
StayAhead Training
the training specialist
bringing people and technology together
Oracle Database
Oracle Development
Oracle Fusion Middleware
Oracle Business Intelligence
Oracle e-Business
MySQL
MariaDB
MongoDB
Unit Testing Frameworks
Maven
Hibernate
Acceptance Testing Frameworks
Java Technology
Web Development HTML & CSS
JavaScript & jQuery
PHP, Python Perl & Ruby
OOAD
Spring
UNIX
Solaris, AIX & HP-UX
Red Hat Linux
SUSE Linux
LPI Linux
Business Analysis ITIL® & PRINCE2®
London | Birmingham | Leeds | Manchester | Sunderland | Bristol | Edinburgh
scheduled | closed | virtual training
www.stayahead.com
+44 (0) 20 7600 6116
sales@stayahead.com

Python Programming 1

Update 2.3

Python Programming 1

4 days training

Python Programming 1 Course Overview

Python is an object oriented rapid development language deployed in many scenarios in the modern world.

This Python Programming 1 course is designed to give delegates the knowledge to develop and maintain Python scripts using the current version (V3) of Python.

There are many similarities between Python V2 and Python V3. The skills gained on this course will allow the delegate to develop their own skills further using Python V2 or V3 to support the development and maintenance of scripts.

The Python Programming 1 course comprises sessions dealing with syntax, variables and data types, operators and expressions, conditions and loops, functions, objects, collections, modules and packages, strings, pattern matching, dates, exception handling, files, and databases.

Exercises and examples are used throughout the course to give practical hands-on experience with the techniques covered.

Skills Gained

The delegate will practice:

- Writing and testing simple scripts
- Representing data using built-in and custom data types
- Building expressions
- Building conditional and iterative statements
- Declaring and calling functions
- Using objects
- Creating and manipulating collections including lists, tuples, and dictionaries

- Creating and manipulating strings
- Creating modules and packages, and using third-party libraries
- Pattern matching
- Working with date and time objects
- Handling exceptions
- Reading from and writing to files and databases
- Coding in an OOP manner

Who will the Course Benefit?

The Python Programming 1 course is aimed at anyone who wants to learn Python as a first language, and developers/engineers who want to migrate to Python from another language, particularly those with little or no object-oriented knowledge.

Course Objectives

This course aims to provide the delegate with the knowledge to be able to produce Python scripts and applications that exploit all core elements of the language including variables, expressions, selection and iteration, functions, objects, collections, strings, modules, pattern matching, exception handling, I/O, and classes.

Requirements

Delegates attending this course should have some previous programming experience and be able to define general programming concepts including: compilation, execution, variables, arrays, sequence, selection, iteration, functions, objects, and classes. Moreover delegates should be able to navigate the filesystem (on the command line ideally), edit and save text files and browse the web. This knowledge can be obtained by attendance on the pre-requisite Introduction to Programming course.

Pre-Requisite Courses

- Introduction to Programming

Follow-On Courses

- Python Programming 2
- Introduction to Python Cryptography
- Apache Web Server
- PHP Programming
- PHP & MySQL for Web Development
- PHP & MariaDB for Web Development
- Perl Programming
- Ruby Programming
- Introduction to MySQL
- Introduction to MariaDB

Notes:

- Course technical content is subject to change without notice.
- Course content is structured as sessions, this does not strictly map to course timings. Concepts, content and practicals often span sessions.

Table of Contents

Chapter 1: Getting Started

Chapter 2: Scripts and Syntax

Chapter 3: Variables and Data Types

Chapter 4: Operators and Expressions

Chapter 5: Conditions and Loops

Chapter 6: Functions

Chapter 7: Objects and Classes

Chapter 8: Lists

Chapter 9: Tuples

Chapter 10: Sets

Chapter 11: Dictionaries

Chapter 12: Strings

Chapter 13: Modules and Packages

Chapter 14: Pattern Matching

Chapter 15: Exception Handling

Chapter 16: Files and the Filesystem

Chapter 17: Databases

Appendix: Case Study

CHAPTER 1

Getting Started

About Python

Python is a high-level, object-oriented programming language comparable to Perl, Ruby, and Java. It was first released in 1991 and is intended to be easier to read/write than some other, similar languages.

Python ships with a standard library of modules to support many of the most common tasks, and its interactive mode makes testing small snippets quick and easy.

As Python code is typically interpreted, it can run on almost any platform including Mac OS X, Windows, Linux, and UNIX.

Python is free to download and to use.

Python Versions

Version	Year	Major features (some, by no means all)
0.9.0	1991	Core data types, functions, modules, classes, exceptions
1.0	1994	Functional programming (lambdas etc.)
1.4	1996	Complex numbers, keyword arguments, data hiding
2.0	2000	Comprehension, garbage collection
2.2	2001	Unification of types and classes, generators
2.3	2003	bool type, yield keyword, enumerate function
2.4	2004	Set objects, generator expressions
2.5	2006	Conditional expressions, the with statement
2.6	2008	multiprocessing and json modules
3.0	2008	Rectify design flaws, **not backwards compatible**
3.1	2009	Ordered dictionaries
2.7	2010	Ordered dictionaries, unittest and argparse modules
3.2	2011	concurrent.futures module
3.3	2012	Virtual environments, unittest.mock module
3.4	2014	Bootstrapping of PIP, enum, statistics, and asyncio modules
3.5	2015	Coroutines, matrix multiplication
3.6	2016	Formatted string literals, numeric literal underscores
3.7	2018	Built-in breakpoint, nanosecond time functions

Python Documentation

The Python docs are accessible at docs.python.org. This address will direct you to the documentation for the latest version of Python (3.7.2 at the time of writing). If you're going to be coding with an older release of the software, then you should include the version number at the end of the URL, e.g. docs.python.org/2.7.

The Python docs include, among other things, a detailed tutorial, language reference, and library reference. It is, in the author's opinion, an excellent resource for those for whom Python is not his/her first language.

In addition to this manual, the tutorial at w3schools.com/python is, perhaps, more accessible for those for whom Python is his/her first language.

Python Implementations

A Python implementation is a program that is capable of executing Python code.

CPython is the de-facto standard implementation and is what you get when you download Python from python.org/downloads. When you use the standard implementation to execute your Python code it is compiled into bytecode, loaded into memory, and then interpreted. The definitions below are provided in the event some or all of these terms are new to you.

Term	Definition
Compilation	The parsing and analysis of source code into either machine code (instructions that the computer's processor understands) or bytecode.
Bytecode	Low-level code intended to be read by an interpreter. Bytecode is platform-agnostic, meaning that it can be executed on any operating system, provided an appropriate interpreter is present.
Interpreter	A program that converts bytecode into machine code. An interpreter is platform-dependent and is also referred to as a virtual machine (VM).

There are many Python implementations available. Some are based on CPython, some are not. What follows is a small sample of the many Python implementations.

Impl.	Description
Jython	Uses Java, not C, to generate the bytecode. This means the generated bytecode can be executed using a Java interpreter.
IronPython	Uses C#, not C, to generate the bytecode. This means the generated Intermediate Language code can be executed using a .NET runtime.
PyPy	Uses Just-In-Time (JIT) compilation to compile some parts of the source code into machine code, instead of bytecode.
Brython	Adapted to HTML5 with an interface to the DOM. Intended to replace JavaScript in the browser.
Tinypy	Minimalist subset – 64KB.

NOTE

A number of Python compilers are also available, enabling the compilation of Python source code or bytecode into machine code.

Installing Python

Windows

To install Python on Windows:

1. Download the installer version of your choice from python.org/downloads.
2. Run the installer, making sure to add Python to the PATH.

NOTE

The 32-bit version of Python will typically work on a 64-bit machine. The 32-bit version consumes less memory but may be less performant than the 64-bit version.

Linux

To install Python on **Debian** derivatives, use apt:

```
$ sudo apt-get install python3
```

To install Python on **Red Hat** derivatives, use yum:

```
$ sudo yum install python3
```

To install Python on **SUSE** derivatives, use zypper:

```
$ sudo zypper install python3
```

Mac OS X

A version of Python usually ships with OS X but it is likely to be out-of-date. Before installing Python you'll have to install GCC (the GNU Compiler Collection) which can be obtained by downloading XCode from https://developer.apple.com/xcode/.

Python is typically installed using a package manager like Homebrew. Instructions for installing Homebrew may be found here: https://brew.sh/#install.

To install Python on Mac OS X, use Homebrew:

```
$ brew install python3
```

Using the Interpreter

Assuming Python was added to the PATH during installation, the interpreter may be invoked on the command line as follows (assuming version 3.7):

```
$ python3.7
```

To execute a Python script, add the script path as an argument.

```
$ python3.7 my_script.py
```

To execute a Python module, add the –m option following by the module name.

```
$ python3.7 –m my_script
```

To execute a Python command or suite of commands, add the –c option followed by the command(s) surrounded in quotes.

```
$ python 3.7 –c 'print("Hello world"); print("This is Python")'
```

Interactive mode

When the interpreter is invoked without any options or arguments, it is said to be in interactive mode. Interactive mode is useful for testing small snippets of code.

```
$ python3.7
Python 3.7 (default, February 5 2019, 09:00:00)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information
$ >>>
```

The interpreter waits for a command to be input from standard in (usually the keyboard), evaluates said command, writes the result to standard out (usually the screen), and then resumes waiting for input. This cycle is referred to as REPL (read, evaluate, print, loop).

```
$ >>> print("Hello world")
Hello world
$ >>>
```

NOTE

Where the command is composed of more than one line, pressing the Enter key will result in a secondary prompt (…).

To exit interactive mode, type an end-of-file character (Control + D on UNIX/OS X, Control + Z on Windows) or execute the quit function.

```
$ >>> quit()
```


Python Editors

There are many editors available with which to write Python code. Some are written specifically for Python, and some are generic. Python typically ships with an editor named Idle, which, assuming Python was added to the PATH during installation, may be invoked on the command line as follows (assuming version 3.7):

```
$ idle3.7
```

Tabled below are some of the more popular Python-capable editors.

Editor	Description
PyCharm	Full-featured and dedicated for Python; paid and community eds.
Spyder	Optimised for data science; open source
Thonny	Designed for beginners; bundled with Python
Eclipse + PyDev	For those developers already familiar with Eclipse
Sublime Text	Good generic editor; not free but may be used in evaluation mode
Visual Studio Code	Good generic editor; open source

CHAPTER 2

Scripts and Syntax

Script Naming

According to PEP (Python Enhancement Proposal) 0008…

> *Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability.*

NOTE

A module is just a Python source code file that exposes classes/functions/variables for use in other scripts/modules.

The following example script names adhere to this convention:

- my_script.py
- myscript.py

Comments

In a Python source code file, any line or part thereof that is prefixed with the hash symbol (#) is a comment and will be ignored by the interpreter.

According the PEP 0008:

- Comments should be complete sentences
- The first word of a comment should be capitalised (unless it's an identifier)
- Each sentence of a comment should end with a period
- Two spaces should be inserted between comment sentences
- Block comments should be indented to the same level as that of the code
- Inline comments should be separated from the code by at least two spaces

Documentation strings (docstring)

A docstring is a string literal inserted as the first line in a public module, class, or function, and is, effectively, its documentation. The value of the docstring is assigned to the __doc__ attribute of the object in question (more about objects and attributes later). PEP 0257 dictates that docstrings ought to be surrounded by triple quotes.

```
def my_function():
  """A description of my function."""
  ...
```

A docstring may comprise more than one line.

```
def circumference(rad):
  """Calculate the circumference of a circle.

  Arguments:
  rad -- the radius of the circle

  Return value:
  cir -- the circumference of the circle
  """
  ...
```


Statements

A single-line statement is terminated with a newline character.

```
name = "David"
age = 30
```

Two or more statements may be written on one line using the semicolon as a delimiter.

```
name = "David"; age = 30
```

Multi-line statements (indenting)

Conditional and loop statements, and function and class declarations are multi-line statements – they each (typically) comprise more than one line. Consider the following conditional statement made up of three lines:

```
if age >= 17:
    print("You are legally entitled to drive a car")
    print("But you must first obtain a license")
print("I am not a part of the if statement")
```

The colon indicates the beginning of a code block. The statements within the code block *must* be indented. In this case, the two indented print statements form part of the conditional statement. The block ends when the indenting ends.

PEP 0008 dictates that four spaces should be used to indent each statement in a block, and that spaces are preferred over tabs. This a convention, however, and not a rule. The important thing is that you're consistent. Four spaces is not the same as a tab in the interpreter's eyes.

Maximum line length

PEP 0008 dictates that each line should be no more than 79 characters long.

A statement may be split over multiple lines where it is composed of a comma separated list of elements.

```
names = [
    "David",
    "Sarah",
    "Tom",
    "Jane"
]
```

In this case, convention dictates the use of a hanging indent.

You can split a long expression by wrapping it in brackets...

```
if (this_thing_with_a_very_long_name
    == that_thing_with_a_very_long_name):
```

or braces…

```
if {this_thing_with_a_very_long_name
    == that_thing_with_a_very_long_name}:
```

Alternatively, you may use the backslash to achieve the same result.

```
if this_thing_with_a_very_long_name \
    == that_thing_with_a_very_long_name:
```


Names

Most of the variables, functions, and classes you create will be named.

```
my_number = 1

def my_function():
    pass

class my_class:
    pass
```

In this case, my_number, my_function, and my_class are all names. The names that you choose are subject to rules and conventions.

The rules

- A name must start with a letter or an underscore
- The remainder of the name may be composed of letters, digits, and underscores
- Names are case sensitive

The conventions

- Variable and function names should be lower case, with words separated by underscores to improve readability, e.g.:

  ```
  my_variable
  my_function
  ```

- Constant names should be upper case, with words separated by underscores to improve readability, e.g.:

  ```
  MY_CONSTANT
  ```

- The first letter of a class name should be capitalised, as should the first letter of each new word in the name, e.g.:

  ```
  MyClass
  ```

- Module and package names should be lower case. Underscores may be used in module names but their use in package names is discouraged, e.g.:

  ```
  mymodule
  mypackage
  ```

- Never use lower case l (el) or upper case O (oh) as single letter names as they are not easily distinguishable from the numbers 1 (one) and 0 (zero).

Console IO

To write simple Python scripts that do something useful, we need to be able to read input from standard in (the keyboard) and write output to standard out (the screen). To do this, we exploit two built-in functions – input and print.

The input function

The built-in input function is passed an optional prompt and returns the value input by the user (as a string).

```
$ >>> value = input('Enter a value: ')
$ Enter a value: Hello world
$ >>> value
$ 'Hello world'
```

The user types Hello world and presses Enter in response to the prompt

NOTE

In this example the lines prefixed with $ >>> represent the code to be evaluated, while the lines prefixed with $ represent the result of the evaluation.

The print function

The built-in print function is passed one or more values to be written to standard out. It does not return anything.

```
$ >>> print("Hello world")
$ Hello world
```

When the print function is passed more than one value, each is written to standard out delimited by a space.

```
$ >>> print("Hello world", "this is Python")
$ Hello world this is Python
```

The print function might be used to print the value input by the user (or a variant of it).

```
$ >>> value = input('Enter a value: ')
$ Enter a value: Hello world
$ >>> print("You entered", value)
$ You entered Hello world
```


Script Execution

To execute a Python script, invoke the interpreter with the script path as an argument.

```
$ python3.7 my_script.py
```

To execute a Python script as a module, invoke the interpreter with the –m option following by the module name (excluding the extension).

```
$ python3.7 –m my_script
```

Exercises

Console IO

In this exercise you will create a simple script that prompts the user to input some information about him/herself, and then prints that same information back to the console.

1. Create a script named ch2_console_io.py.
2. Prompt the user to input his/her name and capture it in variable named name.
3. Prompt the user to input his/her age and capture it in a variable named age.
4. Print the user's name and age to the console. You might try doing this with one invocation of the built-in print function.
5. Execute the script in a terminal window.

CHAPTER 3

Variables and Data Types

Introduction

Variable

In Python, a variable is a named reference/pointer to an object (some data). Unlike some other languages, there are no primitive-type variables in Python. Everything is an object, which means that every variable is a reference-type variable. Consider the following:

```
x = 1
```

The variable x is assigned a reference to the number 1. Or to put it another way, the variable x points to the segment of memory containing the number 1.

NOTE

The assignment operator (=) dictates that a reference/pointer to the thing on the right is assigned to the variable on the left.

Assigning one variable to another has the effect of copying the reference from the variable on the right to the variable on the left.

```
y = x
```

The variable y is assigned a reference to the number 1, just like x.

Data type

Every object has a type. Consider the following:

```
name = "David"
email = "david@mail.com"
```

The objects "David" and "david@mail.com" are str (string) type objects. Python has a number of built-in data types that provide for the storage of simple data – numbers, strings of characters, lists etc. You can create your own data types too. See the chapter on Objects and Classes for more information.

Literal

The term literal is used to refer to a fixed value.

```
name = "David"
age = 30
hobbies = ["reading", "walking", "eating out"]
```

In this case, `"David"` is a string literal, `30` is an integer literal, and `["reading", "walking", "eating out"]` is a list literal. Of course, in Python, each of these literals is also an object. In fact, every literal is an object.

The type function

The built-in type function is used to determine the data type of a literal or the object referenced by a variable.

```
$ >>> type("David")
$ <class 'str'>

$ >>> name = "David"
$ >>> type(name)
$ <class 'str'>
```

Dynamic typing

A variable need not be declared before it is assigned. Indeed, doing so will yield an error:

```
$ >>> age
$ NameError: name 'age' is not defined
```

Python is a dynamically-typed language. That means that a variable is assigned a type only when it is assigned a value (at runtime). It also means that the data type of a variable may change over the course of its life.

```
$ >>> x = 1
$ >>> type(x)
$ <class 'int'>

$ >>> x = "one"
$ >>> type(x)
$ <class 'str'>
```

Strong typing

Python is a strongly-typed language. That means that the type of an object cannot change. Consider the following example:

```
$ >>> "Age: " + 30
$ TypeError: must be str, not int
```

An integer literal cannot be added to a string literal. That is, an integer cannot be coerced into being a string.

NOTE

The difference between dynamic and strong typing is subtle, but important. The type of a variable is dynamic (can change) whilst the type of an object cannot.

Numbers

There are four built-in number types with which we ought to be familiar: bool, int, float, and complex. The standard library includes other number types too: the fractions module provides support for rational numbers, and the decimal module provides support for user-definable precision.

bool

A Boolean object is one whose value is either True or False. Since Python version 3, True and False are keywords.

```
$ >>> my_bool = True
$ >>> type(my_bool)
$ <class 'bool'>
```

bool is a number type because the values True and False are equivalent to the numbers 0 (zero) and 1 (one) respectively.

```
$ >>> True == 1
$ True
$ >>> False == 0
$ True
```

NOTE

The == symbol is used to test for equality; it is the equality operator.

int

An integer object is one whose value is a whole number.

```
$ >>> my_int = 3
$ >>> type(my_int)
$ <class 'int'>
```

NOTE

In Python 2, there are two whole number types: int and long. Values of type int occupy 32 bits of memory, while values of type long can occupy any amount of memory. In Python 3, values of type int can occupy any amount of memory.

float

A floating point object is one whose value is a fractional number.

```
$ >>> my_float = 19.95
$ >>> type(my_float)
$ <class 'float'>
```

complex

A complex object is one whose value has both real and imagined fractional parts. Each part may be accessed using attributes of the resultant complex object.

```
$ >>> my_complex = 1+2j
$ >>> my_complex.real
$ 1.0
$ >>> my_complex.imag
$ 2.0
$ >>> type(my_complex)
$ <class 'complex'>
```

Binary, octal, and hexadecimal numbers

Binary numbers are prefixed with 0b.

```
$ >>> my_binary_number = 0b1010
$ >>> my_binary_number
$ 10
```

Octal numbers are prefixed with 0o.

```
$ >>> my_octal_number = 0o12
$ >>> my_octal_number
$ 10
```

NOTE

In Python 2, octal numbers are prefixed with 0 (zero) only.

Hexadecimal number are prefixed with 0x.

```
$ >>> my_hex_number = 0xa
$ >>> my_hex_number
$ 10
```

Underscores

Underscores may be included in number literals to improve readability.

```
$ >>> my_big_number = 1_000_000
```


Floating point accuracy

Floating point arithmetic has some limitations. Let us calculate

```
$ >>> 0.3 - 0.1
$ 0.19999999999999998
```

Now, that is clearly incorrect. How about checking if

```
$ >>> 0.1 + 0.1 + 0.1 == 0.3
$ False
```

What is going on? Well, like all data, floating point numbers are represented in computer memory in binary. Unfortunately, many decimal fractions are recurring sequences in binary. For example, the decimal number 0.1 is 0.0001100110011001100110011001100... in binary. This means we can only represent it approximately using a finite number of bits. This is what is known as a rounding error and is related to the computer hardware and not the programming language. These rounding errors are of the order of 2^{53}, which is insignificant for most applications. However, as we saw, we can have some cases of unexpected (or just plain wrong) results. Where higher accuracy is required, we can look to the Python decimal or fractions modules. For heavy numerical work in Python there is the NumPy package.

Collections

This section provides an introduction only to the various built-in collection types. Each collection type is covered in detail later in the course.

str

A string is a collection of characters. In Python, a string literal may be surrounded by either single or double quotes.

```
$ >>> my_string = 'Hello'
$ >>> type(my_string)
$ <class 'str'>

$ >>> my_string = "Hello"
$ >>> type(my_string)
$ <class 'str'>
```

list

A list is a collection of values, that is, a collection of anything. In Python, a list literal comprises zero or more comma separated values surrounded by square brackets.

```
$ >>> my_list = [1, 2, 3]
$ >>> my_list = [1, 2, 3]
$ >>> type(my_list)
$ <class 'list'>
$ >>> odd_numbers = [1, 3, 5, 7, 9]
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
```

A list can hold data of varied types.

```
$ >>> order = ["Milk", 0.85, 2]  # don't do this!
```

We can even have a lists of lists (making a kind of matrix)

```
>>> scores = [
... [11,12,13],
... [21,22,23]
... ]
>>> scores
[[11, 12, 13], [21, 22, 23]]
>>> scores[1]
[21, 22, 23]
>>> scores[1][2]
23
```


Each element of a list is indexed, with the first element having an index of zero.
A list element is accessible by its index.

```
$ >>> second_name = names[1]
$ >>> second_name
$ 'Sarah'

$ >>> names[2] = "Thomas"
$ >>> names[2]
$ 'Thomas'
```

tuple

A tuple is similar to a list but its elements cannot be modified. That is, after creation, elements cannot be added, edited, or removed. In Python, a tuple literal comprises one or more comma separated values surrounded by round brackets.

```
$ >>> my_tuple = ("Ball", 1.99)
$ >>> type(my_tuple)
$ <class 'tuple'>
```

Just like a list, a tuple element is accessible by its index.

```
$ >>> price = my_tuple[1]
$ >>> price
$ 1.99
```

Any attempt to change a tuple element will yield an error.

```
$ >>> my_tuple[1] = 2
$ TypeError: 'tuple' object does not support item assignment
```

set

A set is an unordered collection of hashable objects that does not permit duplicate values. It does, however, permit the adding and removing of elements. In Python, a set literal comprises one or more comma separated values surrounded by curly braces.

```
$ >>> my_set = {1, 1, 2, 3, 3, 4, 5}
$ >>> my_set
$ {1, 2, 3, 4, 5}
$ >>> type(my_set)
$ <class 'set'>
```

Unlike a tuple/list, a set does not support indexing. So the following will yield an error:

```
$ >>> second_number = my_set[1]
$ TypeError: 'set' object does not support indexing
```

Any attempt to change a set element will yield an error.

```
$ >>> my_set[0] = -1
$ TypeError: 'set' object does not support item assignment
```

dict

A dict (dictionary) is a collection of key value pairs. In Python, a dictionary literal comprises zero or more comma separated key value pairs surrounded by curly braces. Each key value pair is delimited by a colon.

```
$ >>> my_dict = {'name': 'David', 'age': 30}
$ >>> type(my_dict)
$ <class 'dict'>
```

Dictionary values are not indexed. Each value in a dictionary is accessible by its key.

```
$ >>> name = my_dict['name']
$ >>> name
$ 'David'

$ >>> my_dict['age'] = 31
$ >>> my_dict['age']
$ 31
```

A dictionary's keys and values may be of any type.

```
$ >>> accounts = {
$ ...     6001: {'name': 'Smith', 'balance': 50.0},
$ ...     6002: {'name': 'Jones', 'balance': 25.0}
$ ... }
```

In this case the dictionary keys are of type int (account number), while the dictionary values are of type dict (account name and balance). The following examples demonstrate the getting of the balance for account number 6001.

```
$ >>> smith_account = accounts[6001]
$ >>> smith_account['balance']
$ 50.0
```

or (better)…

```
$ >>> smith_account_balance = accounts[6001]['balance']
$ >>> smith_account_balance
$ 50.0
```

The following examples demonstrate the setting of the balance for account number 6001.

```
$ >>> accounts[6001] = {'name': 'Smith', 'balance': 100.0}
```

or (better)…

```
$ >>> accounts[6001]['balance'] = 100.0
```


None

None is a Python keyword but is also a value that may be assigned to a variable (much like True and False). It is intended to indicate null/no value.

```
$ >>> x = None
$ >>> print(x)
$ None
```

NOTE

Evaluating a variable assigned None in interactive mode yields no output.

None is not equivalent to 0 (zero) or False.

```
$ >>> x = None
$ >>> x == 0
$ False
$ >>> x == False
$ False
```

Type Conversion

Type conversion is the process of creating a new object from an existing one, where the newly created object is of a different type. It is not possible, as the name suggests, to convert or change the type of an object (see the Strong Typing section on p.4).

The examples below demonstrate type conversion using literals, but the results would be the same if we were to substitute variables assigned the same values.

To bool

Any object may be converted to a Boolean using the bool built-in function.

The following rules apply:

- Zero numbers convert to False
- Empty collections convert to False
- None converts to False

```
$ >>> bool(0)
$ False
$ >>> bool(1)
$ True
$ >>> bool('')
$ False
$ >>> bool('Hello')
$ True
$ >>> bool([])
$ False
$ >>> bool[1, 2, 3])
$ True
$ >>> bool(None)
$ False
```

There a few gotchas of which you should be aware:

- Both 0 (positive zero) and -0 (negative zero) numbers convert to False
- A string with whitespace (a space/tab/carriage return/line feed) is not empty

To int

Either Boolean value, or any string comprising a valid whole number only (positive or negative) may be converted to an integer using the int built-in function.

```
$ >>> int(True)
$ 1
$ >>> int('123')
$ 123
$ >>> int('-1')
$ -1
```

To float

Either Boolean value, any integer, or any string comprising a valid number (positive or negative and including scientific notation) may be converted to a floating point number using the float built-in function.

```
$ >>> float(True)
$ 1.0
$ >>> float(123)
$ 123.0
$ >>> float('-1')
$ -1.0
$ >>> float('1.2')
$ 1.2
$ >>> float('1.2e-3')
$ 0.0012
```

To str

Any object may be converted to a string using the str built-in function.

```
$ >>> str(True)
$ 'True'
$ >>> str(123)
$ '123'
$ >>> str([1, 2, 3])
$ '[1, 2, 3]'
$ >>> str(None)
$ 'None'
```

NOTE

The str built-in function is a special case. Whilst the result of converting simple things like numbers is predictable, the same cannot be said for complex objects. See the chapter on Objects and Classes for more information.

To list

Any string, tuple, set, or dict may be converted to a list using the list built-in function.

NOTE

Any iterable object may be converted to a list.

```
$ >>> list('Hello')
$ ['H', 'e', 'l', 'l', 'o']

$ >>> list((1, 2, 3))
$ [1, 2, 3]

$ >>> list({1, 2, 2, 3})
$ [1, 2, 3]

$ >>> list({'a': 1, 'b': 2, 'c': 3})
$ ['a', 'b', 'c']
```

Conversion of a dictionary results in a list composed of the dictionary keys, not its values.

To tuple

Any string, list, set, or dict may be converted to a tuple using the tuple built-in function.

NOTE

Any iterable object may be converted to a tuple.

```
$ >>> tuple('Hello')
$ ('H', 'e', 'l', 'l', 'o')

$ >>> tuple([1, 2, 3])
$ (1, 2, 3)

$ >>> tuple({1, 2, 2, 3})
$ (1, 2, 3)

$ >>> tuple({'a': 1, 'b': 2, 'c': 3})
$ ('a', 'b', 'c')
```

Conversion of a dictionary results in a tuple composed of the dictionary keys, not its values.

To set

Any string, list, tuple, or dict may be converted to a set using the set built-in function.

NOTE

Any iterable object may be converted to a set.

```
$ >>> set('Hello')
$ {'H', 'e', 'l', 'o'}  # note that duplicate values are omitted

$ >>> set([1, 2, 3])
$ {1, 2, 3}

$ >>> set((1, 2, 2, 3))
$ {1, 2, 3}

$ >>> set({'a': 1, 'b': 2, 'c': 3})
$ {'b', 'a', 'c'}
```

Conversion of a dictionary results in a set composed of the dictionary keys, not its values. As neither dictionary nor set is ordered, the keys in the resultant set may not appear in the same order as they do in the dictionary, as is the case here.

To dict

A list of two-tuples may be converted to a dictionary using the dict built-in function.

```
$ >>> dict([('a', 1), ('b', 2)])
$ {'a': 1, 'b': 2}
```

But it not often the case that your data will be organised like this naturally. You're more likely to have two lists which you intend to combine into one dictionary. See the chapter on Dictionaries for more information.

