Laboratory Booklet

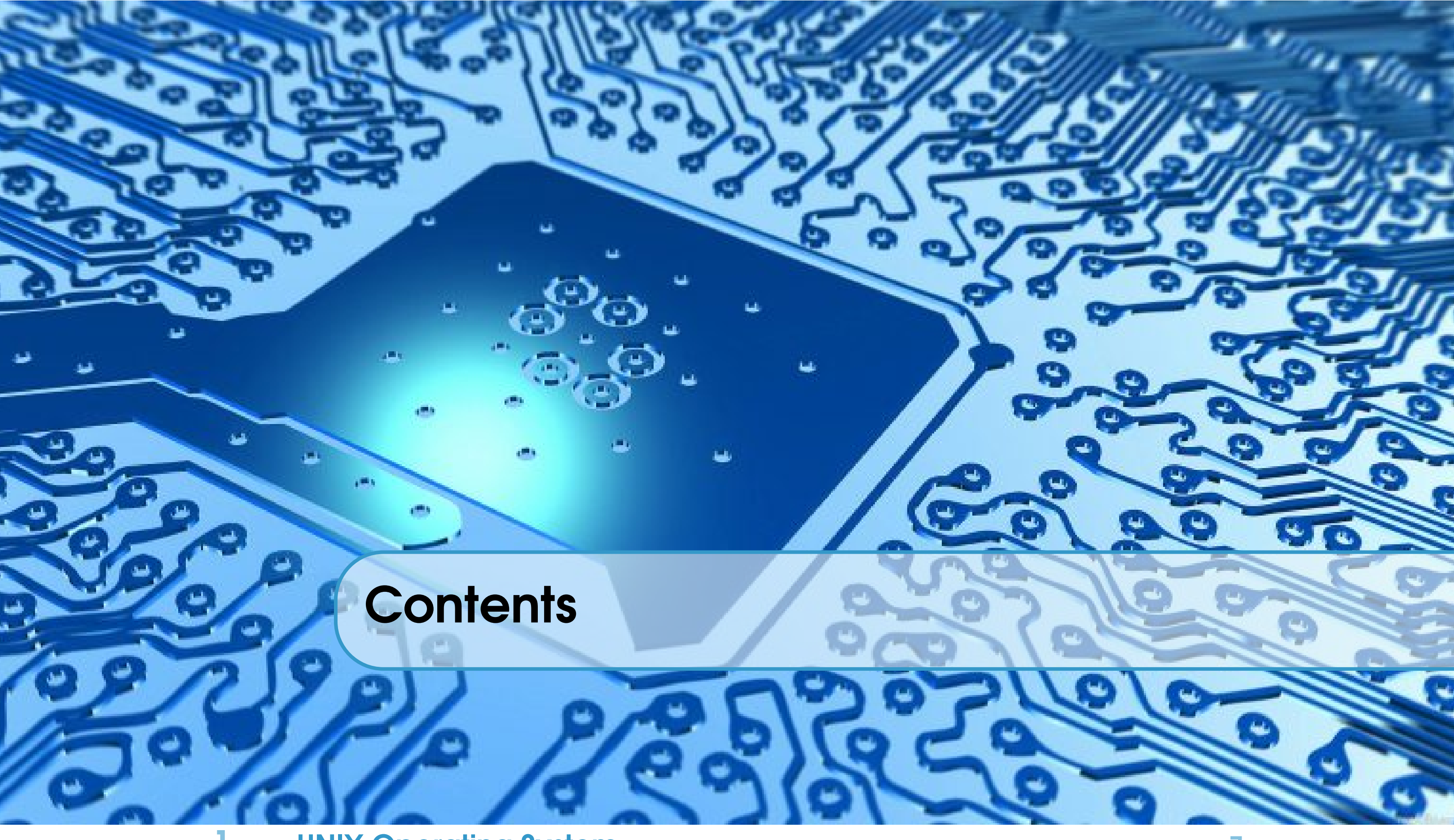
Informatica aa. 2022/2023

AUTHORS:
GIOVANNI ACAMPORA, AUTILIA VITIELLO, MARIACARLA STAFFA
UNIVERSITY OF NAPLES FEDERICO II

NOTE:

: is for commands.

First release, March 2020



Contents

1 UNIX Operating System 7

1.1 What is UNIX? 7

1.2 Types of UNIX 7

1.3 The UNIX operating system 7

1.4 Files and processes 9

1.5 The Directory Structure 9

1.6 Starting an UNIX terminal 9

1.7 Listing files and directories 10

1.8 Making Directories 11

1.9 Changing to a different directory 12

1.10 The directories . and .. 12

1.11 Pathnames 13

1.12 Copying Files 13

1.13 Moving files 14

1.14 Removing files and directories 15

1.15 Displaying the contents of a file on the screen 15

1.16 Searching the contents of a file 16

1.17 Redirection 17

1.18 Redirecting the Output 18

1.19 Appending to a file 19
1.20 Redirecting the Input 19
1.21 Pipes 20
1.22 Setting a Virtual Machine 21

2 Introduction to Python 27
2.1 Introduction to programming 27
2.2 Programming languages 27
2.2.1 Machine Language 28
2.2.2 Assembly Language 28
2.2.3 High-level language 28
2.3 What is Python? 29
2.3.1 Python interpreter 29
2.3.2 Python programming paradigm 30
2.4 Why we should use Python? 31

3 Getting Started with Python 33
3.1 Installing Python 33
3.1.1 Working with Windows 33
3.1.2 Working with the Mac 36
3.1.3 Working with Linux 38
3.2 Anaconda: an alternative Python distribution 39
3.3 Other useful tools 44
3.3.1 PyCharm: Installation and Set Up 45
3.4 How to include a Unix shell (bash) in the PyCharm IDE under Windows 52
3.5 Interacting with Python 54
3.5.1 Interactive mode 54
3.5.2 Scripting mode 55
3.5.3 IDE mode: using PyCharm 56

4 The basic elements of Python 63
4.1 Statements, Variables and Expressions 63
4.2 Variable types 64
4.2.1 Numbers 65
4.3 Dynamic and strongly typed language 66
4.4 Arithmetic and Assignment Operators 67
4.4.1 Arithmetic operators 67
4.4.2 Assignment operators 69

4.5 Output and Input functions 69
4.5.1 Function `print` . . . 70
4.5.2 Function `input` . . . 72
4.6 Type conversions 72
4.7 Exercises 73

5 Strings and Branching Programs . . . 75
5.1 Strings 75
5.2 Branching programs 77
5.2.1 If Statements . . . 77
5.2.2 While Loops . . . 80
5.2.3 For Loops . . . 81
5.2.4 Break statement . . . 83
5.3 Chapter Exercises 83

6 Functions and Structured Data Types . . . 85
6.1 User-defined Functions 85
6.2 Sequence Types 89
6.2.1 Lists . . . 89
6.2.2 Tuples . . . 94
6.3 Dictionaries 97
6.4 Chapter exercises 100

7 Object-Oriented Programming . . . 103
7.1 Objects and Classes 103
7.1.1 Creating a Class . . . 103
7.1.2 Creating objects . . . 104
7.1.3 Special methods . . . 104
7.2 Inheritance 105
7.3 Exceptions 107
7.4 Chapter exercises 108

8 Modules . . . 109
8.1 Introduction to Modules 109
8.2 `Numpy` module 110
8.2.1 Basic array creation . . . 110
8.2.2 Indexing a NumPy array . . . 111
8.2.3 Initializing an array . . . 111
8.2.4 NumPy array attributes . . . 113

8.2.5 Universal functions 113
8.2.6 Numpy (`dot`) function 113
8.2.7 Basic Arithmetic 114
8.2.8 NumPy's special values, `nan` and `inf` 114
8.2.9 Copy an array 115
8.2.10 Changing the shape of an array 115
8.2.11 Concatenate arrays 116
8.2.12 Constants and special symbols 117
8.2.13 Standard Functions 117
8.2.14 Merging and Splitting Arrays 117

8.3 `Matplotlib` module 118

8.3.1 An Interactive Session with PyPlot 118
8.3.2 Basic plotting 119
8.3.3 Plot components 120
8.3.4 Line graph 121
8.3.5 Scatter Graph 124
8.3.6 Bar graph 124

9 Reading and Writing Files 127

9.1 Reading and writing a file 127

9.1.1 Opening a file 127
9.1.2 Reading input from a file 127
9.1.3 Writing an output to a file 129

9.2 Reading and writing a NumPy array to a file 130

1. UNIX Operating System

1.1 What is UNIX?

UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available.

1.2 Types of UNIX

There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, and MacOS X.

1.3 The UNIX operating system

The UNIX operating system is made up of three parts; the kernel,System calls and the shell.

The kernel

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the filestore and communications in response to system calls.

As an illustration of the way that the shell and the kernel work together, suppose a user types **rm myfile** (which has the effect of removing the file **myfile**). The shell searches the filestore for the file containing the program **rm**, and then requests the kernel, through system calls, to execute the program **rm** on **myfile**. When the process **rm myfile** has finished running, the shell then returns the UNIX prompt **%** to the user, indicating that it is waiting for further commands.

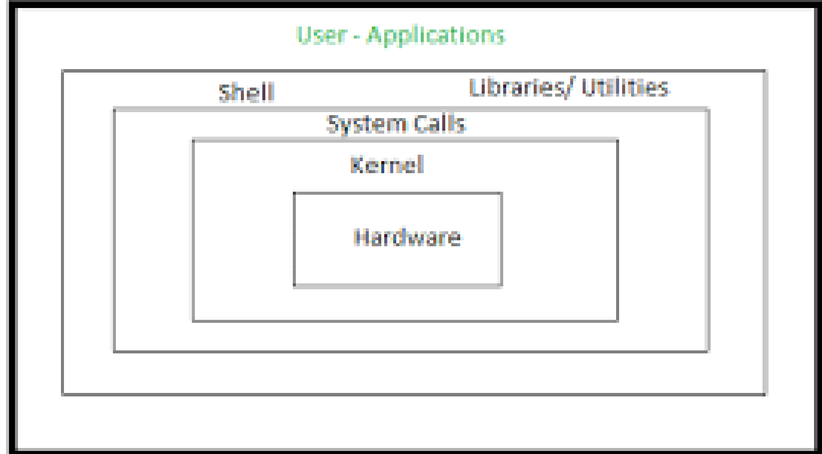


Figure 1.1: Unix Architecture

System Calls

A system call is a way for programs to interact with the operating system. A computer program makes a system call when it makes a request to the operating system's kernel. System call provides the services of the operating system to the user programs via Application Program Interface(API). System calls can be grouped roughly into six major categories: (i) Process control (create process, terminate process, load, execute, etc.); (ii) File management (create file, delete file, open, close, read, write, reposition get/set file attributes, etc.); (iii) Device management (request device, release device, read, write, reposition, get/set device attributes, etc.); (iv) Information maintenance (get/set time or date, get/set system data, etc.); (v) Communication (create, delete communication connection, send, receive messages, etc.); (vi) Protection (get/set file permissions).

The shell

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems).

The adept user can customise his/her own shell, and users can use different shells on the same machine.

For example, it is possible to use the function *Filename Completion* - By typing part of the name of a command, filename or directory and pressing the [**Tab**] key, the shell will complete the rest of the name automatically. If the shell finds more than one name beginning with those letters you have typed, it will beep, prompting you to type a few more letters before pressing the tab key again.

History - The shell keeps a list of the commands you have typed in. If you need to repeat a command, use the cursor keys to scroll up and down the list or type history for a list of previous commands

1.4 Files and processes

Everything in UNIX is either a file or a process. A process is an executing program identified by a unique PID (process identifier).

A file is a collection of data. They are created by users using text editors, running compilers etc.

Examples of files:

- a document (report, essay etc.)
- the text of a program written in some high-level programming language
- instructions comprehensible directly to the machine and incomprehensible to a casual user, for example, a collection of binary digits (an executable or binary file);
- a directory, containing information about its contents, which may be a mixture of other directories (subdirectories) and ordinary files.

1.5 The Directory Structure

All the files are grouped together in the directory structure. The file-system is arranged in a hierarchical structure, like an inverted tree. The top of the hierarchy is traditionally called **root** (written as a slash **/**)

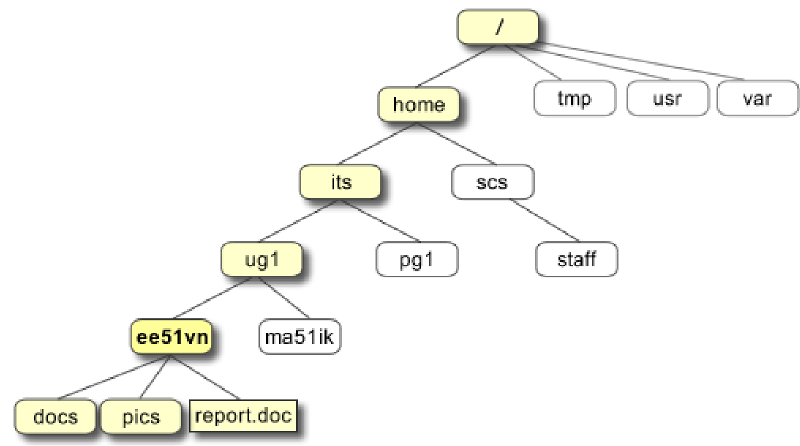


Figure 1.2: Unix Tree.

In the diagram above, we see that the home directory of the undergraduate student **"ee51vn"** contains two sub-directories (**docs** and **pics**) and a file called *report.doc*.

The full path to the file *report.doc* is *"/home/its/ug1/ee51vn/report.doc"*

1.6 Starting an UNIX terminal

To open an UNIX terminal window, click on the "Terminal" icon from Applications/Accessories menus. An UNIX Terminal window will then appear with a % prompt, waiting for you to start entering commands.

Figure 1.3: Unix Terminal

1.7 Listing files and directories

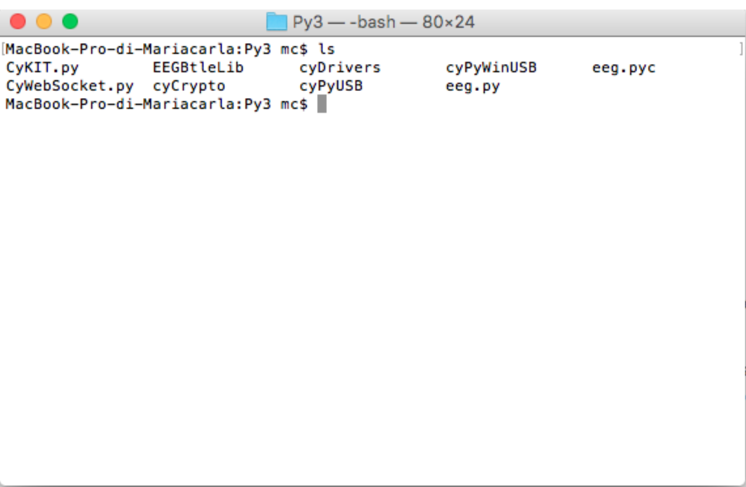
ls (list)

When you first login, your current working directory is your home directory. Your home directory has the same name as your user-name, for example, **ee91ab**, and it is where your personal files and subdirectories are saved.

To find out what is in your home directory, type

```
% ls
```

The ls command (lowercase L and lowercase S) lists the contents of your current working directory.



ls does not, in fact, cause all the files in your home directory to be listed, but only those ones whose name does not begin with a dot (.) Files beginning with a dot (.) are known as hidden files and usually contain important program configuration information. They are hidden because you should not change them unless you are very familiar with UNIX!!!

To list all files in your home directory including those whose names begin with a dot, type

```
% ls -a
```

As you can see, **ls -a** lists files that are normally hidden. **ls** is an example of a command which can

```
Py3 — -bash — 80×24
MacBook-Pro-di-Mariacarla:Py3 mc$ ls
CyKIT.py        EEGBtleLib      cyDrivers       cyPyWinUSB      eeg.pyc
CyWebSocket.py  cyCrypto        cyPyUSB         eeg.py
MacBook-Pro-di-Mariacarla:Py3 mc$ ls -a
.               CyWebSocket.py  cyDrivers       eeg.py
..              EEGBtleLib      cyPyUSB         eeg.pyc
CyKIT.py        cyCrypto        cyPyWinUSB
MacBook-Pro-di-Mariacarla:Py3 mc$
```

take options: -a is an example of an option.

```
Py3 — -bash — 80×24
MacBook-Pro-di-Mariacarla:Py3 mc$ ls
CyKIT.py        EEGBtleLib      cyDrivers       cyPyWinUSB      eeg.pyc
CyWebSocket.py  cyCrypto        cyPyUSB         eeg.py
MacBook-Pro-di-Mariacarla:Py3 mc$ ls -la
total 328
drwxr-xr-x@ 11 mc  staff    374  3 Mar 14:35 .
drwxr-xr-x@  6 mc  staff    204  4 Feb 14:04 ..
-rwxr-xr-x@  1 mc  staff  11307  4 Feb 14:04 CyKIT.py
-rwxr-xr-x@  1 mc  staff  15709  4 Feb 14:04 CyWebSocket.py
drwxr-xr-x@  6 mc  staff    204  4 Feb 14:04 EEGBtleLib
drwxr-xr-x@  5 mc  staff    170  4 Feb 14:04 cyCrypto
drwxr-xr-x@  6 mc  staff    204  4 Feb 14:04 cyDrivers
drwxr-xr-x@ 14 mc  staff    476  4 Feb 14:04 cyPyUSB
drwxr-xr-x@  9 mc  staff    306  4 Feb 14:04 cyPyWinUSB
-rwxr-xr-x@  1 mc  staff  84662  4 Feb 14:04 eeg.py
-rw-r--r--@  1 mc  staff  49580  3 Mar 14:35 eeg.pyc
MacBook-Pro-di-Mariacarla:Py3 mc$
```

The options change the behaviour of the command. There are online manual pages that tell you which options a particular command can take, and how each option modifies the behaviour of the command. . (See later in this tutorial) another option of **ls** is **la**, which lists all the files and subdirectories and their access permissions.

1.8 Making Directories

mkdir (make directory)

We will now make a subdirectory in your home directory to hold the files you will be creating and using in the course of this tutorial. To make a subdirectory called unixstuff in your current working directory type

```
% mkdir unixstuff
```

To see the directory you have just created, type

```
% ls
```

1.9 Changing to a different directory

cd (change directory)

The command **cd *directory*** means change the current working directory to 'directory'. The current working directory may be thought of as the directory you are in, i.e. your current position in the file-system tree.
To change to the directory you have just made, type

```
% cd user
```

Type **ls** to see the contents (which should be empty)

Exercise 1
Make another directory inside the *user* directory called *backups*

1.10 The directories . and ..

Still in the *user* directory, type

```
% ls -a
```

The current directory (.)
In UNIX, (.) means the current directory, so typing

```
% cd .
```

> Note: there is a space between cd and the dot

means stay where you are (the user directory). This may not seem very useful at first, but using (.) as the name of the current directory will save a lot of typing, as we shall see later in the tutorial.

The parent directory (..)
(..) means the parent of the current directory, so typing

```
% cd ..
```

will take you one directory up the hierarchy (back to your home directory). Try it now.

> Note: typing cd with no argument always returns you to your home directory. This is very useful if you are lost in the file system.

1.11 Pathnames

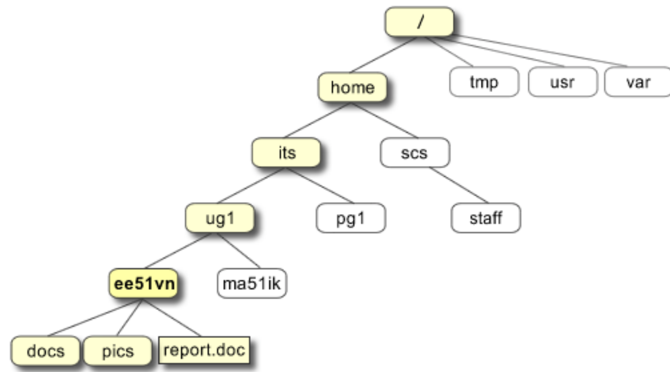
pwd

Pathnames enable you to work out where you are in relation to the whole file-system. For example, to find out the absolute pathname of your home-directory, type **cd** to get back to your home-directory and then type

```
% pwd
```

The full pathname will look something like this -

```
/home/its/ug1/ee51vn
```



which means that *ee51vn* (your home directory) is in the sub-directory *ug1* (the group directory),which in turn is located in the *its* sub-directory, which is in the home sub-directory, which is in the top-level root directory called "/" .

Exercise 2
Use the commands **cd, ls** and **pwd** to explore the file system. (Remember, if you get lost, type **cd** by itself to return to your home-directory)

1.12 Copying Files

cp (copy)

cp *source destination* is the command which makes a copy of *source* that is a file in the directory *destination*.
What we are going to do now, is to take a file stored in an open access area of the file system, and use the cp command to copy it to your unixstuff directory.
First, **cd** to your **unixstuff** directory.

```
% cd ~/unixstuff
```

Then at the UNIX prompt, type,

Command	Meaning
ls	list files and directories
ls -a	list all files and directories
mkdir	make a directory
cd *directory*	change to named directory
cd	change to home-directory
cd ~	change to home-directory
cd ..	change to parent directory
pwd	display the path of the current directory

Figure 1.4: Summary of Unix Commands Part 1

```
% cp /vol/ee/ee-info/Teaching/Unix/science.txt .
```

Note: Don't forget the dot . at the end. Remember, in UNIX, the dot means the current directory.

The above command means copy the file *science.txt* to the current directory.

Exercise 3
Create a backup of your *science.txt* file by copying it to a directory called *backups* previously created into your unixstuff directory.

1.13 Moving files

mv (move)

mv *file1 file2* moves (or renames) file1 to file2
To move a file from one place to another, use the mv command. This has the effect of moving rather than copying the file, so you end up with only one file rather than two.
It can also be used to rename a file, by moving the file to the same directory, but giving it a different name.
We are now going to move the file science.bak in the current directory to directory named backups contained the same in the current directory.

```
% mv science.bak backups
```

Type ls and ls backups to see if it has worked.

1.14 Removing files and directories

rm (remove), rmdir (remove directory)

To delete (remove) a file, use the **rm** command.
As an example, we are going to create a copy of the *science.txt* file then delete it.
Inside your unixstuff directory, type

```
% cp science.txt tempfile.txt
% ls
% rm tempfile.txt
% ls
```

You can use the **rmdir** command to remove a directory (make sure it is empty first). Try to remove the backups directory. You will not be able to since UNIX will not let you remove a non-empty directory.

Exercise 4
Create a directory called *tempstuff* using mkdir, then remove it using the **rmdir** command.

1.15 Displaying the contents of a file on the screen

clear (clear screen)

Before you start the next section, you may like to clear the terminal window of the previous commands so the output of the following commands can be clearly understood.
At the prompt, type

```
% clear
```

This will clear all text and leave you with the % prompt at the top of the window.

cat (concatenate)

The command cat can be used to display the contents of a file on the screen. Type:

```
% cat science.txt
```

As you can see, the file is longer than than the size of the window, so it scrolls past making it unreadable.

less

The command less writes the contents of a file onto the screen a page at a time. Type

```
% less science.txt
```

Press the **[space-bar]** if you want to see another page, and type **[q]** if you want to quit reading. As you can see, less is used in preference to cat for long files.

head

The **head** command writes the first ten lines of a file to the screen. First clear the screen then type

```
% head science.txt
```

Then type

```
% head -5 science.txt
```

What difference did the -5 do to the head command?

tail

The **tail** command writes the last ten lines of a file to the screen.

```
% tail science.txt
```

Q. How can you view the last 15 lines of the file?

1.16 Searching the contents of a file

less

Using **less**, you can search though a text file for a keyword (pattern). For example, to search through *science.txt* for the word 'science', type

```
% less science.txt
```

then, still in **less**, type a forward slash [/] followed by the word to search

```
/science
```

As you can see, **less** finds and highlights the keyword. Type [**n**] to search for the next occurrence of the word.

grep

grep is one of many standard UNIX utilities. It searches files for specified words or patterns. First clear the screen, then type

```
% grep science science.txt
```

As you can see, **grep** has printed out each line containg the word *science*. Or has it ???? Try typing

```
% grep Science science.txt
```

The grep command is case sensitive; it distinguishes between Science and science. To ignore upper/lower case distinctions, use the -i option, i.e. type

```
% grep -i science science.txt
```

To search for a phrase or pattern, you must enclose it in single quotes (the apostrophe symbol). For example to search for spinning top, type

```
% grep -i 'spinning top' science.txt
```

Some of the other options of grep are:
- **v** display those lines that do NOT match
- **n** precede each matching line with the line number
- **c** print only the total count of matched lines

Try some of them and see the different results. Don't forget, you can use more than one option at a time. For example, the number of lines without the words science or Science is

```
% grep -ivc science science.txt
```

wc (word count)

A handy little utility is the wc command, short for word count. To do a word count on science.txt, type

```
% wc -w science.txt
```

To find out how many lines the file has, type

```
% wc -l science.txt
```

1.17 Redirection

Most processes initiated by UNIX commands write to the standard output (that is, they write to the terminal screen), and many take their input from the standard input (that is, they read it from the keyboard). There is also the standard error, where processes write their error messages, by default, to the terminal screen.

cat

We have already seen one use of the **cat** command to write the contents of a file to the screen. Now type **cat** without specifing a file to read

```
% cat
```

Then type a few words on the keyboard and press the [Return] key.

Finally hold the [**Ctrl**] key down and press [**d**] (written as **Ctrl+D** for short) to end the input. What has happened?
If you run the cat command without specifying a file to read, it reads the standard input (the keyboard), and on receiving the 'end of file' (**Ctrl+D**), copies it to the standard output (the screen).
In UNIX, we can redirect both the input and the output of commands.

Command	Meaning
cp *file1 file2*	copy file1 and call it file2
mv *file1 file2*	move or rename file1 to file2
rm *file*	remove a file
rmdir *directory*	remove a directory
cat *file*	display a file
less *file*	display a file a page at a time
head *file*	display the first few lines of a file
tail *file*	display the last few lines of a file
grep *'keyword' file*	search a file for keywords
wc *file*	count number of lines/words/characters in file

Figure 1.5: Summary of Unix Commands Part 2

1.18 Redirecting the Output

>

We use the **>** symbol to redirect the output of a command. For example, to create a file called list1 containing a list of fruit, type

```
% cat > list1
```

Then type in the names of some fruit. Press [Return] after each one.

```
pear
banana
apple
^D {this means press [Ctrl] and [d] to stop}
```

What happens is the cat command reads the standard input (the keyboard) and the > redirects the output, which normally goes to the screen, into a file called list1
To read the contents of the file, type

```
% cat list1
```

Exercise 5
Using the above method, create another file called list2 containing the following fruit: orange, plum, mango, grapefruit. Read the contents of list2

1.19 Appending to a file

»

The form » appends standard output to a file. So to add more items to the file *list1*, type

```
% cat >> list1
```

Then type in the names of more fruit

```
peach
grape
orange
^D (Control D to stop)
```

To read the contents of the file, type

```
% cat list1
```

You should now have two files. One contains six fruit, the other contains four fruit.
We will now use the cat command to join (concatenate) *list1* and *list2* into a new file called *biglist*. Type

```
% cat list1 list2 > biglist
```

What this is doing is reading the contents of *list1* and *list2* in turn, then outputing the text to the file *biglist*
To read the contents of the new file, type

```
% cat biglist
```

1.20 Redirecting the Input

<

We use the < symbol to redirect the input of a command.
The command sort alphabetically or numerically sorts a list. Type

```
% sort
```

Then type in the names of some animals. Press [Return] after each one.

```
dog
cat
bird
ape
^D (control d to stop)
```

The output will be

```
ape
bird
cat
dog
```

Using < you can redirect the input to come from a file rather than the keyboard. For example, to sort the list of fruit, type

```
% sort < biglist
```

and the sorted list will be output to the screen.
To output the sorted list to a file, type,

```
% sort < biglist > slist
```

Use cat to read the contents of the file slist

1.21 Pipes

|

To see who is on the system with you, type

```
% who
```

One method to get a sorted list of names is to type,

```
% who > names.txt
% sort < names.txt
```

This is a bit slow and you have to remember to remove the temporary file called names when you have finished. What you really want to do is connect the output of the who command directly to the input of the sort command. This is exactly what pipes do. The symbol for a pipe is the vertical bar |
For example, typing

```
% who | sort
```

will give the same result as above, but quicker and cleaner.
To find out how many users are logged on, type

```
% who | wc -l
```

Exercise 6
Using pipes, display all lines of list1 and list2 containing the letter 'p', and sort the result.

Command	Meaning
command > *file*	redirect standard output to a file
command >> *file*	append standard output to a file
command < *file*	redirect standard input from a file
command1 | *command2*	pipe the output of command1 to the input of command2
cat *file1 file2* > *file0*	concatenate file1 and file2 to file0
sort	sort data
who	list users currently logged in

Figure 1.6: Summary of Unix Commands Part 2

1.22 Setting a Virtual Machine

In order to ease the use of Unix system (for those who do not use Linux installed on own computer), we will use a virtual machine. To achieve this aim, we need two components:

- a virtual machine image of a Unix system version. We will use Ubuntu 18.04.
- a software to launch our virtual machine such as VirtualBox[1].

Please, download the virtual machine image by the folder "File" in Microsoft Teams and install the software VirtualBox. Once you have on your computer both these components, perform the following procedure to create and start your virtual machine running Ubuntu 18.04.

- To create a new virtual machine, you need to start VirtualBox. The Oracle VM VirtualBox Manager is displayed, as shown in Fig. 1.7.

Figure 1.7: VirtualBox main window

- In the toolbar, click the "Aggiungi" button. The New Virtual Machine Wizard is displayed in a new window, as shown in Fig. 1.8. Add information like in 1.8.

[1] https://www.virtualbox.org/

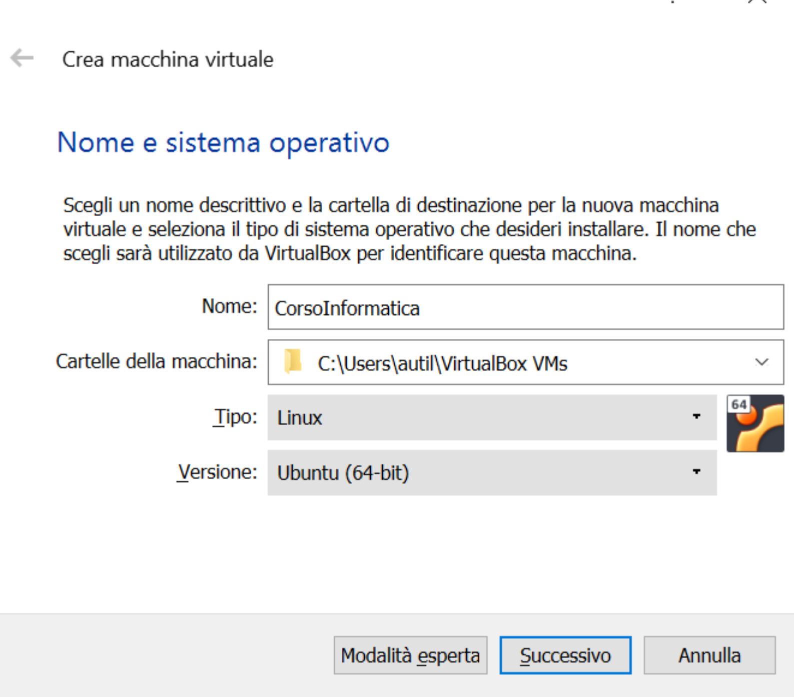


Figure 1.8: VirtualBox window: create new virtual machine

- In the new window (see Fig. 1.9), select the memory space as shown. If your computer has less than 8GB of RAM, select a smaller memory space and click "Successivo".
- In the new window (see Fig. 1.10), select the last option and click on the folder icon. In this way, it will be possible to select the virtual machine image (see Fig. 1.11).
- In the window (see Fig. 1.12) appears the selected image. Click on "Scegli".
- In the window (see Fig. 1.13) click on "Crea".
- In the VirtualBox window (see Fig. 1.14) appears the created virtual machine on the left. Click on "Avvia" to start the virtual machine.
- Linux system starts (see Fig. 1.15). Insert the password: "osboxes.org".

