

# PYTHON

Master Python OOP  
Programming  
with One Guide Only!

A Lot of Coding, Practice and Theory



# **PYTHON – Master Python OOP Programming with One Guide Only!**

**A lot of Coding, Practice, and Theory  
Learn Python with Hands-On Projects**

**Rick Sekulsoki**

Copyright © 2022 Rick Sekuloski  
All rights reserved.  
ISBN:

## PREFACE

## WHO IS THIS BOOK FOR?

## HOW TO GET THE MOST OUT OF THIS BOOK?

## DOWNLOAD THE FILES

## CHAPTER 1 - PYTHON BASICS

## WHAT IS PYTHON?

## WHY IS PYTHON SO POPULAR?

## PYTHON INTERPRETER

## WHY SO MANY PROGRAMMING LANGUAGES?

## GETTING STARTED – HOW TO RUN PYTHON CODE

**WRITE OUR FIRST PYTHON CODE**

**VARIABLES IN PYTHON**

**COMMENTS IN PYTHON**

**DATA TYPES**

**INT – INTEGER (DATA TYPE)**

**FLOAT– FLOATING POINT NUMBER (DATA TYPE)**

**HOW INTEGER AND FLOAT ARE STORED**

**PYTHON POWER OPERATOR (\*\*)**

**PYTHON DOUBLE SLASH (//) OPERATOR: FLOOR DIVISION**

**PYTHON MODULO OPERATOR (%)**

**MATH FUNCTIONS ON INT AND FLOAT DATA TYPES**

**ROUND()**

**ABS()**

## **OPERATOR PRECEDENCE**

### **ORDER OF PRECEDENCE:**

### **RULES FOR NAMING A VARIABLE**

### **REASSIGN VARIABLES TO A NEW VALUE**

### **CONSTANTS IN PYTHON**

### **DUNDER VARIABLES OR DUNDER METHODS**

### **EXPRESSIONS AND STATEMENTS IN PYTHON**

### **AUGMENTED ASSIGNMENT OPERATOR**

### **STR – STRING (DATA TYPE)**

### **STRING CONCATENATION**

### **TYPE CONVERSION OR TYPE CASTING**

### **IMPLICIT TYPE CONVERSION**

### **ESCAPE SEQUENCE**

### **FORMATTED STRINGS**

**HOW STRINGS ARE STORED?**

**SLICING STRINGS – GET A SUBSTRING OF A STRING**

**STRING-SLICING WITH STEPOVER**

**STRING IMMUTABILITY**

**PRACTICE TIME**

**FUNCTIONS AND METHODS – WHAT DIFFERENTIATES THEM?**

**LEN()**

**UPPER()**

**CAPITALIZE()**

**LOWER()**

**FIND()**

**REPLACE()**

**JOIN()**

**PYTHON IN KEYWORD**

**BOOLEANS IN PYTHON (DATA TYPE)**

**LISTS IN PYTHON (DATA TYPE)**

**LIST SLICING**

**PYTHON IN KEYWORD WITH LISTS**

**MULTI-DIMENSIONAL LISTS**

**LISTS METHODS**

**APPEND() METHOD**

**INSERT() METHOD**

**EXTEND() METHOD**

**POP() METHOD**

**REMOVE() METHOD**

**CLEAR() METHOD**

**INDEX() METHOD**

**COUNT() METHOD**

SORT() METHOD

COPY() METHOD

REVERSE() METHOD

USEFUL TIPS AND TRICKS USED ON LISTS

COPY OF THE LIST USING THE SLICING METHOD

GENERATE NEW LIST USING RANGE() FUNCTION

.JOIN()

LIST UNPACKING

NONE IN PYTHON (DATA TYPE)

DICTIONARY IN PYTHON (DATA STRUCTURE)

DICTIONARY – METHODS

GET() METHOD

KEYS()

VALUES() METHOD

ITEMS() METHOD

CLEAR()

COPY()

POP()

POPITEM()

UPDATE()

SETDEFAULT() METHOD

TUPLES IN PYTHON (DATA TYPE)

