserver
import threading
import struct
import copy
import random


class Car:
    def __init__(self, seats, mileage, owner):
        self.__seats = seats
        self.mileage = mileage
        self.owner = owner

    @property
    def seats(self):
        return self.__seats

    @property
    def mileage(self):
        return self.__mileage

    @mileage.setter
    def mileage(self, mileage):
        self.__mileage = mileage

    @property
    def owner(self):
        return self.__owner

    @owner.setter
    def owner(self, owner):
        self.__owner = owner

    def __str__(self):
        return f'{self.seats}, {self.mileage}, {self.owner}'


class Finish(Exception):
    pass


def main():
    filename = os.path.join(os.path.dirname(__file__), 'car_registration.dat')
    cars = load(filename)
    print(f'Loaded {len(cars)} car registrations')
    RequestHandler.Cars = cars  # set Cars attribute into RequestHandler
    server = None
    try:
        server = CarRegistrationServer(('', 9653), RequestHandler)
        server.serve_forever()
    except Exception as err:
        print('ERROR', err)
    finally:
        if server is not None:
            server.shutdown()
            save(filename, cars)
            print(f'Save {len(cars)} car registrations')


def load(filename):
    if not os.path.exists(filename):
        # Generate fake data
        cars = {}
        owners = []
        for forename, surname in zip(
```

```
                ("Warisha", "Elysha", "Liona",
                 "Kassandra", "Simone", "Halima", "Liona", "Zack",
                 "Josiah", "Sam", "Braedon", "Eleni"),
                ("Chandler", "Drennan", "Stead", "Doole", "Reneau",
                 "Dent", "Sheckles", "Dent", "Reddihough", "Dodwell",
                 "Conner", "Abson")):
            owners.append(forename + " " + surname)
        for license in (
                "1H1890C", "FHV449", "ABK3035", "215 MZN",
                "6DQX521", "174-WWA", "999991", "DA 4020", "303 LNM",
                "BEQ 0549", "1A US923", "A37 4791", "393 TUT", "458 ARW",
                "024 HYR", "SKM 648", "1253 QA", "4EB S80", "BYC 6654",
                "SRK-423", "3DB 09J", "3C-5772F", "PYJ 996", "768-VHN",
                "262 2636", "WYZ-94L", "326-PKF", "EJB-3105", "XXN-5911",
                "HVP 283", "EKW 6345", "069 DSM", "GZB-6052", "HGD-498",
                "833-132", "1XG 831", "831-THB", "HMR-299", "A04 4HE",
                "ERG 827", "XVT-2416", "306-XXL", "530-NBE", "2-4JHJ"):
            mileage = random.randint(0, 100000)
            seats = random.choice((2, 4, 5, 6, 7))
            owner = random.choice(owners)
            cars[license] = Car(seats, mileage, owner)
        return cars

    try:
        with gzip.open(filename, 'rb') as fh:
            cars = pickle.load(fh)
            print(cars)
            return cars
    except (EnvironmentError, pickle.UnpicklingError) as err:
        print(f'server cannot load data: {err}')
        sys.exit(1)


def save(filename, cars):
    try:
        with gzip.open(filename, 'wb') as fh:
            pickle.dump(cars, fh, 3)
    except (EnvironmentError, pickle.UnpicklingError) as err:
        print(f'server failed to save data: {err}')
        sys.exit(1)


class CarRegistrationServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass


class RequestHandler(socketserver.StreamRequestHandler):
    CarsLock = threading.Lock()
    CallLock = threading.Lock()

    Call = dict(
        GET_CAR_DETAILS=lambda self, *args: self.get_car_details(*args),
        CHANGE_MILEAGE=lambda self, *args: self.change_mileage(*args),
        CHANGE_OWNER=lambda self, *args: self.change_owner(*args),
        NEW_REGISTRATION=lambda self, *args: self.new_registration(*args),
        SHUTDOWN=lambda self, *args: self.shutdown(*args)
    )

    def handle(self) -> None:
        SizeStruct = struct.Struct('!I')
        size_data = self.rfile.read(SizeStruct.size)
        size = SizeStruct.unpack(size_data)[0]
        data = pickle.loads(self.rfile.read(size))

        try:
            with RequestHandler.CallLock:
                function = self.Call[data[0]]
            reply = function(self, *data[1:])
        except Finish:
            return
        data = pickle.dumps(reply, 3)
        self.wfile.write(SizeStruct.pack(len(data)))
        self.wfile.write(data)
```