Figures in this section are related to VirtualBox running on Windows platform.

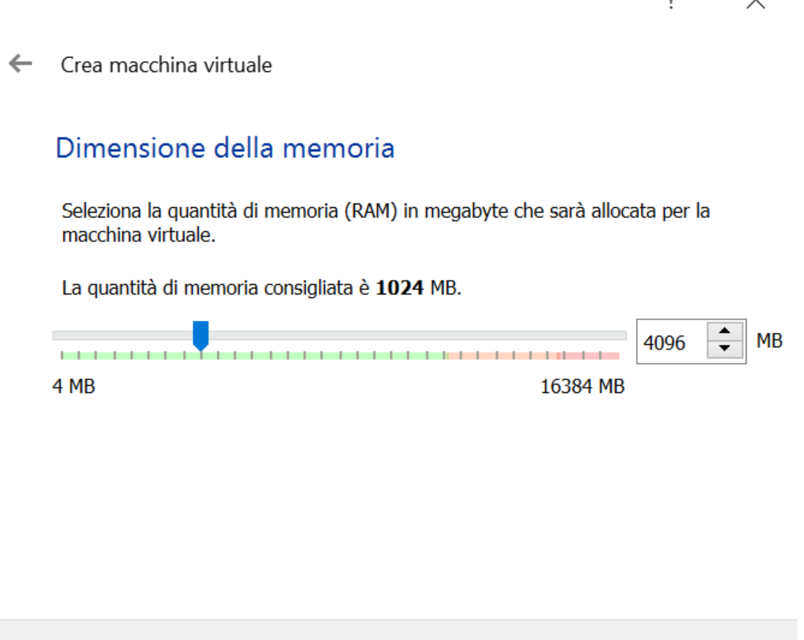


Figure 1.9: VirtualBox window: select memory space

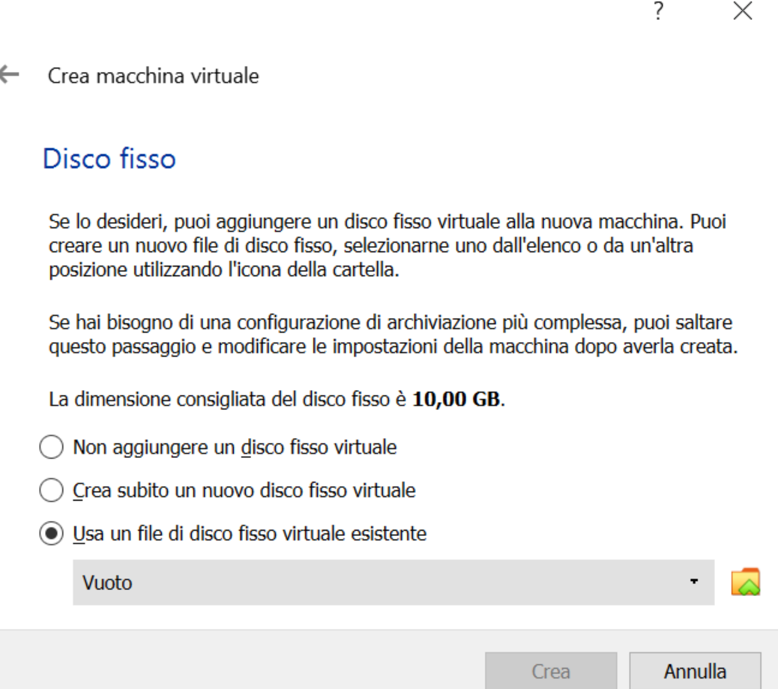


Figure 1.10: VirtualBox window: select memory space

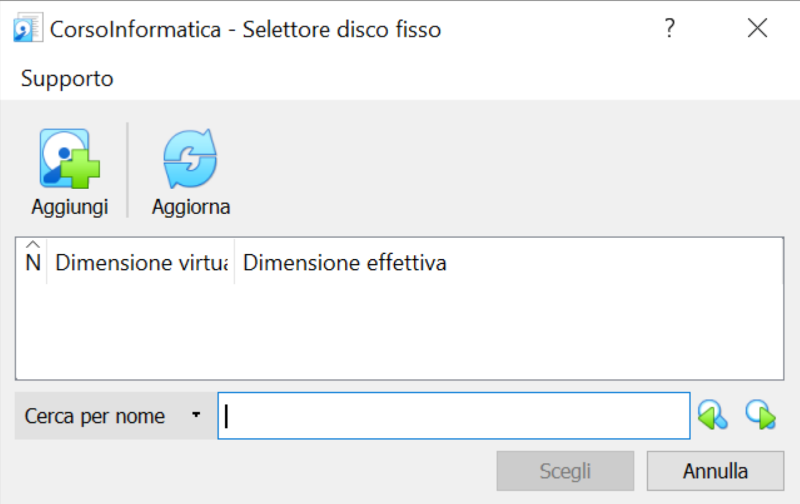


Figure 1.11: VirtualBox window: select virtual machine image

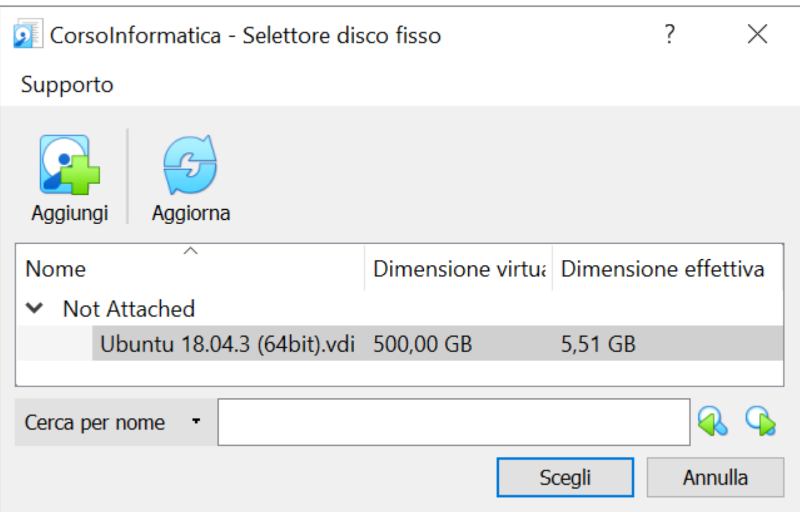


Figure 1.12: VirtualBox window: select virtual machine image

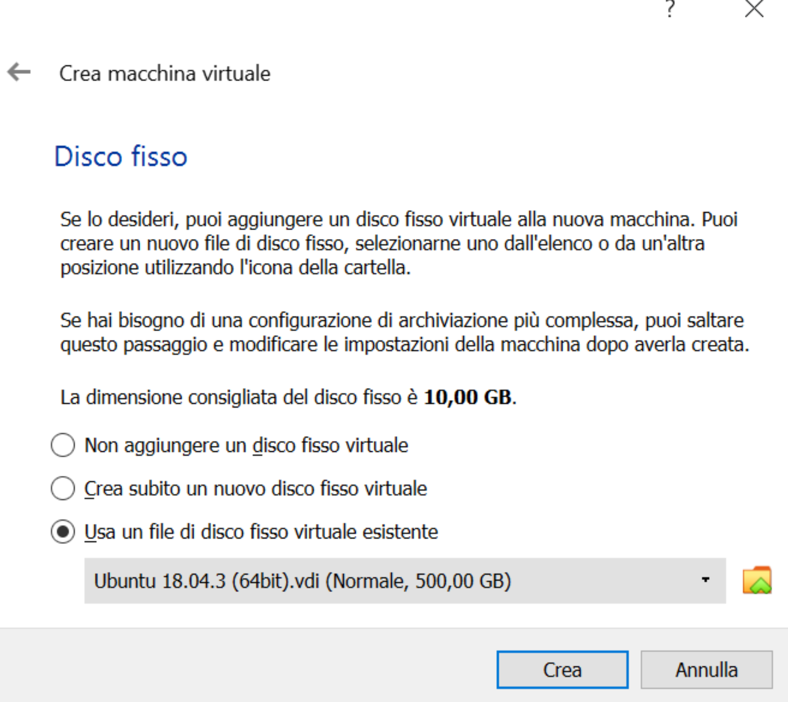


Figure 1.13: VirtualBox window: create virtual machine image

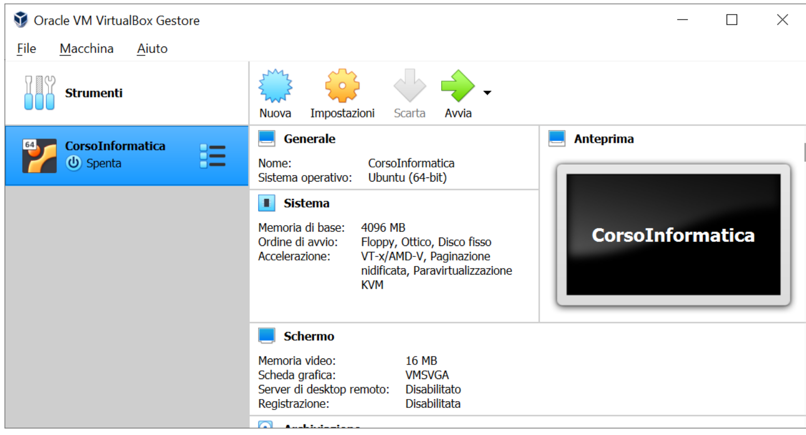


Figure 1.14: VirtualBox window: start virtual machine

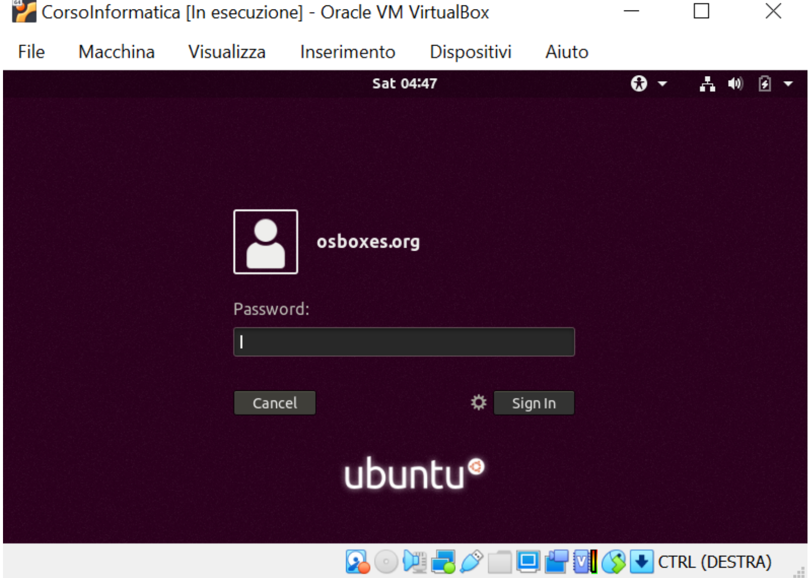


Figure 1.15: Login Ubuntu 18.04

2. Introduction to Python

This chapter presents an introduction to programming languages and, in particular, to the Python language. In particular, this chapter discusses its history, the motivations for its success and details how it works. Contents of this chapter are extracted in part from the following sources [GV18][Ste15][Hun19a][Ced10].

2.1 Introduction to programming

An *algorithm* is a finite sequence of effective steps that solve a problem. A step is effective if it is unambiguous and possible to perform. The number of steps must be finite (rather than infinite) so that all of the steps can be completed. Recipes, assembly instructions for furniture or toys, and the steps needed to open a combination lock are examples of algorithms that we encounter in everyday life. A *computer program* is a sequence of instructions that control the behaviour of a computer. The instructions tell the computer when to perform tasks like reading input and displaying results, and how to transform and manipulate values to achieve a desired outcome.

An algorithm must be translated into a computer program before a computer can be used to solve a problem. The translation process is called programming and the person who performs the translation is referred to as a programmer. The mean used to perform the translation process are the computer *programming languages*. In other words, a computer program is an algorithm written by using a programming language.

2.2 Programming languages

Computers cannot write programs on their own as they do not understand human needs unless we communicate with the computer through programming languages. A programming language is a computer language engineered to communicate instructions to a machine. There are different types of programming languages.

2.2.1 Machine Language

Machine language, also called machine code, is a low-level computer language that is designed to be directly understandable by a computer and it is the language into which all programs must be converted before they can be run. It is entirely comprised of binary, 0's and 1's. In machine language, all instructions, memory locations, numbers and characters are represented in 0's and 1's. For example, a typical piece of machine language might look like, 00000100 10000000.

The main advantage of machine language is that it can run and execute very fast as the code will be directly executed by a computer and the programs efficiently utilize memory. Some of the disadvantages of machine language are: (i) Machine language is almost impossible for humans to use because it consists entirely of numbers; (ii) Machine language programs are hard to maintain and debug; (iii) Machine language has no mathematical functions available; (iv) Memory locations are manipulated directly, requiring the programmer to keep track of every memory location.

2.2.2 Assembly Language

Machine language is extremely difficult for humans to read because it consists merely of patterns of bits (i.e., 0's and 1's). Thus, programmers who want to work at the machine language level instead usually use assembly language, which is a human-readable notation for the machine language. Assembly language replaces the instructions represented by patterns of 0's and 1's with alphanumeric symbols also called as mnemonics in order to make it easier to remember and work with them including reducing the chances of making errors. For example, the code to perform addition and subtraction is,

```
ADD 3, 5, result
SUB 1, 2, result
```

The use of mnemonics is an advantage over machine language. Since the computer cannot understand assembly language, a program called *assembler* converts the alphanumeric symbols written in assembly language to machine language and this machine language can be directly executed on the computer. Some of the disadvantages of Assembly language are: (i) there are no symbolic names for memory locations; (ii) it is difficult to read; (iii) assembly language is machine-dependent making it difficult for portability.

2.2.3 High-level language

High-level language is more like human language and less like machine language. High-level languages are written in a form that is close to our human language, enabling programmers to just focus on the problem being solved. High-level languages are platform independent which means that the programs written in a high-level language can be executed on different types of machines.

A program written in the high-level language is called *source program* or source code and is any collection of human-readable computer instructions. However, for a computer to understand and execute a source program written in high-level language, it must be translated into machine language. This translation is done using either *compiler* or *interpreter*. The advantages of high-level programming languages are: (i) they are easier to modify, faster to write code and debug as it uses English like statements; (ii) they implement portable code, as it is designed to run on multiple machines.

A wide variety of different languages have been created, each of which has its own strengths and weaknesses. Popular programming languages currently include Java, C++, JavaScript, PHP, C# and

Python, among others. In our course, we will study Python programming language, and so, details on this language and motivations for its choice are given in the next sections.

2.3 What is Python?

Python is a free general-purpose programming language whose the history dates back to the late 1980s. More in detail, Python was conceived in the late 1980s and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands. The first ever version of Python (i.e., Python 1.0) was introduced in 1991. Python 2 was launched in October 2000 and has been, and still is, very widely used. However, Python 2 end of life plan was initially announced back in 2015, and, it is expected to 2020. Python 3.0 was released on 3 December 2008 after a long testing period. It is a major revision of the language that is not completely backward-compatible with previous versions. In this course, we will work with Python 3.x version.

Python is an interpreted and multi-paradigm programming language. These features will be discussed in the remaining of this section. As well as the core language, there are very many libraries available for Python. These libraries extend the functionality of the language and make it much easier to develop applications. These libraries cover web frameworks such as Django/Flask, email clients such as smtplib (a SMTP email client) and imaplib (an IMAP4 email client), the generation of Microsoft Excel files using the Python Excel library, graphics libraries such as Matplotlib and PyOpenGL, machine learning using libraries such as SKLearn and TensorFlow. How to manage libraries in Python will be discussed in the next chapters.

2.3.1 Python interpreter

A computer to understand and execute a source program written in high-level language, it must be translated into machine language. This translation is done using either compiler or interpreter.

Briefly, a compiler is a system software program that transforms high-level source code written by a software developer in a high-level programming language into a low-level machine language. The process of converting high-level programming language into machine language is known as compilation. Compilers translate source code all at once and the computer then executes the machine language that the compiler produced. The generated machine language can be later executed many times against different data each time. Programming languages like C, C++, C# and Java use compilers.

Python is not a pre-compiled language. It is what is known as an interpreted language (although even this is not quite accurate). Indeed, the Python source code is executed directly using an *interpreter* rather than first compiling it and then executing the resulting machine language. An interpreter is a program that reads source code one statement at a time, translates the statement into machine language, executes the machine language statement, then continues with the next statement.

Python actually uses an intermediate model in that it actually converts the plain text English style Python program into an intermediate 'pseudo' machine code format and it is this intermediate format that is executed. This is illustrated in Fig. 2.1.

The way in which the Python interpreter processes a Python program is broken down into several steps. The steps shown here are illustrative (and simplified) but the general idea is correct.

- First the program is checked to make sure that it is valid Python. That is a check is made that the program follows all the rules of the language and that each of the commands and operations etc. is understood by the Python environment.

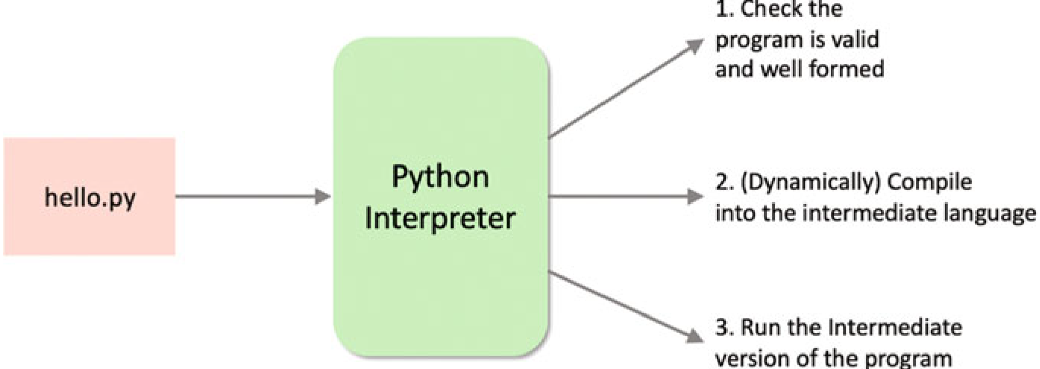


Figure 2.1: Python interpreter [Hun19a]

- It then translates the plain text, English like commands, into a more concise intermediate format that is easier to execute on a computer. Python can store this intermediate version in a file which is named after the original file but with a '. pyc' extension instead of a '.py' extension (the 'c' in the extension indicates it contains the compiled version of the code).
- The compiled intermediate version is then executed by the interpreter. When this program is rerun, the Python interpreter checks to see if a '.pyc' file is present. If no changes have been made to the source file since the '.pyc' was created, then the interpreter can skip steps 1 and 2 and immediately run the '.pyc' version of the program.

It is generally faster to run compiled code than to run a program under an interpreter. This is largely because the interpreter must analyze each statement in the source code each time the program is executed and then perform the desired conversion, whereas this is not necessary with compiled code because the source code was fully analyzed during compilation. However, the intermediate model used by Python makes Python program execution faster with respect to other common interpreters. Moreover, it can take less time to interpret source code than the total time needed to both compile and run it, and thus interpreting strategy is the most appropriate when developing and testing source code for new programs.

2.3.2 Python programming paradigm

There are several different programming paradigms that a programming language may allow developers to code in, these are:

- **Procedural Programming** in which a program is represented as a sequence of instructions that tell the computer what it should do explicitly. Procedures and/or functions are used to provide structure to the program; with control structures such as if statements and loop constructs to manage which steps are executed and how many times. Languages typifying this approach include C and Pascal.
- **Declarative Programming language** such as Prolog and SQL, that allow developers to describe how a problem should be solved, with the language/environment determining how the solution should be implemented.
- **Object Oriented Programming** approaches that represent a system in terms of the objects that form that system. Each object can hold its own data (also known as state) as well as define behaviour that defines what the object can do. A computer program is formed from a set of these objects co-operating together. Languages such as Java and C# typify the object oriented

approach.

- **Functional Programming languages** decompose a problem into a set of functions. Each function is independent of any external state, operating only on the inputs they received to generate their outputs. The programming language Haskell is an example of a functional programming language.

Some programming languages are considered to be hybrid languages; that is they allow developers to utilise a combination of difference approaches within the same program. Python is an example of a *hybrid programming* or *multi-programming* language as it implements procedural, functional and object-oriented programming.

2.4 Why we should use Python?

Hundreds of programming languages are available today, from mature languages like C and C++, to newer entries like Ruby and C#, to enterprise juggernauts like Java. Choosing a language to learn is difficult. Although no one language is the right choice for every possible situation, Python is a good choice for a large number of programming problems, and it's also a good choice if you're learning to program. Hundreds of thousands of programmers around the world use Python, and the number grows every year. This increased interest is driven by several different factors:

- Python is readable. You might think that a programming language needs to be read only by a computer, but humans have to read your code as well: whoever debugs your code (quite possibly you), whoever maintains your code (could be you again), and whoever might want to modify your code in the future. In all of those situations, the easier the code is to read and understand, the better it is.
- Python is an excellent cross-platform language. It runs on many platforms: Windows, Mac, Linux, UNIX, and so on. Because it's interpreted, the same code can run on any platform that has a Python interpreter, and almost all current platforms have one.
- Python has the availability of a wide range of libraries (modules) that can be used to extend the basic features of the language.
- Python is free. Python was originally, and continues to be, developed under the open source model, and it's freely available. You can download and install practically any version of Python and use it to develop software for commercial or personal applications, and you don't need to pay a dime.

3. Getting Started with Python

This chapter guides you through downloading, installing, and starting up Python. At this writing, Python 3.8.2 is the most current version. After years of refinement, Python 3 is the first version of the language that is not fully backward-compatible with earlier versions, so be sure to get a version of Python 3. Contents of this chapter are extracted in part by the following sources [Mue18][17].

3.1 Installing Python

Installing Python is a simple matter, regardless of which platform you are using. The first step is to obtain a recent distribution for your machine; the most recent one can always be found at `www.python.org`. It is worth noting that this distribution does not contain all possible packages that you need. In the next chapters, we will discuss how to add packages to this distribution. Another issue is that you may already have an earlier version of Python installed on your machine. Indeed, many Linux distributions and MacOS come with Python 2.x as part of the operating system. However, we will use Python 3, but Python 3 is not compatible with Python 2, so, any platform you use, you need to install Python 3. However, there's no need to worry, because different Python versions on the same computer will not cause a conflict.

Hereafter, we discuss the installation for the three main platforms: Windows, Mac OS X, and Linux.

3.1.1 Working with Windows

Microsoft Windows does not come with Python pre-installed as standard, so you're going to have to install it yourself manually. Thankfully, it's an easy process to follow. Indeed, the installation process on a Windows system follows the same procedure that you use for other application types. The following procedure should work fine on any Windows system, whether you use the 32-bit or the 64-bit version of Python.

1. Start by opening your web browser to `www.python.org/downloads/` (see Fig. 3.1. Look for the button detailing the download link for Python 3.x.x (the current version is Python 3.8.2).

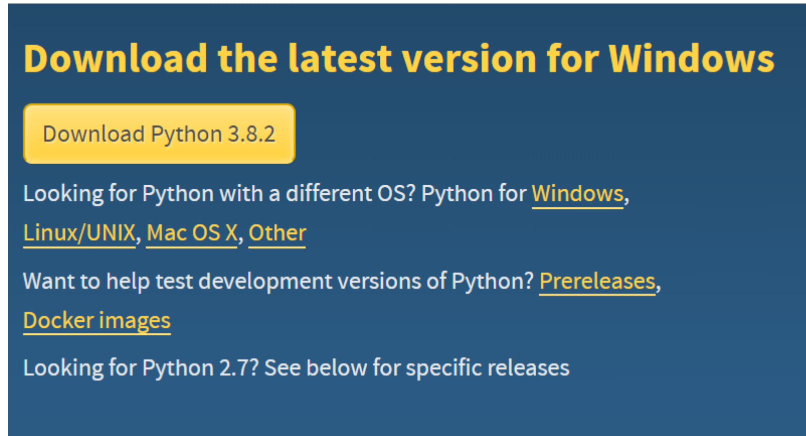


Figure 3.1: Downloading Python 3 for Windows

2. Click the download button for version 3.8.2, and save the file. When the file is downloaded, double-click the executable and the Python installation wizard will launch. In this wizard, we suggest to select "Add Python 3.8 to PATH" (at the bottom of the wizard in Fig. 3.2). This setting enables you to access Python from anywhere on your hard drive. If you do not select this setting, you must manually add Python to the path later. Moreover, from this wizard, you have two choices (see Fig. 3.2: *Install Now* and *Customise Installation*). We recommend clicking for the customise installation link.

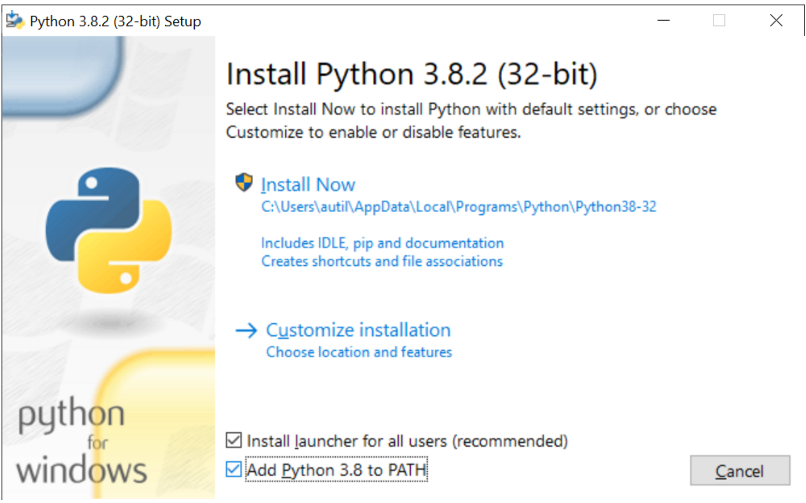


Figure 3.2: Python Installing Wizard for Windows

3. Once selected the customise installation link, we move to the next page (see Fig. 3.3, where it is possible to change the default parameters for installation. In our installation, ensure all boxes are ticked and click the Next button.
4. The next page of options (see Fig. 3.4 include some interesting additions to Python. Ensure the Associate file with Python, Create Shortcuts, Add Python to Environment Variables, Precompile Standard Library and Install for All Users options are ticked. These make using

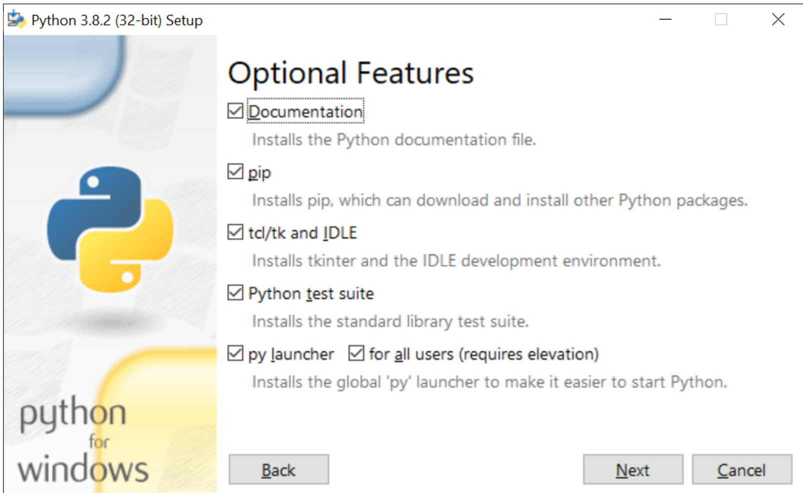


Figure 3.3: Python Installing Wizard for Windows: Custom features page

Python later much easier. Moreover, this page asks you to provide the name of an installation directory for Python. Using the default destination will save you time and effort later. However, you can install Python anywhere you desire. Finally, click Install when you're ready to continue.

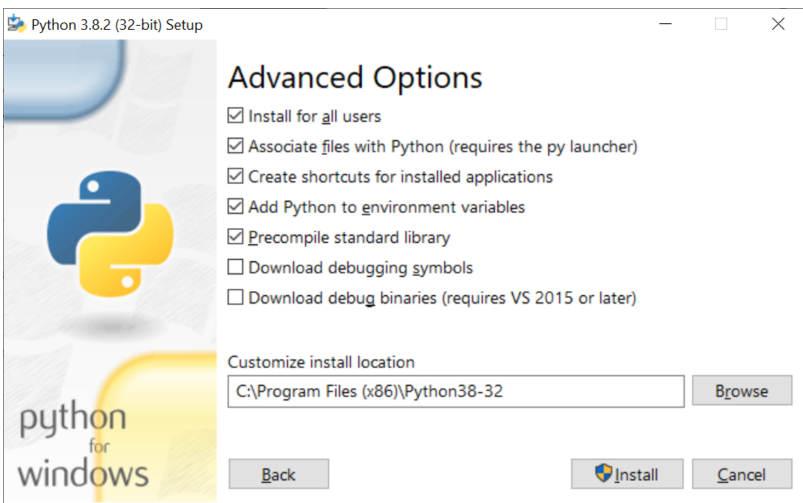


Figure 3.4: Python Installing Wizard for Windows: Advanced options page

5. You may need to confirm the installation with the Windows authentication notification. Simply click Yes and Python will begin to install. Once the installation is complete the final Python wizard page will allow you to view the latest release notes, and follow some online tutorials (see Fig. 3.5). Before you close the install wizard window, however, it's best to click on the link next to the shield detailed *Disable Path Length Limit*. This will allow Python to bypass the Windows 260 character limitation, enabling you to execute Python programs stored in deep folders arrangements. Again, click Yes to authenticate the process; then you can Close

the installation window.

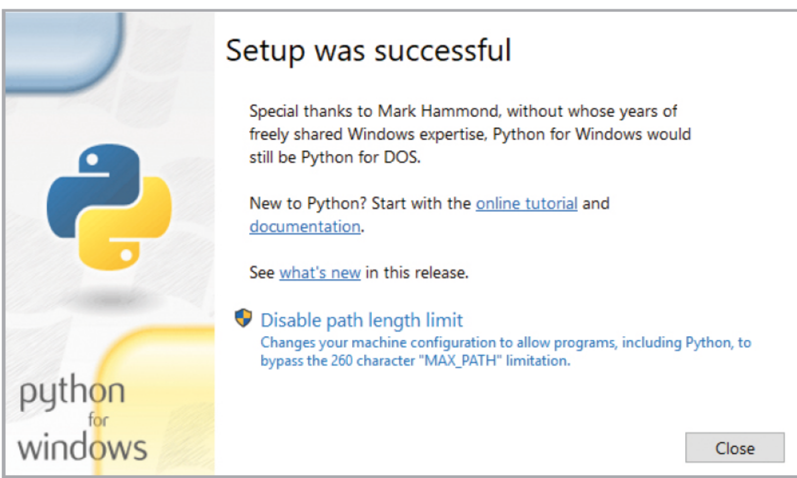


Figure 3.5: Python Installing Wizard for Windows: Final page

6. Now, we can check if the installation was successful. Open the *Command Prompt* to enter in the Windows command line environment. To enter Python within the command line, you need to type `python` and press keyboard button *Enter*. If Python has been successfully installed, the message displayed in Fig. 3.6 is shown. To leave Python command line environment, you need to type `exit()`.

```
Prompt dei comandi
Microsoft Windows [Versione 10.0.18362.720]
(c) 2019 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\autil>python
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

C:\Users\autil>
```

Figure 3.6: To enter Python in the Windows Command Prompt

3.1.2 Working with the Mac

Apple's operating system comes with Python installed. However, Apple does not update Python very often and you're probably running an older version (maybe Python 2). So it makes sense to install the last version of Python as follows.