Exercises

Built-in data types

In this exercise you will create a script comprising airline passenger information. You must choose the most appropriate data type for each item of data. In practice, it is unlikely that you will hard-code data into a script in this manner, but the purpose of the exercise is to make you familiar with the built-in data types at your disposal. Don't forget about Python's naming conventions.

1. Create a script named ch3_data_types.py.
2. Declare and assign one variable for each of the items of data tabled below.

Description	Value
ID	1234
First name	John
Last name	Doe
Checked bags	False
Visited countries	Latvia, Guyana, Yemen, Uzbekistan
Flight	LGW to CDG
Flight time	1.25

3. Print each variable and its data type to the console.

CHAPTER 4

Operators and Expressions

Introduction

Operators don't vary much between modern languages, and Python is no exception. In this chapter we table the various operators for reference and provide detailed information only where we think it's needed.

For completeness - an operator is a special symbol that can be applied to one or more values, referred to as operands, and that either changes the operand (unary operator) or evaluates to a value (binary operator). An expression is a statement that yields a new value. That value is often assigned to a variable.

```
new_value = num + 1
```

In this case, the + (plus symbol) is the operator, the variable num and the integer literal 1 are the operands, and the part of the statement on the right of the assignment operator is the expression.

Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (returns the remainder)
**	Exponentiation (to the power)
//	Floor division

Division

A division operation always yields a float, even if the operands are of type int.

```
$ >>> 9 / 2
$ 4.5
```

The floor division operator yields an int.

```
$ >>> 9 // 2
$ 4
```

Implicit type conversion (promotion)

If an arithmetic expression comprises one int operand and one float operand, then the int value is implicitly converted to a float (promoted).

```
$ >>> 9 + 2.0
$ 11.0
```

Non-numeric arithmetic

Sounds silly doesn't it. But in Python, arithmetic operators can be applied to anything.

```
$ >>> "ha" * 3
$ 'hahaha'

$ >>> [1, 2, 3] + [4, 5, 6]
$ [1, 2, 3, 4, 5, 6]
```

For more information about how this is achieved, and how you can exploit it in your own custom data types, see the chapter on Objects and Classes.

Assignment Operators

Operator	Description
=	Assignment
+=	Compound assignment (addition)
-=	Compound assignment (subtraction)
*=	Compound assignment (multiplication)
/=	Compound assignment (division)
%=	Compound assignment (modulus)
**=	Compound assignment (exponentiation)
//=	Compound assignment (floor divition)

NOTE

Python includes a set of bitwise assignment operators too but they're beyond the scope of this course.

Multiple assignment

Many variables may be assigned values on one line. The following…

```
$ >>> x = 1
$ >>> y = 2
$ >>> z = 3
```

…may be rewritten as:

```
$ >>> x, y, z = 1, 2, 3
```

Assignment operator chaining

Assignment expressions may be chained together. The following…

```
$ >>> x = 1
$ >>> y = 1
```

…may be re-written as:

```
$ >>> x = y = 1
```

As always, the expression on the right of the assignment operator is evaluated first. In this case, y = 1 is evaluated first, then x = y.

Increment and decrement

If you're coming to Python from another language you may be surprised to learn that Python does not have increment/decrement operators. One of the main reasons for this is that, due to the way in which things like iteration is done in Python, increment/decrement operators are rarely needed.

```
$ >>> x = 1
$ >>> x++
$ SyntaxError: invalid syntax
$ >>> x += 1
$ >>> x
$ 2
```

Comparison Operators

Operator	Description
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
is	References the same object as
is not	Does not reference the same object as
in	Is a member of
not in	Is not a member of

Testing for equality (==)

Testing for equality seems like a simple thing…

```
$ >>> 2 == 2
$ True
```

but what about…

```
$ >>> x == y
```

What are we comparing? The objects referenced by the variables x and y, or the references/pointers themselves? The answer is that it *should* be the former, that is, testing for equality should compare the objects referenced by the variables x and y. And indeed this is the case for all the data types we've discussed thus far – bool, int, float, complex, str, list, tuple, set, and dict, but it may not be the case for custom data types. See the chapter on Objects and Classes for more information.

```
$ >>> "Hello" == "Hello"
$ True
$ >>> [1, 2, 3] == [1, 2, 3]
$ True
$ >>> {'a': 1, 'b': 2} == {'a': 1, 'b': 2}
$ True
$ >>> [1, 2, 3] == [3, 2, 1]
$ False
$ >>> {'a': 1, 'b': 2} == {'b': 2, 'a': 1}
$ True
```

The last two of the examples above may seem a little odd. The two lists are not equivalent because lists are ordered, and so the elements at each index are compared in turn. The two dictionaries are equivalent because dictionaries are unordered, and so the test only serves to determine if the content of each dictionary is the same.

NOTE

Further to the points made above, tuples are ordered, like lists, so…

```
$ >>> (1, 2, 3) == (3, 2, 1)
$ False
```

But sets are unordered, so…

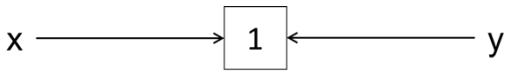
```
$ >>> {1, 2, 3} == {3, 2, 1}
$ True
```

Testing for identity (is)

The is operator should not be confused with the equality operator. The is operator is used to compare references/pointers. That is, do these two variables reference/point to the same object or not?

```
$ >>> x = 1
$ >>> y = 1
$ >>> x == y
$ True
$ >>> x is y
$ True
```

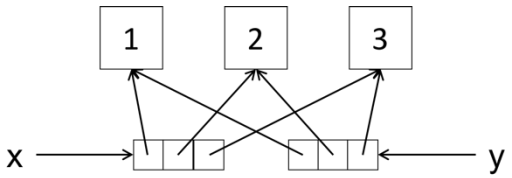
The integers referenced by x and y are equivalent and, given that there can only ever be one integer literal with a value of 1 (one), the two variables reference the same object.



But what about lists?

```
$ >>> x = [1, 2, 3]
$ >>> y = [1, 2, 3]
$ >>> x == y
$ True
$ >>> x is y
$ False
```

The lists referenced by x and y are equivalent (the contents are the same) but are separate objects. That is, x does not reference the same object as does y.



NOTE

Despite the fact that a string is a collection, like a list, with regards equality, strings behave like numbers.

```
$ >>> x = "Hello"
$ >>> y = "Hello"
$ >>> x == y
$ True
$ >>> x is y
$ True
```

That is, there can only ever be one string literal with a value of "Hello".

Testing for membership (in)

The in operator is used to test for the presence of a value in a collection.

```
$ >>> 'e' in 'Hello'
$ True
$ >>> 2 in [1, 2, 3]
$ True
```

In the same way that the conversion of a dictionary to a list results in a list composed of the dictionary keys, and not its values, testing a value for membership in a dictionary means testing that the value is present in the dictionary's set of keys.

```
$ >>> 'b' in {'a': 1, 'b': 2}
$ True
$ >>> 1 in {'a': 1, 'b': 2}
$ False
```

NOTE

It is, of course, possible to test that a value is present in a dictionary's set of values. See the chapter on Dictionaries for more information.

Logical Operators

Operator	Description
and	Boolean AND: yields True if each expression is True
or	Boolean OR: yields True if either expression is True
not	Boolean NOT: inverts the value

Comparison operator chaining

Python allows for the chaining of comparison operators. Consider the following:

```
$ >>> age >= 18 and age <= 35
```

This is a pretty typical logical expression, but can be rewritten as…

```
$ >>> 18 <= age <= 35
```

Logical operators with non-Boolean values

We know that any value may be converted to a Boolean, and so we might assume that logical operators can be applied to any value.

```
$ >>> 1 and 0
$ 0
```

You might have expected the result to be False, given that 0 (zero) is False when converted to a Boolean, and 1 (one) is likewise True. But instead the result is 0 (zero).

The following rules are applied:

- `A and B`: if `A` is false, then the result is `A`, else the result is `B`
- `A or B`: if `A` is true, then the result is `A`, else the result is `B`

To ensure the result of a compound expression is a Boolean, make sure that each constituent expression is a Boolean.

```
$ >>> bool(1) and bool(0)
$ False
```


Operator Precedence

The full table of operator precedence is provided below for reference, but most of the time it will suffice to know that any expression enclosed by the grouping operator (parentheses) takes precedence over everything else.

```
$ >>> 3 + 2 * 4  # multiplication takes precedence over addition
$ 11
$ >>> (3 + 2) * 4
$ 20
```

Operator Precedence (from highest to lowest)	
(expressions…), [expressions…], {expressions…}, {key: value…}	Grouping, tuple, list, set, dictioary
x(args…), x[index:index], x[index], x.attribute	Function call, slicing, subscription, attribute reference
**	Eponentiation
+x, -x , ~x	Positive, negative, bitwise not
*, / , %, //	Multiplication, division, modulus, floor division
+, -	Addition, subtraction
<<, >>	Shift left, shift right
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
<, <=, ==, !=, =>, >, is, is not, in, not in	Less than, less than or equal to, equal to, not equal to, greather than or equal to, greater than, is, is not, in, not in
not x	Boolean NOT
and	Boolean AND
or	Boolean OR
if else	Conditional expression
lamda	Lambda expression

Exercises

Operators and expressions

This exercise does not require you to create a script, though you may want to execute small snippets on the command line using Python's interactive mode.

Complete the table below by writing the result of each expression.

Arithmetic	**Result**
7 % 3	
7 ** 3	
7 // 3	
7 + 3.5	
["ho"] * 3	
"The meaning of life is " + 42	
Unary	**Result**
not False	
not 2	
not ""	
not [1]	
Comparison	**Result**
"a" > "b"	
[1, 2, 3] < [1, 1, 4]	
{"a", "b", "c"} == {"b", "a", "c"}	
3 == 3	
3 is 3	
[] == []	
[] is []	
"o" in "bobble"	
"john" in {"name": "john"}	
Logical	**Result**
True and bool(0)	
bool([]) or bool("Hello")	

Appendix

Bitwise operators are used only in cases where data is to be manipulated at the bit level and so is not something all developers will have cause to use.

Bitwise operators

Operator	Description
&	Bitwise AND: 1 if both bits are 1
|	Bitwise OR: 1 if either bit is 1
^	Bitwise XOR: 1 if one bit is 1 and the other is 0
<<	Shift left: shift the pattern to the left by n 0 bits
>>	Shift right: shift the pattern to the right by n 0 bits

Some examples:

```
$ >>> a = 0b0101  # decimal 5
$ >>> b = 0b0110  # decimal 6

$ >>> a & b
$ 4               # 0b0100

$ >>> a | b
$ 7               # 0b0111

$ >>> a ^ b
$ 3               # 0b0011

$ >>> a << 1
$ 10              # 0b1010

$ >>> a >> 1
$ 2               # 0b0010
```

CHAPTER 5

Conditions and Loops

Introduction

Conditional and loop statements typically comprise one or more blocks of code. As described in chapter 2, a Python block must be indented. The start of a block is indicated by the presence of a colon (:), and each statement within the block must be indented using the same amount of whitespace (the convention is to use four spaces). The block ends when the indenting ends.

Conditional Statements

Conditional statements are somewhat easier to learn in Python than they are in other languages, because there is only one type – the if statement.

The if statement

An if statement takes the following form:

```
if <expression>:
    <statement(s)>
```

If the expression evaluates to True, the statement(s) in the block that follow are executed.

```
$ >>> x = 2
$ >>> if x > 1:
$ ...     print("x is greater than 1")
$ ...
$ x is greater than 1
```

The expression need not be a Boolean expression. As any value can be converted to a Boolean, any expression that yields a value may be used.

```
$ >>> x = 2
$ >>> if x:
$ ...     print("bool(x) is True")
$ ...
$ bool(x) is True
```

NOTE

You might recall from chapter 3 that any non-zero, non-empty value evaluates to True when passed to the built-in bool function.

To demonstrate the importance of consistent indenting, consider the following:

```
$ >>> x = 2
$ >>> if x > 1:
$ ...     print("Hello")
$ ...    print("x is greater than 1")
$ ...
$ IndentationError: unindent does not match any outer indentation level
```

We expect both print statements to be executed. The problem is that the first print statement is indented using four spaces, while the second print statement is indented using three spaces.

Recall that the block ends when the indenting ends. There is no special symbol to indicate the end of the block.

```
$ >>> x = 2
$ >>> if x > 1:
$ ...     print("x is greater than 1")
$ ...
$ >>> print("Outside the block/the if statement")
```

NOTE

The example above can't be executed in one go in interactive mode. By its very nature a REPL shell can only evaluate one statement at a time (unless that statement involves the invoking of a function in which multiple statements are executed).

The else and elif clauses

A two branch conditional statement is achieved with the addition of the else clause to an if statement. A two branch if statement takes the following form:

```
if <expression>:
    <statement(s)>
else:
    <statement(s)>
```

If the expression evaluates to True, the statement(s) in the first block are executed. If the expression evaluates to False, the first block is skipped and the statement(s) in the second block are executed.

```
$ >>> x = 2
$ >>> if x > 2:
$ ...     print("x is greater than 2")
$ ... else:
$ ...     print("x is not greater than 2")
$ ...
$ x is not greater than 2
```

A three or more branch conditional statement is achieved with the addition of the elif clause to an if else statement. A three branch if statement takes the following form:

```
if <expression>:
    <statement(s)>
elif <expression>:
    <statement(s)>
else:
    <statement(s)>
```

Note that two expressions are required in this case.

```
$ >>> x = 2
$ >>> if x > 2:
$ ...     print("x is greater than 2")
$ ... elif x < 2:
$ ...     print("x is less than 2")
$ ... else:
$ ...     print("x is equal to 2")
$ ...
$ x is equal to 2
```

Only the first branch associated with an expression that evaluates to True is executed. That is, following the execution of a branch, the interpreter steps out of the if statement without evaluating any more expressions.

```
$ >>> x = 2
$ >>> if x > 1:
$ ...     print("x is greater than 1")
$ ... elif x != 1:
$ ...     print("x is not equal to 1")
$ ... else:
$ ...     print("None of the expressions evaluated to True")
$ ...
$ x is greater than 1
```

In this case, only the first branch is executed despite the second expression also evaluating to True.

Nested statements

Conditional statements can be nested, for example

```
>>> x = 12
>>> if x % 2 == 0:
...     print("x is even.")
...     if x % 3 == 0:
...             print("It is also divisible by 3")
...     else:
...             print("But, not divisible by three")
... else:
...     print("x is odd.")
...
>>> x is even.
>>> It is also divisible by 3
```

Note that nested conditions can be difficult to read and often can be avoided in favour of an unnested statement.

One-liners

One branch if statements may be written on one line…

```
$ >>> if x > 1: print("x is greater than 1")
```

…even when the block comprises more than one statement:

```
$ >>> if x > 1: print("Hello"); print("x is greater than 1")
```

NOTE

The latter of the two examples above is, in the author's view, bad practice. It's ambiguous and hard to read. Unless you're intimately familiar with the language, it's not obvious that the second print statement is a part of the preceding if statement.

Conditional expression

A conditional expression is one in which either of two values is assigned to a variable depending on the result of a boolean expression. It takes the following form:

```
$ >>> x = <value1> if <expression> else <value2>
```

Consider the following example:

```
$ >>> action = "Get up" if hour >= 7 else "Go back to sleep"
```

NOTE

For those of you familiar with one or more other languages, this may appear similar to what you know as the ternary operator. It is, indeed, the Python equivalent.

Iterative Statements

There are two types of iterative statement in Python – while and for.

The while statement

A while statement takes the following form:

```
while <expression>:
    <statement(s)>
```

If the expression evaluates to True, the statement(s) in the block that follow are executed. The cycle is repeated until the expression evaluates to False.

```
$ >>> x = 1
$ >>> while x <= 3:
$ ...     print(x)
$ ...     x += 1
$ ...
$ 1
$ 2
$ 3
```

The for statement

If you've written code in another language, you'll almost certainly be familiar with a for loop of the form:

```
for (<init_statement>; <expression>; <update_statement>) {
    ...
}
```

There is no such statement in Python. Python for loops are used exclusively to iterate over collections (or iterables, to be precise).

A for statement takes the following form:

```
for <element> in <collection>:
    <statement(s)>
```

The block is executed once for each element in the collection.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> for name in names:
$ ...     print(name)
$ ...
$ David
$ Sarah
$ Tom
$ Jane
```

NOTE

The variable to which each subsequent element is assigned (name in this case) need not have been declared outside of the loop.

The range function

What if you want to perform some task *n* times but you don't have a collection? You could use a while loop, but doing so is considerably slower than a for loop that uses the built-in range function.

The range function returns a range of numbers and can be invoked in a variety of ways.

```
$ >>> my_range = range(5)        # start = 0; stop = 5; step = 1
$ >>> list(my_range)
$ [0, 1, 2, 3, 4]

$ >>> my_range = range(1, 5)     # start = 1; stop = 5; step = 1
$ >>> list(my_range)
$ [1, 2, 3, 4]

$ >>> my_range = range(1, 10, 2) # start = 1; stop = 10; step = 2
$ >>> list(my_range)
$ [1, 3, 5, 7, 9]
```

You'll have noticed that we're converting the returned value to a list. That's because the range function returns a range type object. You needn't convert it to a list when using it in a for loop though.

```
$ >>> for number in range(5):
$ ...     print(number)
$ ...
$ 0
$ 1
$ 2
$ 3
$ 4
```

NOTE

Since Python version 3, the range function generates each number on demand. This means it consumes far less memory than might otherwise be the case. If you're using Python version 2, you should consider using the xrange function instead.

Break

Like other languages, the break statement is used to terminate a loop prematurely.

```
$ >>> while(True):
$ ...     option = input("More? (y/n): ")
$ ...     if option == "n":
$ ...         print("Bye")
$ ...         break
$ ...     print("Some more...")
$ ...
$ More? (y/n): y
$ Some more...
$ More? (y/n): y
$ Some more...
$ More? (y/n): n
$ Bye
```

In this case, the infinite loop is terminated if the user enters the letter n at the prompt.

Continue

Also like other languages, the continue statement is used to jump to the next iteration, skipping one or more statements in the process.

```
$ >>> for number in range(5):
$ ...     if number == 3:
$ ...         continue
$ ...     print(number)
$ ...
$ 1
$ 2
$ 4
$ 5
```

The optional else clause

The else clause may be added to an iterative statement where break is also present. The else block will only be executed if the loop exits cleanly, that is, without terminating prematurely.

```
$ >>> for line in file:
$ ...     if line.startswith("ERROR"):
$ ...         print("The file has one or more errors")
$ ...         break
$ ...     print(line)
$ ... else:
$ ...     print("File processed with no errors")
$ ...
```

Exercises

Conditions

In this exercise you will create a script that prompts the user to input his/her birth year, and then prints to the console the generation of which he/she is a member, e.g. Baby Boomer, Generation X, etc.

1. Create a script named ch5_conditions.py.
2. Prompt the user to input his/her birth year and capture it in a variable named birth_year.
3. Convert birth_year to an int.
4. If the user was born between 1946 and 1965, print Baby Boomer to the console.
5. If the user was born between 1966 and 1976, print Generation X to the console.
6. If the user was born between 1977 and 1994, print Millennial to the console.
7. If the user was born in 1995 or beyond, print Generation Z to the console.

Loops

In this exercise you will create a number guess game. It will prompt the user to guess the magic number between 1 and 10. If the user guesses correctly, it will print a winner message and exit. If the user guesses incorrectly, he/she will be prompted again. The user will be given three go's after which, if he/she has not guessed correctly the script will print a loser message.

1. Create a script named ch5_loops.py.
2. Import the random module as follows:
   ```
   import random
   ```
3. Declare and assign a variable as follows:
   ```
   magic_number = random.randint(1, 10)
   ```
4. Code a loop that iterates three times.
5. Inside the loop…
   a. Prompt the user to input a guess and capture it in a variable named user_guess.
   b. If the user's guess equals the magic number, print a winner message to the console and break out of the loop.
   c. If the user's guess does not equal the magic number, print an appropriate message, e.g. too low or too high.
6. If the loop exits normally, the user has not guessed correctly so print a suitable consolation message to the console.

CHAPTER 6

Functions

Introduction

A function is a named and callable group of statements. It is the principle mechanism for creating re-usable code. A function declared in one script may be called (resulting in the execution of its statements) in any number of other scripts.

NOTE

A method is a function that is also a property of an object, though the terms function and method are often used interchangeably.

A function declaration takes the following form:

```
def <function_name>(*<parameter(s)>):
    <statement(s)>
    *return <return_value>
```

Both def and return are keywords, and each of the parameter list and return statement is optional. Consider the following example function declaration:

```
$ >>> def calc_circumference(radius):
$ ...     return 2 * radius * 3.14
$ ...
```

A function call/invocation takes the following form:

```
<function_name>(*<argument(s)>)
```

Consider the following example function call/invocation:

```
$ >>> circumference = calc_circumference(4)
$ >>> circumference
$ 25.12
```

Parameters

A parameter is a variable that stores a reference to one of the objects passed to the function when it is called. Consider again the previous example:

```
$ >>> def calc_circumference(radius):
$ ...     return 2 * radius * 3.14
$ ...
$ >>> circumference = calc_circumference(4)
$ >>> circumference
$ 25.12
```

When the function is called a reference to the integer literal 4 (the argument) is copied into the parameter named radius.

NOTE

An argument is an object reference passed to a function when it is called.

A function may be declared with any number of parameters.

```
$ >>> def calc_hypotenuse(a, b):
$ ...     a_squared = a ** 2
$ ...     b_squared = b ** 2
$ ...     sum_of_squares = a_squared + b_squared
$ ...     return sum_of_squares ** 0.5
$ ...
```

Default values

Parameters are, by default, required. That is, when you call a function you must pass one argument for each specified parameter.

```
$ >>> hypotenuse = calc_hypotenuse(3)
$ TypeError: missing 1 required positional argument: 'b'
```

Sometimes it is desirable to declare a function with optional parameters. This is achieved by providing said parameters with default values.

```
$ >>> def open_account(name, balance=0):
$ ...     print("Account name:", name)
$ ...     print("Account balance:", balance)
$ ...
```

The parameter, balance, is assigned a default value of 0 (zero). The open_account function may now be called in one of two different ways.

```
$ >>> open_account("Smith", 100)
$ Account name: Smith
$ Account balance: 100
```



```
$ >>> open_account("Smith")
$ Account name: Smith
$ Account balance: 0
```

Named (keyword) arguments

Arguments are, by default, positional. That is, when you call a function you must pass arguments in the order specified by the parameter list. Failure to do so will result in spurious results.

```
$ >>> def open_account(name, balance=0):
$ ...     print("Account name:", name)
$ ...     print("Account balance:", balance)
$ ...
$ >>> open_account(100, "Smith")
$ Account name: 100
$ Account balance: Smith
```

You can, however, pass arguments in any order provided each argument is named.

```
$ >>> open_account(balance=100, name="Smith")
$ Account name: Smith
$ Account balance: 100
```

In this case each argument is prefixed with the corresponding parameter name and the assignment operator.

The Return Value

A function may be declared to return a value. This value may be a variable...

```
$ >>> def sqrt(number):
$ ...     result = number ** 0.5
$ ...     return result
$ ...
```

…or a literal:

```
$ >>> def sqrt(number):
$ ...     return number ** 0.5
```

Either way, what is returned is a copy of the reference to said object. When the function is called, the return value (object reference) may be assigned to a variable.

```
$ >>> sqrt_of_25 = sqrt(25)
```

Output as input

Sometimes the value returned by a function serves as the input to yet another function. For example, the following two lines…

```
$ >>> sqrt_of_25 = sqrt(25)
$ >>> print("The square root of 25 is", sqrt_of_25)
```

…may be refactored as one:

```
$ >>> print("The square root of 25 is", sqrt(25))
```

The value returned by the sqrt function serves as input to the print function.

NOTE

If you find yourself declaring a variable that is used only once, it is likely that the variable is not needed, as is the case in the first of the examples above.

Multiple return values

Earlier we said that a function may be declared to return *a* value, implying that only one value may be returned. This is true, though the following is valid:

```
$ >>> def calc_min_max(numbers):
$ ...     min, max = 100, 0
$ ...     for number in numbers:
$ ...         if number < min: min = number
$ ...         if number > max: max = number
$ ...     return min, max
$ ...
```


It appears as though the calc_min_max function is returning two values – min and max. In fact, the two values are packed into a tuple, a reference to which is then returned.

```
$ >>> min_max = calc_min_max([31, 16, 25, 83, 40, 59, 4, 66])
$ >>> min_max[0]
$ 4
$ >>> min_max[1]
$ 83
```

When you call a function that returns a tuple, the tuple's values may be unpacked into separate variables, making it appear as though the function does indeed return more than one value.

```
$ >>> min, max = calc_min_max([31, 16, 25, 83, 40, 59, 4, 66])
$ >>> min
$ 4
$ >>> max
$ 83
```

NOTE

Objects of type str, list, set, and dict may also be unpacked in this way, e.g.:

```
$ >>> name = "Tom"
$ >>> first, second, third = name
$ >>> second
$ 'o'
```

None returned

If a function does not explicitly return something, then None is returned.

```
>>> def sqrt(number):
...     result = number**0.5
...
>>> result = sqrt(25)
>>> result
>>> result == None
True
>>> type(result)
<class 'NoneType'>
>>>
```

None is a singleton with its own type.

Variable Scope

The scope of a variable determines its longevity and the points within your script at which it's accessible.

Local

A variable declared inside a function is said to be local. It exists in memory only whilst the function is executing or until you remove it. A local variable is accessible only within the function in which it is declared.

```
$ >>> def calc_circumference(radius):
$ ...     PI = 3.14
$ ...     return 2 * radius * PI
$ ...
$ print(PI)
$ NameError: name 'PI' is not defined
```

NOTE

Parameters are local too, e.g. radius in the example above.

Global

A variable declared outside of any function is said to be global. It exists in memory for as long as the program is executing or until you remove it. A global variable is accessible by all of the script's functions.

```
$ >>> greeting = "Hello"
$ >>> def greet(name):
$ ...     print(greeting, name)
$ ...
$ >>> greet("David")
$ Hello David
```

But what about this...

```
$ >>> greeting = "Hello"
$ >>> def greet(name):
$ ...     greeting += name  # attempt to change greeting
$ ...     print(greeting)
$ ...
$ >>> greet("David")
$ UnboundLocalError: local variable 'greeting' ...
```

The error suggests that greeting is a local, and not a global variable. Why? A variable that is assigned a value inside a function is assumed to be local to that function. In this case we are attempting change the *local* variable, greeting, before it is assigned an initial value.

Consider a variation on the above.

```
$ >>> greeting = "Hello"
$ >>> def greet(name):
$ ...     greeting = "Hi"  # attempt to change greeting?
$ ...     print(greeting, name)
$ ...
$ >>> greet("David")
$ Hi David
$ >>> print(greeting)
$ Hello
```

It would appear that the global variable, greeting, is changed inside the greet function. In fact, the variable named greeting inside the greet function is local, and is unrelated to the global variable that shares its name. The global variable is said to be *shadowed* by the local variable. We also say the local variable *masks* the global variable.

If you intend to assign a value to a global variable within a function, then you must declare it as being global using the global keyword.

```
$ >>> greeting = "Hello"
$ >>> def greet(name):
$ ...     global greeting  # declare greeting as global
$ ...     greeting = "Hi"
$ ...     print(greeting, name)
$ ...
$ >>> greet("David")
$ Hi David
$ >>> print(greeting)
$ Hi
```

THE RULES

1. **A variable that is only referenced inside a function is assumed to be global.**
2. **A variable that is assigned a value inside a function is assumed to be local, unless it is explicitly declared as being global using the global keyword.**

Args and Kwargs

*args

A parameter prefixed with an * (asterisk) indicates that a variable number of arguments (zero or more) is expected. This parameter is typically named args though it need not be; the asterisk is the important part. Given the interpreter cannot know how many arguments will be passed, a parameter of this type must be positioned in the parameter list after all required parameters.

```
$ >>> def subscribe(name, *hobbies):
$ ...     print("Your name is", name)
$ ...     if len(hobbies) > 0:
$ ...         print("Your hobbies include:")
$ ...         for hobby in hobbies:
$ ...             print("-", hobby)
$ ...
$ >>> subscribe("David", "Reading", "Walking")
$ Your name is David
$ Your hobbies include:
$ - Reading
$ - Walking
```

In this case the arguments `"Reading"`, and `"Walking"` are packed into a tuple, a reference to which is then copied into the parameter named hobbies.

The same thing might have been achieved by coding the function to expect a list/tuple. Which to choose? It depends on the circumstances. *args implies an optional collection of things, where a regular parameter implies a collection is required even if it's empty. If the user of your function is unlikely to have a collection to pass to your function, and the number of values is small, then you should probably use *args. If, on the other hand, the user of your function is likely to have a collection to pass to your function, and the number of values is large, then you should probably use a regular parameter.

What if you have a list or tuple that you want to pass to a function declared with *args?

```
$ >>> my_hobbies = ["Reading", "Walking"]
$ >>> subscribe("David", my_hobbies)
$ Your name is David
$ Your hobbies include:
$ - [Reading, Walking]
```

The subscribe function expects a variable number of hobby arguments which it will pack into a tuple. Instead, it gets a pre-packed list of hobbies. Nonetheless it packs the list of hobbies into a tuple, and the result is a tuple comprising a list of hobbies.

You can unpack an existing collection so as to pass it where *args is expected by prefixing the argument with an asterisk.

```
$ >>> my_hobbies = ["Reading", "Walking"]
$ >>> subscribe("David", *my_hobbies)
$ Your name is David
$ Your hobbies include:
$ - Reading
$ - Walking
```

**kwargs

A parameter prefixed with ** (double asterisk) indicates that a variable number of named (keyword) arguments (zero or more) is expected. This parameter is typically named kwargs though it need not be; the double asterisk is the important part. A parameter of this type must be positioned in the parameter list after all required arguments and the *args parameter if present.