TUPLE METHODS

COUNT()

INDEX()

SETS IN PYTHON (DATA TYPE AND DATA STRUCTURE)

ACCESSING ITEMS IN SETS

CHECK THE SET LENGTH

CREATE A SET FROM A LIST

CONVERT A SET TO A LIST

PYTHON SET METHODS

COPY() METHOD

CLEAR() METHOD

DIFFERENCE() METHOD

DIFFERENCE\_UPDATE() METHOD

DISCARD() METHOD

INTERSECTION() METHOD

INTERSECTION\_UPDATE() METHOD

ISDISJOINT() METHOD

UNION()

ISSUBSET()

ISSUPERSET()

## SUMMARY

## CHAPTER 2 – PYTHON CONDITIONAL, LOGICAL OPERATORS AND LOOPS

### CONTROL STRUCTURES

#### IF-STATEMENT

#### IF-ELSE STATEMENT

#### ELIF

### UNDERSTANDING PYTHON INDENTATION

### PYTHON OPERATORS

#### ARITHMETIC OPERATORS

#### ASSIGNMENT OPERATORS

##### ASSIGN OPERATOR ‘=’

##### ADD AND ASSIGN +=

##### SUBTRACT AND ASSIGN -=

##### MULTIPLY AND ASSIGN \*

DIVIDE AND ASSIGN /=

MODULUS AND ASSIGN %=

DIVIDE(FLOOR) AND ASSIGN //=

EXPONENT AND ASSIGN \*\*=

BITWISE AND &=

BITWISE OR |=

BITWISE XOR ^=

BITWISE RIGHT SHIFT AND ASSIGN

BITWISE LEFT SHIFT AND ASSIGN

COMPARISON OPERATORS

LOGICAL OPERATORS

TRUTHY VS FALSY VALUES

BOOLEAN CONTEXT

THE BUILT-IN BOOL() FUNCTION

TERNARY OPERATOR

SHORT-CIRCUITING

PYTHON IDENTITY OPERATORS

THE 'IS' OPERATOR VERSUS THE '=='

MEMBERSHIP OPERATORS

PYTHON LOOPS

NESTING LOOPS

WHILE LOOP

ITERABLES

ITEMS() METHOD

VALUES() METHOD

KEYS() METHOD

THE BUILT-IN RANGE() FUNCTION

ENUMERATE 0.

BREAK, CONTINUE AND PASS STATEMENTS

PRACTICE TIME

HERE IS MY SOLUTION (CODE ONLY)