1. Just as with the Windows set up procedure, start by opening your web browser to `www.python.org/downloads/` and look for the button detailing the download link for Python 3.x.x (the current version is Python 3.8.2).
2. Click the download button for version 3.8.2. This will automatically download the latest version of Python and depending on how you have got your Mac configured, it automatically

starts the installation wizard (see Fig. 3.7). With the Python installation wizard open, click on the Continue button to begin the installation.

Figure 3.7: Python Installing Wizard for the Mac

3. Once clicked the Continue button, the next page contains some information. When ready, click Continue again. The next page contains details related the Software License Agreement. When ready, click Continue again.
4. Finally you are be presented with the amount of space Python will take up on your system and an Install button (see Fig. 3.8), which you need to click to start the actual installation of Python 3.x on to your Mac. You may need to enter your password to authenticate the installation process. Once the installation is ended, the Installation Successful prompt will be displayed (see Fig. 3.9).

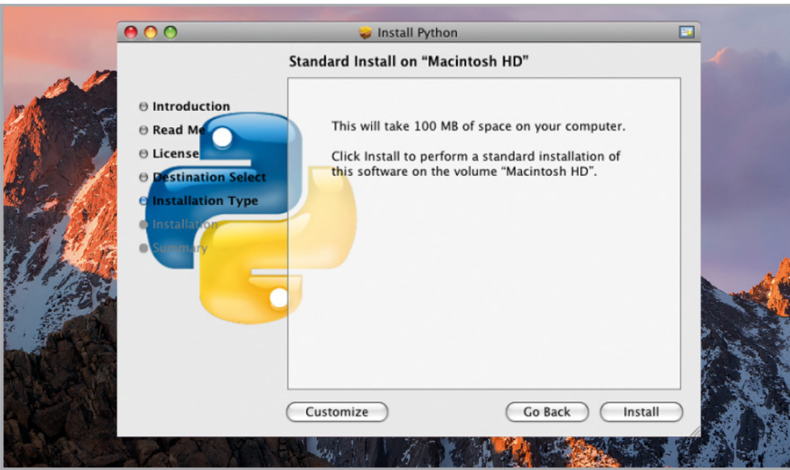


Figure 3.8: Python Installing Wizard for the Mac: Installing Python dialog box

5. Now, we can check if the installation was really successful. Open the *Terminal* and enter the command `python3` and press keyboard button *Enter*. If Python has been successfully installed,

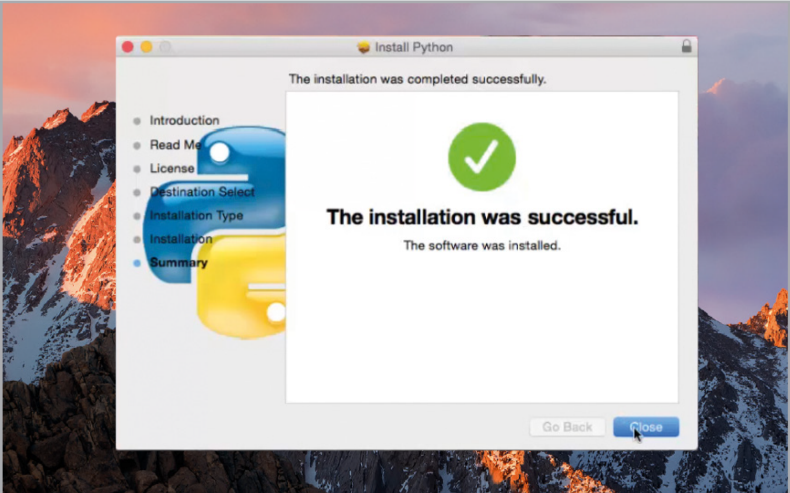


Figure 3.9: Python Installing Wizard for the Mac: Install Succeeded dialog box

the message displayed in Fig. 3.10 is shown. To leave Python command line environment, you need to type `exit()`.

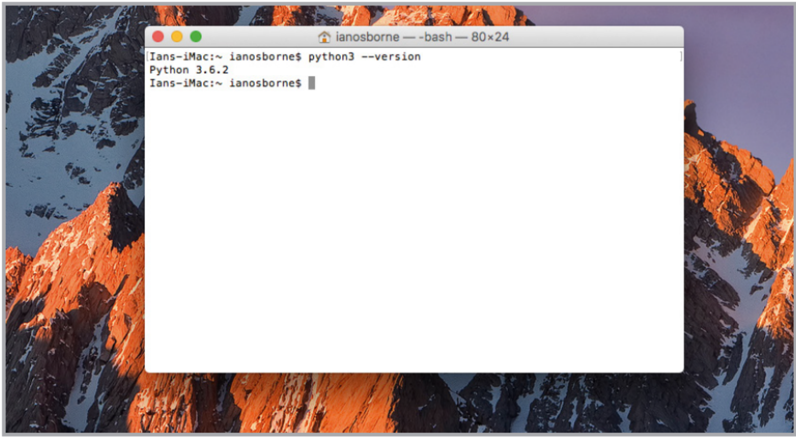


Figure 3.10: To enter Python in the Mac Terminal

3.1.3 Working with Linux

Python version 2.x is already installed in most Linux distributions but as we're going to be using Python 3.x, there's a little work we need to do first to get hold of it. Linux is such a versatile operating system that it's often difficult to nail down just one way of doing something. Different distributions go about installing software in different ways. However, in this chapter, the standard Linux installation using terminal commands that should work on any system is shown.

1. If you have never built any software on your system before, you must install the build essentials, SQLite, and bzip2 or the Python installation will fail. Therefore, you need to type these commands into the Terminal (press keyboard button *Enter* after every command):

```
sudo apt-get install build-essential
sudo apt-get install libsqlite3-dev
sudo apt-get install libbz2-dev
```

2. Just as with the Windows set up procedure, open your web browser to `www.python.org/downloads/` and look for the button detailing the download link for Python 3.x.x (the current version is Python 3.8.2) to download the source Python-3.8.2.tar.xz file.
3. In the Terminal, go the Downloads folder by entering: `cd Downloads/`. Then unzip the contents of the downloaded Python source code with: `tar -xvf Python-3.6.2.tar.xz`. Now enter the newly unzipped folder with `cd Python-3.6.2/`.
4. Within the Python folder, enter:

```
./configure
sudo make altinstall
```

5. Now, we can check if the installation was really successful. Open the *Terminal* and enter the command `python3` and press keyboard button *Enter*. If Python has been successfully installed, a message like that displayed for Windows and the Mac is shown.

3.2 Anaconda: an alternative Python distribution

In addition to the distribution of Python that you can get directly from `Python.org`, a distribution called Anaconda is gaining popularity. Anaconda is a free and open source distribution of the Python programming language for data science and machine-learning related applications such as large-scale data processing, predictive analytics, and scientific computing, that aims to simplify package management and deployment. Package versions are managed by the package management system conda, which makes it quite simple to install, run, and update complex data science and machine learning software libraries like Scikit-learn, PyTorch, TensorFlow, and SciPy.

To install Anaconda, system requirements are: a minimum of 4 GB RAM (8 GB recommended); 5 GB of disk space and one of the following operating systems: Windows 8 or newer, 64-bit macOS 10.13+, or Linux, including Ubuntu, RedHat, CentOS 6+, and others.

The first step of the Anaconda installation is to download the installer for your platforms (Windows, MacOS, Linux) by visiting the website `https://www.anaconda.com/distribution/` and by clicking the "DOWNLOAD" link under the Python 3.7 version (see Fig. 3.11.

Now, different steps are required according to your platform.

Installing Anaconda on Windows

Steps to install Anaconda on Windows:

1. Run the installer .exe. You will be presented with the Anaconda Setup screen. Click on Next button (see Fig. 3.12).
2. Now you will be presented with licensing terms, read and click “I Agree”. (see Fig. 3.13).
3. Select an install for “Just me” and click Next (see Fig. 3.14).
4. The wizard asks where to install Anaconda on disk, as shown in Fig. 3.15. We suggest to use the default location and click Next.
5. You see the Advanced Installation Options, shown in 3.16. These options are selected by default, and no good reason exists to change them in most cases. Click Install.
6. You see an Installing dialog box with a progress bar. When the installation process is over, you see a Next button enabled. Click Next.

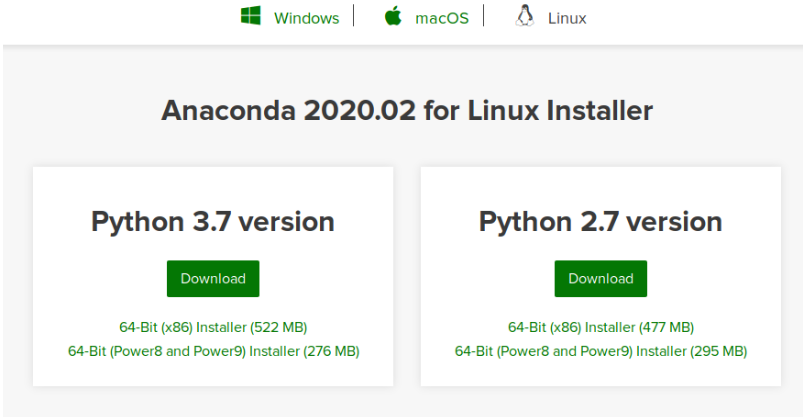


Figure 3.11: Anaconda download website

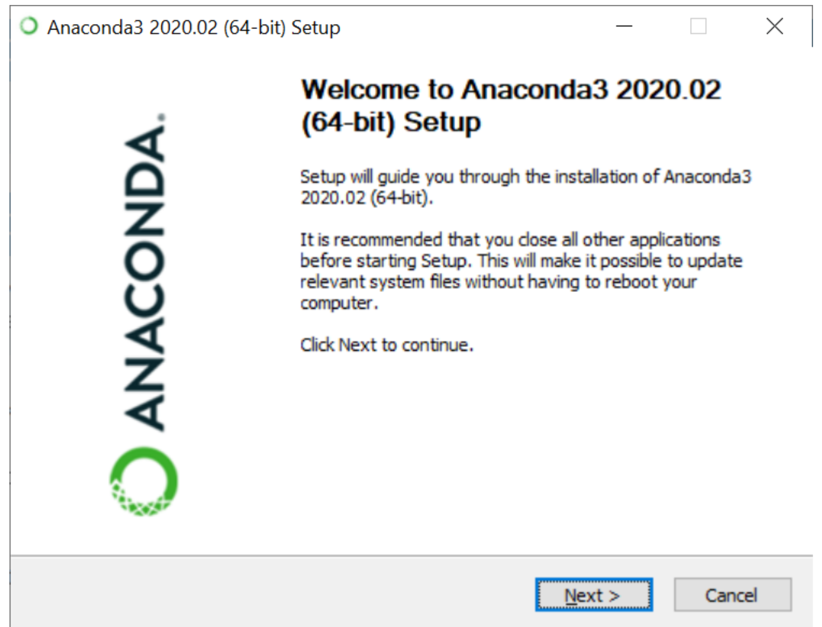


Figure 3.12: Anaconda Installation on Windows: Welcome phase

7. The wizard tells you that the installation is complete. Click Finish (see Fig. 3.17.
8. When the installation process is over, you can reach Anaconda by the Start Menu. Moreover, you can run python by using Anaconda prompt (see Fig. 3.18.

Installing Anaconda on Linux

You have to use the command line to install Anaconda on Linux; you're given no graphical installation option. The following procedure works for Ubuntu 18.04.

1. Open a Terminal window and type the following commands:

```
cd /tmp
curl -O https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-x86_64.sh
```

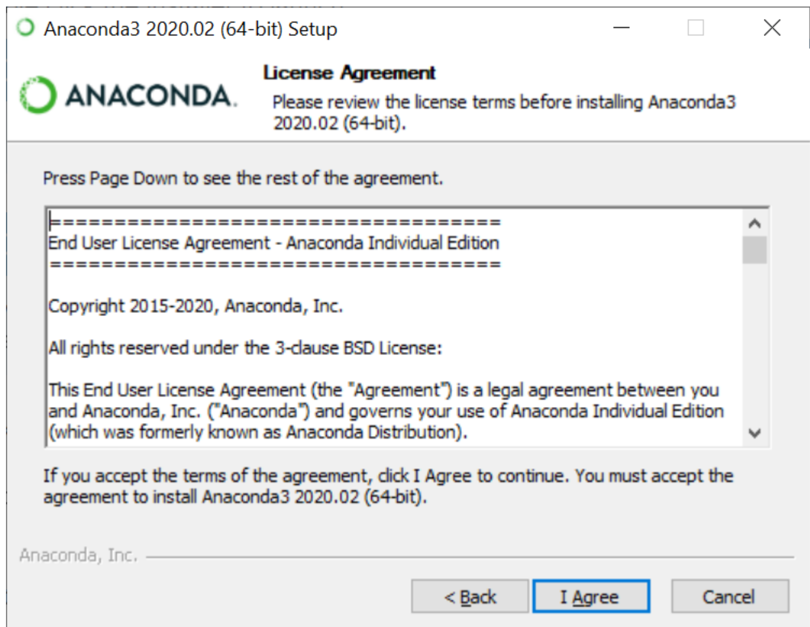


Figure 3.13: Anaconda Installation on Windows: Licensing Terms

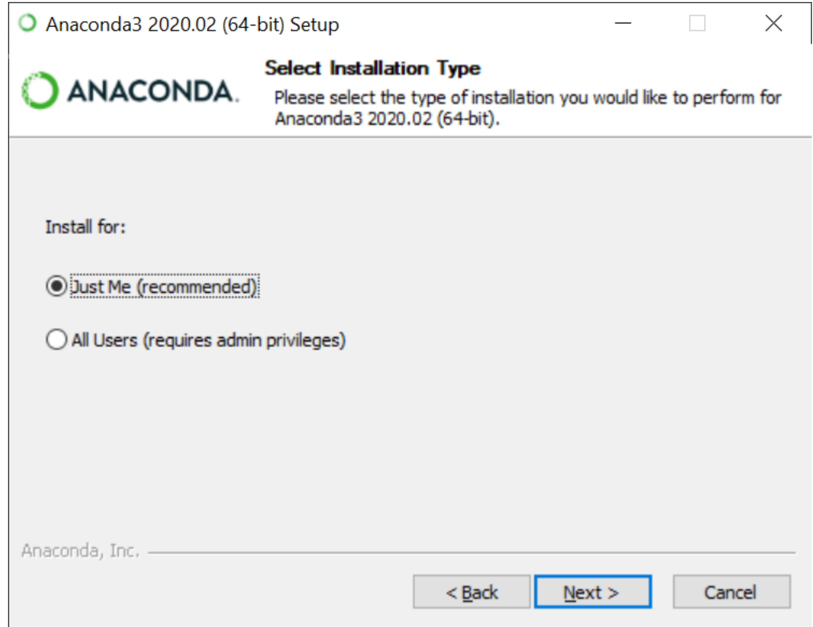


Figure 3.14: Anaconda Installation on Windows: Options

If command curl is not found (See Fig. 3.19, type the following command:

```
sudo apt install curl
```

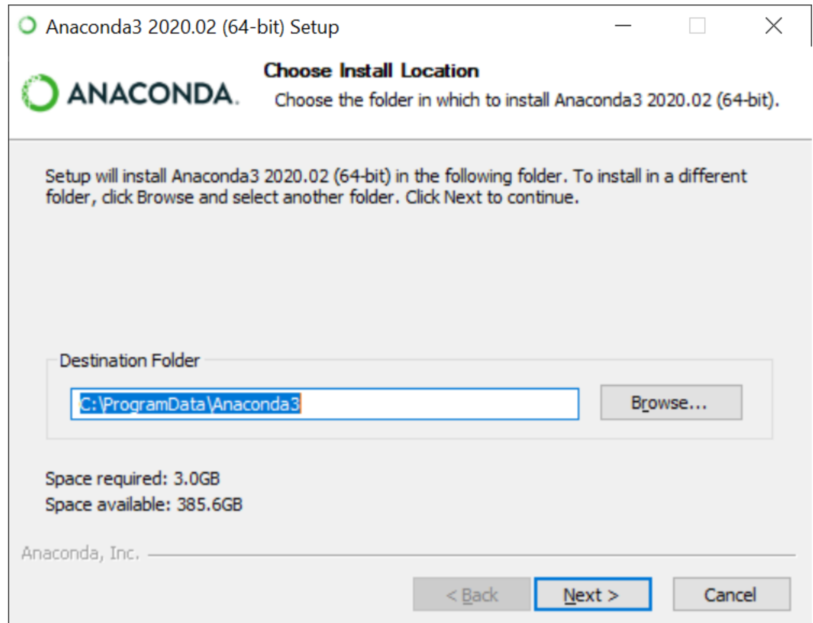


Figure 3.15: Anaconda Installation on Windows: Location

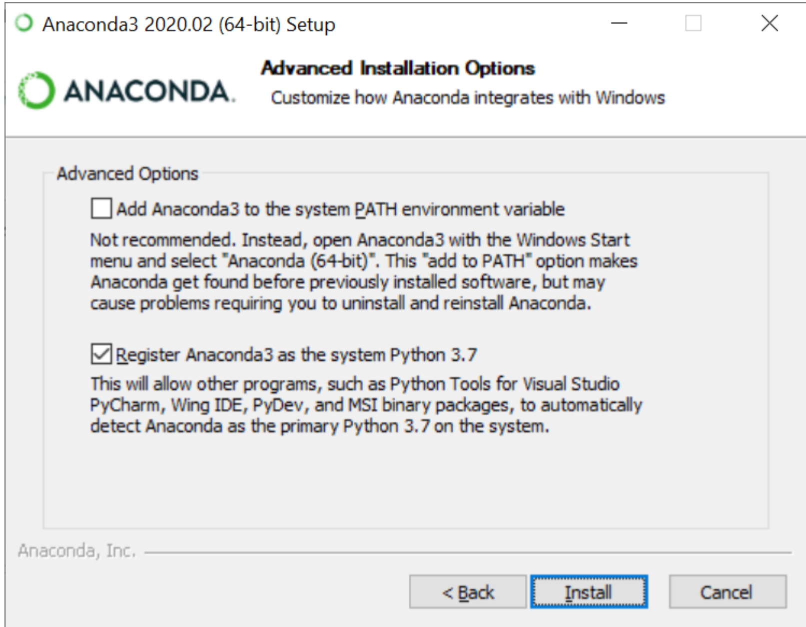


Figure 3.16: Anaconda Installation on Windows: Advanced options

2. Once `curl` command is installed (see Fig. 3.20), type the following commands:

```
curl -O https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-x86_64.sh
```

3. Once downloaded the anaconda package (see Fig. 3.21), type the following command:

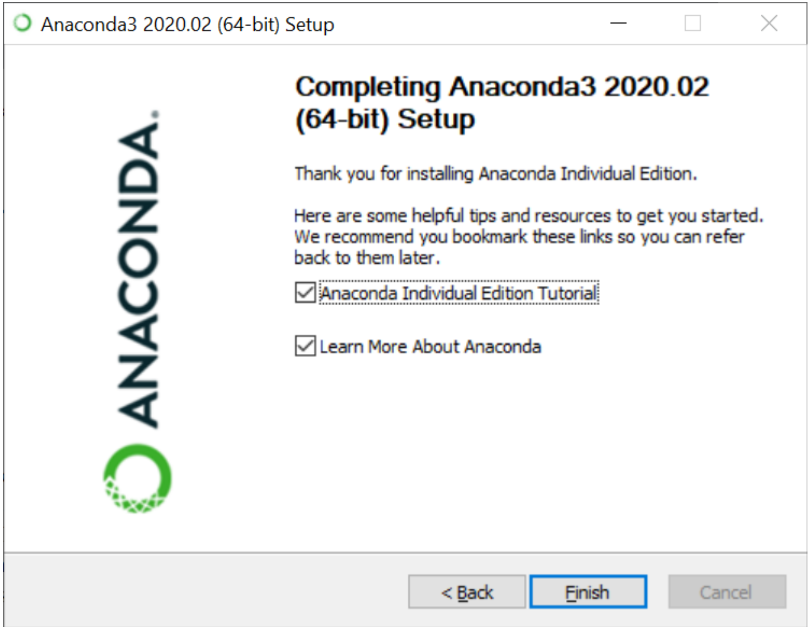


Figure 3.17: Anaconda Installation on Windows: Final phase

Figure 3.18: Anaconda prompt

```
bash Anaconda3-2020.02-Linux-x86_64.sh
```

4. You'll receive the output in Fig. 3.22 to review the license agreement by pressing ENTER until you reach the end.
5. When you get to the end of the license, type yes as long as you agree to the license to complete installation (see 3.23).
6. When you get to the end of the installation, you can see the message in Fig. 3.24.
7. When the installation process is over, ypu can activate the installation with the following command:

   ```
   source ~/.bashrc
   ```

   Once executed, you can run python by opening the Terminal window and type the `python` command (see Fig. 3.25.

Figure 3.19: Anaconda Installation on Linux: Curl command not found

Figure 3.20: Anaconda Installation on Linux: Curl command installation

Figure 3.21: Anaconda Installation on Linux: Download completed

3.3 Other useful tools

We will need a Text Editor. Any Text Editor can be good. So, no installation should be required, because any platform comes with a Text Editor installed. Another important tool that we will need is

```
osboxes@osboxes:/tmp$ curl -O https://repo.anaconda.com/archive/Anaconda3-2020.0
2-Linux-x86_64.sh
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  521M  100  521M    0     0  3375k      0  0:02:38  0:02:38 --:--:-- 3527k
osboxes@osboxes:/tmp$ bash Anaconda3-2020.02-Linux-x86_64.sh

Welcome to Anaconda3 2020.02

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>
```

Figure 3.22: Anaconda Installation on Linux: License agreement

```
osboxes@osboxes: /tmp
File  Edit  View  Search  Terminal  Help
  wheel              pkgs/main/linux-64::wheel-0.34.2-py37_0
  widgetsnbextension pkgs/main/linux-64::widgetsnbextension-3.5.1-py37_0
  wrapt              pkgs/main/linux-64::wrapt-1.11.2-py37h7b6447c_0
  wurlitzer          pkgs/main/linux-64::wurlitzer-2.0.0-py37_0
  xlrd               pkgs/main/linux-64::xlrd-1.2.0-py37_0
  xlsxwriter         pkgs/main/noarch::xlsxwriter-1.2.7-py_0
  xlwt               pkgs/main/linux-64::xlwt-1.3.0-py37_0
  xmltodict          pkgs/main/noarch::xmltodict-0.12.0-py_0
  xz                 pkgs/main/linux-64::xz-5.2.4-h14c3975_4
  yaml               pkgs/main/linux-64::yaml-0.1.7-had09818_2
  yapf               pkgs/main/noarch::yapf-0.28.0-py_0
  zeromq             pkgs/main/linux-64::zeromq-4.3.1-he6710b0_3
  zict               pkgs/main/noarch::zict-1.0.0-py_0
  zipp               pkgs/main/noarch::zipp-2.2.0-py_0
  zlib               pkgs/main/linux-64::zlib-1.2.11-h7b6447c_3
  zstd               pkgs/main/linux-64::zstd-1.3.7-h0b5b093_0


Preparing transaction: done
Executing transaction: done
installation finished.
Do you wish the installer to initialize Anaconda3
by running conda init? [yes|no]
[no] >>>
```

Figure 3.23: Anaconda Installation on Linux: Final phase

PyCharm. PyCharm installation procedure is reported in the following subsection. The need for a Text Editor and PyCharm will be discussed in the remaining part of the chapter.

3.3.1 PyCharm: Installation and Set Up

PyCharm is cross-platform, with the availability of Windows, MacOS and Linux versions. There are two main editions: the Community Edition is open-source, whereas, the Professional Edition is released under a proprietary license but it add features like scientific tools, web development, Python Web Frameworks, Database and SQL support. In this course, we will use the Community Edition.

To install PyCharm, system requirements are: a minimum of 4 GB RAM (8 GB recommended); 2.5 GB of disk space; a display with a minimum resolution of 1024-by-768 pixels. You must already have installed either Python 2 (2.6.x or 2.7.x series) or Python 3 (3.4.x or higher).

The first step of the PyCharm installation is to download PyCharm related to your platforms (Windos, MacOS, Linux) by visiting the website `https://www.jetbrains.com/pycharm/download/` and by clicking the "DOWNLOAD" link under the Community Section (see Fig. 3.26.

Now, different steps are required according to your platform.

Installing Pycharm on Windows

Steps to install PyCharm on Windows:

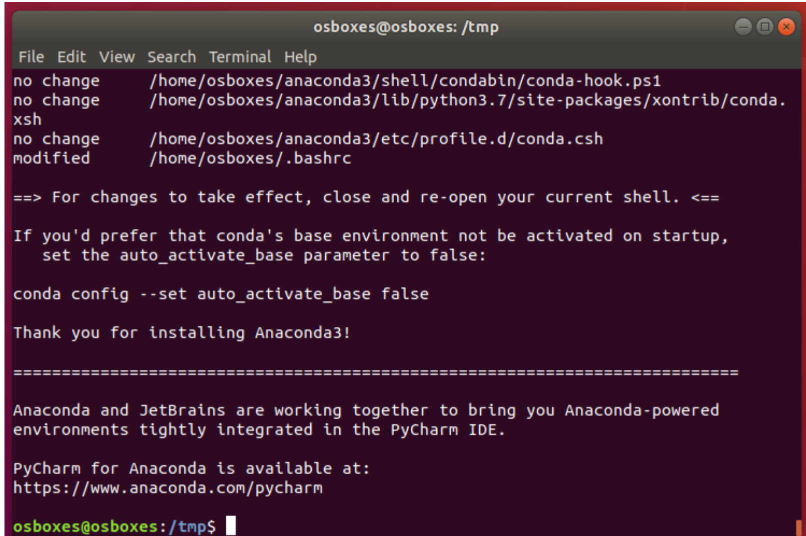


Figure 3.24: Anaconda Installation on Linux: Final message

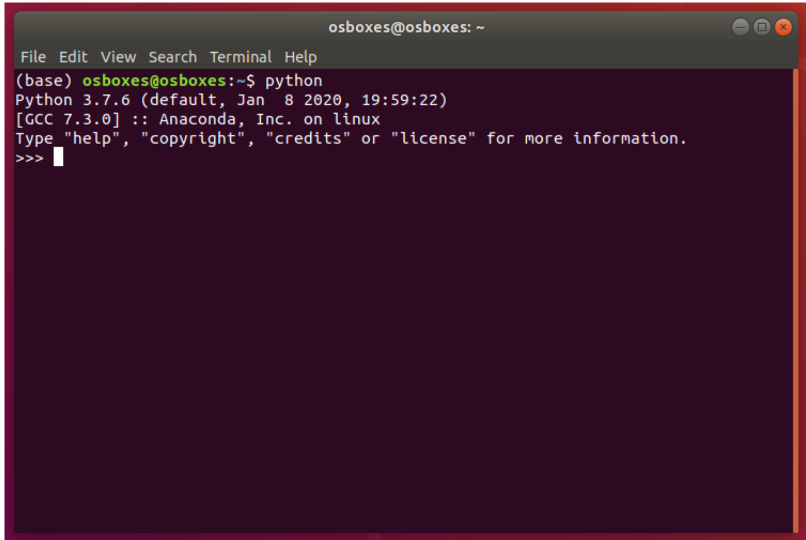


Figure 3.25: Python shell in Linux

1. Run the installer .exe. You will be presented with the PyCharm Community Edition Setup screen. Click on Next button (see Fig. 3.27.
2. Now you will be presented with Choose Install Location screen. Go with the default destination folder to install PyCharm Community Edition (see Fig. 3.28. Click on Next button.
3. On the Installation Options screen, you can create a desktop shortcut if you want (64-bit launcher is suggested) and click on "Next" (see Fig. 3.29).
4. In the Choose Start Menu Folder phase, let the default option to be selected as JetBrains and click on the Install button to proceed (see Fig. 3.30. It will take some time for the installation to finish.

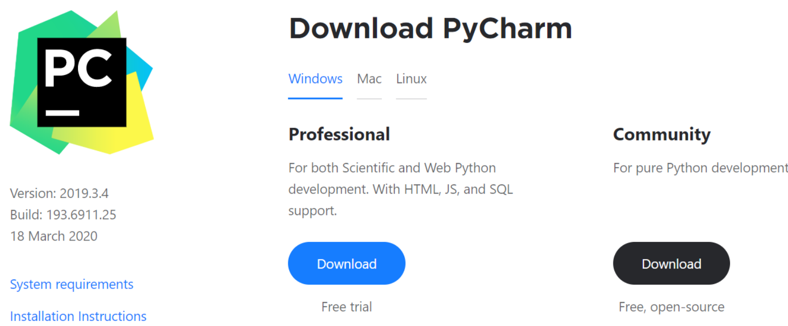


Figure 3.26: PyCharm download website

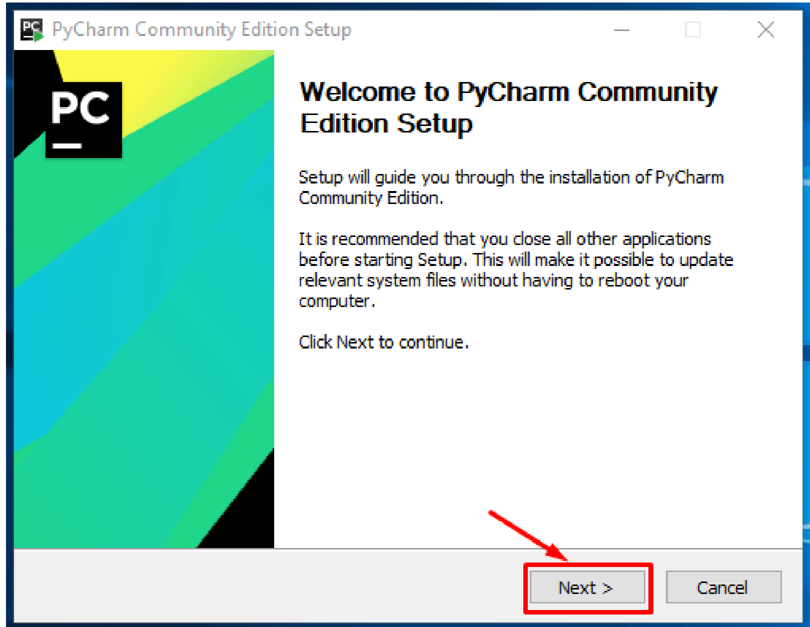


Figure 3.27: PyCharm Installation on Windows: Welcome phase

5. Once installation finished, you should receive a message screen that PyCharm is installed. Make sure to checkmark the “Run PyCharm Community Edition” checkbox to launch it automatically after hitting the Finish button and click “Finish”.
6. After closing the setup wizard, the PyCharm will launch up but first, it will complete installation by asking you to set up some of the important configurations for your convenience as described below.

Installing PyCharm on the Mac

Steps to install PyCharm on the Mac:

1. Click on the file .dmg downloaded by the PyCharm website to begin the installation.
2. After launching the dmg file you will then drag PyCharm into your Application folder (see Fig. 3.31.