```

    def shutdown(self, *ignore):
        self.server.shutdown()
        raise Finish()

    def get_car_details(self, license):
        with RequestHandler.CarsLock:
            car = copy.copy(self.Cars.get(license, None))
        if car is not None:
            return True, car.seats, car.mileage, car.owner
        return False, 'This license is not registered'

    def change_mileage(self, license, mileage):
        if mileage < 0:
            return False, 'Cannot set a negative mileage'
        with RequestHandler.CarsLock:
            car = self.Cars.get(license, None)
            if car is not None:
                if car.mileage < mileage:
                    car.mileage = mileage
                    return True, None
                return False, 'Cannot wind the odometer back'
        return False, 'This license is not registered'

    def change_owner(self, license, owner):
        with RequestHandler.CarsLock:
            car = self.Cars.get(license, None)
            if car is not None:
                car.owner = owner
                return True, None
        return False, 'This license is not registered'

    def new_registration(self, license, seats, mileage, owner):
        if not license:
            return False, 'Cannot set an empty license'
        if seats not in {2, 4, 5, 6, 7, 8, 9}:
            return False, 'Cannot register car with invalid seats'
        if mileage < 0:
            return False, 'Cannot set a negative mileage'
        if not owner:
            return False, 'Cannot set an empty owner'

        with RequestHandler.CarsLock:
            if license not in self.Cars:
                self.Cars[license] = Car(seats, mileage, owner)
                return True, None
        return False, 'Cannot register duplicate license'


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 classs methods are used in preference to the second classs 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 librarys 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
db.close()	Closes the connection.
db.commit()	Commits any pending transaction to the database; does nothing for databases that don't support transactions.
db.cursor()	Returns a databse cursor object through which queries can be executed.
db.rollback()	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
c.arraysize	The (readable/writable) number of rows that `fetchall()` will return if no size is specified
c.fetchmany(size)	Returns a sequence of rows (each row it self being a sequence); `size` default to `c.arraysize`
c.fetchall()	Returns a sequence of all the rows that have not yet been fetched
c.fetchone()	Returns the next row of the query result set as a sequence, or `None` when the results are exhausted.
c.description	A read-only sequence of 7-tuples (`name`, `type_code`, `display_size`, `internal_size`, `precision`, `scale`, `null_ok`), describing each successive column of cursor `c`
c.execute(sql, params)	Executes the SQL query in string `sql`, replacing each palceholder with the corresponding parameter from the `params` sequence of mapping if given
c.execute(sql, seqofparams)	Executes the SQL query once for each item in the `seq_of_params` sequence of sequences or mappings; this method should not be used for operations that create result sets (such as `SELECT` statements)
c.close()	Closes the cursor, `c`; this is done automatically when the curosr 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

Spliting strings

Validation

> The core of regular expression is **matchding**. 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 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 speical characters are `\.^$?+*{}[]()|`. Most of Python's standard string escapes can also be sued 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 there 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. Alse, - 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 that the shorthands can be used inside character sets, so 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 `re.DOTALL` flag; or inside a character class matches a literal .
\d	Matches a Unicode digit; or [**0-9**] with the `re.ASCII` flag
\D	Matches a Unicode nondigit; or [ˆ**0-9**] with the `re.ASCII` flag
\s	Matches a Unicode whitespace; or [**\t\n\r\f\v**] with the `re.ASCII` flag
\S	Matches a Unicode nonwhitespace; or [ˆ **\t\n\r\f\v**] with the `re.ASCII` flag
\w	Matches a Unicode "word" character; or [**a-zA-Z0-9_**] with the `re.ASCII` flag
\W	Matches a Unicode non-"word" character; or [ˆ**a-zA-Z0-9_**] with the `re.ASCII` 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 f<u>ee</u>l, 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
e? or e{0,1}	Greedily match zero or one occurence of expression e
e+ or e{1,}	Greedily match one or more occurences of expression e
e* or e{0,}	Greedily match zero or more occurences of expression e
e{m}	Match exactly m occurences of expression e
e{m,}	Greedily match at least m occurences of expression e
e{,n}	Greedily match at most n occurences of expression e
e{m,n}	Greedily match at least m and at most n occurences of expression e