PYTHON FUNCTIONS

FUNCTION ARGUMENTS AND PARAMETERS

POSITIONAL AND KEYWORD ARGUMENTS

DEFAULT ARGUMENTS AND PARAMETERS

FUNCTION RETURN

NESTED FUNCTIONS

DOCSTRINGS IN PYTHON

PYTHON \*ARGS AND \*\*KWARGS

SCOPE

MORE ABOUT SCOPES

## PYTHON NONLOCAL KEYWORD

## CHAPTER 3 PYTHON INSTALLATION

### PYTHON INSTALLATION

#### MACOS PYTHON INSTALLATION

#### WINDOWS OS PYTHON INSTALLATION

#### LINUX USER ONLY - PYTHON INSTALLATION

### DEVELOPER TOOLS

#### EXPLORE THE VISUAL STUDIO CODE

#### USEFUL TERMINAL/COMMAND PROMPT COMMANDS

### CODE FORMATTING

#### WHAT IS PEP?

#### PYCHARM INSTALLATION

#### WHAT ABOUT CODE FORMATTING IN PYCHARM?

### SUMMARY

## CHAPTER 4 – OBJECT ORIENTED PROGRAMMING – ADVANCED PYTHON

### CLASS KEYWORD

### WHY OBJECT-ORIENTED PROGRAMMING?

### THE PRINCIPLES OF OBJECT-ORIENTED PROGRAMMING

### WHAT OTHER LANGUAGES SUPPORT OBJECT ORIENTED PROGRAMMING?

### HOW CAN WE CREATE CLASSES IN PYTHON?

### WHAT ARE CLASSES AND OBJECTS IN PYTHON?

### CLASS CONSTRUCTOR, ATTRIBUTES, METHODS

### CLASS OBJECT ATTRIBUTE

#### INIT CONSTRUCTOR

#### CONSTRUCTOR WITH DEFAULT VALUES

#### CREATE CLASS METHODS USING @CLASSMETHOD

#### STATIC METHOD @STATICMETHOD

**1<sup>ST</sup> PILLAR OF OOP - ENCAPSULATION**

**2<sup>ST</sup> PILLAR OF OOP – ABSTRACTION**

**PYTHON PRIVATE VS PUBLIC VARIABLES**

**3<sup>RD</sup> PILLAR OF OOP – INHERITANCE**

**METHOD OVERRIDING**

**ISINSTANCE() FUNCTION – PYTHON**

**4<sup>TH</sup> PILLAR OF OOP – POLYMORPHISM**

**SUPER() FUNCTION**

**CODE INTROSPECTION IN PYTHON**

**ID0.**

**DUNDER/MAGIC METHODS**

**MULTIPLE INHERITANCE**

**MRO – METHOD RESOLUTION ORDER**

**SUMMARY**

## CHAPTER 5- PHYTON – MODULES AND PACKAGES

### WHY WE NEED PYTHON MODULES?

### PYTHON PACKAGES

### IMPORT MODULES USING FROM-IMPORT SYNTAX

### IMPORT MODULES USING FROM-IMPORT \* SYNTAX

### PYTHON MODULE ALIAS

### PYTHON \_\_NAME\_\_

### PYTHON BUILT-IN MODULES

### THE BIGGEST REASON WHY PYTHON IS A GREAT PROGRAMMING LANGUAGE

### INSTALL PYTHON PACKAGES USING PYPI REPOSITORY

### SUMMARY

## CHAPTER 6 – WORKING WITH FILES IN PYTHON

### OPEN

### WRITE AND APPEND TO FILES

**WRITE TO A FILE**

**APPEND**

**FILE PATHS**

**TRY – EXCEPT BLOCK FOR ERROR HANDLING**

**SUMMARY**

**CHAPTER 7 – ERROR HANDLING**

**ERRORS IN PYTHON**

**ERROR HANDLING**

**EXCEPT BLOCK**

**FINALLY BLOCK**

**RAISE AN EXCEPTION**

**SUMMARY**

**PROJECTS**

## PROJECT\_1: GUESS THE RANDOM NUMBER

### CODING HINTS

## PROJECT\_2: TEACHER, STUDENT APP

### LET'S START WITH REMOVING AND RENAMING FILES

### PASSWORD HASHING

### DATETIME MODULE

### SCREENSHOTS AND FILES

### STARTER CODE WITH COMMENTS ON WHAT CODE YOU SHOULD ADD

#### MAIN.PY FILE:

#### LOGIN.PY FILE:

#### REGISTER.PY

#### STUDENT\_TASKS.PY

#### APPENDIX A – PROJECT

#### SOLUTIONS

**PROJECT\_1: GUESS THE RANDOM NUMBER SOLUTION:**

**PROJECT\_2: TEACHER, STUDENT APP SOLUTION**

**APPENDIX B: MY BESTSELLER BOOKS**

**APPENDIX C: MORE EXERCISES AND LEARN MORE ABOUT  
JAVASCRIPT, HTML, AND PHP**

**APPENDIX D: RESOURCES**

## Preface

This book is a perfect guide for someone that wants to learn Python. You will find that this book explains complex concepts in an easy-to-understand manner. After covering the basics, we will shift our focus to some intermediate features that every Python programmer should know. I'm not a professional writer but I'm a professional web developer and programmer, therefore my goal is to teach you programming through perfect examples and clean and executable code. When I first started learning Python, I was afraid of how hard this language could be, but I can assure you that Python, like any other programming language, only requires patience, practice, and of course a good guide with practical examples. The goal of my book is to help you bring your current skills to a professional level. If you are looking for additional reference materials then I can only suggest the Python documentation. If you have already looked at the Python documentation then you know it can be long, scary, and overwhelming. Do not worry because this book will save you months of researching, planning, and sleepless nights.