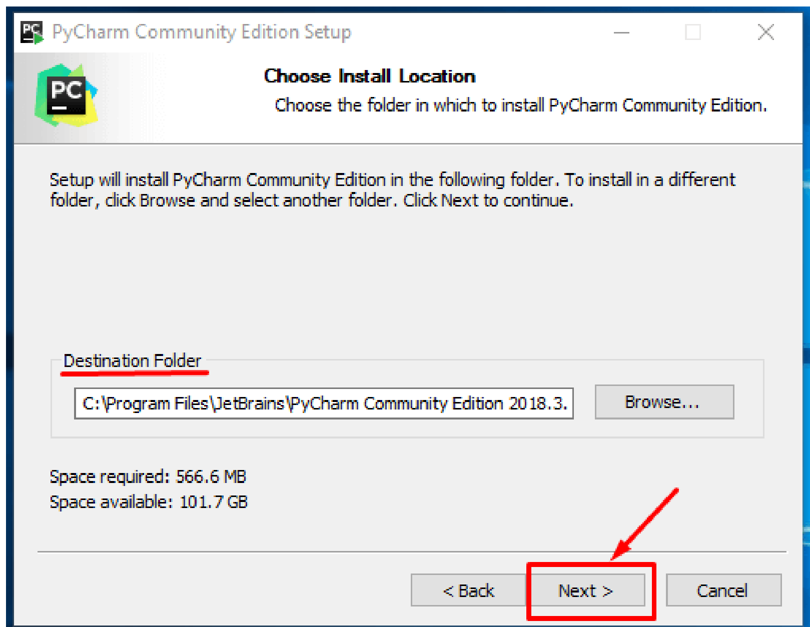


Figure 3.28: PyCharm Installation on Windows: Choose Location

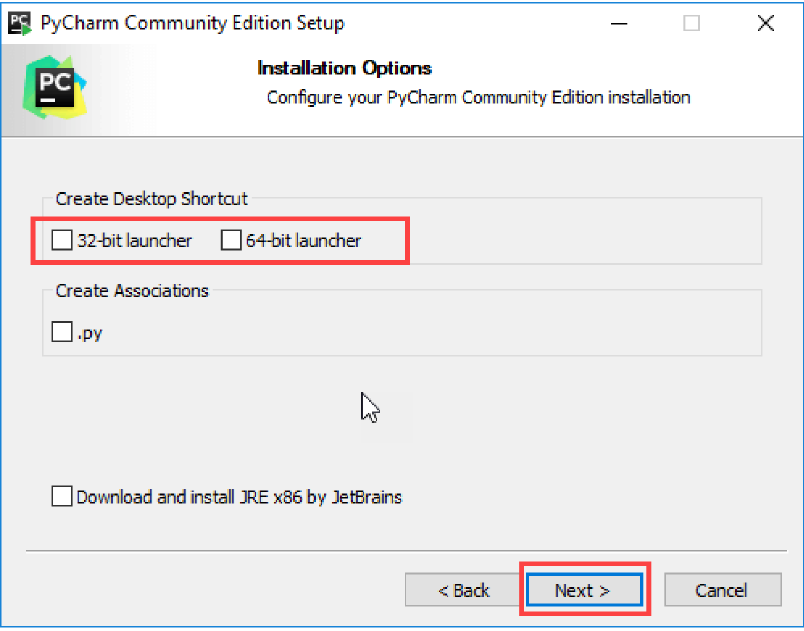


Figure 3.29: PyCharm Installation on Windows: Options

3. In your Applications Folder, double click on PyCharm to open the Application.
4. On your first launch, you must configure PyCharm as described below.

Installing PyCharm on Linux

Steps to install PyCharm on Linux:

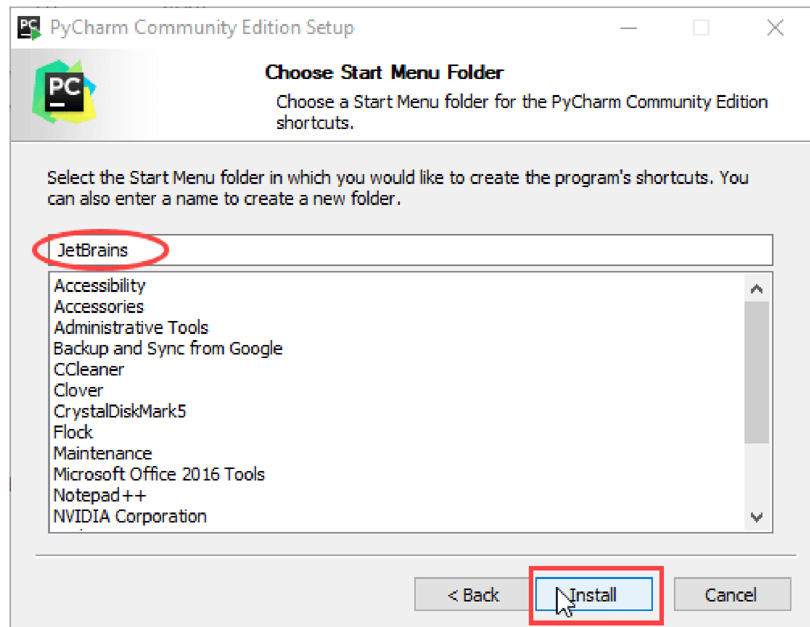


Figure 3.30: PyCharm Installation on Windows: Final phase

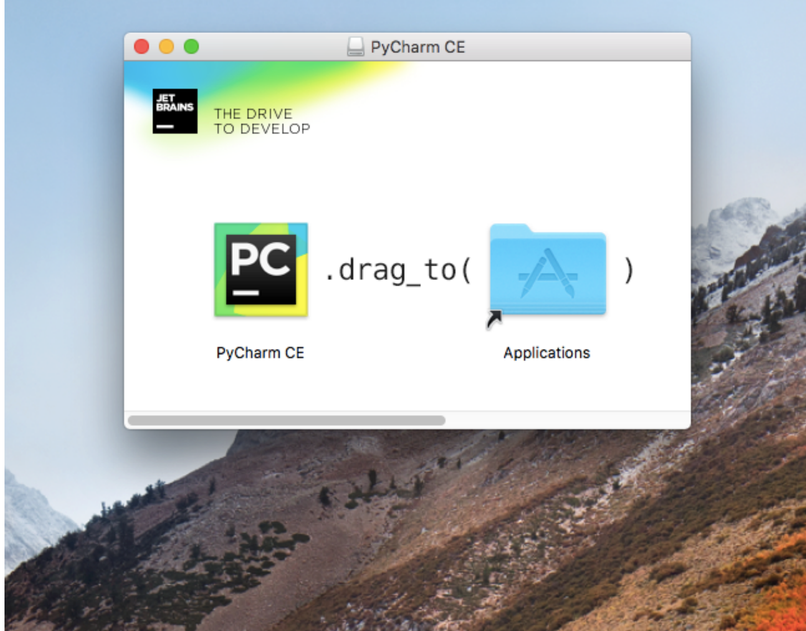


Figure 3.31: PyCharm Installation on the Mac: Mount the image

1. Double click the file tar.gz downloaded by the PyCharm website and let the Archive Manager extract the files.
2. Then go to the folder you have just extracted. Now double-click the Pycharm Community folder once you have extracted the files
3. Inside of the PyCharm folder, double-click the bin folder. Inside the bin folder, locate the file

called pycharm.sh: (see Fig. 3.32.

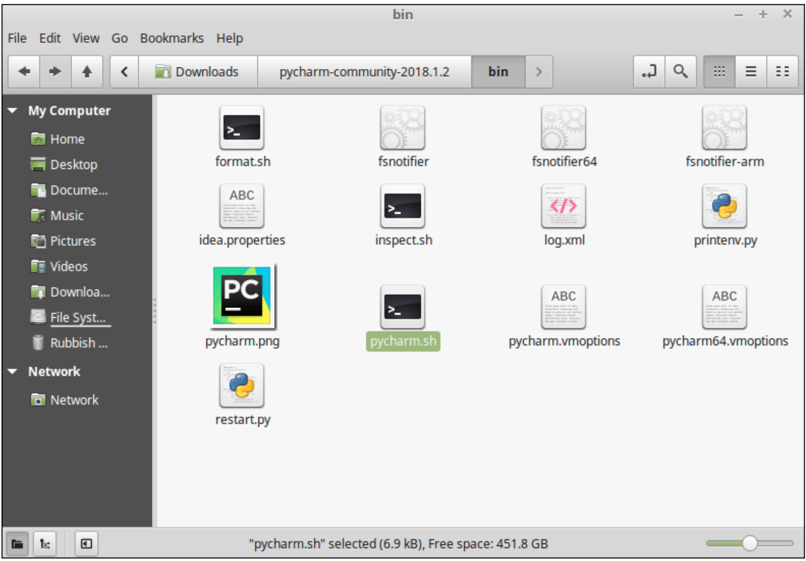


Figure 3.32: PyCharm Installation on Linux

4. Double-click the file pycharm.sh. You may see a dialog box asking you if you want to run the file. Click the Run button.
5. Hopefully, Pycharm will then start up. From now on, you need to double click the pycharm.sn file in the bin folder whenever you want to fire up Pycharm. Or you can create a launcher for it from your desktop.
6. On your first launch, you must configure PyCharm as described below.

In some versions of Linux such us Ubuntu 18.04, you can use `snap` package. In particular, you can follow the steps below:

1. Open a terminal window and type the following command:

```
sudo snap install pycharm-community --classic
```

2. Once pressed the keyboard button *Enter*, the download and installation of PyCharm start (see Fig. 3.33.
3. Once installed, you will see the message in Fig. 3.34).
4. On your first launch, you must configure PyCharm as described below.

Configure PyCharm

When you launch PyCharm for the first time, a configuration procedure is necessary. Steps for this configuration are similar for the three platforms: Windows, MacOs and Linux. The main steps are:

1. In the Import PyCharm settings phase (see Fig. 3.35), select the “Do not import settings” and click OK as shown above.
2. In the Privacy Policy phase (see Fig. 3.36), checkmark the confirmation checkbox of user Agreement as highlighted above. Now click the Continue button to proceed.
3. In the Data Sharing phase (see Fig. 3.36), click on the Don’t send button to proceed as highlighted above.

Figure 3.33: PyCharm Installation on Ubuntu 18.04

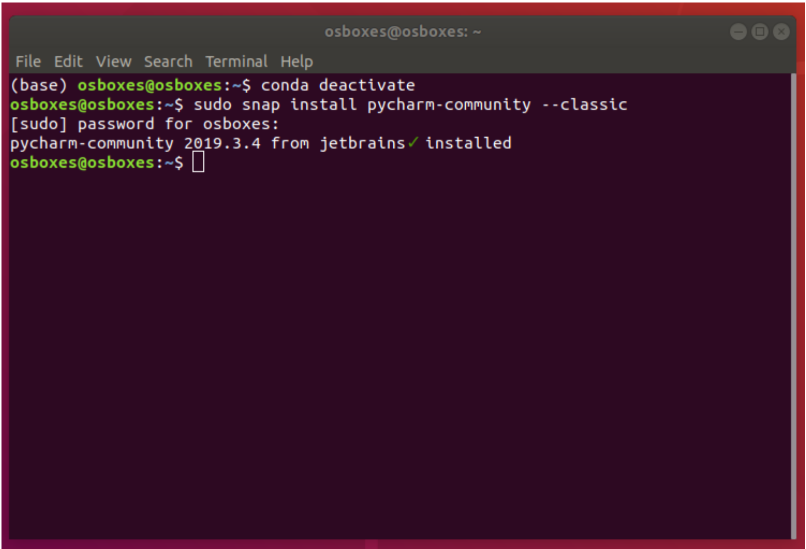


Figure 3.34: PyCharm Installation Completed on Ubuntu 18.04

4. In the Customize PyCharm phase (see Fig. 3.38), it is recommended to set the settings as default, in order to do that click on the Skip Remaining and Set Defaults button. After the Customization phase is complete, you will see the main Window of PyCharm IDE, from where you can start creating your New Project in Python and develop your applications (see Fig. 3.43).

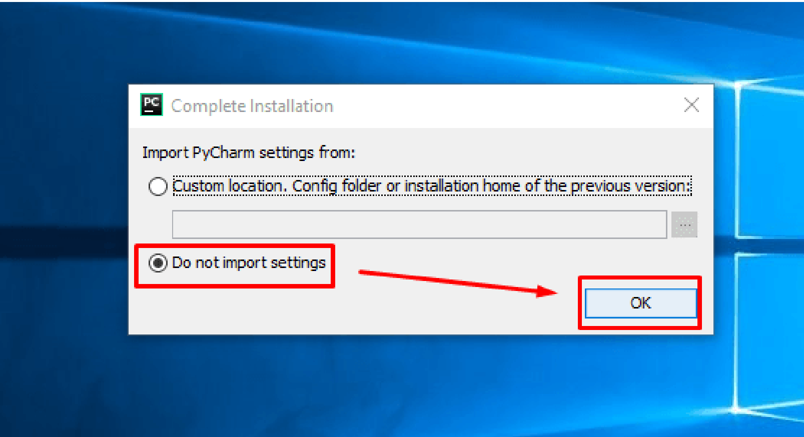


Figure 3.35: PyCharm Configuration: Import Settings

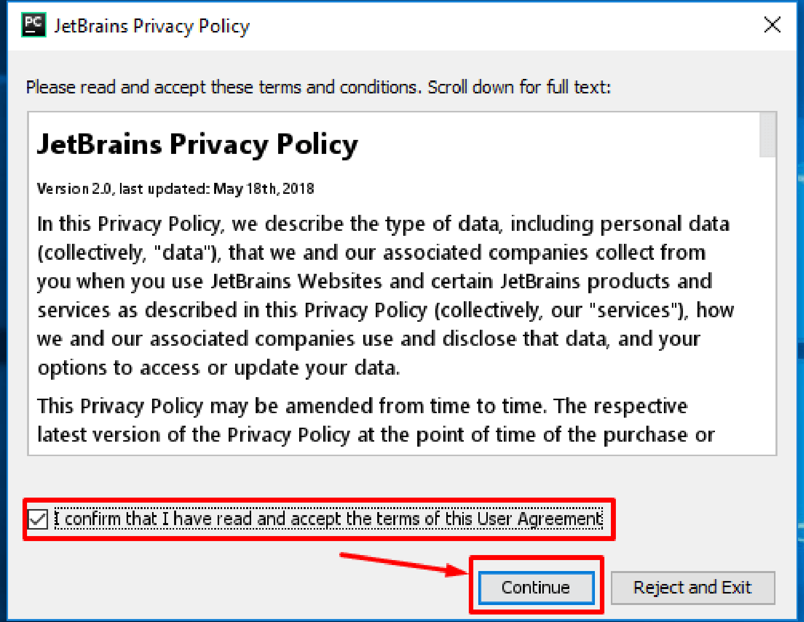


Figure 3.36: PyCharm Configuration: Agreement

3.4 How to include a Unix shell (bash) in the PyCharm IDE under Windows

What textual shell is supported by the PyCharm IDE ?

PyCharm Terminal include a textual shell. It supports different shells depending on the Operating System of your computer. Namely,

- The shell supported by the PyCharm IDE under Mac OS and Linux is the classical **bash shell** (the one we aim at using for our purposes)
- The shell supported by the PyCharm IDE under Windows is the classical Command line interface **cmd**.

the following steps will allow you to substitute the *cmd* shell with the *bash* one.

1 Download and install Cygwin app from www.Cygwin.com (Note: Both version at 32 and

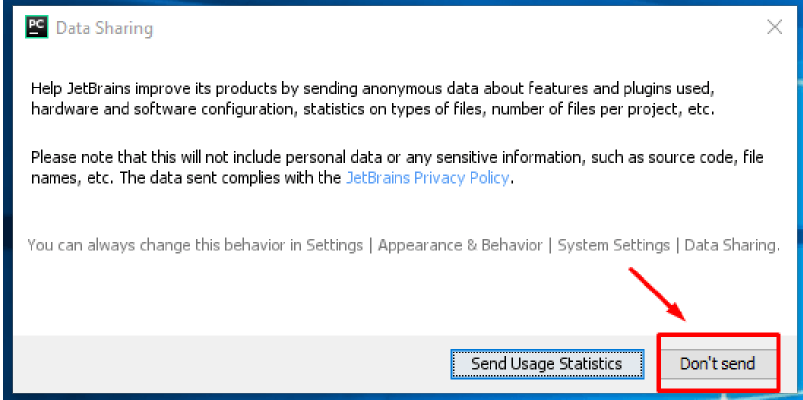


Figure 3.37: PyCharm Configuration: Sharing data

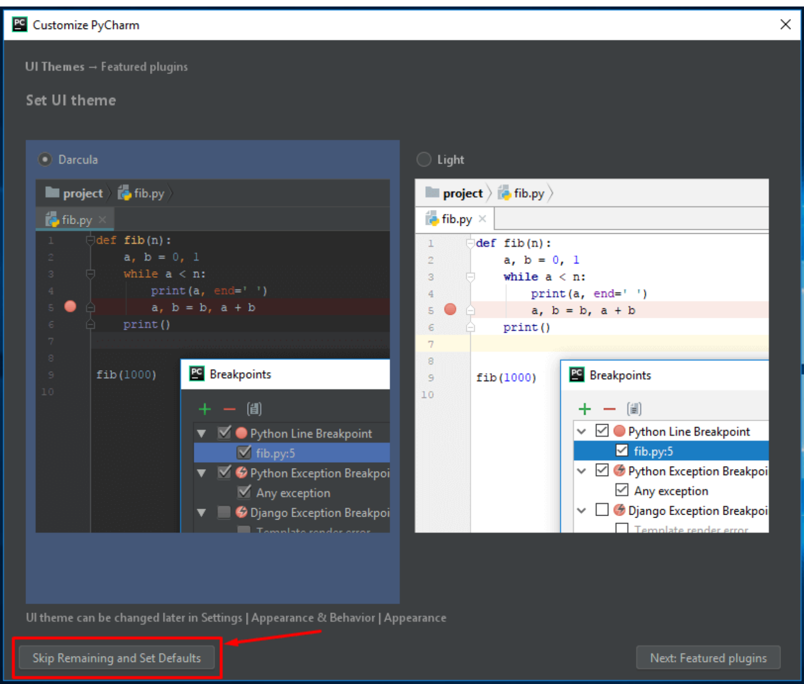


Figure 3.38: PyCharm Configuration: Sharing data

64-bit exist. Make sure to select the right one)

2. Follow the installation setup with default options. Cygwin will be installed in C:
3. Open PyCharm IDE and Go to File/Settings/
4. Select Tools/Terminal
5. Click on the Browse button to select the “Shell Path” and select C:.bat
6. Apply the changes and click on “ok”
7. Now go on the PyCharm IDE Terminal and digit some unix commands to check the shell is changed (es. pwd, ls, etc.) . If the commands are executed the unix shell has been correctly integrated in PyCharm.

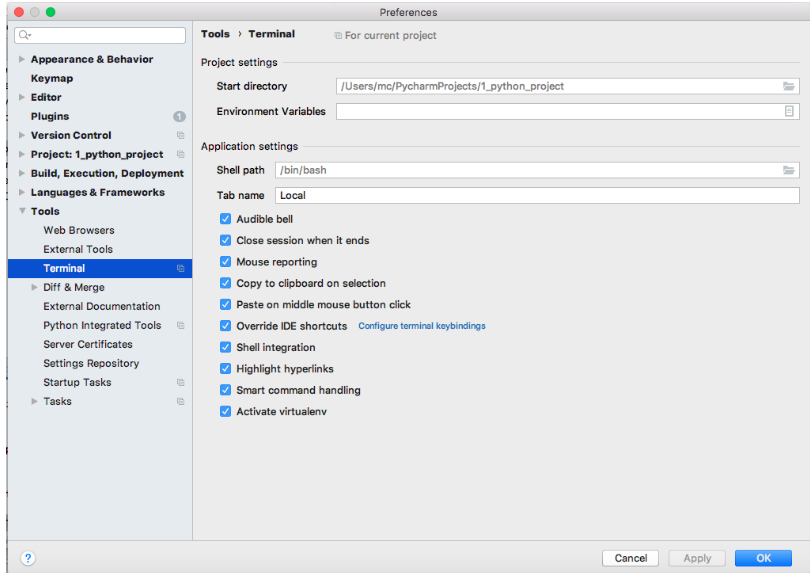


Figure 3.39: PyCharm Terminal shell Configuration

3.5 Interacting with Python

A Python program consists of a list of instructions, resembling a mixture of English words and mathematics and collectively referred to as code. We'll see exactly what form the instructions take in a moment, but first we need to know how and where to enter them into the computer. There are several ways in which you can run a Python program as described below

- Interactive mode: you interactively enter commands at the Python shell;
- Scripting mode: you write commands in a file and running it using the `Python` command;
- IDE mode: you use a Python IDE (Integrated Development Environment) such as PyCharm;
- Notebook mode: you enter commands in your web browser thanks to the Jupyter Notebook library.

Hereafter, more details about the first three strategies are given. As for the last one, due to its complexity is left to be discussed in the future chapters.

3.5.1 Interactive mode

This mode allows you running Python code using the Python shell. Depending on your platform, to enter the Python Shell, you can open a Command Prompt window (in Windows) or a Terminal window (for the Mac and Linux) and type `python` into the prompt. Once executed, in the window, you will be greeted by a message (which will vary depending on your operating system and precise Python version) that tells you what version of Python you are running, along with some other information. An example of this message is:

```
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The three chevrons (>>>) are the prompt and tell you that the computer is ready for you to type

something in. When you see this prompt you can type any command in the Python language at the keyboard and the computer will carry out that command immediately.

■ **Example 3.1** Run a Python code that:

- prints out the string 'Hello World';
- adds 5 and 4 together and gets the result 9;
- stores the string 'Janet' in a variable called *name*;
- prints out the contents of the variable *name*.

■

Fig. 3.40 shows how to run the Python code for Example 3.1 by using the Python shell.