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 "airplace" 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 *caputure 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 group[1]. 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-

[1]Backreferences 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 wont 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**
^	Matches at the start; also matches after each newline with the `re.MULTILINE` flag
$	Matches at the end; also matches before each newline with the `re.MULTILINE` flag
\A	Matches at the start
\b	Matches at a "word" boundary
\B	Matches at a non-"word" boundary
\Z	Matches at the end
(?=e)	Matches if the expression e matches at this assertion but does not advance over it – called lookahead or positive lookahead
(?!e)	Matches if the expression e does not match at this assertion and does not advance over it – called negative lookahead
(?<=e)	Matches if the expression e matches immediately before this assertion – called positive lookbehind
(?<!e)	Matches if the expression e does not match immediately before this assertion – called negative lookbehind

Table 13.3: Regular Expression Assertions

The key value regex is as follows:

```
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 [ ]. Heres the *key=value* regex with comments:

```
r'''
^[ \t]*                # start of lien and optional leading whitespace
(?P<key>\w+)           # the key text
[ \t]*=[ \t]*          # the equals with optional surrounding whitespace
(?P<value>[^\n]+)      # the value text
(?<![ \t])             # negative lookbehind to avoid trailing whitespace
'''
```

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

```
r'''
<img\s+                              # start of the tag
[^>]*?                               # any attributes that precede the src
src=                                 # start of the src attribute
(?:
    (?P<quote>["'])                  # opening quote
    (?P<qimage>[^\1>]+?)             # image filename
    (?P=quote)                       # closing quote matching the opening quote
|
    (?P<uimage>[^"' >]+              # unquoted image filename
)
[^>]*?                               # any attribute that follow the src
>
'''
```

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

```
import re

# manner 1
text = '#C0C0AB'
match = re.search(r'#[\dA-Fa-f]{6}\b', text)

# manner 2
color_re = re.compile(r'#[\dA-Fa-f]{6}\b')
match = color_re.search(text)

# flag
match = re.search(r'#[\dA-F]{6}\b', text, re.IGNORECASE)
match = re.search(r'(?i)#[\dA-F]{6}\b', text)
```

The methods provided by match objects are listed in Figure 13.4

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

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

Example (extract image filenames):

```
text = '''
<img src="/images/stickman.gif" alt="Stickman" width="24" height="39">
<img src="https://www.w3schools.com/images/lamp.jpg" alt="Lamp" width="32" height="32">
'''
image_re_text = r'''
<img\s+                              # start of the tag
```

Flag	Meaning
re.A or re.ASCII	Makes \b, \B, \s, \S, \w, and \W assume that strings are ASCII; the default is for these character class shorthands to depend on the Unicode specification
re.I or re.IGNORECASE	Makes the regex match case-insensitively
re.M or re.MULTILINE	Makes ^ match at the start and after each newline and $ match before each newline and at the end
re.S or re.DOTALL	Makes . 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

```
[^>]*?                                  # any attributes that precede the src
src=                                    # start of the src attribute
(?:
    (?P<quote>["'])                     # opening quote
    (?P<qimage>[^\1>]+?)                # image filename
    (?P=quote)                          # closing quote matching the opening quote
|
    (?P<uimage>[^"' >]+)                # unquoted image filename
)
[^>]*?                                  # any attribute that follow the src
>
'''
image_re = re.compile(image_re_text, re.IGNORECASE | re.VERBOSE)
image_files = []
for match in image_re.finditer(text):
    image_files.append(match.group("qimage") or match.group("uimage"))
for image_file in image_files:
    print(image_file)
# /images/stickman.gif
# https://www.w3schools.com/images/lamp.jpg
```

Example (convert html to text):