```
$ >>> def subscribe(name, *hobbies, **other_details):
$ ...     print("Your name is", name)
$ ...     if len(hobbies) > 0:
$ ...         print("Your hobbies include:")
$ ...         for hobby in hobbies:
$ ...             print("-", hobby)
$ ...     if len(other_details) > 0:
$ ...         print("Other details:")
$ ...         for key, value in other_details.items():
$ ...             print("-", key, ":", value)
$ ...
$ >>> subscribe("David", "Reading", "Walking", age=30, county="Surrey")
$ Your name is David
$ Your hobbies include:
$ - Reading
$ - Walking
$ Other details:
$ - age : 30
$ - county : Surrey
```

In this case the named (keyword) arguments `age=30` and `county="Surrey"` are packed into a dictionary, a reference to which is then copied into the parameter named other_details.

NOTE

Each dictionary object has a method named items that returns an iterable collection of tuples, where each tuple has two elements – a key and a value.

The same thing might have been achieved by coding the function to expect a dictionary. The guidelines that apply to *args apply also to **kwargs.

What if you have a dictionary that you want to pass to a function declared with **kwargs? As with *args, you can unpack an existing dictionary so as to pass it where **kwargs is expected by prefixing the argument with a double asterisk.

```
$ >>> my_details = {"age": 30, "county": "Surrey"}
$ >>> subscribe("David", "Reading", "Walking", **my_details)
$ Your name is David
$ Your hobbies include:
$ - Reading
$ - Walking
$ Other details:
$ - age : 30
$ - county : Surrey
```


The pass Keyword

The pass keyword represents a null operation. That is, when it is executed by the interpreter nothing happens. It is useful where one or more statements is required syntactically but none are to be executed. The pass statement is often used as a placeholder in functions that do not, as yet, contain any code.

```
$ >>> def calc_area(width, height, depth):
$ ...     pass
$ ...
```

The pass statement is not limited to use within functions. It might be used, for example, in a conditional or loop statement.

```
$ >>> if name == None or len(name) == 0:
$ ...     pass

$ >>> for account in accounts:
$ ...     pass
```

Recursive Functions

Recursion is when a function calls itself. Recursive functions typically implement algorithms that at their core have a repetitive, usually simple, step. In code, this step includes a function call to itself. Of course, a practical recursion cannot be indefinite and requires a *base* case.

Let us look at an example function that calculates the factorial $n!$ of a *natural* number $n \geq 0$. The n! = n(n-1)(n-1) …. 1. Now, note that n! = n (n-1)! and that we have a *base* case when n=1. Let us code the function

```
>> def factorial(n):
...     if n == 1: # base case
...             return 1
...     return n * factorial(n-1) # recusive call
...
>>> factorial(1) # 1!
1
>>> factorial(5) # 5! = 5 x 4 x 3 x 2 x 1
120
```


Exercises

Functions

In this exercise you will create a module and a script. The module will have 2 functions. The first function will create and return a dating profile based on age and a list of hobbies. The second function will accept two dating profiles and on comparing them will return a number representing the quality of the match. The script will call the first function in your module to create two dating profiles, and then call the second function to determine the quality of the match. The number returned by the second function will be printed to the console.

1. Create a module named ch6_profile_matcher.py.
2. Declare a function named build_profile that builds and returns a dating profile. Note that a dating profile will be a dictionary that looks something like this:

```
p1 = {
    "age": 25,
    "hobbies": ["reading", "eating out", "travelling"]

}
```

3. Inside the build_profile function…
   a. Declare a variable named profile and assign it a dictionary as follows:

```
profile = {
    "age": 0,
    "hobbies": []
}
```

   b. Prompt the user to input the age of the profile and capture it in a variable named p_age.
   c. Convert p_age to an int.
   d. Assign p_age to profile["age"].
   e. Code an infinite loop as follows…
   f. Prompt the user to input a hobby and capture it in a variable named hobby.
   g. Append the hobby to profile's list of hobbies (more about appending to lists later in the course).

```
profile["hobbies"].append(hobby)
```

   h. Prompt the user to input y to input more hobbies, or n to stop, and capture it in a variable named more.
   i. If more is n, break out of the loop.
4. Finish the function by adding a return statement to return the profile dictionary.

5. Declare a function named match that will accepts two dating profiles .

6. Inside the match function…
   a. Declare a variable named match_quality and assign it 0 (zero).
   b. If the first profile's age is within five years of the second profile's age, increment the match_quality variable by one.
   c. For each hobby in the first profile's list of hobbies, if there is a matching one in the second profile's list of hobbies, increment the match_quality variable by one.
   d. Finish the function by adding a return statement to return match_quality.
7. Create a script named ch6_functions.py.
8. Import the build_profile and match function from the ch6_profile_matcher module as follows:

```
from ch6_profile_matcher import build_profile, match
```

9. Call the build_profile function and assign the result to a variable called profile1.

10. Call the build_profile function again and assign the result to a variable called profile2.

11. Call the match function, passing profile1 and profile2 as arguments, and capture the return value in a variable named match_quality.

12. Print the match_quality variable to the console.

On execution of ch6_functions.py the output will be as or similar to that shown below:

```
Create 1st profile
Please enter age: 25
Enter a hobby: Reading
More hobbies? (y/n):y
Enter a hobby: Cinema
More hobbies? (y/n):n
Create 2nd profile
Please enter age: 29
Enter a hobby: Reading
More hobbies? (y/n):y
Enter a hobby: Walking
More hobbies? (y/n):n

Match Quality: 2
```

CHAPTER 7

Objects and Classes

Introduction to Objects

Everything in Python is an object. But what is an object, exactly? An object is a collection of related data and functions. An object is the means of grouping stuff that belongs together. Take, for example, a bank account. A bank account is comprised of data including the account number, name, and balance. In Python, as in other object oriented languages, the functions associated with the data (deposit and withdraw, in this case) are wrapped up in the object along with the data.

NOTE

Technically, an object's functions (its methods) do not inhabit the same memory space as does the object's data. The methods associated with an object need not be duplicated for each object of a given type. It would be more precise to say that each object has access to a set of related methods.

Attributes and the Dot Notation

An object's data and functions are its attributes. Function attributes are referred to as methods. A bank account object might have three data attributes: number; name; and balance, and two methods: deposit and withdraw.

An object's attributes are accessed using the dot notation.

```
<object>.<attribute>
```

The thing on the left of the (.) dot references the object, while the thing on the right of the dot references the attribute.

Consider the following:

```
$ >>> my_complex = 1+2j
$ >>> my_complex.real
$ 1.0
$ >>> my_complex.imag
$ 2.0
```

The complex object referenced by the variable, my_complex, has data attributes – real and imag – the values of which may be accessed using the dot notation.

Consider the following:

```
$ >>> my_string = "Hello"
$ >>> my_string_upper = my_string.upper()
$ >>> my_string_upper
$ 'HELLO'
```

The string object referenced by the variable, my_string, has many methods, each of which may be accessed using the dot notation. In this case the the method, upper, returns an upper case version of "Hello".

The dir function

How is one expected to know what attributes are available on a given object? The built-in dir function expects an object reference and returns a list of attributes.

```
$ >>> my_string = "Hello"
$ >>> for attribute in dir(my_string):
$ ...     print(attribute)
$ ...
$ __add__
$ __class__
$ __contains__
```

And the list goes on.

Dunder attributes

When you list the attributes of a given object you will typically see a large number whose name is prefixed and suffixed with two underscores. These are referred to as dunder (double underscore) attributes.

NOTE

Dunder methods are sometimes referred to as magic or special methods.

Dunder methods are not intended to be called directly. Take, for example, the dunder method __bool__, which is available to all objects of the built-in types, e.g. int, float, str etc. Now consider the following:

```
$ >>> my_number = 42
$ >>> bool(my_number)
$ True
```

The built-in bool function is used to perform type conversion, in this case from int to bool. But what is actually happening here? The built-in bool function calls the __bool__ dunder method on the object referenced by the my_number variable. Therefore, the code above is functionally equivalent to:

```
$ >>> my_number = 42
$ >>> my_number.__bool__()
$ True
```

The first of the two examples here is preferred. As we said, dunder methods are not intended to be called directly.

In Python, dunder methods dictate the behaviour of operators. Listing the attributes of a number type object will yield dunder attributes including __add__ (addition), __mul__ (multiply), __eq__ (equality) etc. This…

```
$ >>> x = 1
$ >>> x + 2
$ 3
```

…is functionally equivalent to:

```
$ >>> x = 1
$ >>> x.__add__(2)
$ 3
```

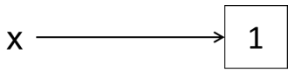
When you come to create your own data types you can override these dunder methods to customise the behaviour. What, for example, should happen when one bank account is added to another?

Mutability

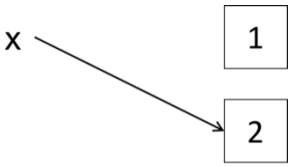
Some objects can be changed; some cannot. An object that can be changed is mutable, while an object that cannot be changed is immutable. Consider the following:

```
$ >>> x = 1
$ >>> x += 1
$ >>> x
$ 2
```

Numbers are immutable. For example, 1 is always 1 and cannot be changed. In the example above, it is the variable, x, that is changed, and not the object. After execution of the first line the variable, x, stores a reference to the object, 1.



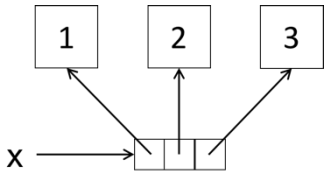
After execution of the second line the variable, x, is changed. It now stores a reference to the object, 2. A new object must be created because numbers are immutable.



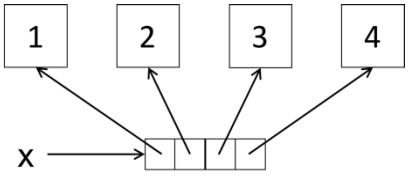
Consider the following:

```
$ >>> x = [1, 2, 3]
$ >>> x.append(4)
$ >>> x
$ [1, 2, 3, 4]
```

Lists are mutable. Elements can be added, updated, and removed. In the example above, it is the object, [1, 2, 3], that is changed, and not the variable, x. After execution of the first line the variable, x, stores a reference to the object, [1, 2, 3].



After execution of the second line the object is changed. It now has four elements. A new object need not be created because lists are mutable.



So, which objects are mutable and which are immutable?

Data Type	Mutable/Immutable
bool	Immutable
int	Immutable
float	Immutable
complex	Immutable
str	Immutable
list	Mutable
tuple	Immutable
set	Mutable
dict	Mutable

When it comes to the built-in data types, only list, set, and dict objects are mutable. Everything else is immutable.

Strings are interesting. Consider the following:

```
$ >>> my_string = "Hello"
$ >>> my_string.upper()
$ 'HELLO'
$ >>> my_string
$ 'Hello'
```

The upper method returns a new, upper case version, of the original string object, but it does not change the original. Why? Because strings are immutable. Like any immutable object, any attempt to change it will result either in a new object, or an error.

```
$ >>> my_tuple = (1, 2, 3)
$ >>> my_tuple.append(4)
$ AttributeError: 'tuple' object has no attribute 'append'
```

tuple objects have no append method because tuples are immutable. But what if we were to do the following?

```
$ >>> my_tuple = (1, 2, 3, 4)
```

Have we changed the object? No. A new tuple object has been created and the variable, my_tuple, has changed. It now references the new tuple object.

The id function

The built-in id function expects an object reference and returns a unique number identifying the object. The id function can be used to determine if two or more variables reference unique objects, or the same object.

```
$ >>> x = 1
$ >>> y = 1
$ >>> id(x)
$ 4297644416
$ >>> id(y)
$ 4297644416
```

In this case both variable, x and y, reference the same object. Why? Because numbers are immutable. There is no need to create two immutable objects with the same value. Instead, the variable, y, is assigned a reference to the existing object.

```
$ >>> x = [1, 2, 3]
$ >>> y = [1, 2, 3]
$ >>> id(x)
$ 4316722120
$ >>> id(y)
$ 4316722056
```

In this case the variables reference different objects, even though the contents of those objects is the same. Why? Because lists are mutable. We expect to be able to modify each of the lists independently of the other.

NOTE

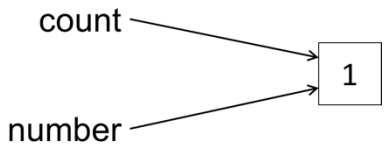
The id assigned each object is unique to the object and to the environment in which the script is executing. You're most unlikely to see the same ids in the classroom/at home/at work etc.

Pass by reference

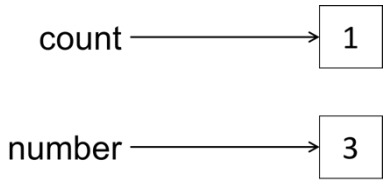
Why does all this mutability stuff matter? Consider the following:

```
$ >>> count = 1
$ >>> def increment(number, amount):
$ ...     # other business logic
$ ...     number += amount
$ ...
$ >>> increment(count, 2)
$ >>> count
```

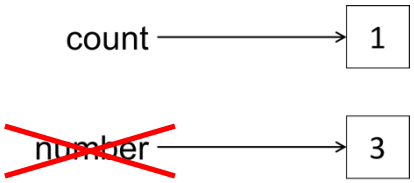
What is the value of count on the final line of the example above? It's 1 (one). When the increment function is called, the value of count (a reference to the object, 1) is copied into the parameter, number. At this point both the count and number variables reference the same immutable number, 1.



The amount, 2, is added to the number. As number references an immutable object, a new object is created with the value, 3.



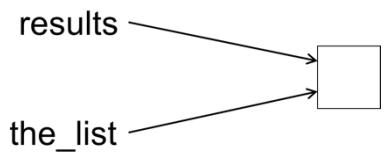
The function ends, and the local variable, number, is deleted from memory.



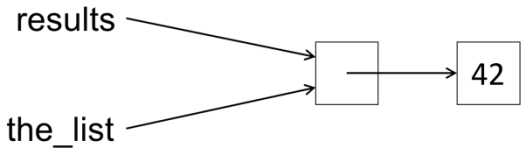
Now consider the following:

```
$ >>> results = []
$ >>> def add_result(the_list, result):
$ ...     # other business logic
$ ...     the_list.append(result)
$ ...
$ >>> add_result(results, 42)
$ results
```

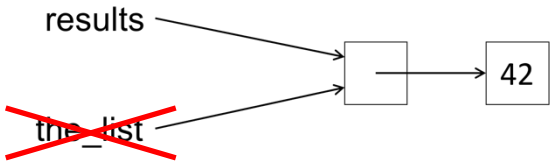
What is the value of results on the final line of the example above? It's [42]. When the add_result function is called, the value of results (a reference to the empty list object) is copied into the parameter, the_list. At this point both the results and the_list variables reference the same mutable list.



The result, 42, is appended to the_list. As the_list references a mutable object, the object is changed. No new object is created.



The function ends, and the local variable, the_list, is deleted from memory.



When we pass a value, be it a variable or a literal, to a function we are, in fact, passing a reference to the object in question. The resultant behaviour depends on whether that object is mutable or immutable.

Introduction to Classes

You have seen that string objects have many methods, e.g. upper…

```
$ >>> my_string = "Hello"
$ >>> my_string_upper = my_string.upper()
$ >>> my_string_upper
$ 'HELLO'
```

…but where is the upper method declared? When a string object is created it is not empty. On the contrary it has data attributes and lots of methods.

An object is not created from nothing. It is constructed based on a template in which its data attributes and methods are declared. The name given to the template from which objects are created is a class.

Each of the data types you've thus far encountered, i.e. bool, int, float, complex, str, list, tuple, set, and dict is a class. And you will encounter many more as you import and use standard library and third party modules. You might choose to declare your own classes so as to create objects that represent the data in your specific business domain.

NOTE

Some languages allow for the creation of empty objects, that is, objects that have no associated template. Python is not one of them. Every object has an associated class.

Class Declaration and Instantiation

A class is declared using the class keyword and takes the following form:

```
class <ClassName>:
    # data attribute and method declarations
```

According to PEP (Python Enhancement Proposal) 0008…

> *Class names should normally use the CapWords convention.*

Now, you might be thinking that the built-in classes, e.g. int and str, break with this convention and you'd be right. But the proposal also states…

> *Note that there is a separate convention for builtin names: most builtin names are single words (or two words run together), with the CapWords convention used only for exception names and builtin constants.*

The dunder __init__ method is declared in almost every class. It is short for initialisation and is called when the class is used to create an object. Consider the following:

```
$ >>> class MyClass:
$ ...     def __init__(self):
$ ...         print("Initialising a MyClass object"...)
$ ...
```

You might be wondering about the parameter named self. It's a tricky one to explain but here goes. By the time the __init__ method is called the interpreter has already created an empty object in memory ready for initialisation. A reference to that object is automatically copied into the first parameter – self. Now, the parameter need not be named self. It could be named anything you want but the convention is to name it self.

To instantiate a class is to create an object based upon it. To instantiate a class, we call the class name as if it were a function.

```
$ >>> my_object = MyClass()
$ Initialising a MyClass object...
```

Note that the dunder __init__ method is called automatically when we instantiate a MyClass object. This is the same for all classes.

What is the data type of the object referenced by my_object?

```
$ >>> my_object = MyClass()
$ Initialising a MyClass object...
$ >>> type(my_object)
$ <class '__main__.MyClass'>
```

The data type is MyClass. The preceding __main__ part is the module name. See chapter 13 for more information about modules.

You might be thinking that objects of the built-in types, e.g. int and str, are not created by calling the class name as if it were a function. For example:

```
$ >>> my_int = 2
$ >>> my_str = "Hello"
```

That is true, of course, but this is only because the interpreter is built in such a way as to permit the creation of certain types in shorthand ways. We could, instead, have done…

```
$ >>> my_int = int(2)
$ >>> my_str = str("Hello")
```

…the result of which is the same.

Data Attributes

How do we specify that an object of a given class should have some data attributes? If you're migrating to Python from another language you might be inclined to do something like this…

```
$ >>> class Customer:
$ ...     name = ""
$ ...     email = ""
$ ...
```

…but that would be wrong. Data attributes must be assigned to the object in question. The code example above has the affect of assigning the data attributes (name and email) to the template (class), and not to each of the objects created based upon the template.

But how do we assign data attributes to the object from inside the class? After all, the class is the template for creating objects. In the previous section we said that a reference to the object is automatically copied into the first parameter – self. So we should assign data attributes to that.

```
$ >>> class Customer:
$ ...     def __init__(self):
$ ...         self.name = ""
$ ...         self.email = ""
$ ...
```

Now each Customer object will have name and email data attributes.

```
$ >>> my_customer = MyCustomer()
$ >>> my_customer.name = "David"
$ >>> my_customer.email = "david@mail.com"
$ >>> my_customer.name
$ 'David'
$ >>> my_customer.email
$ 'david@mail.com'
```

As it stands, when a Customer object is created its name and email data attributes reference empty strings. This may not be acceptable if, for example, business rules dictate that the customer name and email address must not be empty. To solve this problem, we might add additional parameters to the dunder __init__ method.

```
$ >>> class Customer:
$ ...     def __init__(self, name, email):
$ ...         self.name = name
$ ...         self.email = email
$ ...
```

We must now provide a name and email address at the point at which the Customer object is created.


```
$ >>> my_customer = MyCustomer("David", "david@mail.com")
$ >>> my_customer.name
$ 'David'
$ >>> my_customer.email
$ 'david@mail.com'
```

Remember that a reference to the object is automatically copied into the first parameter – self – so we need only pass in two arguments when we, indirectly, call the dunder __init__ method. The string literal "David" is copied into the parameter named name, and the string literal "david@mail.com" is copied into the parameter named email. The values of those parameters are then assigned to data attributes with the same names.

NOTE

Data attributes may be assigned inside methods other than __init__ but this is, in our view, bad practice. It is best to initialise all data attributes in the __init__ method so that one need only look in one place to see what data attributes objects of a given class have.

By default, the value of a data attribute may be changed.

```
$ >>> my_customer = MyCustomer("David", "david@mail.com")
$ >>> my_customer.email
$ 'david@mail.com'
$ >>> my_customer.email = david@stayahead.com
$ >>> my_customer.email
$ 'david@stayahead.com'
```

This applies to objects created based upon your own custom classes, but not necessarily to objects of built-in types. Recall the discussion regarding mutability. A Customer object is mutable because its data attributes may be changed.

Finally, you can add data attributes to an object post-creation…

```
$ >>> my_customer = MyCustomer("David", "david@mail.com")
$ >>> my_customer.age = 30
$ >>> my_customer.age
$ 30
```

…but this is generally to be avoided. Why? Because one of the advantages of declaring a class is that all objects created based upon that class have the same set of data attributes, (though the values will vary). This enables consistent processing. If you add a data attribute to one of your objects but not the others of the same type, then the objects can no longer be processed in a consistent manner.

Methods

To specify that an object of a given class should have one or more methods, you simply add one or more function declarations to the class.

```
$ >>> class Circle:
$ ...     def __init__(self, radius):
$ ...          self.radius = radius
$ ...     def circumference(self):
$ ...         return 2 * self.radius * 3.14
$ ...
```

Note that, like the dunder __init__ method, the circumference method we've declared has a parameter named self. Unless explicitly specified otherwise, a reference to the object is automatically copied into the first parameter – self. This is necessary so as to enable the accessing of the object's data attributes.

An object's methods are accessed using the dot notation just like its data attributes.

```
$ >>> my_circle = Circle(3)
$ >>> my_circle.radius
$ 3
$ >>> my_circle.circumference()
$ 18.84
```

In this case the circumference method makes use of the data attribute – radius. Good practice dictates that each of an object's methods should either use or change one or more of its data attributes. If it does not, then it probably shouldn't be associated with the object – the method should not be a part of the class. Consider the following:

```
$ >>> class Route:
$ ...     def __init__(self, point_a, point_b):
$ ...         self.point_a = point_a
$ ...         self.point_b = point_b
$ ...     def to_miles(self, kms):
$ ...         return kms * 0.621371
$ ...
```

Should the to_miles method be associated with a Route object? No. It neither uses nor changes the object's data attributes – point_a or point_b.

Composition

One custom object may be composed of one or more other custom objects. Consider the following example classes:

```
$ >>> class Product:
$ ...     def __init__(self, desc, price):
$ ...         self.desc = desc
$ ...         self.price = price
$ ...
$ >>> class Order:
$ ...     def __init__(self, customer):
$ ...         self.customer = customer
$ ...         self.products = []
$ ...     def grand_total(self):
$ ...         grand_total = 0
$ ...         for product in self.products:
$ ...             grand_total += product.price
$ ...         return grand_total
$ ...
```

Objects of type Order are composed of zero or more Product objects. Note the data attribute named products that is assigned an empty list. To it will be added Product objects. Consider the following example using both Product and Order classes.

```
$ >>> p1 = Product("Lamp", 19.95")
$ >>> p2 = Product("Mug", 3.90)
$ >>> p3 = Product("Television", 499.0)
$ >>> my_order = Order("David")
$ >>> my_order.products.append(p1)
$ >>> my_order.products.append(p2)
$ >>> my_order.products.append(p3)
$ >>> my_order.grand_total()
$ 522.85
```

We could have achieved something similar using dictionaries, of course. But objects of custom classes are, arguably, easier to work with and to read. Adding a product to an order dictionary might look something like this:

```
$ >>> my_order['products'].append(p1)
```

What is more, a class can be declared in such a way as to ensure that each object created based upon it has the same set of data attributes.

Exercises

Objects

In this exercise you will create a class to model a Training Course object. The class will contain both data and method attributes. The data attributes will consist of a course title, a duration (in days), a course price and a list of delegate names. The Training Course object will also require methods: a method to add a name to the delegate list and a method to display the total revenue generated for the Training Course object.

1. Create a script called ch7_training_course.py.
2. Create a class called TrainingCourse
3. Add a dunder __init__ method to ensure that a TrainingCourse object is created with the following data attributes: *title*, *duration*, *pricePerPerson* and an empty list called *delegates*.
4. Add a method called *addDelegates* that will accept a delegate name as a parameter and then append it to the *delegates* list i.e. delegates.append(name).
5. Add method called *getTotalRevenue* which will return the total revenue generated for the Training Course object i.e. number of delegates * pricePerPerson.
6. Create a TrainingCourse object with the data attributes set to the following values:

   title: Python Programming 1
   duration: 4
   price per person: 1600
   delegates: [ ]
7. Use the addDelegate method to assign some names to the Training Course object's *delegates* list.
8. Display to the console the total revenue generated for the TrainingCourse object.

CHAPTER 8

Lists

Introduction

A list is a collection of values, that is, a collection of anything. In Python, a list literal comprises zero or more comma separated values surrounded by square brackets.

```
$ >>> my_list = [1, 2, 3]
$ >>> type(my_list)
$ <class 'list'>
```

A list object's properties are tabled below.

Property	Yes/No	Description
Mutable?	Yes	A list can both grow and shrink, and its contents may be changed.
Key value pairs?	No	N/A
Ordered?	Yes	Each element in a list is indexed, with the first element at index zero. Iterating over a list yields the elements in order from index zero to index (length – one).
Duplicates?	Yes	N/A

For quick reference, a list object's methods are tabled below. Each of these methods is described in detail in the sections that follow.

Method	Description
index(<element>): <index>	Returns the index of the first instance of the given element or raises an error if the element does not exist in the list
append(<element>)	Appends the given element to the end of the list
extend(<other_list>)	Appends all of the elements of the given other_list to the end of the list
insert(<index>, <element>)	Inserts the given element at the given index
remove(<element>)	Removes the first instance of the given element or raises an error if the element does not exist in the list
pop(<index>): <element>	Removes and returns the element at the given index
clear()	Removes all elements from the list
sort()	Sorts the list
copy(): <set>	Returns a shallow copy of the list
count(<element>): <number>	Returns the number of instances of the given element
reverse()	Reverses the order of the elements in the list

Indexing

A list element is accessed by specifying its index between square brackets.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> second_name = names[1]
$ >>> second_name
$ 'Sarah'
```

NOTE

As a list is mutable, the index of a given element may change over time.

Python supports negative indexing too. The index -1 yields the last element in the list, -2 the second last element and so on.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> second_to_last_name = names[-2]
$ >>> second_to_last_name
$ 'Tom'
```

Any attempt to access a non-existent list element will result in the raising of an IndexError.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names[4]
$ IndexError: list index out of range
```

The **index** method is used to find the index of a given element.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> index_of_tom = names.index("Tom")
$ >>> index_of_tom
$ 2
```

A ValueError is raised in the event the given element does not exist in the list.

Slicing

Slicing provides for the extracting of a subset of list elements into a new list but does not change the original list. It requires the addition of the : (colon) between the square brackets. The number to the left of the colon is the beginning index (inclusive), and the number to the right of the colon is the ending index (exclusive).

```
$ >>> fib = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
$ >>> fib[3:7]  # indices 3 through 6
$ [2, 3, 5, 8]
```

Either or both indices may be omitted. If the beginning index is omitted, the first element of the subset is the one at index 0 (zero). If the ending index is omitted, the last element of the subset if the one at index (length – 1).

```
$ >>> fib = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
$ >>> fib[:3]  # indices 0 through 2
$ [0, 1, 1]

$ >>> fib[5:]  # indices 5 through (length - 1)
$ [5, 8, 13, 21, 34]
```

The indices may be negative.

```
$ >>> fib = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
$ >>> fib[-4:-1]  # indices (length - 4) through (length - 2)
$ [8, 13, 21]
```

The step value is 1 (one) by default, and so the following yields an empty list.

```
$ >>> fib = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
$ >>> fib[-4:1]  # invalid - you can't reach 1 from -4 with a step of 1
$ []
```

The solution is to specify the step.

```
$ >>> fib = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
$ >>> fib[-4:1:-1]
$ [8, 5, 3, 2, 1]
```

Note the addition of a second colon and a third number. The third number is the step. Setting a negative step makes it easy to reverse the elements in a list.

```
$ >>> fib = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
$ >>> fib[::-1]
$ [34, 21, 13, 8, 5, 3, 2, 1, 1, 0]
```

NOTE

To summarise, slicing creates a new list from an existing one and takes the form:

```
<list_name>[<begin_index_incl>:<end_index:excl>:<step>]
```

All of the parts are optional. The begin index is 0 (zero) by default, the end index is (length – 1) by default, and the step is 1 (one) by default.

Updating

A list element is updated by assigning a new value at the given index.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names[2] = "Thomas"
$ >>> names[2]
$ 'Thomas'
```

Two or more elements may be updated at a time by exploiting slicing-style syntax.

```
$ >>> odd_numbers = [1, 3, 4, 6, 9]
$ >>> odd_numbers[2:4] = [5, 7]
$ >>> odd_numbers
$ [1, 3, 5, 7, 9]
```

Appending & Inserting

The **append** method is used to add an element to a list. The newly added element will be appended to the end of the list.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names.append("Fred")
$ >>> names
$ ['David', 'Sarah', 'Tom', 'Jane', 'Fred']
```

The **extend** method is used to add all the elements of one list to another.

```
$ >>> odd_numbers = [1, 3, 5]
$ >>> more_odd_numbers = [7, 9]
$ >>> odd_numbers.extend(more_odd_numbers)
$ >>> odd_numbers
$ [1, 3, 5, 7, 9]
```

The same thing can be achieved using the += operator.