```
Prompt dei comandi - python
(c) 2019 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\autil>python
Python 3.8.2 (tags/v3.8.2:7b3ab59, Feb 25 2020, 22:45:29) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello world')
Hello world
>>> 5+4
9
>>> name = 'Janet'
>>> print(name)
Janet
>>>
```

Figure 3.40: Interactive mode: examples of commands

Using the Python shell can be a useful way to quickly try individual Python commands when you are not sure how something works, but it is not the main way that we will use Python commands. Normally, we want to type in an entire Python program at once, consisting of many commands one after another, then run the whole program together. For this aim, we can use the scripting mode discussed in the next subsection.

3.5.2 Scripting mode

In this mode, we can store the Python commands into a file with extension .py. This creates a program file that can then be run as an argument to the `python` command to enter in the Command Prompt window (in Windows) or in a Terminal window (for the Mac and Linux).

For example, we can write a file, called example.py (see Fig. 3.41, with the Python code for Example 3.1. To run the example.py program on your pc, you can open the Command Prompt window (in Windows) or a Terminal window (for the Mac and Linux) and type the `python` command followed by the name of the file as shown in Fig. 3.42. It is worth noting that the `python` command must be run after positioning in the directory where the file is contained. To position the prompt in the directory containing the file to be run, you can use the `cd` command followed by the relative path of the directory.

The scripting mode is convenient because it allows you to store your Python programs and run them when you want. Moreover, the same Python program stored in a file can be run on whatever platform (Windows, Linux or Mac). This illustrates the cross platform nature of Python and is just one of the reasons why Python is so popular.

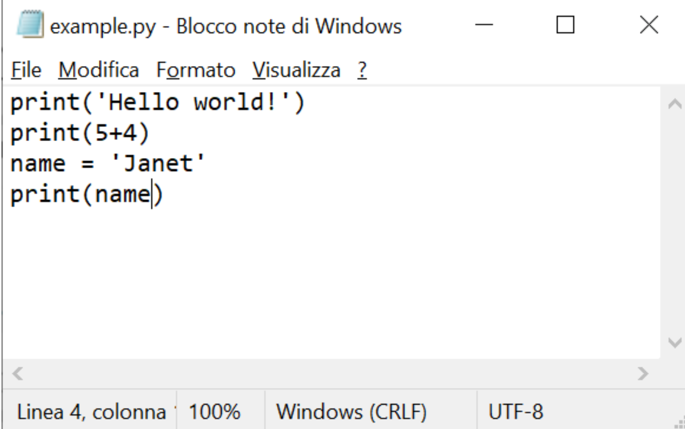


Figure 3.41: Scripting mode: writing a file with the examples of commands.

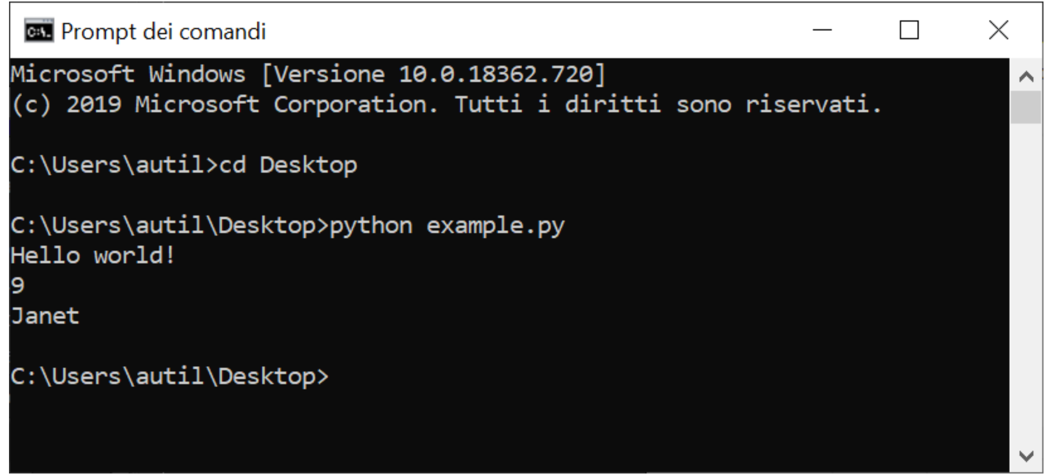


Figure 3.42: Scripting mode: running example.py file in the Command Prompt window

3.5.3 IDE mode: using PyCharm

A good question to ask is, why do you need an IDE to work with Python if the command-line tool (interactive and scripting modes) works fine? The advantages to use an IDE are several, but, we cite two reasons in particular:

- creating better code: a good IDE contains a certain amount of intelligence. This intelligence help you, for example, by suggesting alternatives when you type the incorrect keyword, or by identifying that a certain line of code simply won't work as written.
- debugging functionality: finding bugs (errors) in your code is a process called debugging. Writing perfect code on the first pass is nearly impossible. Hence, even the most expert developer in the world spends time debugging. Consequently, the debugging capabilities of your IDE are critical. Unfortunately, the debugging capabilities of the previous modes are almost nonexistent.

In this course, we will use one of the most known Python IDEs such as PyCharm. This tool is developed by the Czech company JetBrains. It provides code analysis, a graphical debugger, an

integrated unit tester, integration with version control systems and supports web development with Django web framework (for installation see Section 3.3.1).

To use Pycharm for the Example 3.1, it is necessary to perform the following steps:

- the first step is to create a project. If you're on the Welcome screen (see Fig. 3.43), click "Create New Project". If you have already got a project open, choose "New Project" from Menu "File".

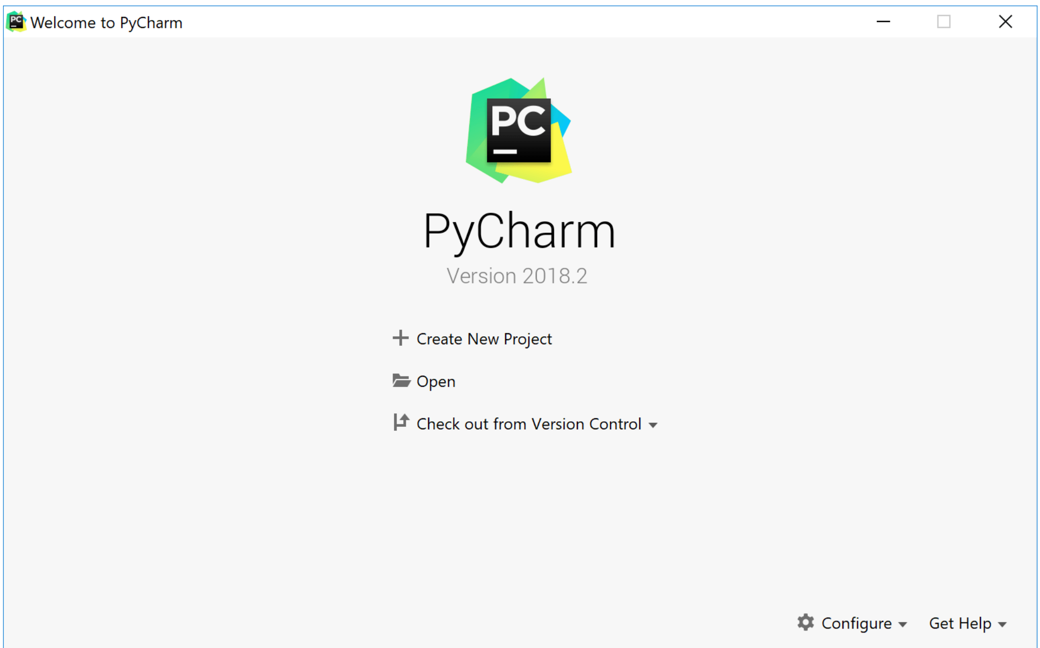


Figure 3.43: PyCharm window to create a project

- The New Project dialog window opens (see Fig. 3.44). Pycharm permits users to create different kinds of projects. In this case, we will choose "Pure Python" option. This template will create an empty python project for us. You can change the default name of the project in the field "Location" replacing the word "untitled". We give the new project name "Example". When you create a python project, it is necessary to associate with it a virtual environment. We discuss this in detail in the next chapters. For this example, we use the default option. Then click the Create button at the bottom of the New Project dialog. If you have already got a project open, after clicking Create PyCharm will ask you whether to open a new project in the current window or in a new one. Choose Open in current window - this will close the current project, but you will be able to reopen it later.
- Select the project root in the Project tool window, then select "New ..." from the menu "File" (see Fig. 3.45.
- Choose the option Python file from the popup (see Fig. 3.46), and then type the new filename "example.py". Then, press the keyboard button *Enter*. PyCharm creates the new Python file and opens it for editing.
- Edit the file "example.py" with the code for the Example 3.1 (see Fig. 3.47.
- Once edited the file example.py completely, you can right-click the name tab of the file "example.py", and from the context menu choose to run the script (see Fig. 3.48.

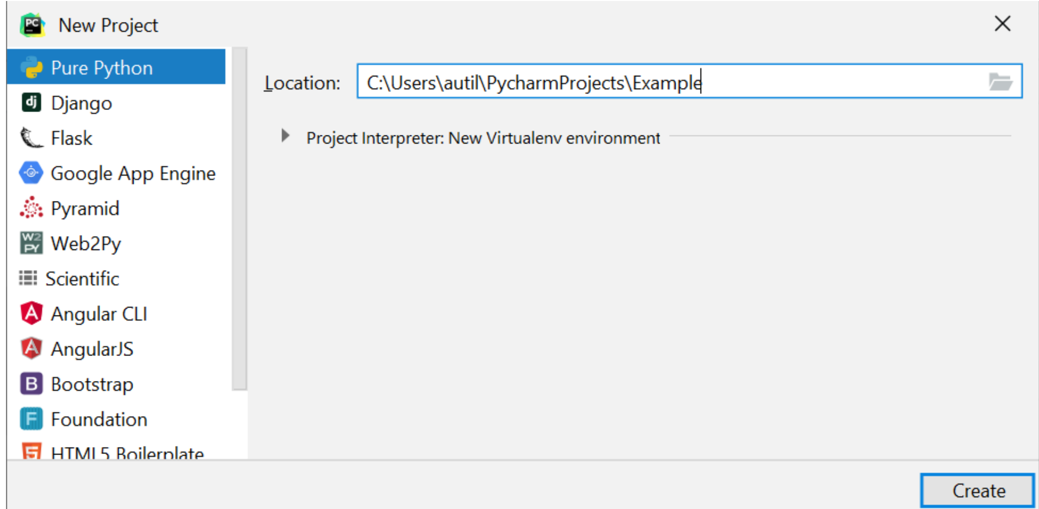


Figure 3.44: PyCharm window to set options for a Python project

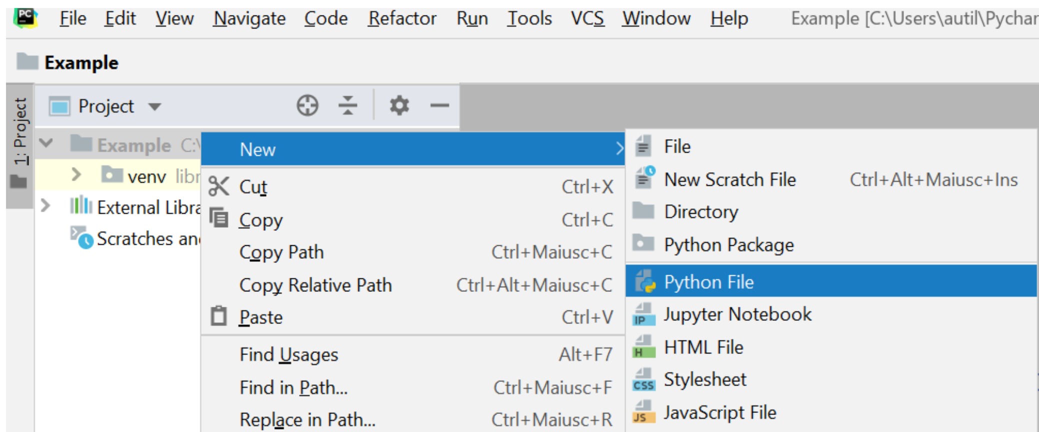


Figure 3.45: Create a Python file in PyCharm

- A console appears in the Run tool window at the bottom of the PyCharm window showing the output of the written code (see Fig. 3.49).

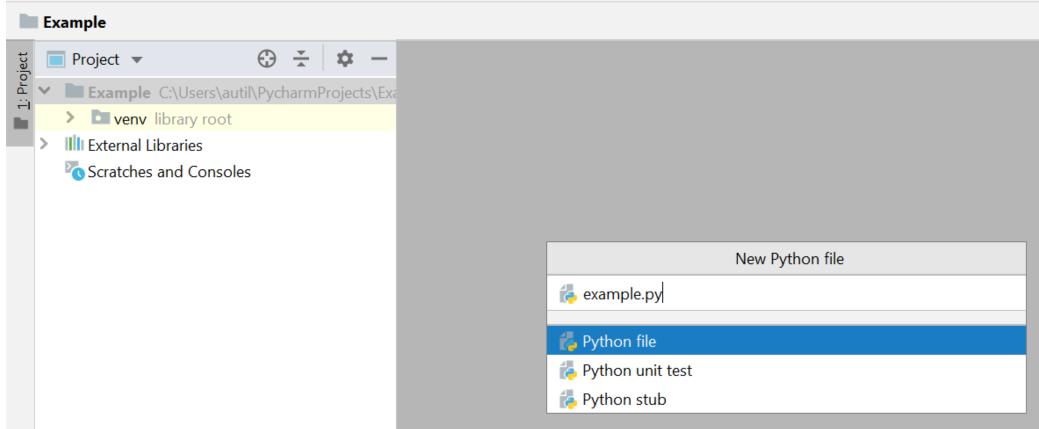


Figure 3.46: Give a name to a Python file in PyCharm

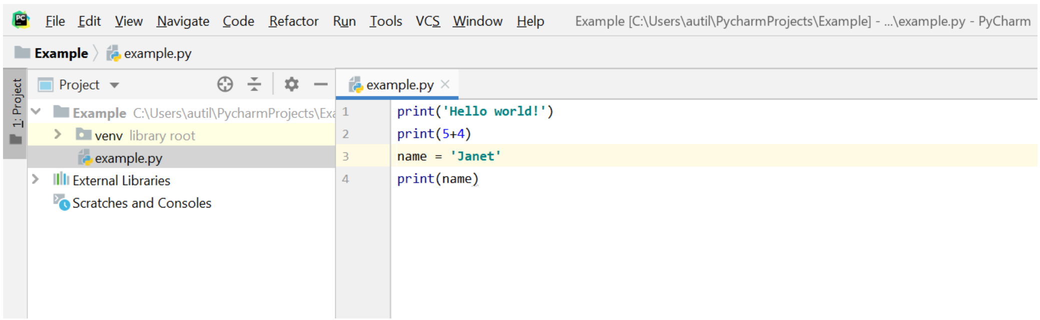


Figure 3.47: Editing file example.py in PyCharm

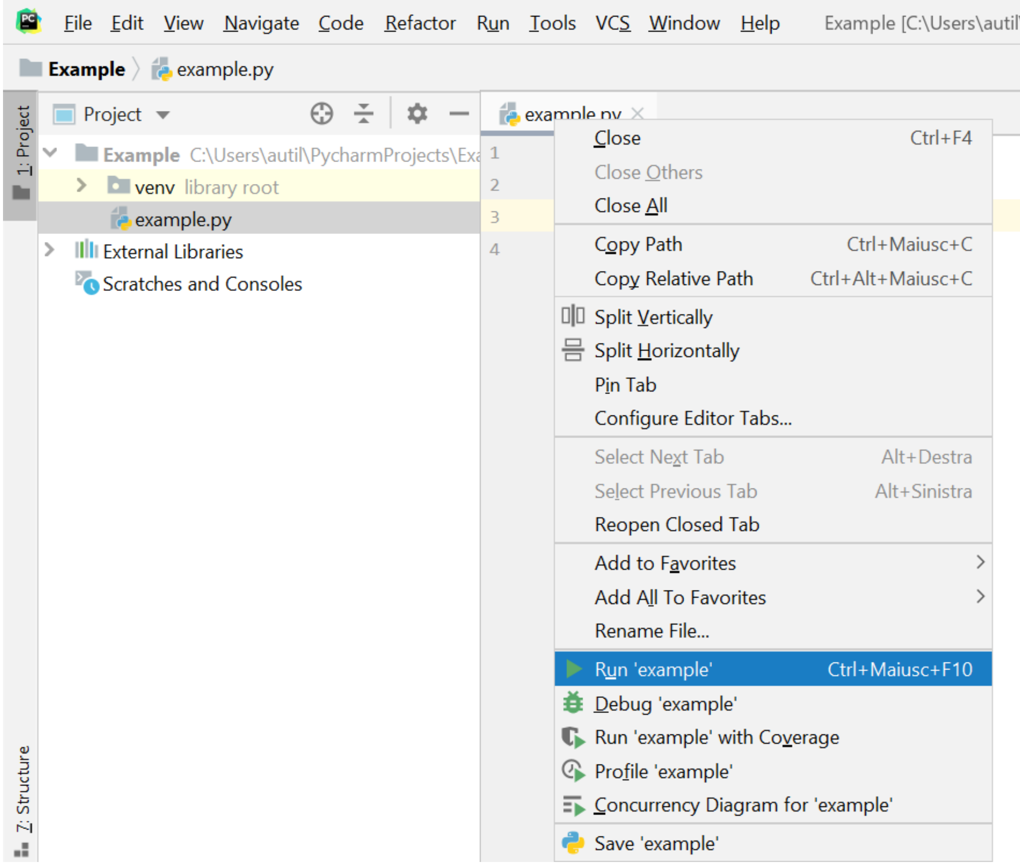


Figure 3.48: Run file example.py in PyCharm

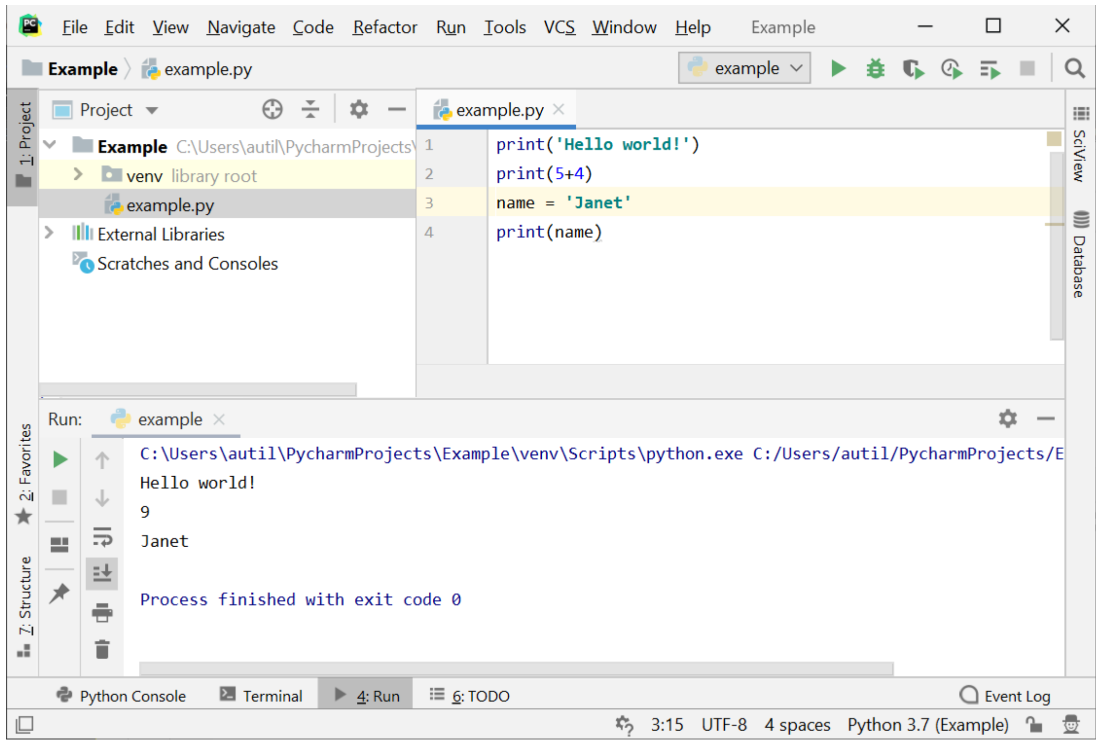


Figure 3.49: IDE mode: running example.py in PyCharm

4. The basic elements of Python

This chapter presents the fundamentals of Python Programming language which you need to know before writing simple Python programs such as operators, variables and types. Contents of this chapter is extracted in part by the following sources [GV18][Ste15].

4.1 Statements, Variables and Expressions

A statement is an instruction that the Python interpreter can execute. Python program consists of a sequence of statements. Statements are everything that can make up a line (or several lines) of Python code. Our first example of a statement in Python is this:

```
x=1
```

This is an *assignment statement*. It tells the computer that there is a variable called x and we are assigning it the value 1. You can think of the variable as a box that stores a value for you, so that you can come back and retrieve that value at any later time, or change it to a different value. In other words, variables represent quantities of interest in a program - which in physics usually means numbers or sets of numbers to represent quantities like positions, velocities, forces, fields, voltages and probabilities.

The name of a variable is a so-called *identifier* that may be one or more characters in the following format:

- identifiers can be a combination of letters (in lowercase or uppercase), digits (0 to 9) and underscore (_).
- identifiers are case-sensitive. For example, the words computer and Computer are different identifiers.
- An identifier cannot start with a digit but is allowed everywhere else.
- Keywords (i.e., reserved words that have predefined meaning in Python as we will see later) cannot be used as identifiers;
- One cannot use spaces and special symbols like !, @, #, $, % etc. as identifiers.
- Identifier can be of any length.

Many of the programs you will write will contain large numbers of variables representing the values of different things and keeping them straight in your head can be a challenge. It is a very good idea - one that is guaranteed to save you time and effort in the long run - to give your variables meaningful names that describe what they represent. If you have a variable that represents the energy of a system, for instance, you might call it `energy`. For more complex concepts, the Python naming convention is to use the underscore symbol "_" to create variable names with more than one word, like `maximum_energy`. Of course, there will be times when single-letter variable names are appropriate (such as the use of variables `i` and `j` in looping constructs as we will see later). For instance, there's no reason why you cannot call your velocity variable simply `v` if that seems natural to you.

Going back to talking about an assignment statement, the general format for assigning values to variables is as follows:

$$variable_name = expression$$

The equal sign (=) is known as *assignment operator* and its aim is to assign values to variables.

> **Note**: Assignment operator should not be confused with the = used in algebra to denote equality. Hence, assignment statements should not be confused with mathematical equations.

In the general format, the operand to the left of the assignment operator is the name of the variable and the operand to the right is a so-called *expression*. An expression is an arrangement of values and operators which are evaluated to make a new value. This value is stored in the variable on the execution of the assignment statement.

> **Note**: An expression, when used in interactive mode is evaluated by the interpreter and result is displayed instantly. But the same expression when used in the other modes (e.g. written in a scripting file) does not show any output altogether. You need to explicitly print the result.

4.2 Variable types

Variable types specify the type of data to be stored and manipulated within a program. Basic data types of Python are:

- **Numbers**: Integers, floating point numbers and complex numbers fall under Python numbers category. More details are given in the remaining part of the section.
- **Booleans**: There are just two values of type boolean: `True` and `False` (they are treated as reserved words). Booleans may not seem very useful at first, but they are essential when you start using conditional statements. In fact, since a condition is really just a yes-or-no question, the answer to that question is a Boolean value, either True or False. More details are given in the next chapters.
- **Strings**: A string consists of a sequence of one or more characters, which can include letters, numbers, and other types of characters. A string can also contain spaces. You can use single quotes or double quotes to represent strings and it is also called a string literal. Multiline strings can be denoted using triple quotes, ''' or """. For example,

```
s1 = 'This is single quote string'
s2 = "This is double quote string"
s3 = '''This is
              Multiline
```

```
                         string'''
s4= "a"
```

More details are given in the next chapters.
- **None**: it is a special data type in Python. It is used to represent the absence of a value.

4.2.1 Numbers

The main numerical types in Python are:

- **Integer**: Integer variables can take integer values such as 1, 0, or -286784. Both positive and negative values are allowed.
- **Float**: A floating-point variable, or "float" for short, can take real, or floating-point, values such as 3.14159, 6.63×10^{34}, or 1.0. Notice that a floating-point variable can take an integer value like 1.0 (which after all is also a real number). A floating point number is accurate up to 15 decimal places;
- **Complex**: A complex variable can take a complex value, such as $1+2j$ or $3.50.4j$. Notice that in Python the unit imaginary number is called j, not i.

You might be asking yourself Why do we need different types? Could not all values, including integers and real numbers, be represented with complex variables, so that we only need one type of variable? In principle they could, but there are great advantages to having the different types:

- **avoiding waste of memory**. Indeed, the values of the variables in a program are stored by the computer in its memory, and it takes twice as much memory to store a complex number as it does a float, because the computer has to store both the real and imaginary parts. Even if the imaginary part is zero (so that the number is actually real), the computer still takes up memory space storing that zero. This may not seem like a big issue given the huge amounts of memory computers have these days, but in many physics programs we need to store enormous numbers of variables—millions or billions of them—in which case memory space can become a limiting factor in writing the program.
- **reducing execution time**. Indeed, calculations with complex numbers take longer to complete, because the computer has to calculate both the real and imaginary parts. Again, even if the imaginary part is zero, the computer still has to do the calculation, so it takes longer either way. Many of our physics programs will involve millions or billions of operations. Big physics calculations can take days or weeks to run, so the speed of individual mathematical operations can have a big effect.

By summarising, if you need to work with complex numbers then you will have to use complex variables, but if our numbers are real, then it is better to use a floating-point variable. Similar considerations apply to floating-point variables and integers. If the numbers we are working with are genuinely non-integer real numbers, then we should use floating-point variables to represent them. But if we know that the numbers are integers then using integer variables is usually faster and takes up less memory space.

Note: There is also another reason to use integer type if you need to represent an integer quantity: integer variables are in some cases actually more accurate than floating-point variables. In detail, floating-point calculations on computers are not infinitely accurate (typically 16 decimal places in the modern computers). This means that the value 1 assigned to a floating-point variable may actually be stored on the computer as 0.9999999999999999. In many cases the difference will not matter much, but what happens, for instance, if something special is supposed to take place in your program if, and only if, the number is less than 1? In that case, the difference between 1 and 0.9999999999999999 could be crucially important

4.3 Dynamic and strongly typed language

Once discussed the variable types, it is obvious to ask: how do you tell the computer what type you want a variable to be? The name of the variable is no help. Indeed, a variable called `x` could be an integer or it could be a complex variable. The answer is that Python is a *dynamically-typed* language: python interpret associates the type to a variable at run-time by considering the value that we give it. Thus, for instance, if we say

```
x = 1
```

then x will be an integer variable, because we have given it an integer value. If we say

```
x = 1.5
```

on the other hand then it will be a float. If we say

```
x = 1+2j
```

it will be complex.

The type of a variable can change as a Python program runs. For example, suppose we have the following two lines one after the other in our program:

```
x = 3
x = 4.5
```

If we run this program then after the first line is executed by the computer `x` will be an integer variable with value 3. But immediately after that the computer will execute the second line and x will become a float with value 4.5. In other words, the type of the variable `x` has changed from integer to float.

Note: If you have previously programmed in one of the so-called *static-typed* languages, such as C and Java, then you'll be used to creating variables with a declaration such as `int i` which means that we are declaring an integer variable called i. In such languages the types of variables are fixed once they are declared and cannot change. There is no equivalent declaration in Python. Variables in Python are created when you first use them, with types which are deduced from the values they are given and which may change when they are given new values.

However, although you can change the types of variables in this way, it does not mean you should. It is considered poor programming to use the same variable as two different types in a single program, because it makes the program significantly more difficult to follow and increases the chance that you may make a mistake in your programming. If x is an integer in some parts of the

program and a float in others then it becomes difficult to remember which it is and confusion can ensue. A good programmer, therefore, will use a given variable to store only one type of quantity in a given program. If you need a variable to store another type, use a different variable with a different name. Thus, in a well written program, the type of a variable will be set the first time it is given a value and will remain the same for the rest of the program.

Moreover, Python is also a strongly typed language as the interpreter keeps track of all the variables types. In a strongly typed language, you are simply not allowed to do anything that's incompatible with the type of data you are working with. For example,

```
>>> 5 + 10
15
>>> 5 + "10"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In the first statement, we add 5 and 10 that are integer values, and we obtain the output 15 because types are are compatible. However, in the second statement, when we try to add 5, which is an integer type, with "10" which is string type, then it results in Traceback as they are not compatible. In Python, Traceback is printed when an error occurs. The last line tells us the kind of error that occurred which in our case is the unsupported operand type(s).

4.4 Arithmetic and Assignment Operators

Operators are symbols, such as +, –, =, >, and <, that perform certain operation to manipulate data values and produce a result based on some rules. An operator manipulates the data values called *operands*. In this section, we start with discussing the arithmetic and assignment operators.

4.4.1 Arithmetic operators

Arithmetic operators are used to execute arithmetic operations such as addition, subtraction, division, multiplication etc. The Table 4.1 shows all the arithmetic operators.

An important rule about arithmetic in Python is that the type of result a calculation gives depends on the types of the variables that go into it.

If the operands of an operation are variables of the same type - integer, float, complex - then when they are added together the result will also have the same type and this will be the type of variable x. For example,

```
>>> 9.2+4.8
14.0
>>> 6-3
3
>>> 2.0**2.0
4.0
>>> 45//10
4
```

If the operands of an operation are of different types, then the end result has the more general of the two types that went into it. This means that if you add a float and an integer, for example, the end

Operator	Operator	Name Description
+	Addition operator	Adds two operands, producing their sum.
−	Subtraction operator	Subtracts the two operands, producing their difference.
*	Multiplication operator	Produces the product of the operands.
/	Division operator	Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor.
%	Modulus operator	Divides left hand operand by right hand operand and returns a remainder.
**	Exponent operator	Performs exponential (power) calculation on operators.
//	Floor division operator	Returns the the integer part of the first operand divided by the second operand.

Table 4.1: List of Arithmetic Operators

result will be a float. If you add a float and a complex variable, the end result will be complex. For example,

```
>>> 2**2.0
4.0
>>> 45//10.0
4.0
```

The division operation, however, is slightly different: it follows basically the same rules except that it never gives an integer result. Only floating-point or complex values result from division. For example,

```
>>> 5.0/2
2.5
>>> 5/2
2.5
```

You can combine several mathematical operations together to make a more complicated expression, like x+2*y-z/3. When you do this the operations obey rules similar to those of normal algebra. Multiplications and divisions are performed before additions and subtractions. If there are several multiplications or divisions in a row they are carried out in order from left to right. Powers are calculated before anything else. You can also use parentheses () in your algebraic expressions, just as you would in normal algebra, to mark things that should be evaluated as a unit. For example,

```
>>> 2+3*2
8
>>> 3*4/2
6.0
>>> 3+2**3
11
>>> (3+2)**3
125
```

4.4.2 Assignment operators

Assignment operators are used for assigning the values generated after evaluating the right operand to the left operand. Assignment operation always works from right to left. Assignment operators are either simple assignment operator or compound assignment operators. Simple assignment is done with the equal sign (=) and simply assigns the value of its right operand to the variable on the left. For example,

```
>>> x = 5
>>> x = x + 1
>>> x
6
```

In the second assignment statement `x = x + 1`, an integer value of 1 is added to the variable x on the right side and the value 6 after the evaluation is assigned to the variable x. *Compound assignment operators* support shorthand notation for avoiding the repetition of the left-side variable on the right side. Compound assignment operators combine assignment operator with another operator with = being placed at the end of the original operator. The Table 4.2 shows all the assignment operators. For example,

```
>>> p = 5
>>> p *= 2
>>> p
10
>>> p **= 2
>>> p
25
```

Finally in this section, a nice feature of Python, not available in most other computer languages, is the ability to assign the values of two variables with a single statement. For instance, we can write

```
x,y = 1,2.5
```

which is equivalent to the two statements:

```
x = 1
y = 2.5
```

One purpose for which this type of multiple assignment is commonly used is to interchange the values of two variables. If we want to swap the values of x and y we can write:

```
x,y = y,x
```

and the two will be exchanged. (In most other computer languages such swaps are more complicated, requiring the use of an additional temporary variable.)

4.5 Output and Input functions

There are some tasks that many programs have to perform such as reading input values from the keyboard, print outputs on the screen of the computer, computing the absolute value a number and so on. Python provides so-called *functions* that perform these common tasks, as well as many others.

=	Assignment	Assigns values from right side operands to left side operand.
+=	Addition Assignment	Adds the value of right operand to the left operand and assigns the result to left operand.
=	Subtraction Assignment	Subtracts the value of right operand from the left operand and assigns the result to left operand.
*=	Multiplication Assignment	Multiplies the value of right operand with the left operand and assigns the result to left operand.
/=	Division Assignment	Divides the value of right operand with the left operand and assigns the result to left operand.
**=	Exponentiation Assignment	Evaluates to the result of raising the first operand to the power of the second operand.
//=	Floor Division Assignment	Produces the integral part of the quotient of its operands where the left operand is the dividend and the right operand is the divisor.
%=	Remainder Assignment	Computes the remainder after division and assigns the value to the left operand.

Table 4.2: List of Assignments Operators

We will discuss about functions in detail in the next chapters. For now, we just need to know that a function is called by using its name, followed by parentheses. Many functions require values when they are called, such as the number for which the absolute value will be computed. These values, called *arguments*, are placed inside the parentheses when the function is called. When a function call has multiple arguments they are separated by commas. Many functions compute a result. This result can be stored in a variable using an assignment statement. The name of the variable appears to the left of the assignment operator and the function call appears to the right of the assignment operator. For example, the following assignment statement calls the `abs` function, which computes the absolute value of a given number:

```
r = abs(-3.4)
```

When the `abs` function executes it identifies the absolute value of the number -3.4. Then the returned value is stored in r.

We have introduced functions briefly, in order to understand how to print outputs on the screen of the computer and to read input values from the keyboard.

4.5.1 Function `print`

The function `print()` allows a program to display text on the screen of the computer. It can be called with one argument, which is the value that will be displayed. For example,

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

This code prints the string Hello World!! onto the console. Notice that the string Hello World!! is enclosed within double quotes inside the print() function. Let us consider this other example:

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

In this case, the function `print()` prints the number 1. Another example:

```
>>> x=2
>>> print(x)
2
```

In this case, the function `print()` prints the value currently store in the variable x, not the letter "x". To print the letter "x", this is the code:

```
>>> print("x")
x
```

The function `print()` always prints the current value of the variable at the moment the statement is executed. Thus consider this example:

```
>>> x = 1
>>> print(x)
1
>>> x = 2
>>> print(x)
2
```

First the variable x is set to 1 and its value is printed out, resulting in a 1 on the screen. Then the value of x is changed to 2 and the value is printed again, which produces a 2 on the screen. Multiple values can be printed with one function call of the function `print()` by providing several arguments separated by commas. For example:

```
>>> x = 1
>>> y = 2
>>> print("When x is", x, "the value of y is", y)
When x is 1 the value of y is 2
```

Notice that a space is automatically included between each item when multiple arguments are given. Moreover, the arguments to a function call can also be arbitrarily complex expressions involving mathematical operators. For example. and other function calls. Consider the following statement:

```
>>> x = 2
>>> y = 4
>>> print("The product of", x, "and", y, "is", x * y)
The product of 2 and 4 is 8
```

When it executes, the product $x * y$ is computed and then displayed along with all of the other arguments to the function `print()`.

4.5.2 Function input

Python programs can read input from the keyboard by calling the function `input`. This function causes the program to stop and wait for the user to type something. When the user presses the keyboard button *Enter*, characters typed by the user are returned by the input function. Then the program continues executing. Input values are normally stored in a variable using an assignment statement so that they can be used later in the program. For example:

```
>>> a = input()
```

This code reads a value typed by the user and stores it in a variable named a.

An argument can be provided to the function `input`. This argument is a prompt that tells the user what to enter. The prompt must be string. For example,

```
>>> x = input("Enter the value of x: ")
Enter the value of x:
```

When the computer executes this statement it does two things. First, the statement acts something like a print statement and prints out the string inside the parentheses. Next the computer will stop and wait until the user types something. Then, the value that the user types is assigned to the variable `x`. However, there is a catch: the function `input` always returns a string. In other words, the value entered is always interpreted as a string value, even if the user types a number 1.

Therefore, if we want that the values inserted is interpreted correctly as numbers, not strings, a conversion mechanism is necessary (see the next section).

4.6 Type conversions

There are cases in which we need to convert the type of a variable. For example, when we wish to give a variable an integer value, like 1, but, at the same time, we wish that variable be a float. Or, another example, when we want to convert the string returned by the function `input` in a number. In all these cases, you can explicitly cast, or convert, a variable from one type to another by using a function provided by Python. Some functions used to achieve this aim are:

- function `int`: convert a float number or a string to an integer. For example,

  ```
  >>> float_to_int = int(3.5)
  >>> print(float_to_int)
  3
  >>> string_to_int = int("1")
  >>> print(string_to_int+1)
  2
  ```

  In the first statement, we convert the real number 3.5 in the integer 3. Notice that occurs the truncation of the real number. In the second call of the function `int`, we convert the string "1" in an integer. In this way, it is possible to apply the mathematical operator +;
- function `float`: convert a integer number or a string to a float. For example,

  ```
  >>> int_to_float = float(4)
  >>> print(int_to_float)
  4.0
  >>> string_to_float = float("1")
  ```

```
>>> print(string_to_float+1.0)
2.0
```

In the first statement, we convert the integer number 4 in the float 4.0. In the second call of the function `float`, we convert the string "1" in a float. In this way, it is possible to apply the mathematical operator + and obtain a float;

- function `str`: convert a number to a string. For example,

```
>>> int_to_string = str(4)
>>> print(int_to_string)
4
```

In the first statement, the function `str` is used to convert the integer value 4 in a string.

Sometimes, it can be useful to know the type that the python interpreter associates to a variable. To achieve this aim, Python provides the function `type`. The argument to this function is the variable whose we want to know the type. For example, by using this function, it is possible to check if the aforementioned functions for casting work. Let us consider the following example:

```
>>> int_to_string = str(4)
>>> print(int_to_string)
4
>>> type(int_to_string)
<class 'str'>
>>> type(4)
<class 'int'>
```

In this example, it is possible to know that the type of the variable `int_to_string` is string, whereas, the type of the value 4 is integer.

4.7 Exercises

The exercises in this chapter will allow you to put the concepts discussed previously into practice. While the tasks that they ask you to complete are generally small, solving these exercises is an important step toward the creation of larger programs that solve more interesting problems.

Exercise 4.1 — Hello. Write a program that asks the user to enter his or her name. The program should respond with a message that says hello to the user, using his or her name. ■

Exercise 4.2 — Area of a Room. Write a program that asks the user to enter the width and length of a room. Once these values have been read, your program should compute and display the area of the room. The length and the width will be entered as floating-point numbers. Include units in your prompt and output message; either feet or meters, depending on which unit you are more comfortable working with. ■

Exercise 4.3 — Tax and Tip. The program that you create for this exercise will begin by reading the cost of a meal ordered at a restaurant from the user. Then your program will compute the tax and tip for the meal. Use your local tax rate when computing the amount of tax owing. Compute the tip as 18 percent of the meal amount (without the tax). The output from your program should

include the tax amount, the tip amount, and the grand total for the meal including both the tax and the tip. ■

Exercise 4.4 — Sum of the First n Positive Integers. Write a program that reads a positive integer, n, from the user and then displays the sum of all of the integers from 1 to n. The sum of the first n positive integers can be computed using the formula:

$$sum = \frac{(n)(n+1)}{2}$$

■

Exercise 4.5 — Arithmetic. Create a program that reads two integers, a and b, from the user. Your program should compute and display:

- The sum of a and b
- The difference when b is subtracted from a
- The product of a and b
- The quotient when a is divided by b
- The remainder when a is divided by b
- The result of a^b

■

Exercise 4.6 — Day Old Bread. A bakery sells loaves of bread for $3.49 each. Day old bread is discounted by 60 percent. Write a program that begins by reading the number of loaves of day old bread being purchased from the user. Then your program should display the regular price for the bread, the discount because it is a day old, and the total price. Each of these amounts should be displayed on its own line with an appropriate label. ■

5. Strings and Branching Programs

This chapter presents the fundamentals about the string manipulations and branching programs by dealing with `if`, `while` and `for` statements. Contents of this chapter is extracted in part by the following sources [GV18][Ste15].

5.1 Strings

A *string* is a sequence of characters: letters, numbers, punctuation marks, spaces,etc.
In Python, string literals are specified by enclosing a sequence of characters within a matching pair of either single or double quotes.

```
print("This is a string.", 'So is this.')
```

By allowing both types of delimiters, Python makes it easy to include an apostrophe or quotation mark within a string.

```
message = 'He said "Hello"‘
```

Note: Remember to use matching pairs of quotes, single with single, double with double.