```
def html2text(html_text):
    def char_from_entity(match):
        code = html.entities.name2codepoint.get(match.group(1), 0xFFFD)
        return chr(code)

    # (?s) math . include newline
    text = re.sub(r"(?s)<!--.*?-->", "", html_text)  # HTML comments
    text = re.sub(r"<[Pp][^>]*?>", '\n\n', text)  # opening paragraph tags
    text = re.sub(r"<[^>]*?>", '', text)  # any tag
    text = re.sub(r"&#(\d+);", lambda m: chr(int(m.group(1))), text)  # &#165; for ě
    text = re.sub(r"&([A-Za-z]+);", char_from_entity, text)  # named entities
    text = re.sub(r"\n(?:[ \xA0\t]+\n)+", '\n', text)  # linesthat contain only whitespace
    # Replace sequences of two or more newlines with exactly two newlines
    text = re.sub(r"\n\n+", '\n\n', text.strip())
    return text
```

Example (switch name order in fullname):

```
import re

# from Forename Middlename1 ... MiddlenameN Surname
# to Surname,ForenameMiddlename1...MiddlenameN
names = ['Mike Ming Chyson', 'Maël Ming Li']
```

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

Figure 13.4: Match Object Attribute and Methods

```python
new_names = []
for name in names:
    # name = re.sub(r"(\w+(?:\s+\w+)*)\s+(\w+)", r"\2, \1", name)
    name = re.sub(r"(?P<forenames>\w+(?:\s+\w+)*)"
                  r"\s+(?P<surname>\w+)",
                  r"\g<surname>, \g<forenames>",
                  name)
    new_names.append(name)
for name in new_names:
    print(name)
```

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

Example (detect encoding):

```
import re

binary = b''

re_binary = r"""
# A lookbehind assertion that says that the
# match cannot be preceded by a hyphen or a word character.
(?<![-\w])
(?:(?:en)?coding|charset)  # encoding|coding|charset
(?:=(["']?)([-\w]+)(?(1)\1)
|:\s*([-\w]+))
""".encode('utf8')
match = re.search(re_binary, binary, re.IGNORECASE | re.VERBOSE)
encoding = match.group(match.lastindex) if match else b'utf8'
```

Example (split text based on whitespace):

```
text = 'hello world'
re.split(r"\s+", text)
# same to
text.split()
```

Chapter 14

Introduction to parsing

Parsing is a fundamental activity in many programs, and for all but the most trivial cases, it is a challenging topic. Parsing is often done when we need to read data that is stored in a custom format so that we can process it or perform queries on it. Or we may be required to parse a DSL (Domain-Specific Language)these are mini task-specific languages that appear to be growing in popularity. Whether we need to read data in a custom format or code written using a DSL, we will need to create a suitable parser. This can be done by handcrafting, or by using one of Pythons generic parsing modules.

Fortunately, for some data formats, we don't have to write a parse at all.

In fact, Python has built-in support for reading and writing a wide range of data formats, including:

- delimiter-separated data with the `csv` module
- Windows-style `.ini` files with the `configparse` module
- JSON data with the `json` module
- ...

In general, if Python already has a suitable parser in the standard library, or as a third-party add-on, it is usually best to use it rather than to write our own.

When it comes to parsing data formats or DSLs for which no parse is available, rather than handcrafting a parser, we can use one of Python's third-party general-purpose parsing modules.

14.1 BNF syntax and parsing terminology

Parsing is a means of transforming data that is in some structured format into a representation that reflects the data's structure and that can be used to infer the meaning that the data represents. The parsing process is most often done in two phases:

1. lexing (also called lexical analysis, tokenizing, or scanning)
2. parsing proper (also called syntactic analysis)

The lexing phase is used to convert the data into a stream of tokens. In typical cases, each token holds at least two pieces of information: the token's type (the kind of data or language construct being represented), and the token's value (which may be empty if the type stands for itself – for example, a keyword in a programming language).

The parsing phase is where a parser reads each token and performs some semantic action. The parser operates according to predefined set of grammer rules that define the syntax that the data is expected to follow. In multiphase parsers, the semantic action consists of building up an internal representation of the input into memory (called an Abstract Syntax Tree – AST), which serves an input to the next phase. Once the AST has been constructed, it can be traversed, for example, to query the data, or to write the data out in a different format, or to perform computations that correspond to the meaning encoded in the data.

Data formats and DSLs (Domain-Specific Language) (and programming languages generally) can be described using a **grammer** – a set of syntax rules that define what is valid syntax for the data or language. Of course, just because a statement is syntactically valid doesn't mean that it makes sense. Nonetheless, being able to define the grammer is very useful, so much so that there is commonly used syntax for describing grammers – BNF (Backus-Naur Form). Creating a BNF is the first step to creating a parser, and although not formally necessary, for all but the most trivial grammars it should be considered essential.

> In computer science, BackusNaur form or Backus normal form (BNF) is a metasyntax notation for context-free grammars, often used to describe the syntax of languages used in computing, such as computer programming languages, document formats, instruction sets and communication protocols.

In a BNF there are two kinds of items: terminals and nonterminals. A terminal is an item which is in its final form, for example, a literal number or string. A nonterminal is an item that is defined in terms of zero or more other items (which themselves may be terminals or nonterminals). Every nonterminal must ultimately be defined in terms of zero or more terminals.

Figure 14.1 shows an example BNF that defines the syntax of a file of “attributes”, to put things into perspective.