```
$ >>> odd_numbers = [1, 3, 5]
$ >>> more_odd_numbers = [7, 9]
$ >>> odd_numbers += more_odd_numbers
$ >>> odd_numbers
$ [1, 3, 5, 7, 9]
```

The **insert** method is used to insert a new element between existing ones in a list. It expects the index at which to insert the new element, and the element to be inserted. Existing elements are moved to accommodate the new element.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names.insert(2, "Holly")
$ >>> names
$ ["David", "Sarah", "Holly", "Tom", "Jane"]
```


Removing

The **remove** method is used to remove a given element. As a list may comprise duplicates, the remove method will remove only the first instance of the given element. Existing elements are, effectively, moved to close the gap created by the removal.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names.remove("Tom")
$ >>> names
$ ['David', 'Sarah', 'Jane']

$ >>> numbers = [1, 2, 2, 3, 4]
$ >>> numbers.remove(2)
$ >>> numbers
$ [1, 2, 3, 4]
```

A ValueError is raised in the event the given element does not exist in the list.

NOTE

Using the remove method on a list comprising integers may be ambiguous to read. It may seem as though your intention is to remove an element at a given index.

The **pop** method is used to remove and return the element at a given index.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> removed = names.pop(1)
$ >>> removed
$ 'Sarah'
$ >>> names
$ ['David', 'Tom', 'Jane']
```

Calling the pop method without passing an index results in the removal and return of the last element in the list.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> removed = names.pop()
$ >>> removed
$ 'Jane'
$ >>> names
$ ['David', 'Tom', 'Sarah']
```

The **clear** method is used to remove all elements from a list.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names.clear()
$ >>> names
$ []
```

The **del** keyword is used to delete references to objects, and so may be used to remove elements from a list.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> del names[0]
$ >>> names
$ ['Sarah', 'Tom', 'Jane']
```

As with updating, two or more elements may be removed at a time by exploiting slicing-style syntax.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> del names[1:3]
$ >>> names
$ ['David', 'Jane']
```

The del keyword may also be used to remove the entire list.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> del names
$ >>> names
$ NameError: name 'names' is not defined
```

Finally, elements may be removed by assigning an empty list to a subset of the list, again exploiting slicing-style syntax.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names[1:3] = []
$ >>> names
$ ['David', 'Jane']
```

NOTE

To remove an element from a list does not mean that the object referenced is removed. It simply means that the reference is removed. The object may live on in memory long after the reference is removed.

Iterating

For loops are used to iterate over lists. A for statement takes the following form:

```
for <element> in <collection>:
    <statement(s)>
```

The block is executed once for each element in the collection.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> for name in names:
$ ...     print(name)
$ ...
$ David
$ Sarah
$ Tom
$ Jane
```

NOTE

The variable to which each subsequent element is assigned (name in this case) need not have been declared outside of the loop.

The for loop yields only the value of each element, and not its index. When passed a list, the built-in **enumerate** function returns an enumerate object which is, effectively, a list of tuples. Each tuple comprises the element's index and its value.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> for index, name in enumerate(names):
$ ...     print(index, name)
$ ...
$ 0 David
$ 1 Sarah
$ 2 Tom
$ 3 Jane
```

Membership Testing

The **in** operator is used to test for membership, that is, to test whether or not a given value is present in the list.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> "Tom" in names
$ True
$ >>> "Fred" not in names
$ True
```

Sorting

The sort method is used to sort the elements of a list. The sort method applies the < (less than) operator to determine the order of the list's elements. The smallest element will be first in the list after sorting.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names.sort()
$ >>> names
$ ['David', 'Jane', 'Sarah', 'Tom']
```

David is first in the list after sorting because the string "David" is less than all of the other strings, in this case. Why? Strings are compared lexicographically. That is, D comes before J which comes before S which comes before T.

You can sort in reverse order by passing to the function a boolean value.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names.sort(reverse=True)
$ >>> names
$ ['Tom', 'Sarah', 'Jane', 'David']
```

You can change the sort key (the thing on which the comparison is performed) by passing to the sort function a reference to a function that returns said key. The function must accept one argument (the element) and return a value that represents the new key on which the list should be sorted.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names.sort(key=len)
$ >>> names
$ ['Tom', 'Jane', 'Sarah', 'David']
```

The built-in len function works here because it accepts one argument and returns a number (the length of the argument). The names list is, accordingly, sorted by the length of each element.

NOTE

Both sort method parameters, reverse and key, are default parameters. That is, in the declaration of the sort method they are each assigned a default value. That is why the sort method can be called without passing any arguments to it.

Copying

Consider the following:

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> copy = names
```

Does the copy variable store a copy of the names list? No. It stores a reference to the same list object as does the names variable. Making changes to the list using the copy variable will affect the names variable, after all, each variable references the same object.

```
$ >>> copy.append("Fred")
$ >>> copy
$ ['David', 'Sarah', 'Tom', 'Jane', 'Fred']
$ >>> names
$ ['David', 'Sarah', 'Tom', 'Jane', 'Fred']
```

The **copy** method is used to create a shallow copy of a list.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> copy = names.copy()
$ >>> copy.append("Fred")
$ >>> copy
$ ['David', 'Sarah', 'Tom', 'Jane', 'Fred']
$ >>> names
$ ['David', 'Sarah', 'Tom', 'Jane']
```

Calling the copy method results in the creation of a new list object.

So why is it called a shallow copy? Whilst the copy method creates a new list object, it simply copies all of the references from the original list into the new list. If those references refer to mutable objects, then making a change to the elements in one list will affect the other. Consider a list of lists.

```
$ >>> coordinates_1 = [[0, 0], [0, 1], [1, 1]]
$ >>> coordinates_2 = coordinates_1.copy()
$ >>> coordinates_2[2][0] = 0  # change 1st coord of 3rd set to 0
$ >>> coordinates_1
$ [[0, 0], [0, 1], [0, 1]]  # Oops! should not have been changed
```

Each of the coordiantes_1 and coordiantes_2 variables reference separate list objects. However, changing element 2 of the copied list does affect the original list because element 2 is a mutable list. The interpreter changes the list object and not its reference – coordinates_2[2]. Care must be taken when copying a list with mutable elements.

NOTE

The copy module has a function named deepcopy which may be used to create a deep copy of a list.

```
import copy
...
coordinates_2 = copy.deepcopy(coordinates_1)
```

The resultant list is completely detached from the original. That is, the deepcopy method copies each mutable element recursively.

Other List Methods

The **count** method is used to obtain the number of instances of a given element.

```
$ >>> numbers = [1, 1, 2, 3, 4, 4, 5, 5, 5, 6, 7, 7]
$ >>> numbers.count(5)
$ 3
```

The **reverse** method is used to reverse the order of the elements in a list.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names.reverse()
$ >>> names
$ ['Jane', 'Tom', 'Sarah', 'David']
```

The same may be achieved using slicing, though slicing results in the creation of a new list; it does not change the original.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> names_in_reverse = names[::-1]
$ >>> names_in_reverse
$ ['Jane', 'Tom', 'Sarah', 'David']
$ >>> names
$ ['David', 'Sarah', 'Tom', 'Jane']
```


Built-in Functions

There are a number of built-in functions applicable to lists. These are tabled below.

Built-in function	Description
len(<list>): <number>	Returns the number of elements in the list
min(<list>): <element>	Returns the smallest element in the list
max(<list>): <element>	Returns the largest element in the list
sum(<list>): <value>	Returns the sum of the elements in the list
any(<list>): <boolean>	Returns True if any of the elements in the list are True (or would be true following conversion e.g. bool(<element>))
all(<list>): <boolean>	Returns True if all of the elements in the list are True (or would be true following conversion e.g. bool(<element>))
enumerate(<list>): <enumerate>	Returns an enumerate object, effectively a list of tuples, where each tuple comprises an index and the associated element (see p.10)
sorted(<list>): <sorted_list>	Returns a new, sorted list based on the original (very similar to the sort method but creates a new list instead of changing the original)

List Arithmetic

The addition and multiplication operators may be applied to lists. Consider the following:

```
$ >>> odd_numbers = [1, 3, 5]
$ >>> more_odd_numbers = [7, 9]
$ >>> odd_numbers + more_odd_numbers
$ [1, 3, 5, 7, 9]
$ >>> odd_numbers
$ [1, 3, 5]

$ >>> odd_numbers = [1, 3, 5]
$ >>> odd_numbers * 2
$ [1, 3, 5, 1, 3, 5]
$ >>> odd_numbers
$ [1, 3, 5]
```

Note that adding or multiplying two lists results in the creation of a new list. It does not change either of the original lists.

Exercises

Lists

In this exercise you will create a script that emulates a sporting league table. Teams will move up and down the table, some will be relegated and some will be promoted. The team at index 0 (zero) is to be considered on top of the table.

1. Create a script named ch8_lists.py.
2. Declare a variable named table and assign it a list of teams (strings) as follows: Bath, Derby, Gloucester, Lancaster, Newcastle, Plymouth, Salford, and Wakefield.
3. Move Newcastle into 3rd position (remove and then insert).
4. Move Derby into 4th position.
5. Relegate (remove) Salford.
6. Promote (append) Canterbury at the bottom of the table.
7. In one statement, relegate Plymouth and promote York (update).
8. Extract the top two teams into a new list named to_play_in_europe.
9. If Newcastle is to play in Europe, print Newcastle to play in Europe to the console.
10. Code a loop that iterates over the table.
11. Inside the loop print each team to the console. The output should be as follows:

```
Bath
Newcastle
Gloucester
Derby
Lancaster
York
Wakefield
Canterbury
```

CHAPTER 9

Tuples

Introduction

A tuple is similar to a list but its elements cannot be modified. That is, after creation, elements cannot be added, edited, or removed. In Python, a tuple literal comprises one or more comma separated values surrounded by round brackets.

```
$ >>> my_tuple = ("Ball", 1.99)
$ >>> type(my_tuple)
$ <class 'tuple'>
```

A tuple object's properties are tabled below.

Property	Yes/No	Description
Mutable?	No	A tuple cannot grow nor shrink, nor can its contents be changed.
Key value pairs?	No	N/A
Ordered?	Yes	Each element in a tuple is indexed, with the first element at index zero. Iterating over a tuple yields the elements in order from index zero to index (length – one).
Duplicates?	Yes	N/A

For quick reference, a tuple object's methods are tabled below. Each of these methods is described in detail in the sections that follow.

Method	Description
index(<element>): <index>	Returns the index of the first instance of the given element or raises an error if the element does not exist in the list
count(<element>): <number>	Returns the number of instances of the given element

NOTE

Why a tuple instead of a list? A tuple is usually used to provide structure – the grouping of a small number of heterogeneous values, for example, chapter title and page number. Where a list is usually used to provide order – the grouping of a large number of homogenous values, for example, the sales figures for 2018.

Creation

As described previously, a tuple literal comprises one or more comma separated values surrounded by round brackets.

```
$ >>> my_tuple = ("Ball", 1.99)
```

The creation of a tuple with a solitary value is a little unusual. Consider the following:

```
$ >>> my_tuple = ("Ball")
$ >>> type(my_tuple)
$ <class 'str'>
```

It's a string! Why? In this context the round brackets are interpreted as the grouping operator. Recall the operator order of precedence. Brackets are used to group parts of an expression so as to ensure that group is evaluated first. That's what happening here. The interpreter assumes you're grouping parts of an expression.

To create a tuple with a solitary value you must insert a comma after the value.

```
$ >>> my_tuple = ("Ball",)
$ >>> type(my_tuple)
$ <class 'tuple'>
```


Indexing

A tuple element is accessed by specifying its index between square brackets.

```
$ >>> book = ("Past Mortem", "Ben Elton", 2004, 8.22)
$ >>> author = book[1]
$ >>> author
$ 'Ben Elton'
```

Python supports negative indexing too. The index -1 yields the last element in the list, -2 the second last element and so on.

```
$ >>> book = ("Past Mortem", "Ben Elton", 2004, 8.22)
$ >>> publication_year = book[-2]
$ >>> publication_year
$ 2004
```

Any attempt to access a non-existent tuple element will result in the raising of an IndexError.

```
$ >>> book = ("Past Mortem", "Ben Elton", 2004, 8.22)
$ >>> book[4]
$ IndexError: tuple index out of range
```

The **index** method is used to find the index of a given element.

```
$ >>> book = ("Past Mortem", "Ben Elton", 2004, 8.22)
$ >>> index_of_author = book.index("Ben Elton")
$ >>> index_of_author
$ 1
```

A ValueError is raised in the event the given element does not exist in the tuple.

Slicing

Slicing provides for the extracting of a subset of tuple elements into a new tuple but does not change the original tuple. It requires the addition of the : (colon) between the square brackets. The number to the left of the colon is the beginning index (inclusive), and the number to the right of the colon is the ending index (exclusive).

```
$ >>> fib = (0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
$ >>> fib[3:7]  # indices 3 through 6
$ (2, 3, 5, 8)
```

Either or both indices may be omitted. If the beginning index is omitted, the first element of the subset is that at index 0 (zero). If the ending index is omitted, the last element of the subset if that at index (length – 1).

```
$ >>> fib[:3]  # indices 0 through 2
$ (0, 1, 1)

$ >>> fib[5:]  # indices 5 through (length - 1)
$ (5, 8, 13, 21, 34)
```

The indices may be negative.

```
$ >>> fib[-4:-1]  # indices (length - 4) through (length - 2)
$ (8, 13, 21)
```

The step value is 1 (one) by default, and so the following yields an empty list.

```
$ >>> fib[-4:1]  # invalid - you can't reach 1 from -4 with a step of 1
$ ()
```

The solution is to specify the step.

```
$ >>> fib[-4:1:-1]
$ (8, 5, 3, 2, 1)
```

Note the addition of a second colon and a third number. The third number is the step. Setting a negative step makes it easy to reverse the elements in a list.

```
$ >>> fib[::-1]
$ (34, 21, 13, 8, 5, 3, 2, 1, 1, 0)
```

NOTE

To summarise, slicing creates a new tuple from an existing one and takes the form:

`tuple_name>[<begin_index_incl>:<end_index:excl>:<step>]`

All of the parts are optional. The begin index is 0 (zero) by default, the end index is (length – 1) by default, and the step is 1 (one) by default.

Iterating

For loops are used to iterate over tuples. A for statement takes the following form:

```
for <element> in <collection>:
    <statement(s)>
```

The block is executed once for each element in the collection.

```
$ >>> book = ("Past Mortem", "Ben Elton", 2004, 8.22)
$ >>> for prop in book:
$ ...     print(prop)
$ ...
$ Past Mortem
$ Ben Elton
$ 2004
$ 8.22
```

NOTE

The variable to which each subsequent element is assigned (prop in this case) need not have been declared outside of the loop.

The for loop yields only the value of each element, and not its index. When passed a tuple, the built-in **enumerate** function returns an enumerate object which is, effectively, a list of tuples. Each tuple comprises the element's index and its value.

```
$ >>> book = ("Past Mortem", "Ben Elton", 2004, 8.22)
$ >>> for index, prop in enumerate(book):
$ ...     print(index, prop)
$ ...
$ 0 Past Mortem
$ 1 Ben Elton
$ 2 2004
$ 3 8.22
```

Membership Testing

The **in** operator is used to test for membership, that is, to test whether or not a given value is present in the tuple.

```
$ >>> book = ("Past Mortem", "Ben Elton", 2004, 8.22)
$ >>> "Ben Elton" in book
$ True
$ >>> 2005 not in book
$ True
```

Other Tuple Methods

The **count** method is used to obtain the number of instances of a given element.

```
$ >>> numbers = (1, 1, 2, 3, 4, 4, 5, 5, 5, 6, 7, 7)
$ >>> numbers.count(5)
$ 3
```

Built-in Functions

There are a number of built-in functions applicable to tuples. These are tabled below.

Built-in function	Description
len(<tuple>): <number>	Returns the number of elements in the tuple
min(<tuple>): <element>	Returns the smallest element in the tuple
max(<tuple>): <element>	Returns the largest element in the tuple
sum(<tuple>): <value>	Returns the sum of the elements in the tuple
any(<tuple>): <boolean>	Returns True is any of the elements in the tuple are True (or would be true following conversion e.g. bool(<element>))
all(<tuple>): <boolean>	Returns True is all of the elements in the tuple are True (or would be true following conversion e.g. bool(<element>))
enumerate(<tuple>): <enumerate>	Returns an enumerate object, effectively a list of tuples, where each tuple comprises an index and the associated element (see p.7)
sorted(<tuple>): <sorted_list>	Returns a sorted list based on the tuple

Tuple Arithmetic

The addition and multiplication operators may be applied to tuples. Consider the following:

```
$ >>> odd_numbers = (1, 3, 5)
$ >>> more_odd_numbers = (7, 9)
$ >>> odd_numbers + more_odd_numbers
$ (1, 3, 5, 7, 9)
$ >>> odd_numbers
$ (1, 3, 5)

$ >>> odd_numbers = (1, 3, 5)
$ >>> odd_numbers * 2
$ (1, 3, 5, 1, 3, 5)
$ >>> odd_numbers
$ (1, 3, 5)
```

Note that adding or multiplying two tuples results in the creation of a new tuple. It does not change either of the original tuples.

CHAPTER 10

Sets

Introduction

A set is an unordered collection of hashable objects that does not permit duplicate values. It does, however, permit the adding and removing of elements. In Python, a set literal comprises one or more comma separated values surrounded by curly braces.

```
$ >>> my_set = {1, 1, 2, 3, 3, 4, 5}
$ >>> my_set
$ {1, 2, 3, 4, 5}
$ >>> type(my_set)
$ <class 'set'>
```

NOTE

An object is hashable if:

1. **Its __hash__ method returns a value that does not change during its lifetime**
2. **It can be compared to other objects (has an __eq__ method)**

Effectively, a hashable object is an immutable one.

A set object's properties are tabled below.

Property	Yes/No	Description
Mutable?	Yes	A set can grow and shrink but its elements, courtesy of their being hashable, cannot be changed.
Key value pairs?	No	N/A
Ordered?	No	N/A
Duplicates?	No	N/A

For quick reference, a set object's methods are tabled below. Each of these methods is described in detail in the sections that follow.

Method	Description
add(<element>)	Adds the given element to the set provided it is not already a member
update(<collection>)	Adds each of the elements in the given collection to the set provided it is not already a member
remove(<element>)	Removes the given element or raises an error if the element does not exist in the set
discard(<element>)	Removes the given element from the set
pop(): <element>	Removes and returns an arbitrary element
clear()	Removes all elements from the set
copy(): <set>	Returns a copy of the set

Method (parameters and return value omitted for brevity)	Description
union	Returns a new set that represents a union of the set and the given other set
intersection	Returns a new set that represents the intersection of the set and the given other set
difference	Returns a new set that represents the difference between the set and the given other set
symmetric_difference	Returns a new set that represents the symmetric difference between the set and given other set
intersection_update	Removes all the elements from the set that do not also exist in the given other set
difference_update	Removes all the elements from the set that also exist in the given other set
symmetric_difference_update	Removes all the elements from the set that exist in both it and the given other set
isdisjoint	Returns True if the intersection between the set and the given other set yields an empty set
issubset	Returns True if the set is a subset of the given other set
issuperset	Returns True if the set is a superset of the given other set

NOTE

Why a set instead of a list? Sets are very useful for removing duplicate elements from a list, and to perform operations such as unions, intersections, and differences.

Creation

As described previously, a set literal comprises one or more comma separated values surrounded by curly braces.

```
$ >>> my_set = {1, 2, 3, 4, 5}
```

The creation of an empty set is a little unusual. Consider the following:

```
$ >>> my_set = {}
$ >>> type(my_set)
$ <class 'dict'>
```

Curly braces are used to construct dictionaries, in addition to sets. A set of empty braces is assumed to be a dictionary and not a set. Therefore, to create an empty set you must use the built-in set function.

```
$ >>> my_set = set()
$ >>> type(my_set)
$ <class 'set'>
```

As described in the section about type conversion in chapter 3, the built-in set function may also be used to create a set from an existing collection.

```
$ >>> my_list = [1, 2, 3, 4, 5]
$ >>> my_set = set(my_list)
$ >>> my_set
$ {1, 2, 3, 4, 5}
```

NOTE

A set is most commonly created from an existing collection, e.g. a list. The built-in set function does not change the existing collection.

Each element in a set must be hashable (immutable). Consider the following:

```
$ >>> my_set = {[1, 2, 3], [4, 5, 6]}
$ TypeError: unhashable type: 'list'
```

list objects are NOT hashable (immutable) and so cannot be added to a set.

Adding

The **add** method is used to add an element to a set.

```
$ >>> my_set = {1, 2, 3}
$ >>> my_set.add(4)
$ >>> my_set
$ {1, 2, 3, 4}
```

Attempting to add an existing element to a set has no effect.

```
$ >>> my_set = {1, 2, 3}
$ >>> my_set.add(2)
$ >>> my_set
$ {1, 2, 3}
```

The **update** method (a bad choice of name in our view) is used to add multiple elements to the set. The update method accepts a collection, e.g. list, tuple, set, dict, string.

```
$ >>> my_set = {1, 2, 3}
$ >>> my_set.update([4, 5, 6])
$ >>> my_set
$ {1, 2, 3, 4, 5, 6}
```

Attempting to add multiple existing elements to a set has no effect.

Removing

The **remove** method is used to remove an element from a set.

```
$ >>> my_set = {1, 2, 3}
$ >>> my_set.remove(2)
$ >>> my_set
$ {1, 3}
```

A KeyError is raised in the event the given element does not exist in the set.

The **discard** method may also be used to remove an element from a set.

```
$ >>> my_set = {1, 2, 3}
$ >>> my_set.discard(2)
$ >>> my_set
$ {1, 3}
```

No error is raised in the event the given element does not exist in the set.

NOTE

You might recall that each list object has a pop method that is used to remove an element at a given index. Each set object has a pop method too but, as a set is unordered, there are no indices. Therefore, the pop method simply removes and returns an element arbitrarily.

The **clear** method is used to remove all elements from a set.

```
$ >>> my_set = {1, 2, 3}
$ >>> my_set.clear()
$ >>> my_set
$ set()
```

Iterating

For loops are used to iterate over sets. A for statement takes the following form:

```
for <element> in <collection>:
    <statement(s)>
```

The block is executed once for each element in the collection.

```
$ >>> my_set = set("David")
$ >>> for char in my_set:
$ ...     print(char)
$ ...
$ v
$ d
$ a
$ i
$ D
```

Remember that sets are not ordered in the way a list or tuple is. This means that iterating over a set may not yield the elements in the order in which they were added.

NOTE

The variable to which each subsequent element is assigned (char in this case) need not have been declared outside of the loop.

Membership Testing

The **in** operator is used to test for membership, that is, to test whether or not a given value is present in the set.

```
$ >>> my_set = {1, 2, 3}
$ >>> 2 in my_set
$ True
$ >>> 4 not in my_set
$ True
```

Sorting

Unlike a list, a set object does not have a sort method. This is because a set does not support the updating of its elements. The built-in sorted function, however, may be used to create a sorted list from an existing set.

```
$ >>> my_set = {"a", "b", "c"}
$ >>> my_set
$ {'c', 'a', 'b'}
$ >>> sorted_list = sorted(my_set)
$ >>> sorted_list
$ ['a', 'b', 'c']
```

You can sort in reverse order by passing to the function a boolean value.

```
$ >>> my_set = {"a", "b", "c"}
$ >>> my_set
$ {'c', 'a', 'b'}
$ >>> sorted_list = sorted(my_set, reverse=True)
$ >>> sorted_list
$ ['c', 'b', 'a']
```

You can change the sort key (the thing on which the comparison is performed) by passing to the function a reference to a function that returns said key. The function must accept one argument (the element) and return a value that represents the new key on which the list should be sorted.

```
$ >>> # my_set comprises tuples of page numbers and line numbers
$ >>> my_set = {(35, 19), (81, 4), (112, 47)}
$ >>> sorted_by_line_number = sorted(my_set, key=lambda t : t[1])
$ >>> sorted_by_line_number
$ [(81, 4), (35, 19), (112, 47)]
```

The key keyword argument is assigned a lambda function. It accepts one argument – a tuple, in this case – and returns the second element of said tuple (the line number). The sort key is now the line number.

NOTE

Both sorted function parameters, reverse and key, are default parameters. That is, in the declaration of the sorted function they are each assigned a default value. That is why the sorted function can be called without passing any arguments to it.

Copying

The **copy** method is used to create a copy of a set.

```
$ >>> my_set = {1, 2, 3}
$ >>> copy = my_set.copy()
$ >>> copy.add(4)
$ >>> copy
$ {1, 2, 3, 4}
$ >>> my_set
$ {1, 2, 3}
```

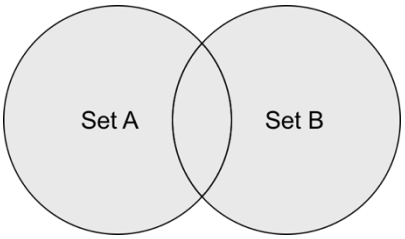
Calling the copy method results in the creation of a new set object.

NOTE

As with a list object, the copy created by the copy method is a shallow one. Unlike a list, however, this is not a problem in the case of a set because its elements must be hashable (immutable).

Union

The union of a collection of sets is the set of all the elements in the collection.



The **union** method is used to perform a union operation.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> u = set_a.union(set_b)
$ >>> u
$ {1, 2, 3, 4, 5, 6, 7}
$ >>> set_a
$ {1, 2, 3, 4, 5}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```

NOTE

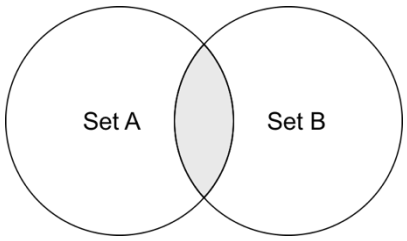
The union method creates a new set; it does not change the original sets.

A union may also be performed using the | (bitwise or) operator.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> u = set_a | set_b
$ >>> u
$ {1, 2, 3, 4, 5, 6, 7}
$ >>> set_a
$ {1, 2, 3, 4, 5}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```


Intersection

The intersection of two sets, A and B, is the set that contains all the elements of A that also exist in B or, equivalently, all the elements of B that also exist in A.



The **intersection** method is used to perform an intersection operation.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> i = set_a.intersection(set_b)
$ >>> i
$ {3, 4, 5}
$ >>> set_a
$ {1, 2, 3, 4, 5}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```

NOTE

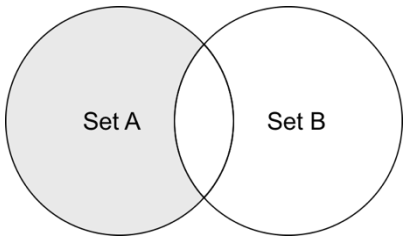
The intersection method creates a new set; it does not change the original sets.

An intersection may also be performed using the & (bitwise and) operator.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> i = set_a & set_b
$ >>> i
$ {3, 4, 5}
$ >>> set_a
$ {1, 2, 3, 4, 5}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```

Difference

The difference between two sets, A and B, is the set that contains all the elements of A that do not exist in B.



The **difference** method is used to perform a difference operation.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> d = set_a.difference(set_b)
$ >>> d
$ {1, 2}
$ >>> set_a
$ {1, 2, 3, 4, 5}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```

NOTE

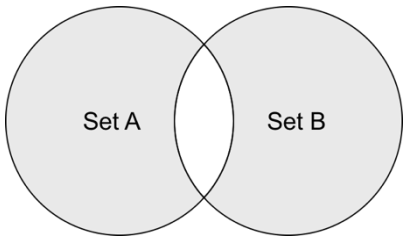
The difference method creates a new set; it does not change the original sets.

A difference may also be performed using the - (subtract) operator.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> d = set_a - set_b
$ >>> d
$ {1, 2}
$ >>> set_a
$ {1, 2, 3, 4, 5}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```


Symmetric Difference

The symmetric difference between two sets, A and B, is the set that contains all the elements of A that do not exist in B, and all the elements of B that do not exist in A.



The **symmetric_difference** method is used to perform a symmetric difference operation.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> sd = set_a.symmetric_difference(set_b)
$ >>> sd
$ {1, 2, 6, 7}
$ >>> set_a
$ {1, 2, 3, 4, 5}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```

NOTE

The symmetric_difference method creates a new set; it does not change the original sets.

A symmetric difference may also be performed using the ^ (exclusive or) operator.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> sd = set_a ^ set_b
$ >>> sd
$ {1, 2, 6, 7}
$ >>> set_a
$ {1, 2, 3, 4, 5}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```

Other Set Methods

The **intersection_update** method is used to remove all elements from set A that do not also exist in set B. It does not return anything.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> set_a.intersection_update(set_b)
$ >>> set_a
$ {3, 4, 5}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```

The **difference_update** method is used to remove all elements from set A that also exist in set B. It does not return anything.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> set_a.difference_update(set_b)
$ >>> set_a
$ {1, 2}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```

The **symmetric_difference_update** method is used to remove all elements from set A that exist in both set A and set B.

```
$ >>> set_a = {1, 2, 3, 4, 5}
$ >>> set_b = {3, 4, 5, 6, 7}
$ >>> set_a.symmetric_difference_update(set_b)
$ >>> set_a
$ {1, 2, 6, 7}
$ >>> set_b
$ {3, 4, 5, 6, 7}
```

The **isdisjoint** method returns True if the intersection between sets A and B yields an empty set.

```
$ >>> set_a = {1, 2, 3}
$ >>> set_b = {4, 5, 6}
$ >>> seta.isdisjoint(sb)
$ True
```

The **issubset** method returns True if set A is a subset of set B.

```
$ >>> set_a = {1, 2, 3}
$ >>> set_b = {1, 2, 3, 4, 5, 6}
$ >>> set_a.issubset(set_b)
$ True
```


The **issuperset** method returns True is set A is a superset of set B.

```
$ >>> set_a = {1, 2, 3, 4, 5, 6}
$ >>> set_b = {1, 2, 3}
$ >>> set_a.issuperset(set_b)
$ True
```

Built-in Functions

There are a number of built-in functions applicable to sets. These are tabled below.