■ **Example 5.1 — String Length.** You can compute the length of a string using Python's `len()` function.

```
>>> length = len("World!")
>>> print(length)
6
```

■

■ **Example 5.2 — String Repetition.** You can produce a string that is the result of repeating a string multiple times by using the operator '*'.

```
>>> dashes = "-" * 5
>>> print(dashes)
-----
```

■

■ **Example 5.3 — String Concatenation.** You can ‘add’ one String onto the end of another with the operator ’+’.

```
>>> firstName = "Harry"
>>> lastName = "Morgan"
>>> name  = firstName + lastName
>>> print("my name is:", name)
my name is: HarryMorgan
```

■

■ **Example 5.4 — Converting Numbers to String.** Use the `str()` function to convert between numbers and strings.

```
>>> balance = 888.88
>>> type(balance)
<class 'float'>
>>> balanceAsString = str(balance)
>>>type(balanceAsString)
<class 'str'>
```

■

■ **Example 5.5 — Accessing a String.** You can access to each char inside a string by using an index number.

```
>>> name = "Mary"
>>> start = name[0]
>>> print(start)
M
>>> final = name[len(name)-1]
>>> print(final)
y
```

■

■ **Example 5.6 — Comparing Strings.** You can check if two strings are equal by using the equality operator "==".

```
>>> name1 = "Mary"
>>> name2 = "Mary"
>>> print(name1 == name2)
True
```

■

■ **Example 5.7 — Operator in.** You can check if a string contains a given substring by using the operator in.

```
>>> name = "John Wayne"
>>> print("Way" in name)
True
```

■

Exercise 5.1 — Welcome. Write a Python program to display a string made of the first 2 and the last 2 chars of the string "Welcome!". ■

Exercise 5.2 — Swap strings. Write a Python program to build a string by concatenating (separated by a space) two strings given by the user after swapping the first two characters of the given strings. ■

5.2 Branching programs

Branching programs contain statements that may or may not be executed when the program runs. Execution still begins at the top of the program and progresses toward the bottom, but some statements that are present in the program may be skipped. This allows performing different tasks for different input values and greatly increasing the variety of problems that a Python program can solve.

5.2.1 If Statements

Python programs make decisions using `if` statements. An `if` statement includes a condition and one or more statements that form the body of the `if` statement. When an `if` statement is executed, its condition is evaluated. If the condition evaluates to `True` then the body of the `if` statement executes, otherwise the body of the `if` statement is skipped and execution continues at the first line after the body of the `if` statement.

■ **Example 5.8 — If statement.** The following program reads a number from the user and displays a message if the number is zero.

```
# Read a number from the user
num =  int(input("Enter a number: "))
# Store the appropriate message in result
if num == 0:
  result = "The number was zero"
  print(result)
```

■

An `if` statement can also be followed by an `else` statement which is optional. The `if-else` statement consists of an `if` part with a condition and a body, and an `else` part with a body (but no condition). When the statement executes its condition is evaluated. If the condition evaluates to `True` then the body of the `if` part executes and the body of the `else` part is skipped. When the condition evaluates to `False` the body of the `if` part is skipped and the body of the `else` part executes.

■ **Example 5.9 — If-else statement.** The following program displays if a number introduced by a user is zero or not.

```
# Read a number from the user
num =  int(input("Enter a number: "))
# Store the appropriate message in result
if num == 0:
  result = "The number was zero"
else:
  result = "The number was not zero"
print(result)
```

■

More in general, an `if-elif-else` statement is used to execute exactly one of several alternatives. The statement begins with an `if` part, followed by one or more `elif` parts, followed by an `else` part. All of these parts must include a body that is indented. Each of the `if` and `elif` parts must also include a condition that evaluates to either `True` or `False`.

■ **Example 5.10 — If-elif-else statement.** The following program displays if a number introduced by a user is zero, less or greater than zero.

```
# Read a number from the user
num = int(input("Enter a number: "))
# Store the appropriate message in result
if num > 0:
  result = "That's a positive number"
elif num < 0:
  result = "That's a negative number"
else:
  result = "That's zero"
# Display the message
print(result)
```

■

The `else` that appears at the end of an `if-elif-else` statement is optional. When the `else` is present, the statement selects exactly one of several options. Omitting the else selects at most one of several options. When an `if-elif` statement is used, none of the bodies execute when all of the conditions evaluate to `False`. Whether one of the bodies executes, or not, the program will continue executing at the first statement after the body of the final `elif` part.

The body of any `if` part, `elif` part or `else` part of any type of `if` statement can contain (almost) any Python statement, including another `if`, `if-else`, `if-elif` or `if-elif-else` statement. When one `if` statement (of any type) appears in the body of another `if` statement (of any type) the `if` statements are said to be *nested*.

■ **Example 5.11 — Nested if statement.** This program begins by reading a number from the user and displays an opportune message according if the number is positive or negative.

```
# Read a number from the user
num = float(input("Enter a number: "))
```

```
# Store the appropriate message in result
if num > 0:
    # Determine what adjective should be used to describe the number adjective = " "
    if num >= 1000000:
        adjective = " really big "
    elif num >= 1000:
        adjective = " big "

    # Store the message for positive numbers including the appropriate adjective
    result = "That's a" + adjective + "positive number"
elif num < 0:
    result = "That's a negative number"
else:
    result = "That's zero"
# Display the message
print(result)
```

■

The condition on an *if* statement can be an arbitrarily complex expression that evaluates to either *True* or *False*. Such an expression is called a Boolean expression, named after George Boole (1815–1864), who was a pioneer in formal logic. An *if* statement's condition often includes a relational operator that compares two values, variables or complex expressions. Python's relational operators are listed in Table 5.1.

Operator	Operator Name	Description
==	Equal to	If the values of two operands are equal, then the condition becomes True.
!=	Not Equal to	If values of two operands are not equal, then the condition becomes True.
>	Greater than	If the value of left operand is greater than the value of right operand, then condition becomes True.
<	Lesser than	If the value of left operand is less than the value of right operand, then condition becomes True.
>=	Greater than or equal to	If the value of left operand is greater than or equal to the value of right operand, then condition becomes True.
<=	Lesser than or equal to	If the value of left operand is less than or equal to the value of right operand, then condition becomes True.

Table 5.1: List of Relation Operators

Boolean expressions can also include Boolean operators that combine and manipulate Boolean values. Python includes three Boolean operators: *not*, *and*, and *or*. The behavior of any Boolean expression can be described by a truth table. A truth table has one column for each distinct variable

in the Boolean expression, as well as a column for the expression itself. The truth tables for the *not*, *and*, and *or* operators are shown below.

x	NOT x
FALSE	TRUE
TRUE	FALSE

x	y	x AND y	x OR y
FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE

■ **Example 5.12 — Compound Boolean Expressions.** The following Python program uses the *or* operator to determine whether or not the value entered by the user is one of the first 5 prime numbers.

```
# Read an integer from the user
x = int(input("Enter an integer: "))
# Determine if it is one of the first 5 primes and report the result
if x == 2 or x == 3 or x == 5 or x == 7 or x == 11:
  print("That's one of the first 5 primes.")
else:
  print("That is not one of the first 5 primes.")
```

■

Exercise 5.3 — Order. Write a program that reads three numbers and displays the message "increasing" if they are in increasing order, "decreasing" if they are in decreasing order and "neither" if they are neither in increasing order nor in decreasing order. In this increasing exercise it means strictly increasing, that is, each value must be greater than the previous one (the same meaning has the term decreasing): the sequence 3 4 4, therefore, should not be considered increasing. ■

Exercise 5.4 — Num Days. Write a program that asks the user to enter the identification number of a month (1 for January, 2 for February, and so on), to then display the number of days from which the selected month is made up. In the case of February, display the message "28 or 29 days". Enter a month: 5 30 days Do not use a different if / else branch for each month, but compose the conditions using the appropriate Boolean operators. ■

5.2.2 While Loops

A `while` loop causes one or more statements to execute as long as, or while, a condition evaluates to *True*. If the `while` loop's condition evaluates to `True` then the body of the loop is executed. When the bottom of the loop body is reached, execution returns to the top of the loop, and the

loop condition is evaluated again. If the condition still evaluates to `True` then the body of the loop executes again. The loop's body continues to execute until the `while` loop condition evaluates to `False`. When this occurs, the loop's body is skipped, and execution continues at the first statement after the body of the `while` loop.

■ **Example 5.13 — While statement.** The following code segment reads values from the user and reports whether each value is positive or negative. The loop terminates when the user enters 0. Neither message is displayed in this case.

```
# Read the first value from the user
x = int(input("Enter an integer (0 to quit): "))
# Keep looping while the user enters a non-zero number
while x != 0:
# Report the nature of the number
    if x > 0:
        print("That's a positive number.")
    else:
        print("That's a negative number.") # Read the next value from the user
    x = int(input("Enter an integer (0 to quit): "))
```

■

Exercise 5.5 — Count by 5. Write a program that displays how to count by 5. In other words, the program must have the following output:
5 * 1 = 5
5 * 2 = 10
....
5 * 10 = 50
■

Exercise 5.6 — Divisible by 7. Write a program that displays all numbers divisible by 7 between 0 and N where N is typed by the user. ■

5.2.3 For Loops

Like `while` loops, `for` loops cause statements that only appear in a program once to execute several times when the program runs. However the mechanism used to determine how many times those statements will execute is rather different for a `for` loop. A `for` loop executes once for each item in a collection. The collection can be a range of integers, the letters in a string, or as we'll see in later chapters, the values stored in a data structure, such as a list.

■ **Example 5.14 — For statement.** The following program uses a `for` loop and the `range` function to display all of the positive multiples of 3 up to (and including) a value entered by the user.

```
# Read the limit from the user
limit = int(input("Enter an integer: "))
# Display the positive multiples of 3 up to the limit
print("The multiples of 3 up to and including", limit, "are:")
```

```
for i in range(3, limit + 1, 3):
    print(i)
```

■

■ **Example 5.15 — For statement with strings.** The following program uses a `for` loop to print the characters of a string read from the user one by one.

```
# Read the string from the user
s = input("Enter a string: ")
# Display the characters one by one
for c in s:
    print(c)
```

■

The statements inside the body of a loop can include another loop. When this happens, the inner loop is said to be *nested* inside the outer loop. Any type of loop can be nested inside of any other type of loop.

■ **Example 5.16 — Nested loop.** The following program uses a `for` loop nested inside a `while` loop to repeat messages entered by the user until the user enters a blank message.

```
# Read the first message from the user
message = input("Enter a message (blank to quit): ")
# Loop until the message is a blank line
while message != "":
# Read the number of times the message should be displayed
n = int(input("How many times should it be repeated? "))
# Display the message the number of times requested
  for i in range(n):
    print(message)
# Read the next message from the user
  message = input("Enter a message (blank to quit): ")
```

■

Exercise 5.7 — Sum Even. Write a program that calculates the sum of all even numbers between 2 and 100 (extremes included). ■

Exercise 5.8 — Sum Squares. Write a program that computes the sum of all of the squares from 1 to 100 (inclusive). ■

Exercise 5.9 — All Powers. Write a program that calculates all powers of 2, from 2**0 to 2**20 (inclusive). ■

Exercise 5.10 — Sum Odd and Even. Write a program to find the sum of all odd and even numbers up to a number (inclusive) specified by the user. ■

5.2.4 Break statement

A useful refinement of the **while** and `for` statement is the `break` statement. The `break` statement allows us to break out of a loop even if the condition in the while statement is not met.

■ **Example 5.17 — Break statement.** The following program will continue looping until you enter a number not greater than 10, except if you enter the number 111, in which case it will give up and proceed with the rest of the program.

```
while x>10:
    print("This is greater than ten. Please try again.")
    x = int(input("Enter a whole number no greater than ten: "))
    if x==111:
        break
```

■

Exercise 5.11 — Break if negative. Write a program that reads five positive numbers by the user and print them after the insertion. If the user inserts a negative number the insertion must terminate. ■

5.3 Chapter Exercises

Exercise 5.12 — Letter. Write a program that asks the user to provide a single letter of the alphabet, then displaying the message "Vowel" if it is a vowel and "Consonant" if it is a consonant. If the user does not type a letter (i.e. a character between "a" and "z" or between "A" and "Z") or type a string longer than one, display an error message. ■

Exercise 5.13 — Numbers. Write a program that reads a set of floating point values and then displays:

- a. the average of the values;
- b. the lower value;
- c. the higher value;
- d. the dynamics of values, that is, the difference between the greater and the lesser value.

■

Exercise 5.14 — Sum Odd digits. Write programs that, using cycles, calculate the sum of all the odd digits of a number acquired in input (if, for example, the number is 32677, the sum to be calculated is 3 + 7 + 7 = 17). ■

Exercise 5.15 — Reverse. Write a program that reads a word and displays it in reverse. For example, if the user supplies the string "Harry", the program must display: yrraH ■

Exercise 5.16 — Vowels. Write a program that reads a word and displays the number of vowels present in it (for this exercise assume that the vowels are a and i or u y). For example, if the user supplies the string "Harry", the program must display the message: 2 vowels. ■

Exercise 5.17 — Binary to decimal. Write a program that reads a number (base 2) and converts it to decimal (base 10). ■

Exercise 5.18 — Decimal to binary. Write a program that reads a number (base 10) and converts it to binary (base 2). ■

6. Functions and Structured Data Types

This chapter deals with the definition and invocation of functions, variable scope, recursion, definition and use of sequences such as lists, tuples and dictionaries. Contents of this chapter is extracted in part by the following sources [GV18][Ste15][Ced10] [Hil16][LL16].

6.1 User-defined Functions

You can create your own functions and use them as and where it is needed. User-defined functions are code blocks created by users to perform some specific task in the program. Functions serve several important purposes: i) they let us divide a program into small parts in such a way that different parts each perform some specific task, then each part can be called as per the requirement (*modularity of code*); ii) they let us write code once and then call it from many locations (*reusability of code*); iii) they allow us to test different parts of our solution individually (*manageability of code*).

In Python, a function definition consists of the `def` keyword, followed by:

- The *name* of the function. The function's name has to adhere to the same naming rules as variables: use letters, numbers, or an underscore, but the name cannot start with a number. Also, you cannot use a keyword as a function name.
- A *list of formal parameters* to the function are enclosed in parentheses and separated by commas. Some functions do not have any parameters at all while others may have one or more parameters.
- A *colon* is required at the end of the *function header*, i.e., the first line of the function definition which includes the name of the function.
- Block of statements that define the *body of the function*. The first statement among the block of statements within the function definition can optionally be a documentation string or *docstring*. Triple quotes are used to represent docstrings.

Defining a function does not execute it. Defining a function simply names the function and specifies what to do when the function is called. *Calling* the function actually performs the specified actions with the *actual parameters*. There must be a one to one correspondence between the formal parameters in the function definition and the actual arguments of the calling function. When a

function is called, the formal parameters are temporarily “bound” to the arguments and their initial values are assigned through the calling function.

■ **Example 6.1 — Trapezium Area function.** Definition of a function to find the area of a trapezium using the formula $area = (1/2) * (base1 + base2) * height$.

```
def area_trapezium(base1, base2, height):
  area = 0.5 * (base1 + base2) * height
  print("Area of the trapezium is", area)
```

■

■ **Example 6.2 — Function call by positional way.** Calling the trapezium area function defined in the Example 6.1 by means of the *positional way*.

```
>>> area_trapezium(2,3,3)
Area of the trapezium is 7.5
```

■

■ **Example 6.3 — Function call by keyword arguments.** Calling the trapezium area function defined in the Example 6.1 by means of the *keyword arguments way*.

```
>>> area_trapezium(base2=3,base1=2, height=3)
Area of the trapezium is 7.5
```

■

In some situations, it might be useful to set a *default value* to the parameters of the function definition. Each default parameter has a default value as part of its function definition. Any calling function must provide arguments for all required parameters in the function definition but can omit arguments for default parameters. If no argument is sent for that parameter, the default value is used. Usually, the default parameters are defined at the end of the parameter list, after any required parameters and non-default parameters cannot follow default parameters.

■ **Example 6.4 — Trapezium Area function with default parameters.** Definition of a function to find the area of a trapezium using the formula $area = (1/2) * (base1 + base2) * height$ where the height parameter is set to 5 by default.

```
def area_trapezium(base1, base2, height=5):
  area = 0.5 * (base1 + base2) * height
  print("Area of the trapezium is", area)
```

■

■ **Example 6.5 — Call a function with default parameters.** Calling the trapezium area function defined in the Example 6.4 by considering 5 as the value of the height.

```
>>> area_trapezium(2,3)
Area of the trapezium is 12.5
>>> area_trapezium(2,3,5)
Area of the trapezium is 12.5
```

■

Most of the times you may want the function to perform its specified task to calculate a value and return the value to the calling function so that it can be stored in a variable and used later. This can be achieved using the optional `return` statement in the function definition. The `return` statement terminates the execution of the function definition in which it appears and returns control to the calling function. It is possible to define functions without a `return` statement, in this case, the function returns `None`.

■ **Example 6.6 — Power.** Define a function that computes and returns the value x to the power of y.

```
>>> def power(x, y):
...   r = 1
...   while y > 0:
...     r = r * x
...     y = y - 1
...   return r
...
>>> value = power(3, 3)
>>> print(value)
27
```

■

When a variable is created inside a function the variable is *local* to that function. This means that the variable only exists when the function is executing and that it can only be accessed within the body of that function. The variable ceases to exist when the function returns, and as such, it cannot be accessed after that time. Conversely, variables assigned outside all function `def`s are global and are available everywhere within the program file. However, these global variables cannot be modified in a function unless one uses the keyword `global`.

■ **Example 6.7 — Variable scope.** An example to better understand the variable scope. In the code, the function named `func` succeeds to access the variable `b` defined outside of the function.

```
>>> def func():
...   a = 5
...   print(a,b)
...
>>> b = 6
>>> func()
5 6
```

■

■ **Example 6.8 — Keyword global.** An example to better understand the use of the keyword `global`. In the code, the function named `func2` succeeds to modify the variable `x` defined outside of the function since it declares it as global.

```
>>> def func2():
...   global x
...   x += 1
...
```

```
>>> x = 4
>>> func2()
>>> print(x)
5
```

■

Functions may also be defined within other functions. It that case, they become local functions, or *nested functions*, known only to the function inside which they are defined. A nested function has full access to all variables in the parent function, i.e. the function within which it is defined.

■ **Example 6.9 — Nested function.** Define a function to calculate and add the surface Area of two cubes.

```
>>> def add_cubes(a, b):
...  def cube_surface_area(x):
...   return 6 * x**2
...  return cube_surface_area(a) + cube_surface_area(b)
...
>>> result = add_cubes(2, 3)
>>> print("The surface area after adding two cubes is", result)
The surface area after adding two cubes is 78
```

■

Short functions can be defined in a compact way, using what is known as a `lambda` function. The syntax consists of `lambda` followed by a series of arguments, colon, and some Python expression resulting in an object to be returned from the function. `lambda` functions are particularly convenient as function arguments.

■ **Example 6.10 — Lambda function.** An example of the use of a lambda function.

```
>>> def treat_xy(f, x, y):
...  return f(x, y)
...
>>>result = treat_xy(lambda x, y: x+2*y, 3, 4)
>>> print(result)
11
```

■

Exercise 6.1 — Check sign. Write a program that reads an integer from the user and displays a message indicating whether the integer is negative, positive or zero. The check task must be made in a function. ■

Exercise 6.2 — Average. Write a program that gets an integer $N > 1$ from the user and computes the average of all integers $i = 1, \ldots, N$. The computation should be done in a function that takes N as input parameter. Print the result to the screen with an appropriate text. ■

Exercise 6.3 — Test range. Write a Python function to check whether a number x is in a given range $[a,b]$. Write a main program that reads three number, a, b and x, and check if x is in the range $[a,b]$. ■

Exercise 6.4 — Letter frequency. Write a function that returns the frequency of a letter in a string text, i.e., the number of occurrences of letter divided by the length of text. Write a main program that reads a string and a letter and displays the frequency of the letter in the string. ■

Any function that calls itself is *recursive* because the function's body (its definition) includes a call to the function that is being defined. In order to reach a solution a recursive function must have at least one case where it is able to produce the required result without calling itself. This is referred to as the base case. Cases where the function calls itself are referred to as recursive cases.

■ **Example 6.11 — Summing Integers recursively.** Define a function that sums all the integers from 0 up to and including the positive integer n.

```
>>> def sum_to(n):
...   if n <= 0:
...    return 0 # Base case
...   else:
...    return n + sum_to(n - 1) # Recursive case
...
>>> sum_to(10)
55
```

■

Exercise 6.5 — Recursive power. Write a Python program to calculate the value a^b recursively. Write a main program that reads two numbers a and b and displays a^b'. ■

6.2 Sequence Types

Up until this point, every variable that we have created has held one value. The value could be a integer, a Boolean, a string, or a value of some other type. While using one variable for each value is practical for small problems it quickly becomes untenable when working with larger amounts of data. Structured types help us overcome this problem by allowing several, even many, values to be stored in one variable.

6.2.1 Lists

Lists are the basic ordered and *mutable* data collection type in Python. You can think of the list as a container that holds a number of items. Lists avoid having a separate variable to store each item which is less efficient and more error prone when you have to perform some operations on these items. Lists can be simple or nested lists with varying types of values. Lists are one of the most flexible data storage formats in Python because they can have values added, removed, and changed.

Lists are constructed using square brackets [ ] wherein you can include a list of items separated by commas. You can create an empty list without any items.

■ **Example 6.12 — Creating lists.** Examples to understand how lists are created in Python.

```
>>> superstore = ["metro", "tesco", "walmart", "kmart", "carrefour"]
>>> print(superstore)
['metro', 'tesco', 'walmart', 'kmart', 'carrefour']
>>> number_list = [4, 4, 6, 7, 2, 9, 10, 15]
>>> print(number_list)
[4, 4, 6, 7, 2, 9, 10, 15]
>>> mixed_list = ['dog', 87.23, 65, [9, 1, 8, 1]]
>>> print(mixed_list)
['dog', 87.23, 65, [9, 1, 8, 1]]
>>> empty_list = []
>>> print(empty_list)
[]
```

■

Lists can be concatenated using the + sign, and the * operator is used to create a repeated sequence of list items. You can check for the presence of an item in the list using `in` and `not in` membership operators. It returns a Boolean `True` or `False`.

■ **Example 6.13 — Basic Operations.** An example to understand as operators +, *, `in`, `not in` in Python.

```
>>> list_1 = [1, 3, 5, 7]
>>> list_2 = [2, 4, 6, 8]
>>> list_1 + list_2
[1, 3, 5, 7, 2, 4, 6, 8]
>>> list_1 * 3
[1, 3, 5, 7, 1, 3, 5, 7, 1, 3, 5, 7]
>>> 3 in list_1
True
>>> 10 in list_1
False
>>> 10 not in list_1
True
```

■

As an ordered sequence of elements, each item in a list can be called individually, through indexing. The expression inside the bracket is called the index. Lists use square brackets [ ] to access individual items, with the first item at index 0, the second item at index 1 and so on. The index provided within the square brackets indicates the value being accessed. Moreover, you can modify a list by replacing the older item with a newer item in its place and without assigning the list to a completely new variable. In addition, slicing of lists is allowed in Python wherein a part of the list can be extracted by specifying index range along with the colon (:) operator which itself is a list. Finally, the colon (:) operator can be used to clone a list.

■ **Example 6.14 — Indexing and Slicing.** An example to understand as indexing and slicing work in Python.

```
>>> fruits = ["grapefruit", "pineapple", "blueberries", "mango", "banana"]
>>> fruits[1]
'pineapple'
>>> fruits[5]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> fruits[-1]
'banana'
>>> fruits[1] = 'apple'
>>> fruits
['grapefruit', 'apple', 'blueberries', 'mango', 'banana']
>>> fruits[1:3]
['apple', 'blueberries']
>>> fruits[:3]
['grapefruit', 'apple', 'blueberries']
>>> fruits[2:]
['blueberries', 'mango', 'banana']
>>> fruits[1:4:2]
['apple', 'mango']
>>> fruits2 = fruits[:]
>>> fruits2[2] = 'strawberries'
>>> fruits
['grapefruit', 'apple', 'blueberries', 'mango', 'banana']
>>> fruits2
['grapefruit', 'apple', 'strawberries', 'mango', 'banana']
```

■

There are many built-in functions for which a list can be passed as an argument (see Table 6.1).

Method	Description
`len()`	The **len()** function returns the numbers of items in a list.
`sum()`	The `sum()` function returns the sum of numbers in the list.
`any()`	The `any()` function returns True if any of the Boolean values in the list is True, else returns False.
`all()`	The `all()` function returns True if all the Boolean values in the list are True, else returns False.
`sorted()`	The `sorted()` function returns a modified copy of the list while leaving the original list untouched.

Table 6.1: Some built-in functions for lists [GV18]

Note: Python assigns boolean values to values of other types. For numerical types like integers and floating-points, zero values are false and non-zero values are true. For strings, empty strings are false and non-empty strings are true.

Example 6.15 — Built-in functions. Examples to understand how to apply some built-in functions to the lists.

```
>>> lakes = ['superior', 'erie', 'huron', 'ontario', 'powell']
>>> numbers = [1, 2, 3, 4, 5]
>>> len(numbers)
5
>>> sum(numbers)
15
>>> max(numbers)
5
>>> min(numbers)
1
>>> any([1, 1, 0, 0, 1, 0])
True
>>> all([1, 1, 1, 1])
True
>>> lakes_sorted_new = sorted(lakes)
>>> lakes_sorted_new
['erie', 'huron', 'ontario', 'powell', 'superior']
```

■

There are several methods associated with the lists (see Table 6.2).

Example 6.16 — List methods. Examples to understand how to apply some methods on the lists.

```
>>> cities = ["london", "paris", "washington", "seattle", "washington"]
>>> cities.count('paris')
1
 >>> cities.index('washington')
2
>>> cities.reverse()
>>> cities
['washington', 'seattle', 'washington', 'paris', 'london']
>>> cities.append('brussels')
>>> cities
['washington', 'seattle', 'washington', 'paris', 'london', 'brussels']
>>> cities.sort()
>>> cities
['brussels', 'london', 'paris', 'seattle', 'washington', 'washington']
>>> cities.pop()
'washington'
>>> cities
['brussels', 'london', 'paris', 'seattle', 'washington']
>>> more_cities = ["brussels", "copenhagen"]
>>> cities.extend(more_cities)
>>> cities
```

Method	Syntax	Description
`append()`	`list.append(item)`	The `append()` method adds a single item to the end of the list. This method does not return new list and it just modifies the original.
`count()`	`list.count(item)`	The `count()` method counts the number of times the item has occurred in the list and returns it.
`insert()`	`list.insert(index, item)`	The `insert()` method inserts the item at the given index, shifting items to the right.
`extend()`	`list.extend(list2)`	The `extend()` method adds the items in list2 to the end of the list.
`index()`	`list.index(item)`	The `index()` method searches for the given item from the start of the list and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the list then `ValueError` is thrown by this method.
`remove()`	`list.remove(item)`	The `remove()` method searches for the first instance of the given item in the list and removes it. If the item is not present in the list then `ValueError` is thrown by this method.
`sort()`	`list.sort()`	The `sort()` method sorts the items in place in the list. This method modifies the original list and it does not return a new list.
`reverse()`	`list.reverse()`	The `reverse()` method reverses the items in place in the list. This method modifies the original list and it does not return a new list.
`pop()`	`list.pop([index])`	The `pop()` method removes and returns the item at the given index. This method returns the rightmost item if the index is omitted.

Table 6.2: Various list methods [GV18]

```
['brussels', 'london', 'paris', 'seattle', 'washington', 'brussels',
'copenhagen']
>>> cities.remove("brussels")
>>> cities
[ 'london', 'paris', 'seattle', 'washington', 'brussels', 'copenhagen']
```

■

A *list comprehension* consists of brackets containing a *variable* or *expression* (First Part) followed by a `for` clause (Middle Part), then predicate True or False using an `if` clause (Last Part). The components expression and predicate are optional.

■ **Example 6.17 — List comprehension.** An example to understand how to work the list comprehension statements.

```
>>> squares = [x**2 for x in range(1, 10)]
>>> squares
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

■

Exercise 6.6 — List order. Display a list of integers entered by the user in ascending order. ■

Exercise 6.7 — Word collection. Read a collection of words entered by the user. Display each word entered by the user only once, in the same order that the words were entered. ■

Exercise 6.8 — Number list. Read a collection of integers from the user. Display all of the negative numbers, followed by all of the zeros, followed by all of the positive numbers. ■

Exercise 6.9 — Some items. Write a Python program that reads a list A of N items from the user and creates a new list with all the elements of the list A that are less than 5. N must be read from the user. ■

Exercise 6.10 — Even items. Write a Python program that reads a list A of N items from the user and creates a new list that has only the even elements of the list A. N must be read from the user. ■

6.2.2 Tuples

A tuple is a finite ordered list of values of possibly different types which is used to bundle related values together without having to create a specific type to hold them. Tuples are immutable. Once a tuple is created, you cannot change its values. A tuple is defined by putting a comma-separated list of values inside parentheses ( ). Each value inside a tuple is called an item.

■ **Example 6.18 — Creating tuples.** Examples to understand how tuples are created in Python.

```
>>> air_force = ("f15", "f22a", "f35a")
>>> fighter_jets = (1988, 2005, 2016, air_force)
>>> fighter_jets
(1988, 2005, 2016, ('f15', 'f22a', 'f35a'))
>>> type(fighter_jets)
<class 'tuple'>
>>> empty_tuple = ()
>>> empty_tuple
()
>>> type(empty_tuple)
<class ‘tuple’>
```

■

Like in lists, you can use the + operator to concatenate tuples together and the * operator to repeat a sequence of tuple items. Moreover, you can check for the presence of an item in a tuple using in and not in membership operators. It returns a Boolean True or False.

■ **Example 6.19 — Basic Operations.** An example to understand as operators `+`, `*`, `in`, `not in` in Python.

```
>>> tuple_1 = (2, 0, 1, 4)
>>> tuple_2 = (2, 0, 1, 9)
>>> tuple_1 + tuple_2
(2, 0, 1, 4, 2, 0, 1, 9)
>>> tuple_1 * 3
(2, 0, 1, 4, 2, 0, 1, 4, 2, 0, 1, 4)
>>> tuple_items = (1, 9, 8, 8)
>>> 1 in tuple_items
True
>>> 25 in tuple_items
False
```

■

Each item in a tuple can be called individually through indexing. The expression inside the bracket is called the index. Square brackets [ ] are used by tuples to access individual items, with the first item at index 0, the second item at index 1 and so on. The index provided within the square brackets indicates the value being accessed. In addition to positive index numbers, you can also access tuple items using a negative index number, by counting backwards from the end of the tuple, starting at 1. Negative indexing is useful if you have a large number of items in the tuple and you want to locate an item towards the end of a tuple. In addition, slicing of tuples is allowed in Python wherein a part of the tuple can be extracted by specifying an index range along with the colon (:) operator, which itself results as tuple type.

■ **Example 6.20 — Indexing and Slicing.** An example to understand as indexing and slicing work in Python.

```
>>> holy_places = ("jerusalem", "bethlehem", "mahabodhi")
>>> holy_places
('jerusalem', 'bethlehem', 'mahabodhi')
>>> holy_places[0]
'jerusalem'
>>> holy_places[3]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> holy_places[-2]
'bethlehem'
>>> holy_places[1:3]
('bethlehem', 'mahabodhi')
```

■

There are many built-in functions for which a tuple can be passed as an argument (see Table 6.3).

■ **Example 6.21 — Built-in functions.** Examples to understand how to apply some built-in functions to the tuples.

Method	Description
`len()`	The **len()** function returns the numbers of items in a tuple.
`sum()`	The `sum()` function returns the sum of numbers in the tuple.
`sorted()`	The `sorted()` function returns a copy of the tuple as a list while leaving the original tuple untouched.

Table 6.3: Some built-in functions for tuples [GV18]

```
>>> years = (1987, 1985, 1981, 1996)
>>> len(years)
4
>>> sum(years)
7949
>>> sorted_years = sorted(years)
>>> sorted_years
[1981, 1985, 1987, 1996]
```

■

There are some methods associated with the tuples (see Table 6.4).

Method	Syntax	Description
`count()`	`tuple.count(item)`	The `count()` method counts the number of times the item has occurred in the tuple and returns it.
`index()`	`tuple.index(item)`	The `index()` method searches for the given item from the start of the tuple and returns its index. If the value appears more than once, you will get the index of the first one. If the item is not present in the tuple then `ValueError` is thrown by this method.

Table 6.4: Various tuple methods [GV18]

■ **Example 6.22 — Tuple methods.** Examples to understand how to apply some methods on the tuples.