```
ATTRIBUTE_FILE ::= (ATTRIBUTE '\n')+
ATTRIBUTE      ::= NAME '=' VALUE
NAME           ::= [a-zA-Z]\w*
VALUE          ::= 'true' | 'false' | \d+ | [a-zA-Z]\w*
```

Figure 14.1: A BNF for a file of attributes

The symbol `::=` means **is defined as**. Nonterminals are written in uppercase italics (e.g., VALUE). Terminals are either literal strings enclosed in quotes or regular expressions. The definitions (on the right of the `::=`) are made up of one or more terminals or nonterminals – these must be encountered in the sequence given to meet the definition.

The BNF in Figure **??** defines the four basic arithmetic operations over itegers, as well as parenthesized subexpressions, and all with the correct precedences and (left to right) associativies.

```
INTEGER        ::= \d+
ADD_OPERATOR   ::= '+' | '-'
SCALE_OPERATOR ::= '*' | '/'
EXPRESSION     ::= TERM (ADD_OPERATOR TERM)*
TERM           ::= FACTOR (SCALE_OPERATOR FACTOR)*
FACTOR         ::= '-'? (INTEGER | '(' EXPRESSION ')')
```

Figure 14.2: A BNF for arithmetic operations

14.2 Parsing manners

Typically, there are two manners when no suitable parser can be used directly.

1. write handcraftd parser
2. write parser with general parser

Chapter 15

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 becuase 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.

15.1 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 15.1.

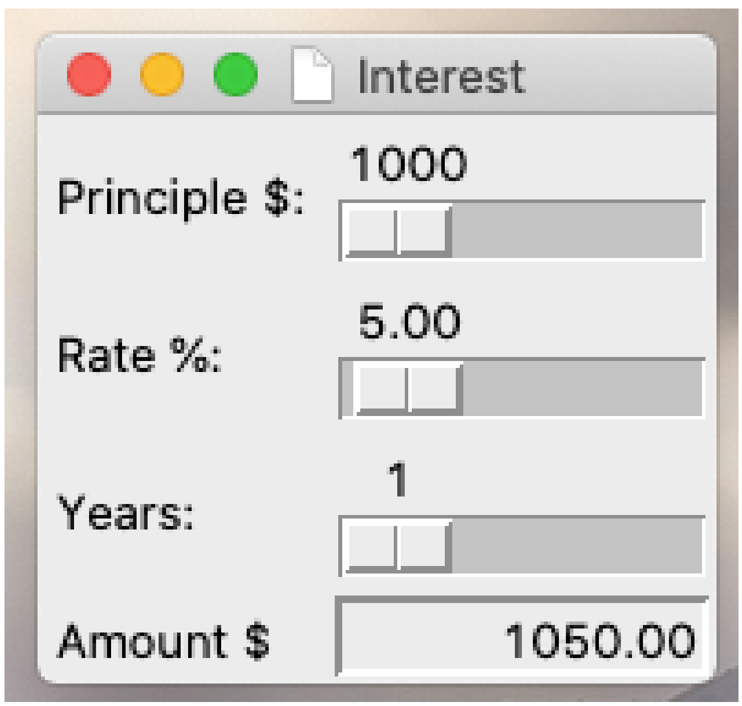

Figure 15.1: The interest program

The corresponding code is shown below:

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