Built-in function	Description
len(<set>): <number>	Returns the number of elements in the set
min(<set>): <element>	Returns the smallest element in the set
max(<set>): <element>	Returns the largest element in the set
sum(<set>): <value>	Returns the sum of the elements in the set
any(<set>): <boolean>	Returns True is any of the elements in the set are True (or would be true following conversion e.g. bool(<element>))
all(<set>): <boolean>	Returns True is all of the elements in the set are True (or would be true following conversion e.g. bool(<element>))
enumerate(<set>): <enumerate>	Returns an enumerate object, effectively a list of tuples, where each tuple comprises an index and the associated element (see p.10)
sorted(<set>): <sorted_list>	Returns a sorted list based on the set

Exercises

Sets

In this exercise you will create a new and improved version of the ch6_profile_matcher.py module by exploiting sets.

1. Make a copy of ch6_profile_matcher.py and name it ch10_profile_matcher.py.
2. Remove the loop code that was previously used to identify the hobbies each of the profiles had in common.
3. Convert p1's list of hobbies into a set and assign it to a variable named s1.
4. Convert p2's list of hobbies into a set and assign it to a variable named s2.
5. Create a new set from the existing ones that contains only those hobbies that exist in each of the original sets. Hint: it's one of either union, intersection, or difference.
6. Increment the match_quality variable by the length of the new set.
7. Make a copy of ch6_functions.py and name it ch10_sets.py.
8. Update the import statement so as to import the match function from the ch10_profile_matcher module.

CHAPTER 11

Dictionaries

Introduction

A dict (dictionary) is a collection of key value pairs. In Python, a dictionary literal comprises zero or more comma separated key value pairs surrounded by curly braces. Each key value pair is delimited by a colon.

```
$ >>> my_dict = {'name': 'David', 'age': 30}
$ >>> type(my_dict)
$ <class 'dict'>
```

In this example, `'name'` and `'age'` are the keys, while `'David'` and `30` are the values.

A dictionary object's properties are tabled below.

Property	Yes/No	Description
Mutable?	Yes	A dictionary can both grow and shrink, and its contents may be changed.
Key value pairs?	Yes	Each key must be hashable (immutable) and, as a consequence, unique.
Ordered?	Yes	Dictionaries are insertion ordered for the CPython implementation of python, as of v3.6.
Duplicates?	Values	Duplicate values are permitted, duplicate keys are not.

For quick reference, a dictionary object's methods are tabled below. Each of these methods is described in detail in the sections that follow.

Method	Description
get(<key>): <value>	Returns the value associated with the given key
setdefault(<key>, <value>)	Returns the value associated with the given key or, if the key does not exist, adds a new key value pair
update(<collection>)	Adds multiple key value pairs to the dictionary; each element of the collection argument must itself be a collection comprising two elements – a key and a value
pop(<key>): <value>	Removes and returns the value associated with the given key; both key and value are removed from the dictionary
popitem(): <item>	Removes and returns an arbitrary key value pair
clear()	Removes all key value pairs from the dictionary
keys(): <keys>	Returns a dict_keys object (effectively a list of keys)
values(): <values>	Returns a dict_values object (effectively a list of values)
items(): <items>	Returns a dict_items object (effectively a list of

Method	Description
	tuples, where each tuple comprises a key and a value)
copy(): <dict>	Returns a shallow copy of the dictionary

Creation

As described previously, a dictionary literal comprises one or more comma separated values surrounded by curly braces, with each key value pair delimited by a colon.

```
$ >>> my_dict = {'name': 'David', 'age': 30}
```

If your keys are strings, as is often the case, then using the built-in dict function simplifies the process of dictionary creation.

```
$ >>> my_dict = dict(name='David', age=30)
$ >>> my_dict
$ {'name': 'David', 'age': 30}
```

Accessing Values

A dictionary value is accessed by specifying its key between square brackets.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> account_name = account['name']
$ >>> account_name
$ 'Smith'
```

Alternatively, the **get** method may be used to access a dictionary value.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> account_name = account.get('name')
$ >>> account_name
$ 'Smith'
```

The get method may be passed a default value in the event the given key does not exist.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> account_credit_limit = account.get('limit', 500.0)
$ >>> account_credit_limit
$ 500.0
```

If the key passed to the get method exists in the dictionary the default value is ignored.

Updating

A dictionary value is updated by assigning a new value for a given, existing key.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> account['balance'] = 200.0
$ >>> account
$ {'number': 1234, 'name': 'Smith', 'balance': 200.0}
```

Adding

A dictionary value is added by assigning a new key value pair.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> account['limit'] = 750.0
$ >>> account
$ {'number': 1234, 'name': 'Smith', 'balance': 100.0, 'limit': 750}
```

The **setdefault** method is used to access a value for a given key or, in the event the key does not exist in the dictionary, add a new key value pair.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> account.setdefault('number')  # 2nd arg is optional
$ 1234
$ >>> account.setdefault('name', 'Jones')  # 2nd arg is ignored here
$ 'Smith'
$ >>> account.setdefault('limit', 500.0)
$ >>> account
$ {'number': 1234, 'name': 'Smith', 'balance': 100.0, 'limit': 500.0}
```

The **update** method (a bad choice of name in our view) is used to add multiple key value pairs to a dictionary. The update method accepts a collection of collections, each with two elements – a key and a value.

```
$ >>> account = {}
$ >>> account.update([('number', 1234), ('name', 'Smith')])
$ >>> account
$ {'number': 1234, 'name': 'Smith'}
```

In the example above we've passed the update method a list of tuples, but we might have passed, for example, a tuple of lists.

Removing

The **pop** method is used to remove and return the value for the given key. Both key and value are removed from the dictionary, but only the value is returned.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> account_name = account.pop('name')
$ >>> account_name
$ 'Smith'
$ >>> account
$ {'number': 1234, 'balance': 100.0}
```

The **popitem** method is used to remove and return an arbitrary key value pair. The return value is a tuple comprising both key and value.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> removed_pair = account.popitem()
$ >>> removed_pair
$ ('balance', '100.0)
$ >>> account
$ {'number': 1234, 'name': 'Smith'}
```

The **clear** method is used to remove all key value pairs from a dictionary.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> account.clear()
$ >>> account
$ {}
```

The **del** keyword is used to delete references to objects, and so may be used to remove a key value pair from a dictionary.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> del account['name']
$ >>> account
$ {'number': 1234, 'balance': 100.0}
```

The del keyword may also be used to remove the entire dictionary.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> del account
$ >>> account
$ NameError: name 'account' is not defined
```

Iterating

For loops are used to iterate over dictionaries. A for statement takes the following form:

```
for <element> in <collection>:
    <statement(s)>
```

The block is executed once for each key value pair in the collection.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> for key in account:
$ ...     print(key)
$ ...
$ number
$ name
$ balance
```

NOTE

By default, each iteration generates the key from the key value pair, and not the value. Furthermore, the variable to which each subsequent element is assigned (key in this case) need not have been declared outside of the loop.

As an explicit alternative to the above, the **keys** method may be used to iterate over the keys of a dictionary.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> for key in account.keys():
$ ...     print(key)
$ ...
$ number
$ name
$ balance
```

The **values** method is used to iterate over the values of a dictionary.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> for value in account.values():
$ ...     print(value)
$ ...
$ 1234
$ Smith
$ 100.0
```


The **items** method is used to iterate over both the key and values of a dictionary. It returns a dict_items object which is effectively a list of tuples.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> for key, value in account.items():
$ ...     print(key, value)
$ ...
$ number 1234
$ name Smith
$ balance 100.0
```

Membership Testing

The **in** operator is used to test for membership, that is, to test whether or not a given key or value is present in the dictionary.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> 'name' in account  # test for the presence of a key
$ True
$ >>> 'Smith' in account.values()
$ True
```


Sorting

Unlike a list, a dictionary object does not have a sort method. This is probably because a dictionary might be sorted by its keys or its values. The built-in sorted function, however, may be used to create a sorted list from an existing dictionary.

Passing a reference to a dictionary object to the sorted function yields a sorted list of keys (recall that iterating over a dictionary object yields its keys, and not its values.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> sorted_keys = sorted(account)
$ >>> sorted_keys
$ ['balance', 'name', 'number']
```

NOTE

The example above works because each of the dictionary's keys is of the same type – string – and can therefore be compared against one another. See below for information about what to do in the event the keys are not of the same type.

We might use the dictionary object's values method to sort the dictionary's values, but…

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> sorted_values = sorted(account.values())
$ TypeError: '<' not supported between instances of 'str' and 'int'
```

…an error is raised because the values in the account dictionary are not of the same type – the number is an integer, the name is a string, and the balance is a float – and, because Python is a strongly typed language, objects cannot be coerced from one type into another. We could overcome this problem by assigning the built-in str function to the sorted function's key parameter.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> sorted_values = sorted(account.values(), key=str)
$ >>> sorted_values
$ [100.0, 1234, 'Smith']
```

The sorted function's key parameter has nothing to do with dictionary keys. It is used to specify what thing should be compared to facilitate the sorting. The str function assigned to the sorted function's key parameter in the example above dictates that it should be the string version of each of the dictionary's values that should be compared.

Each of the approaches thus far described produces a list of either sorted keys or sorted values, but in most cases you will probably want to produce a list of key value pairs, sorted either by key or by value. The dictionary object's items method enables us to do just that.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> sorted_pairs = sorted(account.items())
$ >>> sorted_pairs
$ [('balance', 100.0), ('name', 'Smith'), ('number': 1234)]
```

The resultant list is sorted by key. The sorted function's key parameter might be used to sort the dictionary by its values, instead. This can be achieved by assigning a lambda function to the key parameter as illustrated below

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> sorted_pairs = sorted(account.items(), key=lambda i : str(i[1]))
$ >>> sorted_pairs
$ [('balance', 100.0), ('number': 1234), ('name', 'Smith')]
```

The lambda function is beyond the scope of this course.

Copying

Consider the following:

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> copy = account
```

Does the copy variable store a copy of the account dictionary? No. It stores a reference to the same dictionary object as does the account variable. Making changes to the dictionary using the copy variable will affect the account variable, after all, each variable references the same object.

```
$ >>> copy['balance'] = 200.0
$ >>> copy
$ {'number': 1234, 'name': 'Smith', 'balance': 200.0}
$ >>> account
$ {'number': 1234, 'name': 'Smith', 'balance': 200.0}
```

The **copy** method is used to create a shallow copy of a dictionary.

```
$ >>> account = {'number': 1234, 'name': 'Smith', 'balance': 100.0}
$ >>> copy = account.copy()
$ >>> copy['balance'] = 200.0
$ >>> copy
$ {'number': 1234, 'name': 'Smith', 'balance': 200.0}
$ >>> account
$ {'number': 1234, 'name': 'Smith', 'balance': 100.0}
```

Calling the copy method results in the creation of a new dictionary object.

So why is it called a shallow copy? Whilst the copy method creates a new dictionary object, it simply copies all of the references from the original dictionary into the new dictionary. If those references refer to mutable objects, then making a change to the elements in one dictionary will affect the other. Consider a dictionary that contains a (mutable) dictionary.

```
$ >>> client_1 = {
$ ...     'name': 'StayAhead Training',
$ ...     'address': {
$ ...         'line1': '6 Long Lane',
$ ...         'line2' : 'Barbican',
$ ...         'city': 'London'
$ ...     }
$ ... }
$ >>> client_2 = client1.copy()
$ >>> client_2['name'] = 'Pret A Manger'
$ >>> client_2['address']['line1'] = '1 Long Lane'
$ >>> client_1['name']
$ 'StayAhead Training'
$ >>> client_1['address']['line1']
$ '1 Long Lane'  # Oops! should not have been changed
```

Each of the client_1 and client_2 variables reference separate dictionary objects. Changing the name of client 2 has no affect on client 1 because name is an immutable string. The interpreter creates a new string object for 'Pret A Manger' and then changes the name key reference. Changing the address line 1 of client 2, however, does affect client 1 because address is a mutable dictionary. The interpreter changes the address object and not the address reference. Care must be taken when copying a dictionary with mutable values.

NOTE

The copy module has a function named deepcopy which may be used to create a deep copy of a dictionary.

```
import copy
...
client_2 = copy.deepcopy(client_1)
```

The resultant dictionary is completely detached from the original. That is, the deepcopy method copies each mutable value and each of its mutable values and so on.

Built-in Functions

There are a number of built-in functions applicable to dictionaries. These are tabled below.

Built-in function	Description
len(<dict>): <number>	Returns the number of items in the dictionary
any(<dict>): <boolean>	Returns True is any of the keys in the dictionary are True (or would be true following conversion e.g. bool(<key>))
all(<dict>): <boolean>	Returns True is all of the keys in the dictionary are True (or would be true following conversion e.g. bool(<key>))
sorted(<dict>): <sorted_list>	Returns a sorted list based on the dictionary's keys

Exercises

Dictionaries

In this exercise you will create a script that emulates a bank system. You will hard-code a dictionary of accounts and then prompt the user to search by account number. The main purpose of this exercise is to demonstrate the random access nature of a dictionary. That is, a dictionary does not require that we know the position of a given value in the collection in order to retrieve it, nor must we iterate over the dictionary to do so.

1. Create a script named ch11_dictionaries.py.
2. Copy and paste the following dictionary of accounts:

```
accounts = {
    1234: {
        "number": 1234,
        "name": "Smith",
        "balance": 500.0
    },
    2345: {
        "number": 2345,
        "name": "Jones",
        "balance": -150.0
    },
    3456: {
        "number": 3456,
        "name": "Wilson",
        "balance": 2000.0
    },
    4567: {
        "number": 4567,
        "name": "Thompson",
        "balance": 275.0
    },
    5678: {
        "number": 5678,
        "name": "Davis",
        "balance": 100.0
    }
}
```

   Note that this is a dictionary of dictionaries. Each key is an account number, and each value is an account dictionary.

3. Code an infinite loop.
4. Inside the loop…
   a. Prompt the user to input an account number and assign it to a variable named acc_num.
   b. Convert acc_num to an int.

   c. Access the account associated with acc_num and assign it to a variable named acc.
   d. If acc is not None…
      i. Print the name and balance of the account referenced by acc to the console.
   e. Else, print an account not found message to the console.
   f. Prompt the user to input y to continue searching/filtering, or n to stop, and capture it in a variable named more.
   g. If more is n, break out of the loop.
5. Use the built-in sorted function to create a list of accounts sorted by the account number key in descending order and assign it to a variable named sorted_accounts.
6. Code a for loop that iterates over the list of sorted accounts.
7. Inside the loop print the number, name and balance of each account to the console.

On execution of ch11_dictionaries.py, the output should be as or similar to below:

Enter the account number: 1234

Smith 500.0

More? (y/n): y

Enter the account number: 9999

Account not found

More? (y/n): n

Wilson 2000.0

Smith 500.0

Thompson 275.0

Davis 100.0

Jones -150.0

5678 Davis 100.0

4567 Thompson 275.0

3456 Wilson 2000.0

2345 Jones -150.0

1234 Smith 500.0

CHAPTER 12

Strings

Introduction

A string is a collection of characters. In Python, a string literal may be surrounded by either single or double quotes.

```
$ >>> my_string = 'Hello'
$ >>> type(my_string)
$ <class 'str'>

$ >>> my_string = "Hello"
$ >>> type(my_string)
$ <class 'str'>
```

A string object's properties are tabled below.

Property	Yes/No	Description
Mutable?	No	A string cannot grow nor shrink, nor can its contents be changed.
Key value pairs?	No	N/A
Ordered?	Yes	Each character in a string is indexed, with the first character at index zero. Iterating over a string yields the characters in order from index zero to index (length – one).
Duplicates?	Yes	N/A

The Escape Character

The escape character \ (backslash) is used to embed special characters into a string literal.

```
$ >>> greeting = 'Hello, what\'s your name?'
$ >>> greeting
$ "Hello, what's your name?"
```

NOTE

When a string variable is evaluated in interactive mode, it is written to standard out surrounded by either single or double quotes. When a string variable is written to standard out using the built-in print function, it is not surrounded by quotes.

Tabled below are some of the most commonly used special characters.

Special character	Description
\'	Single quote
\"	Double quote
\n	Line feed (new line)
\t	Tab space
\\	Backslash

Unicode characters may be embedded using the form \u<code>.

```
$ >>> registered_symbol = '\u00ae'
$ >>> registered_symbol
$ '®'
```

Raw strings

Sometimes you won't want the backslash to signal a special character.

```
$ >>> path = 'c:\users\david'
$ >>> print(path)
$ SyntaxError: (unicode error) ...
```

In this case the interpreter expects a Unicode code to follow \u.

A raw string is one in which the \ (backslash) has no special meaning. Raw strings are prefixed with the letter r.

```
$ >>> path = r'c:\users\david'
$ >>> print(path)
$ c:\users\david
```


Triple-quotes

A triple-quoted string is one in which white space is interpreted literally. Consider the following conventional approach:

```
$ >>> my_string = "Hello\nworld"
$ >>> print(my_string)
$ Hello
$ world
```

The same thing can be achieved by surrounding the string with triple-quotes and typing Enter rather than including the line feed special character.

```
$ >>> my_string = """Hello
$ ... world"""
$ >>> print(my_string)
$ Hello
$ world
```

Triple quotes can also be used to insert tab spaces into a string.

Concatenation

In an expression where both operands are strings, the + (addition) operator has the effect of concatenating the strings (creating a new string composed of the existing ones).

```
$ >>> greeting, subject = "Hello", "world"
$ >>> my_string = greeting + " " + subject
$ >>> my_string
$ 'Hello world'
$ >>> greeting
$ 'Hello'
$ >>> subject
$ 'world'
```

The compound assignment operator may be used when you want to change an existing variable. Consider the following variation on the above example:

```
$ >>> greeting, subject = "Hello", "world"
$ >>> greeting += " " + subject
$ >>> greeting
$ 'Hello world'
$ >>> subject
$ 'world'
```

It appears as though the string object referenced by the greeting variable has changed, but strings are immutable. In fact, a new string object is created and the greeting variable is changed such that it references the new string object.

The **join** method is used to concatenate a string with each of the strings in a collection.

```
$ >>> my_list = ["Hello", " ", "world"]
$ >>> my_string = "".join(my_list)
$ >>> my_string
$ 'Hello world'
```

NOTE

In the example above the join method is called on a string literal. This is not uncommon and is perfectly valid.

As Python is a strongly typed language, it is not possible concatenate a string object with, say, an integer object.

```
$ >>> age = 30
$ >>> my_string = "I am " + age + " years old"
$ TypeError: must be str, not int
```

To overcome this problem, convert the non-string object to a string using the built-in str function.

```
$ >>> age = 30
$ >>> my_string = "I am " + str(age) + " years old"
```


Formatting

Concatenation of strings can get messy, particularly when string literals are concatenated with string variables for the purposes of padding and alignment.

Since Python version 3.6 formatting strings has gotten easier with the introduction of f-Strings (formatted string literals). Formatting options prior to v3.6 required either the use of placeholders or the string **format** method.

The basics

An f-String is a string literal that is prefixed with the letter f (upper or lower case) and has one or more sets of curly braces, each one enclosing an expression to be evaluated at runtime. Consider the following example:

```
$ >>> name, age = "David", 30
$ >>> print(f"My name is {name} and I am {age} years old.")
$ My name is David and I am 30 years old.
```

The curly braces may enclose any valid expression including function calls.

```
$ >>> name, age = "David", 30
$ >>> def birth_year(age):
$ ...     return 2019 - age  # dodgy logic!
$ ...
$ >>> print(f"My name is {name} and I was born in {birth_year(age)}")
$ My name is David and I was born in 1989
```

Padding

Padding is achieved by adding a : (colon) with a number (representing the column width) at the end of the expression in curly braces. Consider the following example:

```
$ >>> people = [("David", 30), ("Sarah", 21), ("Tom", 9)]
$ >>> for name, age in people:
$ ...     print(f"{name:6} {age:3}")
$ ...
$ David   30
$ Sarah   21
$ Tom      9
```

NOTE

By default, strings are left aligned and numbers are right aligned.

For floating point numbers, we must add two numbers after the colon delimited by a point and followed by the letter f (for float). The first number is the total width including the decimal point and the fractional part, and the second number is the width of the fractional part.

```
$ >>> products = [("Lamp", 19.95), ("Mug", 3.90), ("TV", 499.0)]
$ >>> for desc, price in products:
$ ...     print(f"{desc:6} {price:6.2f}")
$ ...
$ Lamp   19.95
$ Mug     3.90
$ TV    499.00
```

Alignment

The alignment of a value inside curly braces may be set to left-, right-, or centre-align by prefixing the column width with one of either < (less than symbol), > (greater than symbol), or ^ (caret).

```
$ >>> people = [("David", 30), ("Sarah", 21), ("Tom", 9)]
$ >>> for name, age in people:
$ ...     print(f"|{name:^9}|{age:>3}|")
$ ...
$ |  David  | 30|
$ |  Sarah  | 21|
$ |   Tom   |  9|
```


Indexing

As a string is, effectively, a list of characters, each character may be accessed by specifying its index between square brackets, just like a list.

```
$ >>> name = "David"
$ >>> second_char = name[1]
$ >>> second_char
$ 'a'
```

Python supports negative indexing too. The index -1 yields the last character of the string, -2 the second last character and so on.

```
$ >>> name = "David"
$ >>> second_to_last_char = name[-2]
$ >>> second_to_last_char
$ 'i'
```

Any attempt to access a non-existent list element will result in the raising of an IndexError.

```
$ >>> name = "David"
$ >>> name[5]
$ IndexError: string index out of range
```

The **index** method is used to find the index of a given substring.

```
$ >>> name = "David"
$ >>> index_of_v = name.index("v")
$ >>> index_of_v
$ 2
```

A ValueError is raised in the event the given substring does not exist in the list.

Slicing

Slicing provides for the extracting of a substring into a new string but does not change the original string. It requires the addition of the : (colon) between the square brackets. The number to the left of the colon is the beginning index (inclusive), and the number to the right of the colon is the ending index (exclusive).

```
$ >>> food = "meat and veg"
$ >>> food[5:8]  # indices 5 through 7
$ 'and'
```

Either or both indices may be omitted. If the beginning index is omitted, the first character of the subset is that at index 0 (zero). If the ending index is omitted, the last character of the subset is that at index (length – 1).

```
$ >>> food = "meat and veg"
$ >>> food[:4]  # indices 0 through 3
$ 'meat'

$ >>> food = "meat and veg"
$ >>> food[9:]  # indices 9 through (length - 1)
$ 'veg'
```

The indices may be negative.

```
$ >>> food = "meat and veg"
$ >>> food[-7:-4]  # indices (length - 7) through (length - 5)
$ 'and'
```

The step value is 1 (one) by default, and so the following yields an empty list.

```
$ >>> food = "meat and veg"
$ >>> food[-5:4]  # invalid - can't reach 4 from -5 with a step of 1
$ ''
```

The solution is to specify the step.

```
$ >>> food = "meat and veg"
$ >>> food[-5:4:-1]
$ 'dna'
```

Note the addition of a second colon and a third number. The third number is the step. Setting a negative step makes it easy to reverse the elements in a list.

```
$ >>> food = "meat and veg"
$ >>> food[::-1]
$ 'gev dna taem'
```


NOTE

To summarise, slicing creates a substring from an existing string and takes the form:

```
<string>[<begin_index_incl>:<end_index:excl>:<step>]
```

All of the parts are optional. The begin index is 0 (zero) by default, the end index is (length – 1) by default, and the step is 1 (one) by default.

Iterating

For loops are used to iterate over strings. A for statement takes the following form:

```
for <element> in <collection>:
    <statement(s)>
```

The block is executed once for each character in the string.

```
$ >>> name = "David"
$ >>> for character in name:
$ ...     print(character)
$ ...
$ D
$ a
$ v
$ i
$ d
```

NOTE

The variable to which each subsequent element is assigned (character in this case) need not have been declared outside of the loop.

The for loop yields only the value of each character, and not its index. When passed a string, the built-in **enumerate** function returns an enumerate object which is, effectively, a list of tuples. Each tuple comprises the character's index and its value.

```
$ >>> name = "David"
$ >>> for index, character in enumerate(name):
$ ...     print(index, character)
$ ...
$ 0 D
$ 1 a
$ 2 v
$ 3 i
$ 4 d
```


Membership Testing

The **in** operator is used to test for membership, that is, to test whether or not a given character is present in the string.

```
$ >>> name = "David"
$ >>> "a" in name
$ True
$ >>> "f" not in name
$ True
```

Other String Methods

Remember that string objects are immutable. Any method that purports to change the string does not, but rather it will create a new string object and return a reference to it.

NOTE

In the method descriptions that follow, return values have been included only where we think it's not obvious.

Case

Method	Example
	`$ >>> s = "Hello world"`
islower()	`$ >>> s.islower()` `$ False`
istitle()	`$ >>> s.istitle()` `$ False`
isupper()	`$ >>> s.isupper()` `$ False`
lower()	`$ >>> s.lower()` `$ 'hello world'`
title()	`$ >>> s.title()` `$ 'Hello World'`
upper()	`$ >>> s.upper()` `$ 'HELLO WORLD'`
capitalize()	`$ >>> s.capitalize()` `$ Hello world`
swapcase()	`$ >>> s.swapcase()` `$ 'hELLO WORLD'`

Alignment

Method	Example
	`$ >>> s = "Hello"`
ljust(<width>, <fill_char>)	`$ >>> s.ljust(11, "-")` `$ 'Hello------'`
center(<width>, <fill_char>)	`$ >>> s.center(11, "-")` `$ '---Hello---'`
rjust(<width>, <fill_char>)	`$ >>> s.rjust(11, "-")` `$ '------Hello'`

White space

Method	Example

expandtabs()	`$ >>> s = "Desc\tCost\tQty"`
	`$ >>> s.expandtabs()`
	`$ 'Desc    Cost    Qty'`

lstrip()	`$ >>> s = "  Hello  "`
	`$ >>> s.lstrip()`
	`$ 'Hello  '`

rstrip()	`$ >>> s = "  Hello  "`
	`$ >>> s.rstrip()`
	`$ '  Hello'`

strip()	`$ >>> s = "  Hello  "`
	`$ >>> s.strip()`
	`$ 'Hello'`

isspace()	`$ >>> s = " \t\n\r"`
	`$ >>> s.isspace()`
	`$ True`

NOTE

The lstrip, rstrip, and strip methods described below may be passed an optional string representing the charater sequence to be stripped from the left/right/both ends of the string, for example:

```
$ >>> s = "***Hello***"
$ >>> s.strip("*")
$ 'Hello'
```

Numeric or not

Method	Example

isalnum()	`$ >>> s = "abc123"`
	`$ >>> s.alnum()`
	`$ True`

isalpha()	`$ >>> s = "abc"`
	`$ >>> s.isalpha()`
	`$ True`

isdecimal()
**any character used to form numbers in base 10*
	`$ >>> s = "123"`
	`$ >>> s.isdecimal()`
	`$ True`

isdigit()
**decimal + digits that need special handling*
	`$ >>> s = "123"`
	`$ >>> s.isdigit()`
	`$ True`

isnumeric()
**digit + others e.g. fractions etc.*
	`$ >>> s = "123"`
	`$ >>> s.isnumeric()`
	`$ True`

Curiously, perhaps, none of the methods tabled above can be used to test if a string represents a valid floating point number. For example, none will return True for string comprising a – (minus sign) or . (decimal point).

To test if a string represents a valid floating point number, you should try to convert it to float. This requires exception handling, which is covered in detail in chapter 16, but we'll describe just enough of the basics here to enable you to do the test.

Attempting to convert a string to a number may raise an error if, for example, the string comprises non-numeric characters. If an error is raised, you will know that the string does not represent a valid floating point number and so your test will have failed. If, on the other hand, no error is raised, you will know that the string does indeed represent a valid floating point number and so your test will have passed. Consider the following:

```
$ >>> my_string = input("Enter a valid floating point number: ")
$ >>> try:
$ ...     float(my_string)
$ ...     print("Yes, it\'s valid!")  # will be skipped if error
$ ... except:
$ ...     print("No, it\'s invalid")  # will be skipped if no error
$ ...
```

So, the user will be prompted to enter a valid floating point number. The try block represents some code that may result in the raising of an error. In this case, it's the line `float(my_string)` that may cause problems.

If the value entered can be converted to a float object then no error will be raised, the "Yes it's valid!" message will be written to the console, and the code in the except block will be skipped.

If the value entered cannot be converted to a float object then an error will be raised, and execution will skip the remaining lines in the try block and pass immediately to the except block, where the "No it's invalid" message will be written to the console.

You could create your own function to simplify the test in future.

```
$ >>> def isfloat(a_string):
$ ...     try:
$ ...         float(a_string)
$ ...         return True
$ ...     except:
$ ...         return False
$ ...
```


Search

Method	Example
	`$ >>> s = "Hello world"`
count(<substring>)	`$ >>> s.count("o")` `$ 2`
startswith(<substring>)	`$ >>> s.startswith("He")` `$ True`
endswith(<substring>)	`$ >>> s.endswith("d")` `$ True`
find(<substring>): <index> **returns -1 if not found*	`$ >>> s.find("o")` `$ 4`
index(<substring>): <index> **raises error if not found*	`$ >>> s.index("o")` `$ 4`
rfind(<substring>): <index> **returns -1 if not found*	`$ >>> s.rfind("o")` `$ 7`
rindex(<substring>): <index> **raises error if not found*	`$ >>> s.rfind("o")` `$ 7`

Split

Method	Example

split(<delimiter>): <list>	`$ >>> s = "Jan,Feb,Mar,Apr,May"`
	`$ >>> s.split(",")`
	`$ ['Jan', 'Feb', 'Mar', 'Apr', 'May']`

splitlines(): <list>	`$ >>> s = "Jan\nFeb\nMar\nApr\nMay"`
	`$ >>> s.splitlines()`
	`$ ['Jan', 'Feb', 'Mar', 'Apr', 'May']`

Replace

Method	Example

replace(<substring1>, <substring2>)	`$ >>> s = "Hello world"`
	`$ >>> s.replace("world", "Python")`
	`$ 'Hello Python'`

Encode/decode

Method	Example

encode(<charset>): <bytes>	`$ >>> s = "£"`
	`$ >>> s.encode("utf-8")`
	`$ b'\xc2\xa3'`

decode(<bytes>)	`$ >>> b = b'\xc2\xa3'`
	`$ >>> b.decode("utf-8")`
	`$ '£'`

Exercises

Strings

In this exercise you will create a script that emulates the parsing of a CSV file. You will hard-code a string comprising account records and then parse it into a dictionary of dictionaries ready for processing.