```
>>> channels = ("ngc", "discovery", "animal_planet", "history", "ngc")
>>> channels.count("ngc")
2
>>> channels.index("history")
3
```

■

Exercise 6.11 — Tuple order. Display a tuple of integers entered by the user in ascending order.
■

6.3 Dictionaries

A dictionary is a collection of an unordered set of *key:value* pairs, with the requirement that the keys are unique within a dictionary. Dictionaries are constructed using curly braces , wherein you include a list of key:value pairs separated by commas. Also, there is a colon (:) separating each of these key and value pairs, where the words to the left of the colon operator are the keys and the words to the right of the colon operator are the values. Unlike lists, which are indexed by a range of numbers, dictionaries are indexed by keys. Here a key along with its associated value is called a *key:value* pair. Dictionary keys are case sensitive. You can create an empty dictionary by specifying a pair of curly braces and without any key:value pairs.

■ **Example 6.23 — Creating dictionaries.** Examples to understand how dictionaries are created in Python.

```
>>> fish = {"g": "goldfish", "s":"shark", "n": "needlefish", "b":"barramundi",
"m":"mackerel"}
>>> fish
{'g': 'goldfish', 's': 'shark', 'n': 'needlefish', 'b': 'barramundi',
'm': 'mackerel'}
>>> mixed_dict = {"portable":"laptop", 9:11, 7:"julius"}
 >>> mixed_dict
{'portable': 'laptop', 9: 11, 7: 'julius'}
>>> type(mixed_dict)
<class 'dict'>
>>> empty_dictionary = {}
>>> empty_dictionary
{}
>>> type(empty_dictionary)
<class 'dict'>
```

■

Each individual key:value pair in a dictionary can be accessed through keys by specifying it inside square brackets. The key provided within the square brackets indicates the key:value pair being accessed. Since dictionaries are mutable, you can add a new key:value pair or change the values for existing keys using the assignment operator.

■ **Example 6.24 — Accessing and modifying Dictionaries.** An example to understand how to access or modify elements in dictionaries.

```
>>> renaissance = {"giotto":1305, "donatello":1440, "michelangelo":1511,
"botticelli":1480, "clouet":1520}
>>> renaissance["giotto"] = 1310
>>> renaissance
{'giotto': 1310, 'donatello': 1440, 'michelangelo': 1511, 'botticelli': 1480,
'clouet': 1520}
>>> renaissance["michelangelo"]
1511
>>> renaissance["leonardo"] = 1503
>>> renaissance
```

```
{'giotto': 1310, 'donatello': 1440, 'michelangelo': 1511, 'botticelli': 1480,
'clouet': 1520, 'leonardo': 1503}

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

■

You can check for the presence of a key in the dictionary using `in` and `not in` membership operators.

■ **Example 6.25 — Operator in and Dictionaries.** An example to understand how to use the in operator with dictionaries.

```
>>> clothes = {"rainy":"raincoats", "summer":"tees", "winter":"sweaters"}
>>> "spring" in clothes
False
>>> "spring" not in clothes
True
```

■

There are many built-in functions for which a dictionary can be passed as an argument (see Table 6.5).

Method	Description
`len()`	The **len()** function returns the numbers of items (key:value pairs) in a dictionary.
`any()`	The `any()` function returns Boolean True value if any of the key in the dictionary is True else returns False.
`all()`	The `all()` function returns Boolean True value if all the keys in the dictionary are True else returns False.
`sorted()`	The `sorted()` function by default returns a list of items, which are sorted based on dictionary keys.

Table 6.5: Some built-in functions for dictionaries [GV18]

■ **Example 6.26 — Built-in functions.** Examples to understand how to apply some built-in functions to the dictionaries.

```
>>> presidents = {"washington":1732, "jefferson":1751, "lincoln":1809,
"roosevelt":1858, "eisenhower":1890}
>>> len(presidents)
5
>>> all_dict_func = {0:True, 2:False}
>>> all(all_dict_func)
False
>>> all_dict_func = {1:True, 2:False}
>>> all(all_dict_func)
```

```
True
>>> any_dict_func = {1:True, 2:False}
>>> any(any_dict_func)
True
>>> sorted(presidents)
['eisenhower', 'jefferson', 'lincoln', 'roosevelt', 'washington']
>>> sorted(presidents, reverse = True)
['washington', 'roosevelt', 'lincoln', 'jefferson', 'eisenhower']
>>> sorted(presidents.values())
[1732, 1751, 1809, 1858, 1890]
>>> sorted(presidents.items())
[('eisenhower', 1890), ('jefferson', 1751), ('lincoln', 1809),
('roosevelt', 1858),('washington', 1732)]
```

■

There are some methods associated with the dictionaries (see Table 6.6).

Method	Syntax	Description
`clear()`	`dict_name.clear()`	The `clear()` method removes all the key:value pairs from the dictionary
`get()`	`dict_name.get(key[, default])`	The `get()` method returns the value associated with the specified key in the dictionary. If the key is not present then it returns the default value. If default is not given, it defaults to `None`, so that this method never raises a `KeyError`.
`items()`	`dict_name.items()`	The `items()` method returns a new view of dictionary's key and value pairs as tuples.
`keys()`	`dict_name.keys()`	The `keys()` method returns a new view consisting of all the keys in the dictionary.
`pop()`	`dict_name.pop(key[, default])`	The `pop()` method removes the key from the dictionary and returns its value. If the key is not present, then it returns the default value. If default is not given and the key is not in the dictionary, then it results in `KeyError`.
`update()`	`dict_name.update([other])`	The `update()` method updates the dictionary with the key:value pairs from other dictionary object and it returns None.
`values()`	`dict_name.values()`	The `values()` method returns a new view consisting of all the values in the dictionary.

Table 6.6: Various dictionary methods [GV18]

■ **Example 6.27 — Dictionary methods.** Examples to understand how to apply some methods on the dictionaries.

```
>>> box_office_billion = {"avatar":2009, "titanic":1997, "starwars":2015,
```

```
"harrypotter": 2011, "avengers":2012}
>>> print(box_office_billion.get("frozen"))
None
>>> print(box_office_billion.get("avatar"))
2009
>>> box_office_billion.keys()
dict_keys(['avatar', 'titanic', 'starwars', 'harrypotter', 'avengers'])
>>> box_office_billion.values()
dict_values([2009, 1997, 2015, 2011, 2012])
>>> box_office_billion.items()
dict_items([('avatar', 2009), ('titanic', 1997), ('starwars', 2015),
('harrypotter', 2011),('avengers', 2012)])
>>> box_office_billion.update({"frozen":2013})
>>> box_office_billion
{'avatar': 2009, 'titanic': 1997, 'starwars': 2015, 'harrypotter': 2011,
'avengers': 2012, ' frozen': 2013}
>>> box_office_billion.pop("avatar")
2009
>>> box_office_billion.clear()
{}
```

■

Exercise 6.12 — Dictionary from user. Display a dictionary of N items entered from the user. N is an even number read from user (ask user an even number until the typed number is not an even number). ■

Exercise 6.13 — Occurrences. Write a program that reads a sentence and calculate the number of occurrences of the vocals. Use a dictionary to store the occurrences. ■

6.4 Chapter exercises

Exercise 6.14 — Prime number. A prime number is an integer greater than one that is only divisible by one and itself. Write a function that determines whether or not its parameter is prime, returning True if it is, and False otherwise. Write a main program that reads an integer from the user and displays a message indicating whether or not it is prime. ■

Exercise 6.15 — Perfect number. A perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself. Write a function that determines whether or not its parameter is a perfect number, returning True if it is, and False otherwise. Write a main program that reads a set of integer numbers until the user types the value -1. For each one of the typed numbers, the program displays a message indicating whether or not it is a perfect number. ■

Exercise 6.16 — Common items. Write a Python program that reads two lists A and B from the user. A has N items, whereas, B has M (N and M must be read from the user). Then, the program creates a new list that has only the elements in common between A and B without duplicates. ■

Exercise 6.17 — Matrix sum. Read two matrices, A and B, from the user (tip: to store a matrix as a list of lists). The dimension of the two matrices is NxN where N is read from the user. Finally, display the sum of the matrices A and B. ■

Exercise 6.18 — Anagrams. Determine whether or not two strings read from user are anagrams and report the result. ■

7. Object-Oriented Programming

This chapter deals with the basic concepts of the object-oriented programming in Python. In particular, Python provides full support for object-oriented programming, including encapsulation, inheritance, and polymorphism. Contents of this chapter are extracted in part by the following sources [GV18][Ste15][Van16].

7.1 Objects and Classes

The basic idea of Object-oriented programming (OOP) is that we use *objects* to model real-world things that we want to represent inside our programs and provide a simple way to access their functionality. Object-Oriented Programming is also an approach used for creating neat and reusable code instead of a redundant one. Indeed, it allows dividing the program into self-contained objects or several mini-programs. Every individual object represents a different part of the application having its own logic and data to communicate within themselves. In Python everything is an object and has a type, and you can:

- create new objects of some type;
- manipulate objects;
- destroy objects.

A class is a blueprint from which individual objects are created. An object is a bundle of related state (variables) and behavior (methods). Objects contain variables, which represents the state of information about the thing you are trying to model, and the methods represent the behavior or functionality that you want it to have.

It is important to make a distinction between *creating a class* and *using an instance of the class*.

7.1.1 Creating a Class

Classes are defined by using the `class` keyword, followed by the *ClassName* and a colon. The statements inside a class definition must be indent with respect to the `class` keyword. The statements inside a class definition are mainly related to:

- **data attributes**: think of data as other objects that make up the class. For example, a coordinate is made up of two numbers.
- **methods** (procedural attributes): think of methods as functions that only work with this class how to interact with the object. For example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects.

You can use a special method called **__init__** to initialize some data attributes. Besides the **__init__** method, it is possible to define any additional methods (like a function) that works only with that class. The first parameter in each of these methods is the word *self*. When *self* is used, it is just a variable name to which the object that was created based on a class is assigned. In the method definition, self does not need to be the only parameter and it can have multiple parameters.

■ **Example 7.1 — Class definition.** Creating the definition of a class used to represent a blueprint for coordinates.

```
class Coordinate():
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

■

7.1.2 Creating objects

Object refers to a particular *instance* of a class where the object contains variables and methods defined in the class. Hence, the act of creating an object from a class is called *instantiation*. In order to access to a particular variable or method belonging to the objects, you must use the "." operator.

■ **Example 7.2 — Creating objects.** Creating objects of the class Coordinate (see example 7.1) and accessing to data attributes and methods of these objects.

```
>>> c = Coordinate(3,4)
>>> origin = Coordinate(0,0)
>>> print(c.x)
3
>>> print(origin.x)
0
>>> print(c.distance(origin))
5
```

■

7.1.3 Special methods

A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. These methods start with double underscores and end with double underscores.

■ **Example 7.3 — Special methods.** When you print an object, Python calls the special method **__str__**. The print representation by default is uninformative.

```
>>> print(c)
<__main__.Coordinateobject at 0x7fa918510488
```

■

The special method can be overwritten. For example, the method **__str__** can be overwritten in order to modify the results of the call of the `print` function.

■ **Example 7.4 — Overwriting Special methods.** An example about how it is possible to overwrite the special method **__str__**.

```
>>> class Coordinate(object):
...     def init (self, x, y):
...         self.x = x
...         self.y = y
...     def __str__(self):
...         return "<"+str(self.x)+","+str(self.y)+">"
...
>>> p1= Coordinate(5,4)
>>> print (p1)
<5,4>
```

■

Exercise 7.1 — Arc Length. Write a Python program to calculate the arc length of an angle by assigning values to the radius and angle data Attributes of the class ArcLength. The formula to compute the arc lenght is $2 \cdot 3.141592653589793 \cdot radius \cdot angle/360$ ■

Exercise 7.2 — Bank Account. Write a Python program to simulate a bank account with support for deposit money, withdraw money and show balance operations. ■

7.2 Inheritance

Inheritance enables new classes to receive or inherit variables and methods of existing classes. Inheritance is a way to express a relationship between classes. If you want to build a new class, which is already similar to one that already exists, then instead of creating a new class from scratch you can reference the existing class and indicate what is different by overriding some of its behavior or by adding some new functionality. A class that is used as the basis for inheritance is called a *superclass* or *base class*. A class that inherits from a base class is called a *subclass* or *derived* class. The terms *parent* class and *child* class are also acceptable terms to use respectively. A derived class inherits variables and methods from its base class while adding additional variables and methods of its own. Inheritance easily enables reusing of existing code. To create a derived class, you add a *BaseClassName* after the *DerivedClassName* within the parenthesis followed by a colon.

■ **Example 7.5 — Class Inheritance.** An example of a superclass, namely the class `Person`, and a subclass, namely the class `UPerson`. The class `UPerson` will inherit all attributes and methods of

the parent class and can add new method (`getIdNum`) or overwrite existing methods (*__init__*) for adding new attributes (e.g., an `UPerson` has an idNum besides a name as in `Person`).

```
class Person(object):
    def __init__(self, name):
        """Create a person"""
        self.name = name
        try:
            lastBlank = name.rindex(' ')
            self.lastName = name[lastBlank + 1:]
        except:
            self.lastName = name
        self.birthday = None

    def getName(self):
        """Returns self's full name"""
        return self.name

    def getLastName(self):
        """Returns self's last name"""
        return self.lastName
        ...
    def __str__(self):
        """Returns self's"""
        return self.name

class UPerson(Person):
    nextIdNum = 0  # identification number

    def __init__(self, name):
        Person.__init__(self, name)
        self.idNum = UPerson.nextIdNum
        UPerson.nextIdNum += 1

    def getIdNum(self):
        return self.idNum
    ....
```

■

Exercise 7.3 — Polygon. A polygon is a closed figure with 3 or more sides. Define a class called *Polygon* that has data attributes to store the number of sides *n* and magnitude of each side as a list, *sides*. Tha class has two methods: the first one used to read the magnitude of the Polygon sides and the second one to display the magnitude of the Polygon sides. A triangle is a polygon with 3 sides. Define a class called *Triangle* which inherits from *Polygon* and add to the derived class a method to compute the perimeter length. ■

7.3 Exceptions

There are many things that can go wrong when a program is running. For example, the user can supply a non-numeric value when a numeric value was expected or the user can enter a value that causes the program to divide by 0. All of these errors are *exceptions*. By default, a Python program crashes when an exception occurs. However, we can prevent our program from crashing by catching the exception and taking appropriate actions to recover from it. The programmer must indicate: (i) where an exception might occur in order to catch it using a `try` block; (ii) what code to run to handle the exception when it occurs using one or more `except` blocks. When an exception occurs inside a `try` block, execution immediately jumps to the appropriate `except` block without running any remaining statements in the `try` block. Each `except` block can specify the particular exception that it catches. This is accomplished by including the exception's type immediately after the `except` keyword. Such a block only executes when an exception of the indicated type occurs. An `except` block that does not specify a particular exception will catch any type of exception (that is not caught by another `except` block associated to the same `try` block). If the `try` block executes without raising an exception then all of the `except` blocks are skipped and execution continues with the first line of code following the final `except` block.

Moreover, the `try...except` statement has an optional else block, which, when present, must follow all `except` blocks. It is useful for code that must be executed if the `try` block does not raise an exception. The `try` statement has another optional block which is intended to define clean-up actions that must be executed under all circumstances. A `finally` block is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` block and has not been handled by an except block, it is re-raised after the `finally` block has been executed.

■ **Example 7.6 — Zero division.** Program to check for `ZeroDivisionError` Exception.

```
x = int(input("Enter value for x: "))
y = int(input("Enter value for y: "))
try:
    result = x / y
except ZeroDivisionError:
    print("Division by zero!")
else:
    print("Result is", result)
finally:
    print("Executing finally clause")
```

■

Exercise 7.4 — Average Exception. Write a program which repeatedly reads numbers until the user enters the string 'done'. Once 'done' is entered, print out the total, count, and average of the numbers. If the user enters anything other than a number, detect his mistake using `try...except` statement and print an error message and skip to the next number. ■

We have seen how valuable it is to have informative exceptions when using parts of the Python language. It's equally valuable to make use of informative exceptions within the code you write, so that users of your code can figure out what caused their errors. The way you raise your own exceptions is with the `raise` statement.

■ **Example 7.7 — Fibonacci exception.** An example where we want to let the user know that a negative N is not supported to compute Fibonacci numbers.

```
>>> def fibonacci(N):
...     if N < 0:
...         raise ValueError("N must be non-negative")
...     L = []
...     a, b = 0, 1
...     while len(L) < N:
...         a, b = b, a + b
...         L.append(a)
...     return L
...
>>> fibonacci(10)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fibonacci(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in fibonacci
ValueError: N must be non-negative
```

■

Exercise 7.5 — Sum Raise Exception. Write a program which repeatedly reads integer numbers (if the typed value is not an integer, the program ignores and continues) from the user and raises a general exception (e.g, `Exception`) when the sum of the typed numbers is greater than the number 20. ■

7.4 Chapter exercises

Exercise 7.6 — Music. Write a Python program that includes three classes: Song, Artist and Album. Song attributes are: title, album, track_number. Artist attributes are: name and list of albums. Therefore, write also a method for the class Artist to add an album to the artist. Album attributes are: title, artist, year and list of songs. Therefore, write also a method for the class Album used to add a song to the album. Finally, write some code to test these classes. ■

Exercise 7.7 — Average Exception - 2. Write a program which repeatedly reads numbers until the user enters the string 'done'. Once 'done' is entered, print out the total, count, and average of the numbers. If the user enters anything other than a number, detect his mistake using `try...except` statement and correct the introduced values by setting them to the value 5 (so, the count must be incremented also if a not number is introduced). ■

8. Modules

This chapter deals with Python Modules, and in particular, with two modules very useful for scientific programming, namely `numpy` and `matplotlib`. The content of this chapter is extracted in part from the following sources [Hil16][Pin19][Mil20][Bha18][Hun19b].

8.1 Introduction to Modules

Large programs can consist of hundreds of functions that become difficult to manage and debug if they are all in one source file. By distributing the functions over several source files and grouping related functions together, it becomes easier to test and debug the various functions. To support this, Python has a way to put definitions in a file and use them in a python script or in an interactive instance of the interpreter. Such a file is called a **module**. Definitions from a module have to be *imported* into other modules or into the main module to be used.

Listing 8.1 — Importing a Module. A program gets access to a module through an `import` statement.

```
import module_name
```

Listing 8.2 — Use a Module. You can use a function within an imported module by prepending the name of the module to the function name with the `dot` operator:

```
import module_name
module_name.function_name(<parameter>)
```

Listing 8.3 — Keyword `as`. You can create an alias when you import a module, by using the `as` keyword. This is effectively importing the module in the same way that uses only `import` statement, with the only difference that, now, the module is available by means of the alias.

```
import module_name as alias_name
alias_name.function_name(<parameters>)
```

Listing 8.4 — Keyword `from`. You can choose to import only parts from a module, by using the `from` keyword. When importing using the `from` keyword, do not use the module name when referring to elements in the module.

```
from module_name import function_name
function_name(<parameters>)
```

Listing 8.5 — Operator `*`. You can choose to import all parts from a module, by using the `from` keyword e the operator . The use of this operator is not suggested because it can make some name clashes possible

```
from module_name import *
function_name(<parameters>)
```

8.2 `Numpy` module

NumPy has become the de facto standard package for general scientific programming in Python. Its core object is the `ndarray`, a multidimensional array of a single data type which can be sorted, reshaped, subject to mathematical operations and statistical analysis, written to and read from files, and much more. The NumPy implementations of these mathematical operations and algorithms have two main advantages:

- 1. implemented as precompiled C code (speed of execution)
- 2. NumPy supports vectorization: a single operation can be carried out on an entire array, rather than requiring an explicit loop over the array's elements (See Figure 8.1).

```
c = []
for i in range(n):
    c.append(a[i] * b[i])
```

```
c = a * b
```

Numpy

Figure 8.1: Vectorization idea

8.2.1 Basic array creation

NumPy is used to work with arrays. An array is similar to a list, but it can only contain elements of the same type. The array object in NumPy is called *ndarray*. We can create a NumPy *ndarray* object by using the `array()` function by giving in input a list or tuple of values.

```
import numpy as np
a = np.array( [100, 101, 102, 103], int )
b = np.array( [[1.,2.], [3.,4.]])
```

Note that passing a list of lists creates a two-dimensional array (and similarly for higher dimensions).

If you do not know the element values at the time of creation, there are several methods to create an array of a particular shape filled with default or arbitrary values. The simplest and fastest, `np.empty`, takes a tuple of the array's shape and creates the array without initializing its elements: the initial element values are undefined. There are also helper methods `np.zeros` and `np.ones`, which create an array of the specified shape with elements pre-filled with 0 and 1 respectively.

```
>>> a = np.empty((2,2), dtype=int)
>>> a
array([[1886272883, 1886217518,  841492601],
       [ 690565932,    7208970,    7733353]])
>>> s = np.zeros((3,2))
>>> s
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> t= np.ones((1,2))
>>> t
array([[1., 1.]])
```

8.2.2 Indexing a NumPy array

The manipulation of an array is similar to that seen for lists.

```
>>> import numpy as np
>>> a= np.array([1., 4., 5., 8.])
>>> a[: 2]
array ([1., 4.])
>>> a[3]
8.0
>>> a[0] = 5.
>>> a
array ([5., 4., 5., 8.])
```

Indexing a multidimensional NumPy array is a little different from indexing a conventional Python list of lists: instead of `b[i][j]`, refer to the index of the required element as a tuple of integers, `b[i,j]`.

```
>>> import numpy as np
>>> a= np.array([[1, 5, 6], [3, 6, 8], [5, 2, 7]])
>>> a
array([[1, 5, 6],
       [3, 6, 8]])
>>> a[1]
array([3, 6, 8])
>>> a[1,0]
3
>>> a[0:2]
array([[1, 5, 6],
       [3, 6, 8]])
```

8.2.3 Initializing an array

NumPy arrays can be initialized by using several functions:

The function **numpy.arange([start,]stop, [step,], dtype=None)** takes four parameters: the first three parameters determine the range of the values, while the fourth specifies the type of the elements. Precisely,

- start is the number (integer or decimal) that defines the first value in the array.
- stop is the number that defines the end of the array and isn't included in the array. step is the number that defines the spacing (difference) between each two consecutive values in the array and defaults to 1.
- dtype is the type of the elements of the output array and defaults to None.

```
>>> np.arange (5, dtype = float)
array ([0., 1., 2., 3., 4.])
```

The function numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None) takes six main parameters:

- start: The starting value of the sequence.
- stop: The end value of the sequence, unless endpoint is set to False. In that case, the sequence consists of all but the last of num + 1 evenly spaced samples, so that stop is excluded. Note that the step size changes when endpoint is False.
- num: Number of samples to generate. Default is 50. Must be non-negative.
- endpoint: If True, stop is the last sample. Otherwise, it is not included. Default is True.
- retstep: If True, return (samples, step), where step is the spacing between samples.
- dtype: The type of the output array. If dtype is not given, infer the data type from the other input arguments.

```
>>> np.linspace(1, 20, 5)
array [1. , 5.75, 10.5, 15.25, 20.]
```

The function numpy.identity(n, dtype=None) returns the identity array and takes two parameters:

- n: Number of rows (and columns) in n x n output.
- dtype: Data-type of the output. Default is float.

```
>>> np.identity (4, dtype = float)
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

Exercise 8.1 — Create arrays. Write a Python program to create:

- a 1D array of numbers from 0 to 9 (9 included);
- a 1D array of only odd numbers from 1 to 13 (13 included);
- a vector of size 10 with all zeros except for the sixth value that has to be 11;
- a 2d array 5×5 with 1 on the border and 0 inside.

■

Exercise 8.2 — comb. Create a "comb" of values in an array of length N for which every n-th element is one, but with zeros everywhere else. For example with $N = 101$ and $n = 5$, the resulting array will be: [1 0 0 0 0 1]

8.2.4 NumPy array attributes

A NumPy array is characterized by several attributes. Among these, there are:

- shape: returns the array dimensions along each axes
- ndim: returns the number of axes
- size: returns the total number of elements in the array
- dtype: returns the array data type
- T: returns the transposed array
- data: returns the buffer in memory containing the actual elements of the array
- itemsize: returns the size in byte of each element of the array
- nbytes: returns thr total bytes consumed by the elements of the array.

```
>>> a=np.arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.ndim
1
>>> a.size
6
>>> a.itemsize
4
```

8.2.5 Universal functions

In addition to the basic arithmetic operations of addition, division and more, NumPy provides many other mathematical functions. NumPy allows for vectorization: meaning that functions act on each element of an array and product an array in return without the need for an explicit loop.

```
>>> x = np.linspace(1,5,5)
>>> x
array([1., 2., 3., 4., 5.])
>>> x**2
array([ 1.,  4.,  9., 16., 25.])
>>> x - 1
array([0., 1., 2., 3., 4.])
>>> np.sqrt(x-1)
array([0., 1., 1.41421356, 1.73205081, 2.])
```

8.2.6 Numpy (dot) function

Array multiplication occurs element-wise. Matrix multiplication is implemented by NumPy's dot function (or using matrix objects)

```
>>> a = np.array( ((1,2), (3,4)) )
>>> a
array([[1, 2],
       [3, 4]])
>>> b=a
>>> b
array([[1, 2],
       [3, 4]])
>>> a*b
array([[ 1,  4],
       [ 9, 16]])
>>> np.dot(a,b)
array([[ 7, 10],
       [15, 22]])
```

8.2.7 Basic Arithmetic

The addition, subtraction, multiplication, division, elevation a power, are defined with the relative symbols

```
>>> a = np.array ([1,2,3], float)
>>> b = np.array ([5,2,6], float)
>>> a + b
array ([6., 4., 9.])
>>> a - b
array ([- 4., 0., -3.])
>>> a * b
array ([5., 4., 18.])
>>> b / a
array ([5 ., 1., 2.])
>>> a \% b
array ([1., 0., 3.])
>>> b ** a
array ([5., 4., 216.])
```

8.2.8 NumPy's special values, `nan` and `inf`

NumPy defines two special values to represent the outcome of calculations, which are not mathematically defined or not finite:

- np.nan ("not a number," NaN): represents the outcome of a calculation that is not a well-defined mathematical operation (e.g., 0/0)
- np.inf represents infinity

As an example:

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> np.divide(a,0)     # [0/0 1/0 2/0 3/0]
array([nan, inf, inf, inf])
```

Note: that (`np.nan == np.nan`) is `False`. Hence, do not test `nans` for equality!! Instead, NumPy provides methods `np.isnan()`, `np.isinf()` and `np.isfinite()`.

```
>>> b=np.array([np.inf, np.nan, np.inf])
>>> b
array([inf, nan, inf])
>>> np.isnan(b)
array([False,  True, False])
>>> np.isinf(b)
array([ True, False,  True])
>>> np.isfinite(b)
array([False, False, False])
```

Exercise 8.3 — Create arrays - 2. Write a Python program to:

- create an array of float numbers from 2.0 to 100.0 with step 5. Then, change the numbers contained in the even indices to infinite. Finally, test the created array for finiteness.
- create an array with N fives. N has to be read from user.
- create a MxM identity matrix. M has to be read from user.
- create an array of dimension N. The array has to contain the same value that is a random integer within the range [1,10]. N has to be read from user.

8.2.9 Copy an array

You can make an exact copy of an array using the `copy()` method:

```
>>> a = np.array ([1, 2, 3], float)
>>> b = a
>>> c = a.copy ()
>>> a [0] = 0
>>> a
array ([0., 2., 3.])
>>> b
array ([0., 2., 3.])
>>> c
array ([1., 2., 3.])
```

8.2.10 Changing the shape of an array

NumPy's arrays are stored in memory in C-style, row-major order, that is, with the elements of the last (rightmost) index stored contiguously. In a two-dimensional array, for example, the element a[0,0] is followed by a[0,1]. The array [[1 2], [3 4]] is stored in memory as the sequential elements [1,2,3,4]. Suppose you wish to "flatten" a multidimensional array onto a single axis. NumPy provides two methods to do this: `flatten` and `ravel`. Both flatten the array into its internal (row-major) ordering by respectively creating an independent copy and a copy of the original array.

```
>>> a = np.array( [[1,2,3], [4,5,6], [7,8,9]] )
>>> b = a.flatten()    # create flattened independent copy of a array([1,2,3,4,5,6,7,8,9])
```

```
>>> b[3] = 0            # array([1, 2, 3, 0, 5, 6, 7, 8, 9])
>>> a                   # a is unchanged
array([[1,2,3],[4,5,6], [7,8,9]])

>>> a = np.array( [[1,2,3], [4,5,6], [7,8,9]] )
>>> c = a.ravel()      # create flattened copy of a array([1,2,3,4,5,6,7,8,9])
>>> c[3] = 0            # array([1, 2, 3, 0, 5, 6, 7, 8, 9])
>>> a                   # a is changed
array([[1,2,3],[0,5,6], [7,8,9]])
```

An array may be resized (in place) to a compatible shape with the resize method, which takes the new dimensions as its arguments. If the array doesn't reference another array's data and doesn't have references to it:

- resizing to a smaller shape is allowed and truncates the array;
- resizing to a larger shape pads with zeros.

Array references are created when, for example, one array is a view on another (they share data) or simply by assignment: b=a.

```
>>> a = np.linspace(1, 4, 4) # the array [1. 2. 3. 4.]
>>> print(a)
[1. 2. 3. 4.]
>>> a.resize(2,2)              # reshapes a in place, doesn't return anything
>>> print(a)
[[ 1. 2.][ 3. 4.]]
>>> a.resize(3,2)                  # OK: nothing else references a
>>> print(a)
[[ 1. 2.][ 3. 4.][ 0. 0.]]
```

The size of an array can be changed via the `reshape` method. Please note: a new array is created:

```
>>> a = np.array(range (10), float)
>>> a
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> a = a.reshape((5, 2))
>>> a
array([[0., 1.],
      [2., 3.],
      [4., 5.],
      [6., 7.],
      [8., 9.]])
>>> a.shape
(5, 2)
```

8.2.11 Concatenate arrays

The `concatenate()` method allows the concatenation of two arrays. The concatenation takes place, by default, on the rows (axis parameter = 0)

```
>>> a = np.array ([[1, 2], [3, 4]], float)
>>> b = np.array ([[5, 6], [7,8]], float)
>>> np.concatenate ((a, b))
array ([[1., 2.],[3., 4.], [5., 6.], [7., 8.]])
```

Chaining can also be done for columns (axis parameter = 1)

```
>>> np.concatenate ((a, b), axis = 1)
array ([[1., 2., 5., 6.],
       [3., 4., 7., 8.]])
```

8.2.12 Constants and special symbols

Pi Greek:

```
>>> np.pi
3.1415926535897931
```

Euler constant:

```
>>> np.e
2.7182818284590451
```

8.2.13 Standard Functions

Numpy defines a whole series of standard functions: abs(), sign(), sqrt(), log(), log10(), exp(), sin(), cos(), tan(), arcsin(), arccos(), arctan(), sinh(), cosh(), tanh(), arcsinh(), arccosh(), arctanh()

```
>>> a = np.array([1, 4, 9], float)
>>> np.sqrt(a)
array([1., 2., 3.])
```

8.2.14 Merging and Splitting Arrays

There are 3 methods to merge and split arrays in different ways:

- np.vstack: stack arrays vertically (in sequential rows)
- np.hstack: stack arrays horizontally (in sequential columns)
- np.dstack: stack arrays depthwise (along a third axis)

Given these 3 arrays:

```
>>> a=np.array([0,0,0,0])
>>> b=np.array([1,1,1,1])
>>> c=np.array([2,2,2,2])
>>> np.dstack((a,b,c))
array([[[0, 1, 2],
              [0, 1, 2],
              [0, 1, 2],
              [0, 1, 2]]])
```

The inverse operations, np.vsplit, np.hsplit and np.dsplit split a single array into multiple arrays by rows, columns or depth. In addition to the array to be split, these methods require a argument indicating how to split the array. If this argument is a single integer, the array is split into that number of equal-sized arrays along the appropriate axis.