```
@project: python3
@file: interest
@author: mike
@time: 2021/2/23

@function:
"""
import tkinter
import os
import sys


class MainWindow(tkinter.Frame):
    def __init__(self, parent):
        super().__init__(parent)
        self.parent = parent
        # Lays out the frame using the grid layout manager
        self.grid(row=0, column=0)

        self.principal = tkinter.DoubleVar()
        self.principal.set(1000.0)
        self.rate = tkinter.DoubleVar()
        self.rate.set(5.0)
        self.years = tkinter.IntVar()
        self.amount = tkinter.StringVar()

        principal_label = tkinter.Label(self, text='Principle $:',
                                        anchor=tkinter.W,
                                        underline=0)
        principal_scale = tkinter.Scale(self, variable=self.principal,
                                        command=self.updateUi,
                                        from_=100,
                                        to=10000000,
                                        resolution=100,
                                        orient=tkinter.HORIZONTAL)
        rate_label = tkinter.Label(self, text='Rate %:',
                                   underline=0,
                                   anchor=tkinter.W)
        rate_scale = tkinter.Scale(self, variable=self.rate,
                                   command=self.updateUi,
                                   from_=1,
                                   to=100,
                                   resolution=0.25,
                                   digits=5,
                                   orient=tkinter.HORIZONTAL)
        year_label = tkinter.Label(self, text='Years:',
                                   underline=0,
                                   anchor=tkinter.W)
        year_scale = tkinter.Scale(self, variable=self.years,
                                   command=self.updateUi,
                                   from_=1,
                                   to=50,
                                   orient=tkinter.HORIZONTAL)
        amount_label = tkinter.Label(self, text='Amount $', anchor=tkinter.W)
        actual_amount_label = tkinter.Label(self, textvariable=self.amount,
                                            relief=tkinter.SUNKEN,
                                            anchor=tkinter.E)

        principal_label.grid(row=0, column=0, padx=2, pady=2, sticky=tkinter.W)
        principal_scale.grid(row=0, column=1, padx=2, pady=2, sticky=tkinter.EW)
        rate_label.grid(row=1, column=0, padx=2, pady=2, sticky=tkinter.W)
        rate_scale.grid(row=1, column=1, padx=2, pady=2, sticky=tkinter.EW)
        year_label.grid(row=2, column=0, padx=2, pady=2, sticky=tkinter.W)
        year_scale.grid(row=2, column=1, padx=2, pady=2, sticky=tkinter.EW)
        amount_label.grid(row=3, column=0, padx=2, pady=2, sticky=tkinter.W)
        actual_amount_label.grid(row=3, column=1, padx=2, pady=2, sticky=tkinter.EW)

        principal_scale.focus_set()
        self.updateUi()
        parent.bind('<Alt-p>', lambda *ignore: principal_scale.focus_set())
        parent.bind('<Alt-r>', lambda *ignore: rate_scale.focus_set())
        parent.bind('<Alt-y>', lambda *ignore: year_scale.focus_set())
        parent.bind('<Control-q>', self.quit)
```

```python
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 variables 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 variables 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 widgets behavior and appearance. We have set the `principalLabel`s text appropriately, and set its anchor to `tkinter.W`, which means that the labels 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.

15.2 Main-window-style programs

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

Chapter 16

Environment

16.1 Repo

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

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

You can alter the repo to your preferred one.

16.2 Freeze and install packages

To generate the requirement file:

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

To install the packages from the requirement file:

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

16.3 Upgrade pip

```
python -m pip install --upgrade pip
```