1. Create a script named ch12_strings.py.
2. Copy and paste the following string:

```
file_contents = """
1234,smith     ,500.0
2345,jones     ,-150.0
3456,wilson    ,2000.0
4567,thompson  ,275.0
5678,davis     ,100
"""
```

3. Declare a variable named accounts and assign it an empty dictionary.
4. Split the file contents delimiting on new line characters, and capture the resultant list in a variable named lines.
5. Code a for loop that iterates over the lines list.
6. Inside the loop…
   a. If the line is not an empty string …
      i. Split the line delimiting on commas, and capture the resultant list in a variable named parts.
      ii. Convert the first part (account number) to an int and assign it to a variable named acc_num.
      iii. Strip the second part (account name) of it whitespace and convert it to title case, then assign it to a variable named acc_name.

      As strings are immutable, most string methods return a reference to a new object. This means that string methods may be chained together, e.g.:

```
$ >>> s1 = "Hello word"
$ >>> s2 = s1.replace("world", "Python").upper()
$ >>> s2
$ 'Hello Python'
```

      iv. Convert the third part (account balance) to a float and assign it to a variable named acc_balance.
      v. Assign a new key value pair to the accounts dictionary where the key is the accout number, and the value is a dictionary comprising the account number, name, and balance.

7. Code a for loop that iterates over the values in the accounts dictionary.
8. Inside the loop, print the account dictionary to the console.

On execution of ch12_strings, the output will be as below:

```
{'number': 1234, 'name': 'Smith', 'balance': 500.0}
{'number': 2345, 'name': 'Jones', 'balance': -150.0}
{'number': 3456, 'name': 'Wilson', 'balance': 2000.0}
{'number': 4567, 'name': 'Thompson', 'balance': 275.0}
{'number': 5678, 'name': 'Davis', 'balance': 100.0}
```


CHAPTER 13

Modules and Packages

Module Introduction

A module is a Python script that forms one part of a whole. Large programs are often broken up into modules, with each module exposing some data and/or functions for use by other parts. When compared with its monolithic equivalent, a modular program should be easier to maintain and to test.

Even if you're not involved in contributing to a large and complex program, and instead write only small-ish scripts, you will inevitably use existing modules to help you do things like pattern matching and data processing.

Modules might be categorised as follows:

- Those that you create
- Those that are bundled with the interpreter
- Those that must be installed

There are a *lot* of modules bundled with the interpreter (the standard library) that you can use out-of-the-box. These include modules dealing with:

- Compression
- Date and time
- Mathematical functions
- Encryption
- Data parsing
- Unit testing
- Email
- File comparison
- Internationalisation
- Multithreading/processing
- The operating system
- GUIs

See https://docs.python.org/3/library/ for a complete list.

There are many, many more modules available for installation using pip (Python's package manager). Some of the most popular third-party modules deal with things like:

- HTTP requests
- Image processing
- Web scraping
- Scientific computing
- Data analysis and plotting
- Object relational mapping
- Web apps

See https://pythontips.com/2018/06/03/top-14-most-famous-python-libraries-frameworks/ for more information about these and lots more.

Some Built-in Modules

Let us look at a few built-in modules. These are modules which are included in a typical installation.

The math module

The math module contains functions of real valued numbers in the following areas

1. Number theoretic,
2. Power and logarithmic,
3. Trigonometric,
4. Angular conversion,
5. Hyperbolic,
6. Special and
7. Constants

Let us look at a few functions.

```
>>> import math
>>> math.ceil(4.123) # returns the smallest integer equal to or greater
than arg
>>> math.floor(-4.5), math.floor(4.5) #returns the largest integer less
than or equal to arg
(-5, 4)
>>> math.trunc(4.5) # returns the integer part
4
>>> math.factorial(5) # returns the factorial of arg
120
>>> math.sqrt(5) # returns the square root of arg
2.23606797749979
>>> math.hypot(3,4) # returns sqrt of sum of squares of args
5.0
```

For complex number support look up the `cmath` module.

The random module

The random module contains functions for pseudo-random integer and real valued distributions. It also contains functions for random selection or sampling from sequences. Almost all functionality depends on the `random()` function which generates a random number in the range [0.0, 1.0).

Let us look at a few functions.

```
>>> import random
>>> random.seed(42) # set the seed for reproducability
>>> random.randint(0,10) # select number from [1,10]
10
>>> random.random() # generate random floar in range [0.0, 1.0)
0.11133106816568039
>>> names = ['Jane','Fatima','David','George','Ali']
>>> random.choice(names) # choose a name from names
'David'
>>> random.sample(names, 2) # sample from names 2 unique items
['Fatima', 'Ali']
>>> random.shuffle(names) # shuffle names
>>> names
['Ali', 'George', 'David', 'Jane', 'Fatima']
```

See the `statistics` module for most commonly used statistical functions.

The platform module

The platform module contains functions that examine the underlying platform. This includes information such as the operating system, the python interpreter implementation and versions e.t.c.

Let us look at a few functions.

```
>>> import platform
>>> platform.architecture() # bit and file format for executables
('64bit', 'ELF')
>>> platform.platform() # as much data on underlaying platform
'Linux-5.17.15-76051715-generic-x86_64-with-glibc2.35'
>>> platform.machine() # machine type
'x86_64'
>>> platform.processor() # processor name if available else same as
machine()
'x86_64'
>>> platform.system() # Operating System name
'Linux'
>>> platform.version() # systems release version
'#202206141358~1655919116~22.04~1db9e34 SMP PREEMPT Wed Jun 22 19'
```

```
>>> platform.python_implementation()
'CPython'
>>> platform.python_version()
'3.10.4'
>>> platform.python_version_tuple()
('3', '10', '4')
```

The dir() and help() functions

Two helpful functions in inspecting modules, objects and functions are `dir()` and `help()`. The `dir()` function can be used to inspect the contents of a module. Sor for example

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc', 'ulp']
```

yields a list of all objects (functions, classes e.t.c.) in the math module. In addition to functions such as `'ceil'`, which we have previously used, we see many additional functions. We can choose a function we want to investigate further and call

```
>>> help(math.ceil)
```

which 'opens' a new window with information on the functions shown below.

```
Help on built-in function ceil in module math:

ceil(x, /)
    Return the ceiling of x as an Integral.

    This is the smallest integer >= x.
(END)
```

This is the online documentation on the function.

Creating and Using Modules

Creating and using modules is easy. Consider the following script named mymod.py.

```
# mymod.py
my_string = "Hello world"
def to_miles(kms):
    return kms * 0.621371
```

NOTE

According to PEP (Python Enhancement Proposal) 0008, module names should have short, all-lowercase names.

Assuming mymod.py is saved in the right location (more on that later), its data and functions can be accessed elsewhere. Consider the following:

```
$ >>> import mymod  # import is a keyword
$ >>> print(mymod.my_string)
$ Hello world
$ >>> print(mymod.to_miles(50))
$ 31.0686
```

NOTE

The module name is the script name excluding the .py extension.

The __pycache__

After running the script a *__pycache__* directory appears inside the directory containing the script. This directory, created automatically by the interpreter, contains a cache of bytecode files, one for every imported module. The files have a name like `mymod.cpython-36.pyc`, reflecting the name of an imported module, the python implementation and version used in running the program. The *.pyc* stands for python compiled code. The point of these files is to speed up loading of the modules on subsequent runs.

The Module Search Path

When the Python interpreter encounters an import statement, it looks for the associated script in a number of locations as follows:

1. The current directory
2. The directories associated with the PYTHONPATH environment variable
3. The directories associated with the Python installation

Ironically, you must import the built-in module, sys, to list the directories that comprise the module search path.

```
$ >>> import sys
$ >>> for path in sys.path:
$ ...     print(path)
$ ...
$
$ /Library/Frameworks/Python.framework/Versions/3.6/lib/python36.zip
$ /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6
$ /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/lib-dynload
$ /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages
```

NOTE

The first path is an empty string representing the current directory. Naturally, the list of paths on your system will differ from those listed here.

Importing Modules

The data and functions declared in a module are made available using the import statement which takes the following form:

```
import <module_name(s)>
```

NOTE

You can import multiple modules at once importing a comma separated list.

Importing a module in this way does not make the contents of said module accessible directly. Consider the following:

```
$ >>> import mymod
$ >>> print(my_string)
$ NameError: name 'my_string' is not defined
```

Instead, the import statement creates a separate namespace, in the same way the declaration of a function creates a namespace for local variables. Thus, to access a variable/function/class declared in the imported module, it must be prefixed with the module name using the dot notation.

```
$ >>> import mymod
$ >>> print(mymod.my_string)
$ Hello world
```

Specific names

You may choose to import specific data and functions. An import statement in which you specify the names to be imported takes the following form:

```
from <module_name> import <name(s)>
```

Importing in this way makes the specific data and functions accessible directly.

```
$ >>> from mymod import my_string
$ >>> print(my_string)
$ Hello world
```

There is an inherit danger in doing this, however. If one of the names that you're importing conflicts with an existing name, the existing one will be overwritten.

```
$ >>> my_string = "My name is David"  # will be overwritten by import
$ >>> from mymod import my_string
$ >>> print(my_string)
$ Hello world
```

Wildcard

You can use a wildcard (*) to import all the names in a given module such that each one will be accessible directly.

```
$ >>> from mymod import *
$ >>> print(my_string)
$ Hello world
$ >>> print(to_miles(50))
$ 31.0686
```

We don't recommend doing this, particularly not when dealing with large modules. The scope for naming conflicts is, in our view, too great.

NOTE

Any variable or function whose name begins with an underscore will NOT be imported when the wildcard is used. By convention, a variable or function whose name begins with an underscore is intended for internal use only.

Aliases

You can import specific data and functions and avoid naming conflicts by assigning aliases to the imported names. An import statement in which you specify the names to be imported and assign aliases takes the following form:

```
from <module_name> import <name> as <alias>[, <name> as <alias>, ...]
```

Consider the following:

```
$ >>> my_string = "My name is David"
$ >>> from mymod import my_string as ms
$ >>> print(ms)
$ Hello world
$ >>> print(my_string)
$ My name is David
```

The name, my_string, is not overwritten in this case because the imported name (also my_string) is assigned an alias – ms.

Modules may be assigned aliases too.

```
$ >>> import mymod as m
$ >>> print(m.my_string)
$ Hello world
```


From inside a function

A module, or specific data and functions therein, may be imported from inside a function. This limits the scope of the imported module/names.

```
$ >>> def my_function():
$ ...     import mymod
$ ...     print(mymod.my_string)
$ ...
$ >>> print(mymod.my_string)
$ NameError: name 'mymod' is not defined
```

Since Python version 3 the the import wildcard (*) is not permitted when importing from inside a function.

Executable Modules

Every module is, in fact, executable, but if the module comprises only variable, function, and class declarations, it will do nothing when it is executed. Consider the following:

```
# mymod.py
my_string = "Hello world"
def to_miles(kms):
    return kms * 0.621371

$ python mymod.py
```

The script executes without error, but it does not generate any output. That is, it is not intended to be executed.

Some modules, however, are intended to be executed in addition to their being importable. Consider the following:

```
# mymod.py
my_string = "Hello world"
def to_miles(kms):
    return kms * 0.621371

kms = input("Enter kms to be converted to miles: ")
kms = float(kms)
output = "{:f} kms is {:f} miles".format(kms, to_miles(kms))
print(output)

$ python mymod.py
$ Enter kms to be converted to miles: 50
$ 50 kms is 31.0686 miles
```

Now the mymod module may be executed, resulting in the user being prompted to enter a distance in kms to be converted to miles.

What happens now, however, if we import the module?

```
$ >>> import mymod
$ Enter kms to be converted to miles:
```

The code that follows the declaration of the my_string variable and to_miles function is executed immediately. This is probably not what we want to have happen.

When a module is imported, the interpreter sets its dunder __name__ variable to match the name of the module. When a module is executed, however, the interpreter sets its dunder __name__ variable to '__main__'. This enables you to separate the code that should be executed from the data and functions to be made available on import.

Consider the following variation on the above:

```
# mymod.py
# code to be made available on import
my_string = "Hello world"
def to_miles(kms):
    return kms * 0.621371

# code to be executed
if __name__ == '__main__':
    kms = input("Enter kms to be converted to miles: ")
    kms = float(kms)
    output = "{:f} kms is {:f} miles".format(kms, to_miles(kms))
    print(output)

$ >>> import mymod
$ >>>
```

This time, nothing happens when the module is imported because the dunder __name__ variable is set to match the module name.

```
$ >>> import mymod
$ >>> mymod.__name__
$ 'mymod'
```

Package Introduction

A package is a directory of modules. Packages provide for the organisation of modules into a hierarchical structure. In the same way that modules help to avoid data and function naming conflicts, so do packages help to avoid module naming conflicts.

To make use of packages, code your modules inside a directory in one of the directories in sys.path. Let's assume the following filesystem:

```
- sys.path/
  - mypackage/
    - mymod.py
```

NOTE

Like module names, package names should have short, all-lowercase names.

The package name must now form part of the import statement using the dot notation.

```
$ >>> import mypackage.mymod
```

Of course, this requires that the data and functions in the imported module must be prefixed with both package and module name.

```
$ >>> import mypackage.mymod
$ >>> print(mypackage.mymod.my_string)
$ Hello world
```

Almost all of the same rules that apply to importing modules apply here too. For example, you can import specific names…

```
$ >>> from mypackage.mymod import my_string
$ >>> print(my_string)
$ Hello world
```

…and you can use aliases.

```
$ >>> from mypackage.mymod import to_miles as miles
$ >>> print(miles(50))
$ 31.0686
```

About the only thing you can't do is to import the package on its own (at least not in this context). Whilst syntactically correct, the following does not load the modules inside the specified package into the local namespace.

```
$ >>> import mypackage
$ >>> mypackage.mymod
$ AttributeError: module 'mypackage' has no attribute 'mymod'
```


Package Initialisation

If a script named __init__.py is present in the package directory, then the code therein will be executed when the package (or a module of the package) is imported. This may be used to initialise package-level data.

```
# __init__.py inside mypackage/
config = {"language": "english", "maxrecords": 100}
```

Any module within the package can access the package-level data.

```
# mymod.py inside mypackage/
from mypackage import config
print("Max records:", config["maxrecords"])

$ >>> import mypackage.mymod
$ Max records: 100
```

The __init__.py script may also be used to import modules such that it need not be done in each of the modules in the package. Let's say, for example, that each of the modules in mypackage use the os module, among others. Rather than add the import statements to each of those modules, we could instead add them to the __init__.py script.

```
# __init__.py inside my_package/
print("Importing packages...")
import os
...
print("Done!")

# mymod.py inside mypackage/
import mypackage as p
print("Number of CPUs:", p.os.cpu_count())

$ >>> import mypackage.mymod
$ Importing packages...
$ Done!
$ Number of CPUs: 8
```

Subpackages

A package may contain subpackages to any depth. The import of modules in subpackages is as before, but will require more package names and more dots.
Let's assume the following filesystem:

```
- sys.path/
  - mypackage/
    - mysubpackage/
      - mymod.py
```

The package name *and* subpackage name must now form part of the import statement using the dot notation.

```
$ >>> import mypackage.mysubpackage.mymod
```

And, of course, this requires that the data and functions in the imported module must be prefixed with the package name, subpackage name, and module name.

```
$ >>> import mypackage.mysubpackage.mymod
$ >>> print(mypackage.mysubpackage.mymod.my_string)
$ Hello world
```

A module in one subpackage can access the data and functions in a module of a sibling package using either absolute or relative import. Consider the following filesystem:

```
- sys.path/
  - pkg/
    - sub1/
      - mod1.py
    - sub2/
      - mod2.py
```

The data and functions in mod1 can be accessed from mod2 using an absolute import…

```
# mod2.py
import pkg.sub1.mod1
```

…or a relative one:

```
# mod2.py
import ..sub1.mod1
```

The `..` (two dots) refers to the package one level up.

Installing Packages using PIP

PIP is Python's default package manager and is used to install third-party packages (remember that a package is just a directory of modules).

Since Python version 3.4 PIP is installed by default when you install Python. If you're using an earlier version of Python, see https://pip.pypa.io/en/stable/installing/ for PIP installation instructions.

To download and install a package, execute the following on the command line:

```
$ pip install <package_name>
```

NOTE

Depending on the installation, the name of the PIP executable may or may not include the version number, e.g. pip3.7 as opposed to just pip.

To uninstall a package:

```
$ pip uninstall <package_name>
```

To list the currently installed packages:

```
$ pip list
```

CHAPTER 14

Pattern Matching

Introduction

A regular expression (regex) is a sequence of characters (like a string) that forms a pattern. That pattern is then used to test if a string is a match. Pattern matching is used for, among other things, validation of input data. Consider the following example:

```
$ >>> import re
$ >>> data = input("Enter a four digit number: ")
$ >>> pattern = "^\d{4}$"
$ >>> if re.match(pattern, data):
$ ...     print("It\'s valid!")
$ ... else:
$ ...     print("It\'s invalid")
$ ...
```

The pattern is not a regular string but rather a sequence of characters, all of which in this case, have special meaning. For example, the \d part of the pattern will match any digit, and the {4} part of the pattern is a quantifier. The pattern will match any string comprising exactly four digits.

The set of special characters that can be used in the building of a regular expression is not Python-specific, but the mechanisms for testing strings against regex patterns vary from language to language. This chapter covers both topics. If you're already familiar with regular expressions, then you may want to skip ahead to the Python-specific stuff.

Regex Special Characters

We've divided the set of regex special characters into three groups – metacharacters, sequences, and sets.

Metacharacters

Metacharacters are those interpreted to have special meaning by the regex engine.

Character	Description	Example pattern	Example string	Match
.	Any character (exc. new line)	b.t	bat	Yes
			boat	No
^	The start of the string	^tea	tea please	Yes
			more tea	No
$	The end of the string	tea$	tea please	No
			more tea	Yes
?	Zero or one **of the preceding character*	h.?at	hat	Yes
			heat	Yes
			hihat	No
*	Zero or more **of the preceding character*	yipe*	yip	Yes
			yipe	Yes
			yipeee	Yes
+	One or more **of the preceding character*	no+	n	No
			no	Yes
			nooo	Yes
{n}	Exactly n **of the preceding character*	be{2}p	bep	No
			beep	Yes
|	Either or	hi|bye	hi	Yes
			bye	Yes
()	Capture and group	b(ee)+p	beep	Yes
			beeep	No
			beeeep	Yes
\	Escape **interpret metacharacters literally*	.+\+	Canada+	Yes
			Norway	No
[]	A set of characters	**see the section after next*		

Sequences

A sequence serves to simplify a commonly used pattern.

Character	Description	Example pattern	Example string	Match
\A	The start of the string	\Atea	tea please	Yes
			more tea	No
\Z	The end of the string	tea\Z	tea please	No
			more tea	Yes
\b	The start or end of the word	\btea	teapot	Yes
			protea	No
		tea\b	teapot	No
			protea	Yes
\B	Present, but NOT at the start or end of the word	\Btea	steam	Yes
			protea	Yes
			teapot	No
		tea\B	steam	Yes
			protea	No
			teapot	Yes
\d	A digit	\d	1	Yes
			a	No
\D	NOT a digit	\D	1	No
			a	Yes
\s	A whitespace character *\u0020 is a space	\s	\u0020	Yes
			\t	Yes
			\n	Yes
\S	NOT a whitespace character *\u0020 is a space	\S	\u0020	No
			\t	No
			\n	No
\w	A word character *a-Z, 0-9, and the underscore	\w	a	Yes
			1	Yes
			_	Yes
\W	NOT a word character *a-Z, 0-9, and the underscore	\W	a	No
			1	No
			_	No

Sets

A set is a series of characters enclosed inside square brackets. A character in the string to be tested need only match one of the characters in the series.

Character	Description	Example pattern	Example string	Match
[abc]	Any one of a, b, or c	[abc]	a	Yes
			b	Yes
			c	Yes
			d	No
[a-z]	Any one in the range a-z	[a-z]	m	Yes
			M	No
[^abc]	Any one EXCEPT a, b, or c	[^abc]	a	No
			b	No
			c	No
			d	Yes
[^a-z]	Any one EXCEPT a-z	[^a-z]	m	No
			M	Yes
[A-Za-z]	Any one in the range A-z	[A-Za-z]	m	Yes
			M	Yes

NOTE

In a set, the metacharacters ., ^, $, ?, *, +, {}, |, and () have no special meaning.

The re Module

The re module is a part of the standard library (is bundled with the interpreter) and so may be imported without prior installation. It exposes a number of functions for the purposes of pattern matching.

match

The match function looks for a match at the *start* of the string and will return a match object if found, or None if not.

```
$ >>> import re
$ >>> pattern = "[A-Z]{2}"
$ >>> string_to_test = "EC1A 9HF"
$ >>> result = re.match(pattern, string_to_test)
$ >>> result
$ <sre.SRE_Match object; span=(0, 2), match='EC'>
```

The match object has a number of attributes and methods that can be used to obtain information about the match, for example:

```
$ >>> result.start()  # inclusive index of the start of the match
$ 0
$ >>> result.end()    # exclusive index of the end of the match
$ 2
$ >>> result.span()   # tuple comprising start and end index
$ (0, 2)
```

The match function may be used to test for validity, for example:

```
$ >>> def starts_with_two_upper_case_letters(string_to_test):
$ ...     import re
$ ...     pattern = "[A-Z]{2}"
$ ...     return bool(re.match(pattern, string_to_test))
$ ...
$ >>> string_to_test = "EC1A 9HF"
$ >>> starts_with_two_upper_case_letters(string_to_test)
$ True
```

If your pattern has groups, then the object returned by the match function can be used to extract the value of each group. Consider the following pattern comprising two groups:

```
$ >>> pattern = "(\S+)@(\S+\.\S+)"
```

This is a rudimentary pattern to test for a valid email address. It includes two groups: 1) the part before the @ (the username), and 2) the part after the @ (the domain).

```
$ >>> import re
$ >>> pattern = "(\S+)@(\S+\.\S+)"
$ >>> string_to_test = "david@stayahead.com"
$ >>> result = re.match(pattern, string_to_test)
```

The group method of the match object is passed an index and returns the part of the string that was a match for that group. Note that group 0 (zero) is the part of the string that was a match for the entire pattern. The groups method of the match object returns a tuple of all of the matching parts.

```
$ >>> result.group(0)
$ 'david@stayahead.com'
$ >>> result.group(1)
$ 'david'
$ >>> result.group(2)
$ 'stayahead.com'
$ >>> result.groups()
$ ('david', 'stayahead.com')
```

search

The search function looks for a match *anywhere* in the string and will return a match object if found, or None if not.

```
$ >>> import re
$ >>> pattern = "sea|see"
$ >>> string_to_test = "A sailor went to sea to see what he could see"
$ >>> result = re.search(pattern, string_to_test)
$ >>> result
$ >>> <_sre.SRE_Match object; span=(17, 20), match='sea'>
```

You may have noticed that the resultant match object contains information about the first match only, when in fact the pattern appears three times in the string.

NOTE

Both search and match functions only look for the first instance of a match.

findall

The findall function returns a list of all the matches in the string.

```
$ >>> import re
$ >>> pattern = "sea|see"
$ >>> string_to_test = "A sailor went to sea to see what he could see"
$ >>> result = re.findall(pattern, string_to_test)
$ >>> result
$ >>> ['sea', 'see', 'see']
```

Like the match and search functions, the findall function may be used to test for validity given that, if there are no matches, the function will return an empty list. Remember that, when converted to a Boolean, a non-empty list is True, and an empty list if False.

split

The split function splits the string on each occurrence of the pattern.

```
$ >>> import re
$ >>> pattern = "\d+"
$ >>> string_to_test = "Each12word7is325delimited9by08numbers"
$ >>> result = re.split(pattern, string_to_test)
$ >>> result
$ ['Each', 'word', 'is', 'delimited', 'by', 'numbers']
$ >>> " ".join(result).replace("numbers", "spaces")
$ 'Each word is delimited by spaces'
```

In this example the string comprises words delimited by a random number of digits. The pattern, therefore, matches one or more digits. The last line (a bit of fun) uses the string join method to create a string from the list of words delimited by spaces, and then replaces the word numbers with the word spaces.

sub

The sub function replaces each occurrence of the pattern with a given string. This function takes three arguments: 1) the pattern, 2) the string to replace the occurences of the pattern, and 3) the string to test.

```
$ >>> import re
$ >>> pattern = "\d+"
$ >>> string_to_test = "Each12word7is325delimited9by08numbers"
$ >>> result = re.sub(pattern, " ", string_to_test)
$ >>> result
$ 'Each word is delimited by numbers'
$ >>> result.replace("numbers", "spaces")
$ 'Each word is delimited by spaces'
```

compile

The compile function accepts a pattern and returns a regex object. This object can then be used for matching using its match, search, findall, split, and sub methods. If you intend to match against the same pattern over and over, then doing so using a pattern object is more efficient that calling the various functions on the re module directly.

```
$ >>> import re
$ >>> pattern = "sea|see"
$ >>> string_to_test = "A sailor went to sea to see what he could see"
$ >>> regex = re.compile(pattern)
$ >>> result = regex.findall(string_to_test)
$ >>> result
$ >>> ['sea', 'see', 'see']
```

Note that the findall method is called on the regex object (not on the re module directly) and is passed only the string to test. The pattern is intrinsic to the regex object.

Raw Strings

You might recall from the chapter on Strings that a raw string is one in which the \ (backslash) has no special meaning. This is best demonstrated using Unicode characters. In a regular string, \u indicates that what is to follow is a Unicode. Consider the following:

```
$ >>> copyright = "\u00a9"
$ >>> print(copyright)
$ ©
```

If we prefix the string literal with the letter r, then the backslash has no special meaning and the characters that follow the backslash are interpreted literally.

```
$ >>> copyright = r"\u00a9"
$ >>> print(copyright)
$ \u00a9
```

Consider another example involving the new line character:

```
$ >>> reg_string = "\n"
$ >>> print(reg_string)
$
$
```

And with a raw string:

```
$ >>> raw_string = r"\n"
$ >>> print(raw_string)
$ \n
```

What does all this have to do with pattern matching? Consider the following:

```
$ >>> pattern = ".+\\.+"
```

We want to match one or more of any character (`.+`), then a backslash, then one or more of any character again (`.+`). There are two backslashes because, as we know, backslashes have special meaning in a string and so we must escape the escape character. If we were to use this pattern to test the string `"users\david"` it would NOT find a match.

The problem is that the backslash has special meaning at two levels: 1) at the string level, and 2) at the regex level. We can see how the regex engine will interpret our pattern by passing it to the print function.

```
$ >>> pattern = ".+\\.+"
$ >>> print(pattern)
$ .+\.+
```

Can you see the problem? At the string level, the regex engine will interpret the backslash as having special meaning and escape the escape character. But a single backslash has special meaning at the regex level, too. Now the pattern will match strings with one or more of any character (`.+`) and one or more dots (`\.+`)!

What's to be done? Well we could re-write the pattern as follows…

```
$ >>> pattern = ".+\\\\.+"
$ >>> print(pattern)
$ .+\\.+
```

…or we could use a raw string:

```
$ >>> pattern = r".+\\.+"
$ >>> print(pattern)
$ .+\\.+
```

If we use a raw string, the regex engine will not interpret the backslash as having any special meaning (at the first level) and instead apply the pattern as is.

Exercises

In this exercise you will create a script that emulates the parsing of a text file. You will hard-code a string comprising account records and then parse it into a dictionary of dictionaries ready for processing. Rather than parse the file contents using string methods (as you did previously), you will do so here by coding a regex pattern and using one of the re module functions to extract the information you need.

1. Create a script named ch14_pattern_matching.py.
2. Import the re module as follows:

   ```
   import re
   ```

3. Copy and paste the following string:

   ```
   file_contents = """
   1234 Credit:
   David Smith,500.0
   2345 Debit:
   Sarah Jones,-150.0
   3456 Credit:
   Tom Wilson,2000.0
   4567 Savings:
   Jane Thompson,275.0
   5678 Debit:
   Simon Davis,100.0
   """
   ```

   Note that the file has more information than we need. We're only interested in the account number, (last) name, and balance. Also note that there is no uniform delimiter, and that each record is spread over two lines.

4. Declare a variable named accounts and assign it an empty dictionary.
5. Declare a variable named pattern and assign a regular expression that matches each record and comprises three groups: one for the account number, one for the account name, and one for the account balance.

   Hint: go to regex101.com for help building your regular expression.

6. Call the re module's findall function, passing in the pattern and the file contents, and capture the result in a variable named matches. Note that, given your pattern has three groups, the findall function should return a list of tuples.
7. Code a for loop that iterates over the matches list.
8. Inside the loop…
   a. Convert the first tuple element (account number) to an int and assign it to a variable named acc_num.
   b. Assign the second tuple element (account name) to variable named acc_name.

c. Convert the third tuple element (account balance) to a float and assign it to a variable named acc_balance.

d. Assign a new key value pair to the accounts dictionary where the key is the account number, and the value is a dictionary comprising the account number, name, and balance.

9. Code a for loop that iterates over the values in the accounts dictionary.

10. Inside the loop, print the account dictionary to the console.

On executing ch14_pattern_matching.py, the output will be as below:

```
{'number': 1234, 'name': 'Smith', 'balance': 500.0}
{'number': 2345, 'name': 'Jones', 'balance': -150.0}
{'number': 3456, 'name': 'Wilson', 'balance': 2000.0}
{'number': 4567, 'name': 'Thompson', 'balance': 275.0}
{'number': 5678, 'name': 'Davis', 'balance': 100.0}
```

Optional:

11. Refactor the solution, exploiting automatic tuple unpacking in the for loop declaration for Step 7. You should then be able to do away with the variables acc_num, acc_name, and acc_balance.

NOTE

Recall that tuples may be automatically unpacked, e.g.:

```
$ >>> my_tuple = ("David", 30)
$ >>> name, age = my_tuple
$ >>> name
$ 'David'
$ >>> age
$ 30
```

CHAPTER 15

Exception Handling

Introduction

An exception is an error that occurs during the execution of your script (at runtime).

A syntax error is not an exception but rather a mistake in the spelling or arrangement of the words and symbols that comprise your code. Syntax errors are detected during compilation, and so must be corrected before the script can be executed. Consider the following example syntax error.