```
>>> a = np.arange(6)
>>> a
array([ 0, 1, 2, 3, 4, 5])
>>> np.hsplit(a, 3)
[array([ 0, 1]), array([ 2, 3]), array([ 4, 5])]
```

Exercise 8.4 — Create arrays - 3. Write a Python program to:

- create a 1D array with numbers from 4 to 10 and reshape it to a 2D array (with two rows).
- create a 3x3 matrix with values ranging from 2 to 10 included.
- create a NxM matrix containing only ones and prints its shape. N and M have to read fro user.
- create a 5X2 integer array from a range between 100 to 200 such that the difference between each element is 10

■

Exercise 8.5 — Add Row or Column. Suppose you have a 3x3 array to which you wish to add a row or column. Adding a row is easy with np.vstack. Adding a column requires a bit more work. You can't use np.hstack directly. In fact, if we type

```
a = np.ones((3, 3))
Inp.hstack( (a, np.array((2,2,2))) )
```

we receive a ValueError: all the input arrays must have same number of dimensions. This is because np.hstack cannot concatenate two arrays with different numbers of rows. Find a solution to solve this problem. ■

8.3 `Matplotlib` module

The graphical representation of data, namely plotting, is one of the most important tools for evaluating and understanding scientific data and theoretical predictions. Plotting is not a part of core Python, however, but is provided through one of several possible library modules. The most highly developed and widely used plotting package for Python is matplotlib (http://matplotlib.sourceforge.net/). It is a powerful and flexible program that has become the de facto standard for 2D plotting with Python. Because matplotlib is an external library, it must be imported into any routine that uses it. Therefore, for any program that is to produce 2D plots, you should include the line:

```
import matplotlib.pyplot as plt
```

8.3.1 An Interactive Session with PyPlot

We begin with an interactive plotting session that illustrates some very basic features of matplotlib.

■ **Example 8.1 — First plot.** In this example, the `plot()` function takes a sequence of values which will be treated as the y axis values; the x axis values are implied by the position of the y values within the list. Thus as the list has seven elements in it the x axis has the range 0–6. In turn as the maximum value contained in the list is 3.75, then the y axis ranges from 0 to 4.

```
>>> plt.plot([1, 0.25, 0.5, 2, 3, 3.75, 3.5])
[<matplotlib.lines.Line2D object at 0x000001FFE94FF448>]
>>> plt.show()
```

■

When you run this example in the interactive way, a window should appear with a plot that looks something like the interactive plot window shown in Fig. 8.2.

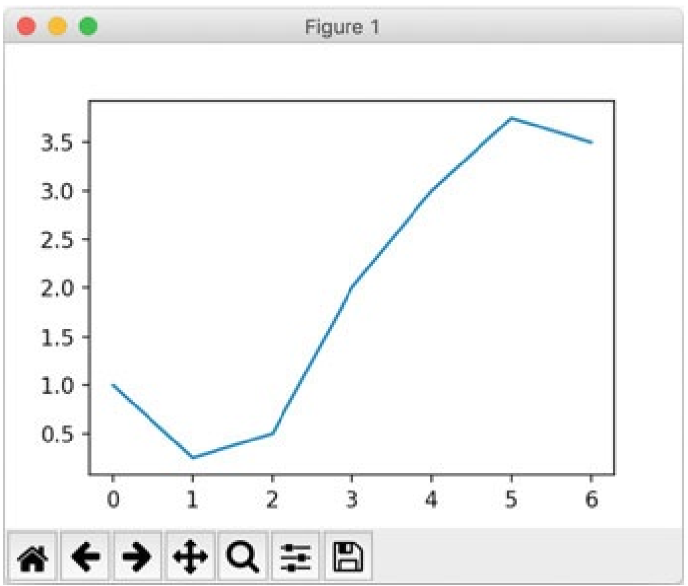


Figure 8.2: Interactive window

If you run this example by using PyCharm, a window showing the plot should appear integrated in the left side of the main window.

8.3.2 Basic plotting

Let's take a closer look at the `plot()` function. It is used to plot x-y data sets and is written like this plot(x, y) where x and y are arrays (or lists) that have the same size. If the x array is omitted, that is, if there is only a single array, as in our example above, the plot function uses 0, 1, ..., N-1 for the x array, where N is the size of the y array. As for the `show()` function, it displays the plot on the computer screen. No screen output is produced before this function is called. Moreover, you can save the plot in a file by using the function `savefig`.

■ **Example 8.2 — Plot sine function.** In this example, the `plot()` function is used to plot the sine function over the interval from 0 to 4π.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 4.*np.pi, 100)
y = np.sin(x)
plt.plot(x, y)
```

```
plt.savefig("plot_sine.png")
plt.show()
```

■

Exercise 8.6 — Plot cosine function. Write a program that plots the cosine function over the interval from $\frac{\pi}{2}$ to $\frac{5\pi}{2}$ by considering 50 points. Moreover, the program saves the plot in a file named "plot_cosine.png" ■

Exercise 8.7 — Plot function. Write a program that plots the function $y = 2x^2 - 3$ over the interval $[-5, 5]$ by considering 30 points. ■

8.3.3 Plot components

There are numerous elements that comprise a Matplotlib graph or plot. These elements can all be manipulated and modified independently. It is therefore useful to be familiar with the Matplotlib terminology associated with these elements, such as ticks, legends, labels etc. The elements that make up a plot are illustrated in Fig. 8.3

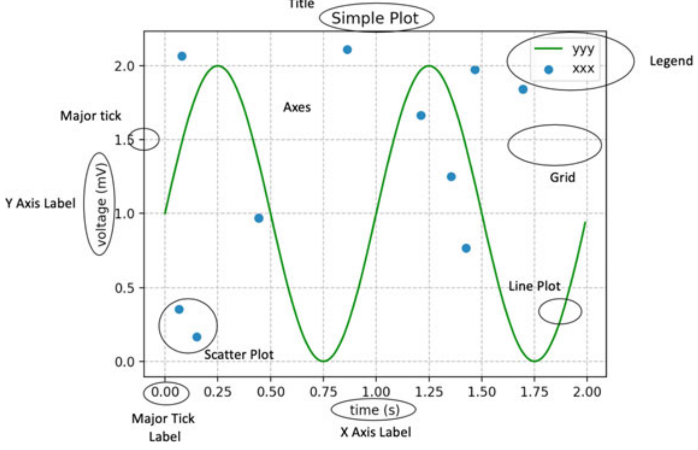


Figure 8.3: Diagram of plot components

In detail, Fig. 8.3 illustrates the following elements:

- **Axes**: An Axes is defined by the `matplotlib.axes.Axes` class. It is used to maintain most of the elements of a figure namely the X and Y Axis, the Ticks, the Line plots, any text and any polygon shapes.
- **Title**: This is the title of the whole figure.
- **Ticks** (Major and Minor): The Ticks are represented by the class `matplotlib.axis.Tick`. A Tick is the mark on the Axis indicating a new value. There can be Major ticks which are larger and may be labeled. There are also minor ticks which can be smaller (and may also be labelled).
- **Tick Labels** (Major and Minor): This is a label on a Tick.
- **Axis**: The `maplotlib.axis.Axis` class defines an Axis object (such as an X or Y axis) within a parent Axes instance. It can have formatters used to format the labels used for the major and minor ticks. It is also possible to set the locations of the major and minor ticks.

- **Axis Labels** (X, Y and in some cases Z): These are labels used to describe the Axis.
- **Plot**: types such as line and scatter plots. Various types of plots and graphs are supported by Matplotlib including line plots, scatter graphs, bar charts and pie charts.
- **Grid**: This is an optional grid displayed behind a plot, graph or chart. The grid can be displayed with a variety of different line styles (such as solid or dashed lines), colours and line widths.

8.3.4 Line graph

A line graph or line plot is a graph with the points on the graph (often referred to as markers) connected by lines to show how something changes in value as some set of values (typically the x axis) changes; for example, over a series to time intervals (also known as a time series).

■ **Example 8.3 — Speed vs Time plot.** In this example, the `plot()` function is used to plot time across the bottom (x axis) against speed (represented by the y axis) (see Fig. 8.4).

```
import matplotlib.pyplot as pyplot
# Set up the data
x = [0, 1, 2, 3, 4, 5, 6]
y = [0, 2, 6, 14, 30, 43, 75]
# Set the axes headings
pyplot.ylabel('Speed', fontsize=12)
pyplot.xlabel('Time', fontsize=12)
# Set the title
pyplot.title("Speed v Time")
# Plot and display the graph
# Using blue circles for markers ('bo')
# and a solid line ('-')
pyplot.plot(x, y, 'bo-')
pyplot.show()
```

■

In the example, it is possible to see that the `plot` function takes a third parameter, this is the string 'bo-'. This is a *coded format string* in that each element of the string is meaningful to the `plot` function. The elements of the string are:

- "b": this indicates the colour to use when drawing the line; in this case the letter 'b' indicates the colour blue;
- "o": this indicates that each marker (each point being plotted) should be represented by a cirlce. The lines between the markers then create the line plot.
- "-': This indicates the line style to use. A single dash ('-') indicates a solid line.

There are numerous options that can be provided via the format string, the following tables summarises some of these.

■ **Example 8.4 — Multiple lines.** When you want to add more lines to the chart, you simply apply as many `plot()` methods as necessary. In this example, the `plot()` function is used to plot two lines related to the actual and fake world population over time (see Fig. 8.5).

```
import matplotlib.pyplot as pyplot
# creating a line plot with multiple lines
```

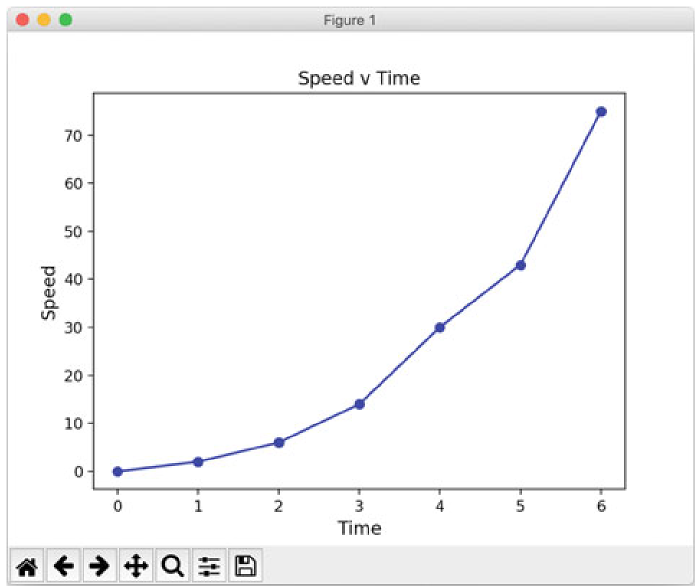


Figure 8.4: Line plot for speed over time

Character	Color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

```
x1, y1 = [ 1600, 1700, 1800, 1900, 2000 ] , [ 0.2, 0.5, 1.1, 2.2, 7.7 ]
x2, y2 = [ 1600, 1700, 1800, 1900, 2000 ] , [ 1, 1, 2, 3, 4 ]
plt.plot(x1, y1, "rx-", label="Actual") # create a red solid line
with x dots
plt.plot(x2, y2, "bo--", label="Fake") # create a blue dashed line
with circle dots
plt.title("World Population Over Time")
plt.xlabel("Year")
plt.ylabel("Population (billions)")
plt.legend( ) # shows labels in best corner
plt.show( )
```

■

Character	Description
‘.’	point marker
‘,’	pixel marker
‘o’	circle marker
‘v’	triangle_down marker
‘^’	triangle_up marker
‘ < ’	triangle_left marker
‘ > ’	triangle_right marker
‘s’	square marker
‘p’	pentagon marker
‘*’	star marker
‘h’	hexagon1 marker
‘ + ’	plus marker
‘x’	x marker
‘D’	diamond marker

Character	Description
‘-’	solid line style
‘–’	dashed line style
‘-.’	dash-dot line style
‘:’	dotted line style

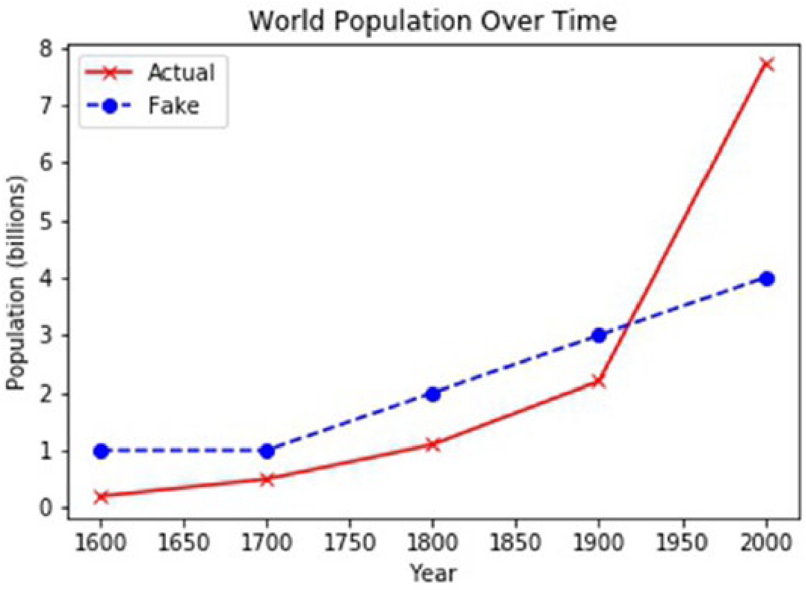


Figure 8.5: Actual and fake population distribution over time

Exercise 8.8 — Cosine vs Sine. Write a program that plots the sine function and the cosine function in the same graph. Both function have to be plotted in the interval $[0, 2\pi]$ by considering 100 points. The sine function has to be plotted in red with pixel marker and solid line style. The cosine function has to be plotted in green with star marker and dash-dot line style. Moreover, the program adds the legend to the graph. ■

8.3.5 Scatter Graph

A Scatter Graph or Scatter Plot is type of plot where individual values are indicated using cartesian (or x and y) coordinates to display values. Each value is indicated via a mark (such as a circle or triangle) on the graph. Matplotlib provides the `scatter()` function in order to display scatter plot. This function is characterized by different parameters such as the color and the marker style as well as the data. As for values for color and marker attributes, it is possible to use the same values of the function `plot()`. Moreover, it is possible to display on the same graph multiple group of data by calling different times the function `scatter()`.

■ **Example 8.5 — Scatter plot.** In this example, the `scatter()` function is used to create a scatter plot where height-weight distribution for twenty people is shown (see Fig. 8.6).

```
# creating a scatter plot to represent height-weight distribution
from random import randint
import matplotlib.pyplot as plt

height = [ randint(140, 180) for x in range(20) ]
weight = [ randint(70, 90) for x in range(20) ]
plt.scatter(weight, height, color='b', marker='o')
plt.title("Height-Weight Distribution")
plt.xlabel("Weight (kg)")
plt.ylabel("Height (cm)")
plt.show( )
```

■

Exercise 8.9 — Subject marks. Write a Python program to draw a scatter plot comparing two subject marks of Mathematics and Science. Use marks of 10 students as reported below. math_marks = [88, 92, 80, 89, 100, 80, 60, 100, 80, 34] science_marks = [35, 79, 79, 48, 100, 88, 32, 45, 20, 30] marks_range = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100] The resulting plot must be similar to the graph displayed in Fig. 8.7. ■

8.3.6 Bar graph

A Bar Chart is a type of chart or graph that is used to present different discrete categories of data. The data is usually presented vertically although in some cases horizontal bar charts may be used. Each category is represented by a bar whose height (or length) represents the data for that category. In order to create bar charts, you can use the function `bar` or the function `barh` for horizontal bars.

■ **Example 8.6 — Bar plot.** In this example, five categories of programming languages are presented along the x axis while the y axis indicates percentage usage. Each bar then represents the usage percentage associated with each programming language (see Fig. 8.8).

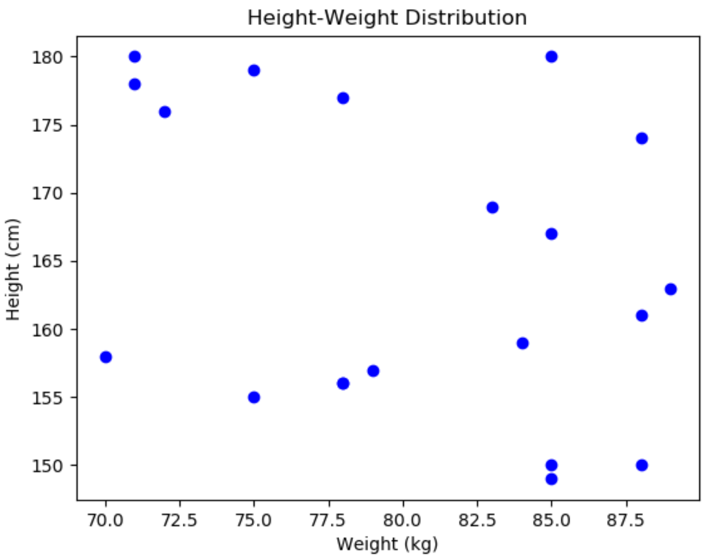


Figure 8.6: Scatter plot for height-weight distribution of 20 people

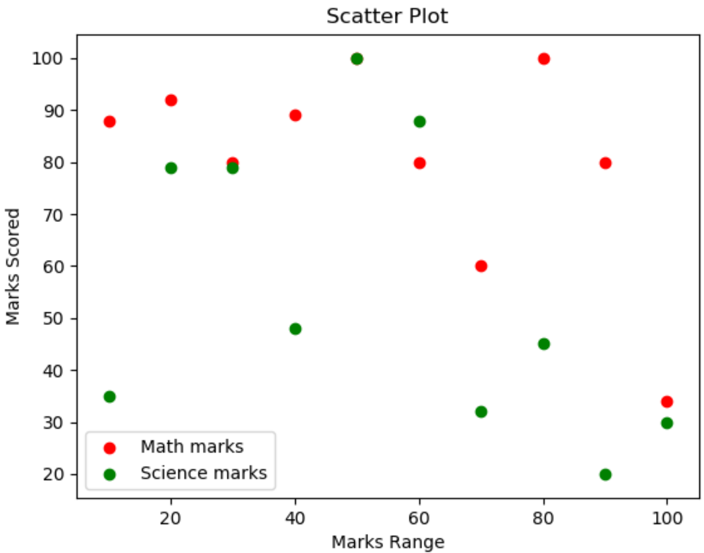


Figure 8.7: Resulting plot for exercise "Subject marks"

```
import matplotlib.pyplot as plt
# Set up the data
labels = ('Python', 'Scala', 'C#', 'Java', 'PHP')
index = (1, 2, 3, 4, 5) # provides locations on x axis
sizes = [45, 10, 15, 30, 22]
# Set up the bar chart
```

```
plt.bar(index, sizes, tick_label=labels)
# Configure the layout
plt.ylabel('Usage')
plt.xlabel('Programming Languages')
# Display the chart
plt.show()
```

■

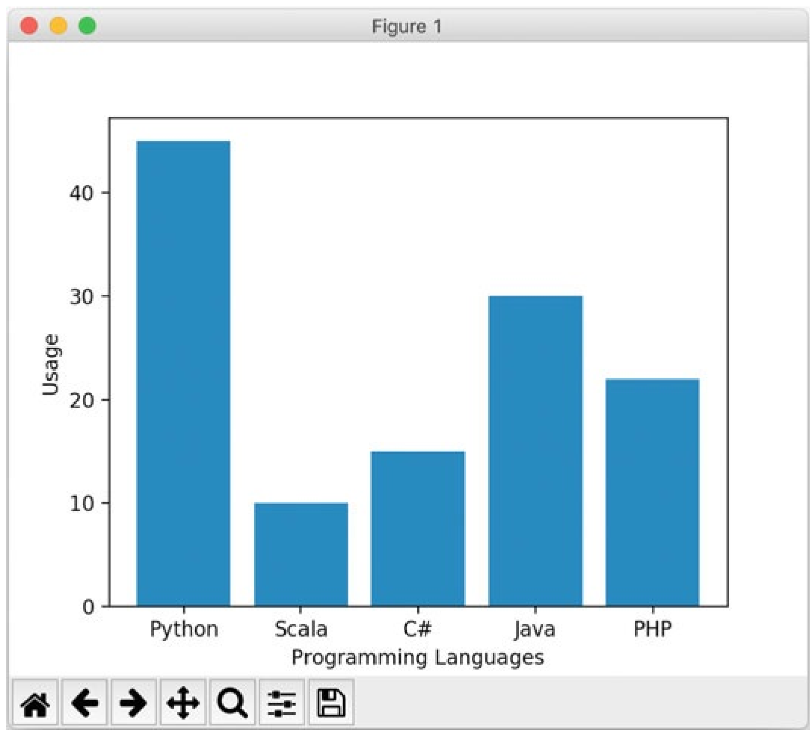


Figure 8.8: Bar plot for programming language usage

Moreover, it is possible to colour different bars in the chart in different colours. The colour to be used for each category can be provided via the color parameter to the `bar()` function. This is a sequence of the colours to apply. For example, we can modify the previous example in the following way:

```
plt.bar(index, sizes, tick_label=labels, color=('red',
'green', 'blue', 'yellow', 'orange'))
```

Exercise 8.10 — Plot letter bars. Write a Python program that ask user to insert a sentence. Then, the program computes the number of occurrences of the letters in the sentence. Finally, the program plots a bar graph by showing the number of occurrences of the letters of the sentence (i.e, the categories are the letters and the height of the bar corresponds to the occurrences). ■

9. Reading and Writing Files

This chapter deals with reading and writing files. The content of this chapter is extracted in part from the following sources [Ste15][GV18].

9.1 Reading and writing a file

Files are relatively permanent. The values stored in them are retained after a program completes and when the computer is turned off. This makes them suitable for storing results that are needed for an extended period of time, and for holding input values for a program that will be run several times.

9.1.1 Opening a file

A file must be opened before data values can be read from it or before new data values are written to it. Files are opened by calling the `open` function. The `open` function takes two arguments: 1) a string that contains the name of the file that will be opened; 2) a string that indicates the access mode for the file. There are different access modes as shown in Table 9.1. A file object is returned by the `open` function as shown below:

```
file_obj = open("input.txt", "r")
```

The file should be closed once all of the values have been read or written. This is accomplished by applying the `close` method to the file object as shown below:

```
file_obj.close()
```

9.1.2 Reading input from a file

There are several methods that can be applied to a file object to read data from a file. These methods can only be applied when the file has been opened in read mode. Attempting to read from a file that has been opened in write mode or append mode will cause your program to crash. The `readline` method reads one line from the file and returns it as a string, much like the input function reads a

Access mode	Description
“r”	Opens the file in read only mode and this is the default mode.
“w”	Opens the file for writing. If a file already exists, then it’ll get overwritten. If the file does not exist, then it creates a new file.
“a”	Opens the file for appending data at the end of the file automatically. If the file does not exist it creates a new file.
“r+”	Opens the file for both reading and writing.
“w+”	Opens the file for reading and writing. If the file does not exist it creates a new file. If a file already exists then it will get overwritten.
“a+”	Opens the file for reading and appending. If a file already exists, the data is appended. If the file does not exist it creates a new file.

Table 9.1: Various access modes [GV18]

line of text typed on the keyboard. Each subsequent call to `readline` reads another line from the file sequentially from the top of the file to the bottom of the file. The `readline` method returns an empty string when there is no further data to read from the file.

Example 9.1 — Read numbers from a file - version 1. Consider a data file that contains a long list of numbers, each of which appears on its own line. The following program computes the total of all of the numbers in such a file.

```python
# Read the file name from the user and open the file
fname = "data.txt"
inf = open(fname, "r")
# Initialize the total
total = 0
# Total the values in the file
line = inf.readline()
while line != "":
    total = total + float(line)
    line = inf.readline()
# Close the file
inf.close()
# Display the result
print("The total of the values in", fname, "is", total)
```

■

Sometimes it is helpful to read all of the data from a file at once instead of reading it one line at a time. This can be accomplished using either the `read` method or the `readlines` method. The `read` method returns the entire contents of the file as one (potentially very long) string. Then further processing is typically performed to break the string into smaller pieces. The `readlines` method

returns a list where each element is one line from the file. Once all of the lines are read with readlines a loop can be used to process each element in the list.

■ **Example 9.2 — Read numbers from a file - version 2.** Consider a data file that contains a long list of numbers, each of which appears on its own line. The following program computes the total of all of the numbers in such a file.

```
# Read the file name from the user and open the file
fname = "data.txt"
inf = open(fname, "r")
# Initialize total and read all of the lines from the file
total = 0
lines = inf.readlines()
# Total the values in the file
for line in lines:
    total = total + float(line)
# Close the file
inf.close()
# Display the result
print("The total of the values in", fname, "is", total)
```

■

9.1.3 Writing an output to a file

When a file is opened in write mode, a new empty file is created. If the file already exists then the existing file is destroyed and any data that it contained is lost. Opening a file that already exists in append mode will cause any data written to the file to be added to the end of it. If a file opened in append mode does not exist then a new empty file is created.

The `write` method can be used to write data to a file opened in either write mode or append mode. It takes one argument, which must be a string, that will be written to the file. Values of other types can be converted to a string by calling the `str` function. Multiple values can be written to the file by concatenating all of the items into one longer string, or by calling the write method multiple times.

Unlike the `print` function, the write method does not automatically move to the next line after writing a value. As a result, one has to explicitly write an end of line marker to the file between values that are to reside on different lines. Python uses to denote the end of line marker. Therefore, when you read a line and you want to print it, you can remove the end of line marker by using the `rstrip` method on the line object.

■ **Example 9.3 — Write numbers to a file.** The following program writes the numbers from 1 up to (and including) a number entered by the user to a file. String concatenation and the escape sequence are used so that each number is written on its own line.

```
# Read the file name from the user and open the file
fname = "data.txt"
outf = open(fname, "w")
# Read the maximum value that will be written
limit = int(input("What is the maximum value? "))
```

```
# Write the numbers to the file with one number on each line
for num in range(1, limit + 1):
    outf.write(str(num) + "\n")
# Close the file
outf.close()
```

■

Exercise 9.1 — Write sentences. Write a Python program that creates a file named "sentences.txt" and writes to it ten sentences, one per line. The sentences must be typed from the user. ■

Exercise 9.2 — Display the Head of a File. Write a Python program that reads the file "sentences.txt" and displays to the user the head (the first 5 lines) of the file. ■

Exercise 9.3 — Remove comments. Write a Python program that removes all of the comments from a file containing the code of the Example 10.1. The new code must be written in a new file named "new_code.txt". ■

Exercise 9.4 — Plot coordinates from a file. Write a Python program that reads the x and y coordinates contained in the file named "coordinates.txt" (x is the first column and y the second one in the file) and plots them with a red line. ■

Exercise 9.5 — Modify coordinates from a file. Write a Python program that reads the x and y coordinates contained in the file named "coordinates.txt" (x is the first column and y the second one in the file) and applies the function named "add_2" on the y coordinates. The function "add_2" simply adds the value 2 to every y coordinates. Then, the program writes a new file named "new_coordinates.txt" with x and y coordinates updated. ■

9.2 Reading and writing a NumPy array to a file

Scientific data are frequently read in from a text file. Columns of values may be either aligned in a fixed-width format or separated by one or more delimiting characters (such as spaces, tabs or commas). Furthermore, there may be a descriptive header and even footnotes to the file, which make it hard to parse directly using Python's string methods. `Numpy` provides several functions to manage files. The function `loadtxt` is used to load data from a file in a multi-dimensional `ndarray` object. The main arguments of this function are:

- fname: The only required argument and represents the name of the file contained data;
- dtype: The data type of the array. By default, the function uses the float type;
- comments: Comments in a file are usually started by some character such as # (as with Python) or %. To tell NumPy to ignore the contents of any line following this character, use the comments argument – by default it is set to #;
- delimiter: The string used to separate columns of data in the file; by default it is None, meaning that any amount of whitespace (spaces, tabs) delimits the data. To read a comma-separated (csv) file, set delimiter=','.

Moreover, it is possible to save a `ndarray` object to a file by using the `savetxt` function.

■ **Example 9.4 — Read and write coordinates from and to a file.** The following program reads the x and y coordinates from the file "coordinates.txt", creates a `ndarray` object. Then, it writes the same coordinates in a second file named "new_file.txt".

```
import numpy
filename = "coordinates.txt"
data = numpy.loadtxt(filename)
x = data[:,0]
y = data[:,1]
filename = "new_file.txt"
outfile = open(filename, "w")
# open file for writing
numpy.savetxt(outfile, data, fmt="%.2f")
```

■

Exercise 9.6 — Write function values to a file. Write a Python program that stores x values and the corresponding f(x) values in a file (x values in the first column and f(x) values in the second one). To create the x values, choose N values equally spaced in the interval [a,b]. N,a,b must be typed from the user. The function f is the logarithm function provided by NumPy. ■

Exercise 9.7 — Scatter plot from a file. Write a Python program that reads data from the file named "numpy_data.txt". This file contains four columns. The first and the second columns represent x and y values for a group of points, whereas, the third and the fourth columns represent x and y values for another group of points. After reading data in the file, plot the points of the first group as red points and the points of the second group as green stars. ■

Bibliography

[17] *BDM's: The Python Manual*. Papercut Limited, 2017. ISBN: 978-1-907306-88-4 (cited on page 33).

[Bha18] Harsh Bhasin. *Python Basics: A Self-teaching Introduction*. Stylus Publishing, LLC, 2018 (cited on page 109).

[Ced10] Vern Ceder. *The Quick Python Book, Second Edition*. 2nd. USA: Manning Publications Co., 2010. ISBN: 193518220X (cited on pages 27, 85).

[GV18] S. Gowrishankar and A. Veena. *Introduction to Python Programming*. 1st. Chapman Hall/CRC, 2018. ISBN: 0815394373 (cited on pages 27, 63, 75, 85, 91, 93, 96, 98, 99, 103, 127, 128).

[Hil16] Christian Hill. *Learning scientific programming with python*. Cambridge University Press, 2016 (cited on pages 85, 109).

[Hun19a] John Hunt. *A Beginners Guide to Python 3 Programming*. 1st. Springer Publishing Company, Incorporated, 2019. ISBN: 3030202895 (cited on pages 27, 30).

[Hun19b] John Hunt. *Advanced Guide to Python 3 Programming*. Springer, 2019 (cited on page 109).

[LL16] Svein Linge and Hans Petter Langtangen. "Programming for Computations-A Gentle Introduction to Numerical Simulations with Python". In: (2016) (cited on page 85).

[Mil20] Connor P Milliken. "Python Projects for Beginners". In: (2020) (cited on page 109).

[Mue18] John Paul Mueller. *Beginning Programming with Python For Dummies*. 2nd. For Dummies, 2018. ISBN: 1119457890 (cited on page 33).

[Pin19] David J Pine. *Introduction to Python for science and engineering*. CRC Press, 2019 (cited on page 109).

[Ste15] Ben Stephenson. *The Python Workbook: A Brief Introduction with Exercises and Solutions*. Springer Publishing Company, Incorporated, 2015. ISBN: 3319142399 (cited on pages 27, 63, 75, 85, 103, 127).

[Van16] Jacob T Vanderplas. *A Whirlwind Tour of Python*. O'Reilly Media, 2016 (cited on page 103).