```
$ >>> print "Hello world"
$ SyntaxError: Missing parentheses in call to 'print'.
```

So how do errors occur in code that is syntactically correct? Errors occur at runtime if:

- The code has logic errors
- The environment in which the code executes undergoes some change
- The user inputs invalid data

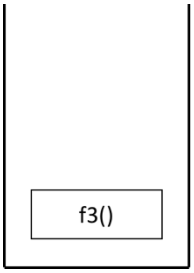
Exception handling is the process of writing code to be executed in the event some exception occurs.

Each time an exception occurs the interpreter creates an error object that contains information about what went wrong. The creation of said error object is referred to as the raising of an error. Assuming the error is not handled at the point at which it is raised, it will be thrown to the calling function. The error is thrown again and again until either it is handled or it reaches the bottom of the call stack, at which time the script will be terminated. Consider the following:

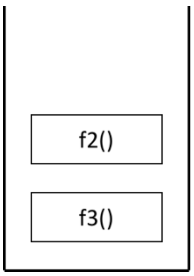
```
$ >>> def f1():
$ ...     print("f1 executing...")
$ ...     raise Exception("My deliberate error")
$ ...     print("f1 done!")
$ ...
$ >>> def f2():
$ ...     print("f2 executing...")
$ ...     f1()
$ ...     print("f2 done!)
$ ...
$ >>> def f3():
$ ...     print("f3 executing...")
$ ...     f2()
$ ...     print("f3 done!")
$ ...
$ >>> f3()
$ f3 executing...
$ f2 executing...
$ f1 executing...
$ Exception: My deliberate error
```

Note that none of the `fn done!` messages were written to the console.

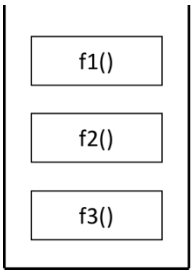
Let's take the execution of this example step-by-step. First, the function f3 is called and so it is pushed onto the call stack.



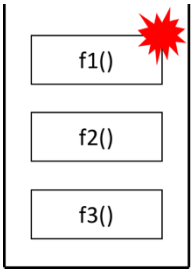
The f3 function commences execution and it writes `f3 executing...` to the console. It then calls the f2 function and so the f2 function is pushed onto the call stack.



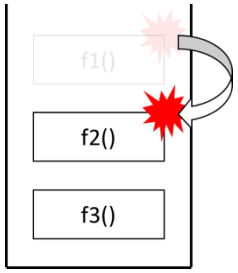
The f3 function is still in execution but must now wait for the f2 function to finish executing before it can continue. The f2 function commences execution and writes `f2 executing...` to the console. It then calls the f1 function. and so the f1 function is pushed onto the call stack.



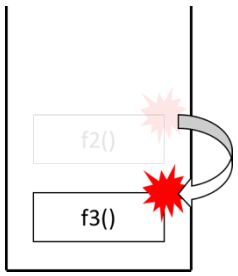
The f2 function is still in execution but must now wait for the f1 function to finish executing before it can continue. The f1 function commences execution and writes `f1 executing...` to the console. It then raises an error (an Exception type object with the message My deliberate error).



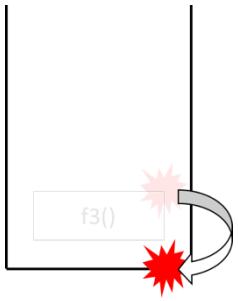
As there is no exception handling code in the f1 function, the error is thrown down the call stack to the function that called f1 – f2, and the f1 function is popped off the stack.



As there is no exception handling code in the f2 function, the error is thrown down the call stack to the function that called f2 – f3, and the f2 function is popped off the stack.



As there is no exception handling code in the function f3, the error reaches the bottom of the call stack. The interpreter writes the Exception object's message to the console and the script is terminated.



So, which types of exception might you want to handle? The answer is only those that are beyond your immediate control, i.e. environmental errors and user errors. The following example would result in a runtime error that should NOT be handled.

```
$ >>> names = ["David", "Sarah", "Tom", "Jane"]
$ >>> print(names[4])  # yields a runtime error that is easy to fix
```

The following example might result in a runtime error that arguably should be handled.

```
$ >>> number = input("Enter a whole number: ")
$ >>> number = int(number)  # might raise an error
```

Why arguably? You might argue that if the user does not enter a valid value then the right course of action is for the script to be terminated. On the other hand, you might argue that if the user does not enter a valid value then he/she should be given a second opportunity to get it right. Each scenario should be judged on its merits.

Try Except

The mechanism for handling exceptions is try except and it takes the following form:

```
try:
    <statement(s)>
except:
    <exception_handling_code>
```

Note that both try and except are keywords.

The statement(s) that may result in the raising of an error are enclosed in a try block. The statement(s) to be executed in the event an error is raised are enclosed in an except block.

If an error is raised, execution proceeds immediately to the except block, skipping the remaining statements in the try block. The statements in the except block are then executed, after which execution continues below as normal.

If no error is raised, all the statements in the try block are executed, the except block is skipped and execution continues below as normal. Consider the following script:

```
# my_script.py
while(True):
    number = input("Enter a whole number: ")
    try:
        number = int(number)
        break
    except:
        print("Invalid input; please try again")
print("You entered the number", number)
```

```
$ python my_script.py
$ Enter a whole number: two
$ Invalid input; please try again
$ Enter a whole number: 2
$ You entered the number 2
```

The code is wrapped in an infinite loop. The user is prompted to enter a whole number. We then *try* to convert the input string to a number. If an error is raised, the break statement is skipped and execution proceeds to the except block where an error message is written to the console. If no error is raised, we break out of the loop and print the `You entered the number...` message to the console.

Catching Specific Exceptions

There are many different error types. You may have encountered some already, e.g. NameError, TypeError, ValueError etc. Different types of problems result in the raising of different error type objects.

Sometimes the statements in a try block may result in the raising of two or more different types of error. If you intend to handle each different type of error in a different way, then you must add more than one except block. Consider the following:

```
$ >>> try:
$ ...     file = open("my_file.dat")
$ ...     for line in file:
$ ...         print(line)
$ ...     file.close()
$ ... except FileNotFoundError:
$ ...     print("I can\'t find that file")
$ ... except PermissionError:
$ ...     print("You do not have permission to read that file")
$ ...
```

NOTE

You can add an except block for more than one type of error by enclosing a comma separated list in round brackets following the except keyword.

The Exception Object

Recall that when an error occurs the interpreter creates an error object that contains information about what went wrong. You can obtain a reference to this object in the except block so as to make use of its error message.

```
$ >>> number = input("Enter a whole number: ")
$ >>> try:
$ ...     number = int(number)
$ ... except ValueError as e:
$ ...     print(e)
$ ...
$ Enter a whole number: two
$ invalid literal for int() with base 10: 'two'
```

The word `as` is a keyword and `e` is the name we're giving to the error object. Writing e to the console has the effect of calling the error object's __str__ dunder method which, in most cases, returns the error message.

NOTE

The calling of an object's __str__ dunder method when you print an object reference is not specific to error objects. It's the default behaviour for all Python objects.

Else and Finally

The else and finally blocks are optional additions to try except.

else

By adding an else block to your try and except blocks, you can add statements to be executed only in the event no errors are raised. Consider the following:

```
$ >>> radius = input("Enter the radius of the circle: ")
$ >>> try:
$ ...     radius = float(radius)
$ ... except:
$ ...     print("Invalid input")
$ ...
$ >>> circumference = radius * 2 * 3.14
$ >>> print("The circumference of the circle is", circumference)
```

The final two lines should only be executed in the event no error is raised in the previous try block. We could solve this problem by reorganising the code.

```
$ >>> radius = input("Enter the radius of the circle: ")
$ >>> try:
$ ...     radius = float(radius)
$ ...     circumference = radius * 2 * 3.14
$ ...     print("The circumference of the circle is", circumference)
$ ... except:
$ ...     print("Invalid input")
$ ...
```

The calculation of the circumference and the print statement have been moved up inside the try block. If an error is raised by the built-in float function, then the two statements that follow will be skipped. However, it is now not immediately obvious which of the three statements in the try block might raise an error. The more statements in a try block, the harder it is to identify the source of the error. The else block provides an alternative.

```
$ >>> radius = input("Enter the radius of the circle: ")
$ >>> try:
$ ...     radius = float(radius)
$ ... except:
$ ...     print("Invalid input")
$ ... else:
$ ...     circumference = radius * 2 * 3.14
$ ...     print("The circumference of the circle is", circumference)
$ ...
```

The statements inside the else block are only executed in the event no error is raised. This serves to neatly divide the statements into three groups: 1) that which may result in the raising of an error, 2) those to be executed in the event an error is raised, and 3) those to be executed in the event no error is raised.

finally

By contrast to the else block, the code in the finally block is executed no matter whether an error is raised or not. Consider the following:

```
$ >>> try:
$ ...     file = open("my_file.dat")
$ ...     for line in file:
$ ...         print(line)
$ ...     file.close()
$ ... except (FileNotFoundError, PermissionError) as e:
$ ...     print(e)
$ ...
```

What happens if an error is raised by during the reading of the file? Execution proceeds immediately to the except block and the file is never closed. This is a classic example of the need for a finally block.

```
$ >>> try:
$ ...     file = open("my_file.dat")
$ ...     for line in file:
$ ...         print(line)
$ ... except (FileNotFoundError, PermissionError) as e:
$ ...     print(e)
$ ... finally:
$ ...     file.close()
$ ...
```


Raising Exceptions

Sometimes you will want to raise your own exceptions. This is necessary when you're coding one part of a large system, and you encounter some problem that must be referred back to the caller for handling. The raising of an exception takes the form:

```
raise <exception_type>(<error_message>)
```

Let's assume you're coding a function to extract and process a subset of data from a comma separated string of values. The function might look something like this:

```
$ >>> def extract_full_name(csv):
$ ...     values = csv.split(",")
$ ...     first_name = values[2].title()
$ ...     last_name = values[3].title()
$ ...     return first_name + " " + last_name
$ ...
```

Let's further assume that if the string passed to your function does not have at least four values, then it should raise an exception and throw it back to the caller.

```
$ >>> def extract_full_name(csv):
$ ...     values = csv.split(",")
$ ...     if len(values) < 4:
$ ...         raise Exception("Invalid CSV: not enough values")
$ ...     first_name = values[2].title()
$ ...     last_name = values[3].title()
$ ...     return first_name + " " + last_name
$ ...
```

In this case we're creating an object of type Exception by invoking its constructor, to which we pass an error message.

Custom Exceptions

In the example above, we raised an exception of type Exception. Exception is the base type for all custom exceptions but, ideally, is not used as it is too generic. That is, its name does not give any indication as to the type of problem. In this case a better name would be CsvFormatException or NotEnoughValuesException.

You can create your own exception type by coding a class that inherits from Exception. For information about how to do that, see the chapter on Object Oriented Programming.

Assertions

A Python assertion is a boolean expression that raises an error if it evaluates to False. If it evaluates to True it does nothing. Assertions are commonly used for debugging as an assertion that is untrue brings your script to an immediate halt. An assertion takes the following form:

```
assert <boolean_expression>[, <error_message>]
```

Consider the following example:

```
$ >>> def is_valid_age(age):
$ ...     assert type(age) is int, "age is not an int"
$ ...     if 18 <= age <= 35:
$ ...         return True
$ ...     else:
$ ...         return False
$ ...
$ >>> is_valid_age(25)
$ True
$ >>> is_valid_age("thirty")
$ AssertionError: age is not an int
```


Built-in Exceptions Hierarchy

We have discussed several kinds of Exceptions and implied a hierarchy in the Object-Oriented sense between Exceptions. Indeed, the FileNotFoundError and PermissionError are subclasses, and therefore examples of, the OSError Exception. We can check this using the inbuilt `issubclass()` function.

```
>>> issubclass(FileNotFoundError, OSError)
True
>>> issubclass(PermissionError, OSError)
True
>>>
```

Below is a table of some common exceptions. The Base field indicates if the class is a base class intended to be inherited from or not. If not a base, a class is a concrete implementation.

Name	Base	Description
BaseException	Base	The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use Exception)
Exception	Base	All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.
SystemExit	Concrete	This exception is raised by the sys.exit() function. It inherits from BaseException instead of Exception so that it is not accidentally caught by code that catches Exception.
KeyboardInterrupt	Concrete	Raised when the user hits the interrupt key (normally Control-C or Delete). Inherits from BaseException not Exception.
LookupError	Base	The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: IndexError, KeyError.
ArithmeticError	Base	The base class for those built-in exceptions that are raised for various arithmetic errors: OverflowError, ZerroDivisionError, FloatingPointError.
LookupError	Base	The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: IndexError, KeyError.

IndexError	Concrete	Raised when a sequence subscript/index is out of range.
KeyError	Concrete	Raised when a mapping (dictionary) key is not found in the set of existing keys.
TypeError	Concrete	Raised when an operation or function is applied to an object of inappropriate type.
ValueError	Concrete	Raised when an operation or function receives an argument that has the right type but an inappropriate value.
ImportError	Concrete	Raised when the `import` statement has troubles trying to load a module. Also raised when the name in `from name import` cannot be found.

A full scheme of Pythons inbuilt Exceptions hierarchy can be found here : Pythons Built-in Exceptions Hierarchy.

Exercises

Exceptions

In this exercise you will create a script called ch16_exceptions.py that defines and invokes a function called get_number. When invoked, the function will prompt the user to enter a number which will be converted to a float and then returned to a variable. Ensure that the function will only exit if the user enters a valid numeric value.

On execution of ch16_exceptions.py the console output will be as shown below:

```
Enter a number: one
Not a number, try again
Enter a number: a
Not a number, try again
Enter a number: 12
12.0
```

CHAPTER 16

Files and the Filesystem

Introduction

The built-in open function and associated data types provide for the reading from and writing to files (file I/O). You saw some examples of this in the previous chapter.

A context manager is an object with a set of dunder methods enabling resources, like files, to be closed automatically. Context managers have rendered finally blocks somewhat redundant.

If you read from and write to files, then you probably parse data too. Data formats like JSON and YAML are now commonplace and so we ought to be able to parse data in these and other formats into Python data structures quickly and efficiently.

The os (operating system) provides functions that enable you to both interrogate and manipulate the filesystem.

File I/O

For quick reference, a file object's methods are tabled below. Each of these methods is described in detail in the sections that follow.

Method	Description
read(<num>): <bytes/string>	Reads num bytes and returns bytes or string depending on whether the file was opened in text or binary mode
tell(): <num>	Returns the current cursor position
seek(<offset>[, <from>]): <num>	Sets the cursor position as an offset from the given starting position
readline(): <bytes/string>	Reads and returns the next line as bytes or string depending on whether the file was opened in text of binary mode
readlines(): <list>	Reads and returns a list (of lines from the current cursor position) of bytes or strings depending on whether the file was opened in text or binary mode
write(<bytes/string>): <num>	Writes the given bytes/string and returns the number of bytes written
flush()	Flushes the buffer to the file on disk
writelines(<list>)	Writes the given list of bytes/strings
close()	Closes the file

Opening a file

The built-in **open** function is used to obtain a handle to a file. This handle is represented by the object returned by the open function. As a minimum, the open function expects to be passed a filename.

```
$ >>> file = open("my_file.dat")  # current directory or...
$ >>> file = open("/Users/home/my_file.dat")
$ >>> file
$ <_io.TextIOWrapper name= my_file.dat' mode='r' encoding='UTF-8'>
```


By default, the file is opened as a text file, in read mode with UTF-8 encoding (more on encoding later). The mode may be specified by passing a second argument to the open function. The full list of modes is tabled below.

Mode	Description
"r"	Open the file for reading (default)
"w"	Open the file for writing; attempts to create a new file; truncates the existing content if the file does exist
"x"	Open the file for exclusive creation; raises exception if file exists
"a"	Open the file for appending at the end; attempts to create a new file if the specified file does not exists;
"t"	Open the file in text mode (default)
"b"	Open the file in binary mode
"+"	Open the file for updating (reading and writing)

If you specify the mode, you must specify exactly one of x, r, w, or a, and at most one +. Consider the following examples:

```
$ >>> file = open("my_file.dat", "w")   # for writing text
$ >>> file = open("my_file.dat", "ab")  # for appending binary
$ >>> file = open("my_file.dat", "r+")  # for reading and writing text
$ >>> file = open("my_file.dat", "x")   # for creation
```

Reading from a file

The file object's **readable** method returns True is the file object is readable, else False.

```
$ >>> # my_file.dat contents: Hello world
$ >>> file = open("my_file.dat")
$ >>> file.readable()
$ True
$ >>> file.close()
```

NOTE

The readable method returns True if the file was opened in read or update mode. If the file on disk does is not readable, then attempting to open it in read or update mode will result in the raising of a PermissionError.

The file object's **read** method reads and returns *n* bytes at a time. It expects to be passed the number of bytes to read. If it is not passed any arguments, it will read and return up to the end of the file. If the file is opened in text mode, then the read method will return a string. If the file is opened in binary mode, then the read method will return one or more bytes.

```
$ >>> # my_file.dat contents: Hello world
$ >>> file = open("my_file.dat")
$ >>> file.read(2)
$ 'He'
$ >>> file.read(3)
$ 'llo'
$ >>> file.close()
```

The file object's **tell** method returns the current cursor position.

```
$ >>> # my_file.dat contents: Hello world
$ >>> file = open("my_file.dat")
$ >>> file.tell()
$ 0
$ >>> file.read(2)
$ 'He'
$ >>> file.tell()
$ 2
$ >>> file.read(3)
$ 'llo'
$ >>> file.tell()
$ 5
$ >>> file.close()
```

The file object's **seek** method is used to set the cursor position. It accepts at least one argument but is better understood, initially, if it is passed two. The first argument is an offset number of bytes, and the second argument is the position from which the offset is to be applied and may be one of:

- 0 – the beginning of the file (the default)
- 1 – the current cursor position
- 2 – the end of the file

The seek method returns the cursor position *after* it is set.

```
$ >>> # my_file.dat contents: Hello world
$ >>> file = open("my_file.dat")
$ >>> file.seek(3, 0)
$ 3
$ >>> file.read(2)
$ 'lo'
$ >>> file.close()
```

NOTE

In text files, only seeks relative to the beginning of the file are allowed.

The file object's **readline** method reads and returns one line at a time.

```
$ >>> # my_file.dat contents: Hello world\nMyname is David
$ >>> file = open("my_file.dat")
$ >>> file.readline()
$ 'Hello world'
$ >>> file.readline()
$ 'My name is David'
$ >>> file.close()
```

Finally, the file object's **readlines** method reads and returns a list of all the lines from the current cursor position.

```
$ >>> # my_file.dat contents: Hello world\nMyname is David
$ >>> file = open("my_file.dat")
$ >>> file.readlines()
$ ['Hello world', 'My name is David']
$ >>> file.close()
```

Perhaps the most common way to read the contents of a (text) file is to iterate over the file object using a for loop.

```
$ >>> # my_file.dat contents: Hello world\nMyname is David
$ >>> file = open("my_file.dat")
$ >>> for line in file:
$ ...     print(line)
$ ...
$ Hello world
$
$ My name is David
$
$ >>> file.close()
```

Note that two new line characters are written to the console for each line in the file. This happens because, by default, the print function adds a new line character to the end of the thing to be printed. To change this behaviour, we must set the print function's end parameter to something other than the new line character.

```
$ >>> # my_file.dat contents: Hello world\nMyname is David
$ >>> file = open("my_file.dat")
$ >>> for line in file:
$ ...     print(line, end='')
$ ...
$ Hello world
$ My name is David
$ >>> file.close()
```

Writing to a file

The file object's **writable** method returns True if the file object is writable, else False.

```
$ >>> file = open("my_file.dat", "w")  # write mode; a for append mode
$ >>> file.writable()
$ True
$ >>> file.close()
```

NOTE

The writable method returns True if the file was opened in write or update mode. If the file on disk does is not writable, then attempting to open it in write or update mode will result in the raising of a PermissionError.

The file object's **write** method writes the given bytes/string, and returns the number of bytes written.

```
$ >>> file = open("my_file.dat", "w")
$ >>> file.write("Hello world\n")
$ 12
$ >>> file.close()
```

The write method does not write to the file on disk, but instead to a buffer. The buffer is flushed to disk when the file is closed. Consider the following:

```
Buffer                                  File on disk
$ >>> file = open("my_file.dat", "w")
$ >>> file.write("Hello ")               Hello
$ 6
$ >>> file.write("world")               Hello world
$ 5
$ >>> file.close()                                      Hello world
```

The file object's **flush** method flushes the write buffer to the file on disk.

```
Buffer                                  File on disk
$ >>> file = open("my_file.dat", "w")
$ >>> file.write("Hello ")               Hello
$ 6
$ >>> file.flush()                                     Hello
$ >>> file.write("world")               world
$ 5
$ >>> file.flush()                                     Hello world
$ >>> file.close()                                     Hello world
```

The file object's **writelines** method writes a list of bytes/strings. The bytes/strings in the list need not be actual lines (terminated with a new line character), as is the case here.

```
$ >>> my_list = ["Hello ", "world"]
$ >>> file = open("my_file.dat", "w")
$ >>> file.writelines(my_list)
$ >>> file.close()
```


Closing a file

As you've seen, the file object's **close** method is used to close a file (release the handle).

```
$ >>> file = open("my_file.dat")
$ >>> # read/write operations
$ >>> file.close();
```

As described in the previous chapter on Exception Handling, if an error is raised whilst reading from or writing to a file, then the file may never be closed. To ensure the file is closed regardless of whether an error is raised, you should close the file in a finally block.

```
$ >>> try:
$ ...     file = open("my_file.dat")
$ ...     # read/write operations
$ ... finally:
$ ...     file.close()
$ ...
```

You might have noticed that in the example above there is no except block. That is ok, and simply means that if an error is raised it will be thrown to the calling function.

Handling exceptions

File I/O operations (including the opening of a file) may result in the raising of errors. You might encounter, for example, a FileExistsError, a FileNotFoundError, or a PermissionError. Each of these Exception data types inherits from (since Python version 3.3, at least) the OSError data type.

To handle such errors, you might wrap your file I/O code in a try block like so:

```
$ >>> try:
$ ...     file = open("my_file.dat")
$ ...     for line in file:
$ ...         print(line, end='')
$ ... except OSError as e:
$ ...     print("Something went wrong", e)
$ ...     # other exception handling code
$ ... finally:
$ ...     file.close()
$ ...
```

In this case we are handling the generic OSError type errors, which means that the except block will be executed if an OSError type error or any one of its sub-type errors is raised, including FileExistsError, FileNotFoundError, and PermissionError (to name but a few).

If you intend to handle different types of errors in different ways, then you should add multiple except blocks, one for each type of error you intend to handle separately.

```
$ >>> try:
$ ...     file = open("my_file.dat")
$ ...     for line in file:
$ ...         print(line, end='')
$ ... except FileNotFoundError:
$ ...     print("File not found")
$ ...     # other exception handling code
$ ... except PermissionError:
$ ...     print("Permission denied")
$ ...     # other exception handling code
$ ... finally:
$ ...     file.close()
$ ...
```


Context Managers

A file object is closed automatically when it is opened as part of a with statement.

```
$ >>> with open("my_file.dat") as file:
$ ...     for line in file:
$ ...         print(line, end='')
$ ...
```

So how does this work? A context manager is an object that implements the dunder methods __enter__ and __exit__. When a context manager object is created as part of a with statement (with is a keyword), then the object's __enter__ method is called automatically when the block is entered, and the object's __exit__ method is called automatically when the block is exited. In the case of a file object, the file is closed inside the __exit__ method.

Each file object is a context manager. Indeed, most objects that represent a handle/connection to an external resource are context managers.

NOTE

Coding a with statement frees you from having to close the file, but it does not free you from exception handling (assuming you intend to do so).

Text Encoding

We have been reading and writing strings of character sequences from and to files. But, data is stored in computer memory as bits and bytes. A seamless conversion is possible because of character mapping and encoding schemes. These map characters to numbers called *code points* which in can be represented in binary. We can get the *code points* associated with a given character using the `ord()` function.

```
>>> ord('A')
65
>>> ord('p')
112
```

Alternatively the `chr()` function will return the character given *a code point*.

```
>>> chr(112)
'p'
>>> chr(66)
'B'
```

At a basic level, computers deal with the binary representation of these *code points*.

Historically we had ASCII (American Standard Code for Information Interchange) which mapped all the English alphabet characters (both upper and lower case), digits 0-9 and some non-printable characters to unique numbers in the range 0-128. For this range only seven bits are required, thus a single 8-bit sequence (a byte) was more than enough. Several mappings collectively referred to as extended ASCII or Latin + 1 were used to exploit the additional bit in order, among other things, to include European language characters.

However, eventually UNICODE was developed to represent all writing systems in use globally. UNICODE defines a *codespace*, which is a mapping of characters to their code points. It is consistent with ASCII and has so far mapped more than 144000 characters, with the potential to expand to over a million. UNICODE code points require between one to four byte(8-bit) sequences to represent. Because any text-based application must be able to determine the start and end of character encoding, we have several schemes based on how a unit sequence is specified. Among them are

- UTF-8
- UTF-16
- UTF-32

UTF stands for Universal Transformation Format. UTF-32 uses a fixed length 32 bits representation while UTF-16 uses 16 or 32 bits per code point. Both can be quite wasteful for some languages, such as English, with code points in the early region of the UNICODE codespace. UTF-8, which was designed to be efficient in memory for languages like English, uses 8 to 32 bits (or 1 to 4 bytes). For the characters that overlap with ASCII UTF-8 uses a single 8-bit sequence which inevitably starts with a binary digit 0. For example, the character 'A' with a code point 65 is mapped to 01000001. For a multi-byte character point, the leading byte (first 8-bit sequence) starts with two or more

bits set to 1 followed by a trailing 0. The remaining bytes, also known as the continuation bytes, reserve the first two bits for the sequence 10, which means they only have 6-bits each for encoding digits.

For example, the first Arabic letter alif has a codepoint 65166 corresponding to binary digit 1111111010001110. This has a length of 16 bits and so requires 3 bytes to encode. The leading byte starts with 1110, which leaves 4 bits for encoding digits. The remaining continuation bytes offer 6 bits each which is 12, giving a total exactly 16 bits for encoding. This happens to be the exact number of bits in the binary representation of our number. So, our 1st, 2nd and 3rd UTF-8 encoding bytes are **11101**111, **10**111010 and **10**001110 respectively. Now if we had a character with a binary code point that did not utilize every available encoding bit, we would simply pad it with preceding 0's. Preceding and not trailing 0's as we do not want to change the place value of the binary digits which would change the code point.

Hexadecimal representation

Unicode codepoints are also commonly represented in hexadecimal form. Given the binary form it is easy to obtain the hexadecimal representation. Just group the binary digits in 4's from the left and convert each group into a hexadecimal digit and amalgamate the results in the same order. For example, 1111111010001110, would be grouped into 1111, 1110, 1000 and 1110. These translate to FE8E, written U+FE8F.

Binary Files

There are two basic types of data that we read and write, text and binary. Images, videos, and music files are all in binary form. Let us open a jpeg image

```
>>> with open('image.jpg','br') as file:
...     chunck = file.read(10)
...
>>> file
<_io.BufferedReader name='image.jpg'>
>>> type(chunck)
<class 'bytes'>
>>> chunck
b'\xff\xd8\xff\xe0\x00\x10JFIF'
```

Note that `open()` returns a `BufferedReader` object. This is an object that reads binary data. We see that `read()` return `bytes`, a built-in class for reading byte streams. `bytes` are objects of immutable ASCII characters or numbers between 0 and 256. They expose a near identical API to strings. Python also offers the mutable `bytearray` counterpart for manipulating binary files (such as image processing).

Writing strings to a binary file object or `bytes` to a text file will raise a `TypeError` Exception.

I/O Layered Abstraction

Like a `BufferedReader` the `TextIOWrapper` object returned when we open a text file is a buffered reader. Buffering is a technique used when we are reading a data stream. It uses an intermediate in-memory storage space (a buffer) to reduce the frequency of system calls, increasing reading and writing efficiency. As well as buffering we saw that the `TextIOWrapper` class offers automatic encoding and decoding of text data. The `io` module implements these features by a layered abstraction. So, when we get a `TextIOWrapper` object after opening a text file in read mode, at the base is a `FileIO` object for reading raw, unbuffered binary data. This is wrapped by a `BufferedReader` which perform the buffering. The `BufferedReader` in turn is wrapped by the `TextIOWrapper` which does the decoding. We would not do this usually, but we can construct the layers ourselves.

```
>>> import io
>>> fileio = io.FileIO('my_file.dat','r')
>>> buffered = io.BufferedReader(fileio)
>>> text_io_wrapper = io.TextIOWrapper(buffered)
>>> text_io_wrapper.readline()
'This is a line\n'
```

The `io` module also defines a `BufferedWriter` and `BufferedRandom` for buffered binary data.

The os Module

The os module is a part of the standard library (is bundled with the interpreter) and so may be imported without prior installation. It exposes a number of functions for interrogating and manipulating the filesystem (among other things).

For quick reference, the relevant os module functions are tabled below. Each of these functions is described in detail in the sections that follow.

Method	Description
getcwd(): <path>	Returns the current working directory
listdir([<path>]): <list>	Returns a list of files and directories in the given directory or the current directory if no arg. passed
mkdir(<path>)	Creates a new directory
makedirs(<path>)	Creates directories recursively
chdir(<path>)	Changes the current working directory
remove(<path>)	Deletes a file
rmdir(<path>)	Deletes an empty directory
rename(<source>, <dest>)	Renames a file or directory

The **getcwd** function returns the current working directory.

```
$ >>> import os
$ >>> os.getcwd()
$ '/Users/home'
```

The **listdir** function returns a list of the files and directories in the given directory or, if no argument is passed, the current working directory.

```
$ >>> import os
$ >>> files_and_dirs = os.listdir("/Users/home")
$ >>> for name in files_and_dirs:
$ ...     print(name)
$ ...
$ Desktop
$ Documents
$ Downloads
$ ...
```

The **mkdir** function is used to create a new directory. It expects a path.

```
$ >>> import os
$ >>> os.mkdir("temp")
```

The **makedirs** function is used to create directories recursively. It expects a path.

```
$ >>> import os
$ >>> os.makedirs("/sales/2019/q1")
```


The **chdir** function is used to change the current working directory. It expects a path.

```
$ >>> import os
$ >>> os.chdir("Documents")
$ >>> os.getcwd()
$ '/Users/home/Documents
```

The **remove** function is used to delete a file. It expects a path.

```
$ >>> import os
$ >>> os.remove("/Users/home/my_file.dat")
```

The **rmdir** function is used to delete an empty directory. It expects a path.

```
$ >>> import os
$ >>> os.rmdir("/Users/home/temp")
```

NOTE

To delete a non-empty directory, you must import the standard library module named shutil, and call its rmtree function passing the path to be deleted, e.g.:

```
$ >>> import shutil
$ >>> shutil.rmtree("/Users/home/sales")
```

The **rename** function is used to rename a file or directory. It expects both source and destination paths.

```
$ >>> import os
$ >>> os.listdir()
$ ['old.txt']
$ >>> os.rename("old.txt", "new.txt")
$ >>> os.listdir()
$ ['new.txt']
```

The errno Module

In the handling errors subsection, we saw that there are various exceptions such as FileExistsError, FileNotFoundError, and PermissionError that might occur when attempting to open a file for reading or writing. We saw that these are examples of the more general OSError Exception. OSError is the generic Exception for system calls. These can arise for example when attempting to address a non-existing device, you run out of memory, there are too many open files, and many more situations. There are too many system call (OS) errors to have a specialised class for each. However, each has a unique error number used to initialize the *errno* attribute of the OSError raised. This can be used to write specialised exceptions.

The `errno` module, via an `errorcode` dictionary, maps error numbers to symbolic constants. Below is code that raises a FileNotFoundException that is captured as the more general OSError instance and maps the error number to its symbol.

```
>>> import errno
>>> try:
...     file = open('somefile.txt')
... except OSError as e:
...     code = e.errno
...     symbol = errno.errorcode[code]
...     print(f"Error no. is {code}, and maps to {symbol}")
...
Error no. is 2, and maps to ENOENT
>>>
```

We see that the error number and symbol for the FileNotFoundException error are 2 and `ENOENT` respectively.

Exercises

File IO

In this exercise you will create a script that reads in from file student names. For each student name the script will prompt the user to input a grade, e.g. A, B, C, etc. The script will then overwrite the file with the student names and their associated grades.

1. Create a text file named students.dat and populate it with the names of your co-delegates, one per line. If you don't have any/many co-delegates you might choose to use the names of your friends/family instead.
2. Create a script named ch17_file_io.py.
3. Open the students.dat file for reading and assign the file reference to a variable named file_read_ref.
4. Declare a variable named results and assign it an empty list.
5. Code a for loop that iterates over the file.
6. Inside the loop…
    a. Strip the new line character from the name.
    b. Prompt the user to input a grade for name and capture it in a variable named grade.
    c. Append the name and grade to the results list.
7. Close the file reference.
8. Open a new file named studentgrades.txt, for writing this time, and assign the reference to a variable named file_write_ref.
9. Code a for loop that iterates over the results list.
10. Inside the loop, write to the file the name and grade separated by a comma and terminated by a new line character.
11. Close the file reference.

After execution of ch17_file_io.py, the console output will be as shown below:

```
Enter grade for David: B
Enter grade for Jane: C
Enter grade for Tom: A
Enter grade for Sarah: D
```

The studentgrades.dat file will be as shown below:

```
David,B
Jane,C
Tom,A
Sarah,D
```

12. What character does the UNICODE codepoint 960 represent?
13. Convert the codepoint 960 to its binary, Hex and UTF-8 encoding (binary) sequence.

CHAPTER 17

Databases

Introduction

PEP (Python Enhancement Proposal) 0249 describes the database API v2.0. It is a specification of a set of classes for interacting with a database. The API specifies what should happen, but not how, and the purpose of it is to standardise the code regardless of the database in use. That is, you ought to be able to switch from a MySQL database to a PostgreSQL one with out having to change your code (much).

The database API v2.0 centres around the Connection and Cursor classes. A Connection object represents a connection to the database, while a Cursor object is used to execute queries and commands.

NOTE

This chapter does not represent exhaustive coverage of the database API v2.0. For more information, see the PEP (link at the top of the page).

Database API Implementations

In most cases there is more than one implementation of the database API v2.0 for each vendor. For example, for MySQL the list includes…

- MySQL for Python
- mysqlclient
- PyMySQL
- mxODBC
- pyodbc
- MySQL Connector/Python
- mypysql

…and more. To see a list of implementations, search the Python Wiki for the vendor.

NOTE

The examples used throughout this chapter assume the PyMySQL database API v2.0 implementation.

The Basics (in 6 Steps)

Before we start to code, we must install the relevant module using PIP. In this case, that's PyMySQL.

```
$ pip install pymysql
```

First, we import the database API v2.0 implementation.

```
$ >>> import pymysql
```

Second, we create a Connection object by calling the connect function, passing it the hostname, username, password (if applicable), and database name.

```
$ >>> import pymysql
$ >>> connection = pymysql.connect("localhost", "user", "pwd", "mydb")
```

Third, we use the Connection object to create a Cursor object.

```
$ >>> import pymysql
$ >>> connection = pymysql.connect("localhost", "root", "pswd", "mydb")
$ >>> cursor = connection.cursor()
```

Fourth, we use the Cursor object to execute a query or command.

```
$ >>> import pymysql
$ >>> connection = pymysql.connect("localhost", "root", "pswd", "mydb")
$ >>> cursor = connection.cursor()
$ >>> cursor.execute("select * from customer")
```

Fifth, and assuming a query, we use the Cursor object to fetch the results.

```
$ >>> import pymysql
$ >>> connection = pymysql.connect("localhost", "root", "pswd", "mydb")
$ >>> cursor = connection.cursor()
$ >>> cursor.execute("select * from customers")
$ >>> results = cursor.fetchall()
```

Sixth and finally, we close the connection. Note that you needn't do this if you use a context manager.

```
$ >>> import pymysql
$ >>> connection = pymysql.connect("localhost", "root", "pswd", "mydb")
$ >>> cursor = connection.cursor()
$ >>> cursor.execute("select * from customers")
$ >>> results = cursor.fetchall()
$ >>> connection.close()
```

Simple, eh?

The Connection Object

Obtaining the connection

The **connect** function returns a Connection object. The specification does not dictate any parameters for the connect function, and so may vary from vendor to vendor. Typically, each parameter is assigned a default value, which means that you need only pass the arguments that are applicable/necessary in any given scenario.

NOTE

PyMySQL's connect function describes 31 parameters, each with a default value, for configuring the Connection object.

A Connection object is a context manager (or should be). This means that the connection will be closed automatically if the object is created as part of a with statement.

```
$ >>> from pymysql import connect
$ >>> with connect("localhost", "user", "pwd", "mydb") as connection:
$ ...     # create a cursor, execute queries etc.
$ ...
```

Creating the cursor

A Connection object's **cursor** method is used to create a Cursor object.

```
$ >>> cursor = connection.cursor()
```

Some implementations provide for the specification of cursor type. For example,…

```
$ >>> cursor = connection.cursor(pymysql.cursors.DictCursor)
```

…results in the creation of a Cursor object that returns each result as a dictionary, as opposed to a tuple (the default).

Managing the transaction

A transaction begins automatically with the first query/command executed using the Cursor object, and ends when either the commit or rollback method is called.

A Connection object's **commit** method commits any pending transaction to the database, while the **rollback** method causes the database to roll back to the start of any pending transaction.

NOTE

Not all databases provide transaction support. If the database does provide transaction support, then its auto-commit feature must be off initially.

Assuming transaction support, then the transaction should be committed only after all of the queries/commands in the given operation have completed without the raising of an error. If an error is raised at any point during the execution of the queries/commands, then the transaction should be rolled back.

```
$ >>> cursor = connection.cursor()
$ >>> try:
$ ...     # execute queries/commands
$ ...     connection.commit()
$ ... except:
$ ...     connection.rollback()
$ ... finally:
$ ...     cursor.close()
$ ...
```

To be even more explicit, you might choose to commit the transaction in the else block (which is only executed in the event no error is raised).

```
$ >>> cursor = connection.cursor()
$ >>> try:
$ ...     # execute queries/commands
$ ... except:
$ ...     connection.rollback()
$ ... else:
$ ...     connection.commit()
$ ... finally:
$ ...     cursor.close()
$ ...
```

Closing the connection

A connection object's **close** method closes the connection. The Connection object is unusable after it is closed and an error will be raised if any operation is attempted on it.

NOTE

Closing a Connection object with a pending transaction results in its being rolled back.

The Cursor Object

Executing queries/commands

A Cursor object's **execute** method is used to execute a query/command.

```
$ >>> cursor.execute("select * from customer")
```

To protect against SQL injection attacks and to improve performance, placeholders should be used to inject parameters into the SQL.

```
$ >>> cursor.execute("select * from customer where id = ?", id)
```

In this example the placeholder is the question mark. Before the query is executed, the value of the parameter – id – is bound to the placeholder.

The database API v2.0 placeholder styles (paramstyles) are tabled below.

Placeholder	Description	Example
qmark	Question mark style	...id = ?
numeric	Numeric, positional style	...id = :1
named	Named style	...id = :id
format	ANSI C printf format codes	...id = %d
pyformat	Python extended format codes	...id = %(id)d

A PyMySQL Cursor object expects either a list, a tuple, or a dictionary composed of the values to be injected to be passed to its execute method. If it's a list or a tuple, then the format-style placeholder may be used. If it's a dictionary, then the pyformat-style placeholder may be used. Consider the following PyMySQL example:

```
$ >>> customer = ("David", "david@mail.com")
$ >>> cursor.execute(
$ ...     "insert into customer (name, email) values (%s, %s)",
$ ...     customer)
```

A Cursor object's **executemany** method is used to execute a command against a collection of parameters. Consider the following PyMySQL example:

```
$ >>> customers = [
$ ...     ("David", "david@mail.com")
$ ...     ("Sarah", "sarah@mail.com")
$ ...     ("Tom", "tom@mail.com")
$ ... ]
$ >>> cursor.executemany(
$ ...     "insert into customer (name, email) values (%s, %s")",
$ ...     customers)
```


Fetching results

The Cursor object's read-only **rowcount** data attribute stores the number of rows that were produced or affected by the last execute invocation, or -1 if no query/command has been executed.

```
$ >>> cursor.rowcount
$ 3
```

You will have noticed that neither the execute nor the executemany method return anything. Having executed a query, the results are available from the Cursor object.

The Cursor object's **fetchone** method is used to fetch the next result set. If there are more results, then a tuple is returned. If there are no more results, then None is returned. An error is raised if the call to execute did not produce a result set.

```
$ >>> cursor.execute("select * from customer where id = ?", id)
$ >>> customer = cursor.fetchone()
$ >>> customer
$ (1, 'David', 'david@mail.com')
```

The Cursor object's **fetchall** method is used to fetch all of the remaining result sets.

```
$ >>> cursor.execute("select * from customer")
$ >>> rows = cursor.fetchall()
$ >>> for row in rows:
$ ...     print(row)
$ ...
$ (1, 'David', 'david@mail.com')
$ (2, 'Sarah', 'sarah@mail.com')
$ (3, 'Tom', 'tom@mail.com')
```

Closing the cursor

A Cursor object's **close** method closes the cursor. The Cursor object is unusable after it is closed and an error will be raised if any operation is attempted on it.

Exercises

Appendix

Case Study

Case study part 1

This is the first part of a case study that will evolve throughout the remaining chapters of this manual. The ultimate goal is to create a script that:

- Reads into memory airline passenger information from a file.
- Enables the user to search and filter the information.
- Enables the user to write selected records to a database table.

In this part, you will create a script that prompts the user to make a choice: to search for a passenger or to filter the passengers. This will require the storage (in memory only, for now) of a list of airline passengers.

If the user chooses to search for a passenger, the script will prompt the user to input the passenger's ID, and will iterate over the list of passengers to find a match. If a matching passenger is found, its details will be printed to the console. If no matching passenger is found, a passenger not found message will be printed to the console.

If the user chooses to filter passengers, the script will prompt the user to input the field name on which to filter, e.g. last name, and then again to input the filter value, e.g. Doe. It will then iterate over the list of passengers to find matches. The details of each matching passenger found will be printed to the console.

1. Download the script pax_list.py from here. If you don't have Internet access, then create a module named pax_list.py and paste into it the code in the addendum to these instructions.
2. Create a script named case_study_part_1.py.
3. Import the list of passengers from the pax_list module as follows:

   ```
   from pax_list import pax_list
   ```

4. Prompt the user to input an s for search or an f for filter, and capture it in a variable named option, e.g.:

   ```
   option = input("[s]earch or [f]ilter: ")
   ```

5. If the user inputs s…
   a. Prompt the user to input a passenger ID and capture it in a variable named Id (not id because that is the name of a built-in function).
   b. Convert Id to an int.
   c. Code a loop that iterates over the pax_list.
   d. Inside the loop…
      i. If the passenger's ID matches the ID input by the user, print the passenger's first and last names to the console and break out of the loop.
   e. If the loop exits normally, a matching passenger has not been found so print a passenger not found message to the console.

6. If the user inputs an f…
    a. Prompt the user to input the field name on which to filter and capture it in a variable named ff (filter field).
    b. Prompt the user to input the filter value and capture it in a variable named fv (filter value).
    c. If ff is "checked_bags", convert fv to a bool.
    d. If ff is "flight_time", convert fv to a float.
    e. Code a loop that iterates over the pax_list.
    f. Inside the loop…
        i. If the passenger's <ff> matches the <fv>, print the passenger's first and last names to the console.

NOTE

Filtering on visited countries and flight time is a little trickier than the others. Each requires a different test than that described above. For example, filtering on visited countries might involve testing for membership, while filtering on flight time might require a threshold.

Case study part 2

In this part you will create a new version of the case study script that enables the user to search/filter the pax_list as many times as he/she wants.

7. Make a copy of case_study_part_1.py and name it case_study_part_2.py.
8. Wrap all of the code, barring the import statement, in an infinite loop.
9. Inside, and at the end of the infinite loop…
    a. Prompt the user to input y to continue searching/filtering, or n to stop, and capture it in a variable named more.
    b. If more is n, break out of the loop.

Addendum

```
pax_list = [
    {
        "id": 1, "fname": "Doug", "lname": "Noon", "checked_bags": False,
        "visited_countries": ["Vatican City", "Ethiopia", "Greece", "Chile"],
        "flight": {"source": "Tallahassee", "dest": "Omaha"}, "flight_time": 15.05
    },
    {
        "id": 2, "fname": "Brad", "lname": "Wren", "checked_bags": False,
        "visited_countries": ["Iceland", "Albania", "Nauru", "Albania", "Benin"],
        "flight": {"source": "Atlanta", "dest":"Fayetteville"}, "flight_time": 3.80
    },
    {
        "id": 3, "fname": "Maya", "lname": "Moore", "checked_bags": False,
        "visited_countries": ["Chile", "Ukraine", "Slovakia", "Brazil", "Malaysia"],
        "flight": {"source": "Jacksonville", "dest":"Long Beach"}, "flight_time": 5.19
    },
    {
        "id": 4, "fname": "Nicole", "lname": "Ross", "checked_bags": True,
        "visited_countries": ["Ecuador", "Egypt", "Myanmar", "Luxembourg", "Swaziland"],
        "flight": {"source":"London", "dest":"Seattle"}, "flight_time": 5.24
    },
    {
        "id": 5, "fname": "Gemma", "lname": "Cann", "checked_bags": True,
        "visited_countries": ["Nauru", "Honduras", "Comoros", "Poland", "Eritrea"],
        "flight": {"source":"Tokyo", "dest":"Miami"}, "flight_time": 9.94
    },
    {
        "id": 6, "fname": "Freya", "lname": "Jarvis", "checked_bags": True,
        "visited_countries": ["United Arab Emirates", "Ethiopia", "Papua New Guinea"],
        "flight": {"source":"Otawa", "dest":"Miami"}, "flight_time": 20.86
    },
    {
        "id": 7, "fname": "Liv", "lname": "Thorne", "checked_bags": False,
        "visited_countries": ["Sri Lanka", "Angola", "Ethiopia", "Uzbekistan"],
        "flight": {"source":"Honolulu", "dest":"Ontario"}, "flight_time": 18.70
    },
    {
        "id": 8, "fname": "Deborah", "lname": "Quinton", "checked_bags": False,
        "visited_countries": ["Tuvalu", "Nigeria", "Senegal", "Zambia", "Samoa"],
        "flight": {"source":"Fremont", "dest":"Memphis"},"flight_time": 12.55
    },
    {
        "id": 9, "fname": "Camden", "lname": "Oldfield", "checked_bags": False,
        "visited_countries": ["Uganda", "Benin", "Philippines", "Iran", "Liberia"],
        "flight": {"source":"Philadelphia", "dest":"Portland"}, "flight_time": 22.83
    }
]
```

Case study part 3

In this part you will create a module that has functions for searching and filtering a list of airline passengers. You will then create a new version of the case study script that uses the module to do the searching and filtering of passengers. This modularisation of the case study seeks to break the problem down into smaller pieces. The functions may be modified without affecting the script, and functions may be tested as independent units.

10. Create a module named case_study_part_3_functions.py.
11. Declare a function named search_pax that accepts a list of passengers and an ID.
12. Inside the function…
    a. Code a loop that iterates over the list of passengers.
    b. Inside the loop…
        i. If the passenger's ID matches the ID parameter, return the passenger.
    c. If the loop exits normally, a matching passenger has not been found so return None.
13. Declare a function named filter_pax that accepts a list of passengers, a filter field (ff), and a filter value (fv).
14. Inside the function…
    a. Declare a variable named matching_passengers and assign it an empty list.
    b. Code a loop that iterates over the pax_list.

c. Inside the loop…

i. If the passenger's <ff> matches the <fv>, append the passenger to the matching_passengers list, e.g.:

```
matching_passengers.append(passenger)
```

d. Return the matching passengers.

15. Make a copy of case_study_part_2.py and name it case_study_part_3.py.
16. Import the search_pax and filter_pax functions from the case_study_part_3_functions module as follows:

```
from case_study_part_3_functions import search_pax, filter_pax
```

17. After the part of the script in which you prompt the user to input the passenger's ID, replace the loop code with a call to the search function, passing in the pax_list and ID input by the user, and capture the return value in a variable named pax.
18. If the returned passenger is not None, print its first and last names to the console, else print a passenger not found message to the console.
19. After the part of the script in which you prompt the user input the filter field and filter value, replace the loop code with a call to the filter function, passing in pax_list, ff, and fv, and capture the return value in a variable named matching_passengers.
20. If the returned list is not empty…
    a. Code a loop that iterates over the matching_passengers list.
    b. Inside the loop, print each passenger's first and last names to the console.
21. Else, print a no matching passengers found message to the console.

Case study part 4a

In this part you will improve the filter_pax function by exploiting list comprehension.

1. Make a copy of case_study_part_3_functions.py and name it case_study_part_4_functions.py.
2. Replace the for loop(s) with list comprehension(s).

Case study part 4b

In this part you will sort by last name the list of matching passengers returned by the filter_pax function.

3. Make a copy of case_study_part_3.py and name it case_study_part_4.py.
4. Update the import statement so as to import the search_pax and filter_pax functions from the case_study_part_4_functions module.
5. Sort by last name the list of matching passengers returned by the filter_pax function. A list object's sort method has a parameter named key which must be passed a reference to a function. In this case, that function will be passed a passenger dictionary and must return the value on which the list is to be sorted. You might, and probably should, use a lambda.

Case study part 5

In this part you will add a function to the case study module for the purpose of formatting a list of passengers. This function will accept a list of passengers and return a formatted string. This will be used to improve the readability of the results of a filter operation.

6. Make a copy of case_study_part_4_functions.py and name it case_study_part_5_functions.py.
7. Declare a function named format_pax that accepts a list of passengers.
8. Inside the function…
    a. Declare a variable named rows and assign it an empty list.
    b. Append to the rows list a formatted string that has column headers as follows:
    - ID, 3 characters wide
    - Last Name, 10 characters wide
    - First Name, 10 characters wide
    - Source, 15 characters wide
    - Dest, 15 characters wide

    …each separated by two spaces.
    c. Code a for loop that iterates over the list of passengers.
    d. Inside the loop…
        i. Append to the rows list a formatted string that comprises the passenger's ID, last name, first name, source, and dest, all left aligned and all with widths to match the header row.
    e. Build a string from the elements in the rows list using the new line character as the delimiter, e.g.:

```
"\n".join(rows)
```

    …and return it.

9. Make a copy of case_study_part_4.py and name it case_study_part_5.py.
10. Update the import statement so as to import the search_pax and filter_pax functions from the case_study_part_5_functions module.
11. Add format_pax to the list of functions to be imported.
12. Within the filter part, replace the for loop in which passenger details are printed to the console, with a call to the format_pax function, passing in the list of passengers. Wrap that call using the built-in print function to print the formatted list of passengers to the console.

Case study part 6

In this part you will make the script more robust by adding to it exception handling code. As it stands, for example, if the user does not input a valid number when searching, the script will crash. Likewise, if the user does not input a valid number when filtering on flight time the script will crash.

13. To keep things tidy, and despite the fact we will not make any changes to it, make a copy of case_study_part_5_functions.py and name it case_study_part_6_functions.py.
14. Make a copy of case_study_part_5.py and name it case_study_part_6.py.
15. Update the import statement so as to import the search_pax, filter_pax, and format_pax functions from the case_study_part_6_functions module.
16. In the search part…
    a. Wrap the conversion to an int of the passenger ID in a try block.
    b. If an error is raised print an appropriate error message to the console.
    c. Make the necessary changes to ensure that the remainder of the search code is only executed if not error is raised.
17. In the filter part…
    a. Wrap the conversion to a float of the flight time in a try block.
    b. If an error is raised print an appropriate error message to the console and then continue immediately to the next iteration of the infinite loop.

Case study part 7

In this part you will change the script such that it reads passengers in from a JSON formatted file. This will involve adding a new function to the case study module.

18. Make a copy of case_study_part_6_functions.py and name it case_study_part_7_functions.py.
19. Declare a function named read_pax_from_json that accepts a file path.
20. Inside the function…

a. Import the json module as follows:

```
import json
```

b. Open the file at the given path for reading and assign the file reference to a variable named file.

c. Parse the contents of the file into a Python data structure (a list of dictionaries in this case) using the appropriate json module function, and capture it in a variable named pax_list.

d. Return the passenger list.

21. Download the file pax_list.json from here. If you don't have Internet accesss, then create a file named pax_list.json and paste into it the JSON in the addendum to these instructions.
22. Make a copy of case_study_part_6.py and name it case_study_part_7.py.
23. Update the import statement so as to import the search_pax, filter_pax, format_pax, and read_pax_from_json functions from the case_study_part_7_functions module.
24. Remove the import for pax_list.
25. Immediately before the commencement of the infinite loop, call the read_pax_from_json function passing it the path for pax_list.json, and assign the resultant list to a variable named pax_list.

Addendum


```
[{"id": 1, "fname": "Doug", "lname": "Noon", "checked_bags": false, "visited_countries":
["Vatican City", "Ethiopia", "Greece", "Chile"], "flight": {"source": "Tallahassee",
"dest": "Omaha"}, "flight_time": 15.05}, {"id": 2, "fname": "Brad", "lname": "Wren",
"checked_bags": false, "visited_countries": ["Iceland", "Albania", "Nauru", "Albania",
"Benin"], "flight": {"source": "Atlanta", "dest": "Fayetteville"}, "flight_time": 3.8},
{"id": 3, "fname": "Maya", "lname": "Moore", "checked_bags": false, "visited_countries":
["Chile", "Ukraine", "Slovakia", "Brazil", "Malaysia"], "flight": {"source":
"Jacksonville", "dest": "Long Beach"}, "flight_time": 5.19}, {"id": 4, "fname": "Nicole",
"lname": "Ross", "checked_bags": true, "visited_countries": ["Ecuador", "Egypt",
"Myanmar", "Luxembourg", "Swaziland"], "flight": {"source": "London", "dest": "Seattle"},
"flight_time": 5.24}, {"id": 5, "fname": "Gemma", "lname": "Cann", "checked_bags": true,
"visited_countries": ["Nauru", "Honduras", "Comoros", "Poland", "Eritrea"], "flight":
{"source": "Tokyo", "dest": "Miami"}, "flight_time": 9.94}, {"id": 6, "fname": "Freya",
"lname": "Jarvis", "checked_bags": true, "visited_countries": ["Bosnia and Herzegovina",
"United Arab Emirates", "Ethiopia", "Papua New Guinea"], "flight": {"source": "Otawa",
"dest": "Miami"}, "flight_time": 20.86}, {"id": 7, "fname": "Liv", "lname": "Thorne",
"checked_bags": false, "visited_countries": ["Sri Lanka", "Angola", "Ethiopia",
"Uzbekistan", "South Africa"], "flight": {"source": "Honolulu", "dest": "Ontario"},
"flight_time": 18.7}, {"id": 8, "fname": "Deborah", "lname": "Quinton", "checked_bags":
false, "visited_countries": ["Tuvalu", "Nigeria", "Senegal", "Zambia", "Samoa"],
"flight": {"source": "Fremont", "dest": "Memphis"}, "flight_time": 12.55}, {"id": 9,
"fname": "Camden", "lname": "Oldfield", "checked_bags": false, "visited_countries":
["Uganda", "Benin", "Philippines", "Iran", "Liberia"], "flight": {"source":
"Philadelphia", "dest": "Portland"}, "flight_time": 22.83}]
```


Case study part 8

In this final part you will change the script such that it enables the user to write filtered passengers to a database, if he/she so chooses. This will involve adding a new function to the case study module.

26. Make a copy of case_study_part_7_functions.py and name it case_study_part_8_functions.py.
27. Declare a function named write_pax_to_db that accepts a list of passengers.
28. Inside the function…
    a. Import the sqlite3 module as follows:

       ```
       import sqlite3
       ```

       sqlite3 is an embedded database meaning that it exists either in memory or as a file on disk as a part of your application. We're using it here because it doesn't require the installation and configuration of an external DBMS.

    b. Establish a connection to a database named paxdb and capture a reference to the Connection object in a variable named conn, e.g.:

       ```
       conn = sqlite3.connect("paxdb")
       ```

    c. Create a Cursor object and assign it to a variable named cursor.
    d. In a try block…
       i. Execute the following query:

          ```
          create table if not exists passenger (
              passenger_id integer,
              fname text,
              lname text)
          ```

       ii. Transform the passenger list into a list of tuples, each comprising the passenger's ID, first name, and last name. You should try to do this by exploiting list comprehension. Assign the resultant list to a variable named records.
       iii. Execute one insert command for each tuple in the records list:

          ```
          insert into passenger values (?, ?, ?)
          ```

    e. In the except block…
       i. Rollback the transaction.
       ii. Print an error message to the console.
    f. If the statements inside the try block execute without raising an error, then commit the transaction and print a success message to the console.
29. Make a copy of case_study_part_7.py and name it case_study_part_8.py.

30. Update the import statement so as to import the search_pax, filter_pax, format_pax, read_pax_from_json, and write_pax_to_db functions from the case_study_part_8_functions module.
31. In the filter part, and after printing the matching passengers to the console…
    a. Prompt the user to input y if they want to write the filtered passengers to the database, or n if not, and capture it in a variable named to_db.
    b. If to_db is y, then call the write_pax_to_db function passing in the list of matching passengers.

NOTE

To test that this works (something is written to the database), you might want to download and install the sqlite browser. If you don't have Internet access you'll just have to write some more code!

StayAhead Training Limited

020 7600 6116

www.stayahead.com