You can visit Python's official websites by clicking on following link:

<https://docs.python.org/3/>

I would also like to hear from you. If you need to contact me, please reach out through any of my social media accounts, and please consider leaving a review with your comment.

My social media accounts:

- Twitter: <https://twitter.com/Rick29702077?s=09>
- LinkedIn: <https://www.linkedin.com/in/rick-sekuloski>
- Facebook:  
<https://www.facebook.com/theodorecodingwebdevelopment/>

- YouTube: <https://www.youtube.com/channel/UCQanUcCNaBg-IM-k0u8z0oQ>

## Who is this book for?

This book is for:

- **Beginners** – If you are a beginner, this book will definitely help you to learn Python
- **Teachers/Educators** – Are you someone that teaches Python? Then this book is an organized guide that will help you and your students learn the new concepts through practical examples and theory
- **Developers** – Are you a developer but need to refresh your memory? This is the best reference guide you will find
- **Anyone** - that is seeking to gain a deep understanding of the Python language

## How to Get the most out of this book?

To get the most out of this book, you will need the following tools:

- Access to internet
- Any of the following: Computer, Laptop, tablet, phone, or an eBook Reader (kindle)
- An up-to-date browser such as Google Chrome, Firefox, Edge, or Safari
- I will use a combination of code editors and IDEs, but you can decide which tool is the most suitable for your needs

If you are using an e-reader to read this book, then please sit back, get comfortable, and enjoy because I will explain every step and include the code and screenshots. But if you like to go through the examples and try to run the code, then please use a computer or laptop. The book will be divided into chapters, and in each chapter will be a new theory coupled with

examples, output, and screenshots. The first few chapters are easy but at the same time, they are the most important ones.

## Download The files

You can download the entire code by visiting my GitHub (*source code management system*) repository page using the following link:

[https://github.com/RickSekuloski/python\\_book1](https://github.com/RickSekuloski/python_book1)

There you will find all the materials (code examples and exercises) that I will use in this book.

Once the file is downloaded to your computer, you will need to unzip the content. Please don't try to open or run the code while it's inside the zipped folders/directories. Ensure you extract the folder into your desired destination, such as the desktop.

You can Unzip the files using the following programs:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux



# **Chapter 1 - Python Basics**

Welcome to the first chapter. This chapter is by far the most important one. Here, the idea is to try to understand the basic Python features because without them you will have a hard time understanding the concepts in the following chapters where more intermediate features will be introduced. In this chapter, we will start by covering the basics of Python, like what Python is and how we can run Python code. Although it is called Python basics, there will be some features that are not that simple to understand, therefore, please make sure you spend enough time learning the concepts before moving to the next section.

## **What is Python?**

According to official documentation, Python is ‘An interpreted, object-oriented, high-level programming language with dynamic semantics. Python is a general computer programming language, meaning it can be used to create applications that will solve different problems. Python can be used to create websites, software, or to even perform data analysis. Learning Python is simple because it uses simple syntax, therefore, even people that are new to programming can learn it. Python emphasizes code readability and this can be achieved because Python supports modules and packages. In simple words, this means that instead of having one large file with thousands of lines of code, we can have multiple files that can communicate and share the code with the main file. This reduces the program maintenance cost and because Python is free to use, it is a perfect programming language. The Python programming language was created by Guido van Rossum and it was first released on February 20, 1991.

## **Why is Python so popular?**

Python is a widely used high-level programming language because the code is written in an English-like format or if you look at the code closely, you will see that is written similarly to the way humans think. Some may argue that Python is slower than other high-level languages like **C**, **C++**, or **Java** but this does not affect the end users. Another big advantage is that Python is more productive compared to **Java** and **C++** because building an application takes less time, effort, and fewer lines of code. Because of its popularity, the Python community is exceptional and large. Having a large community is very important because Python allows us to create packages and modules that can be shared with the rest of the developers. This makes Python a flexible language with different real-world applications like:

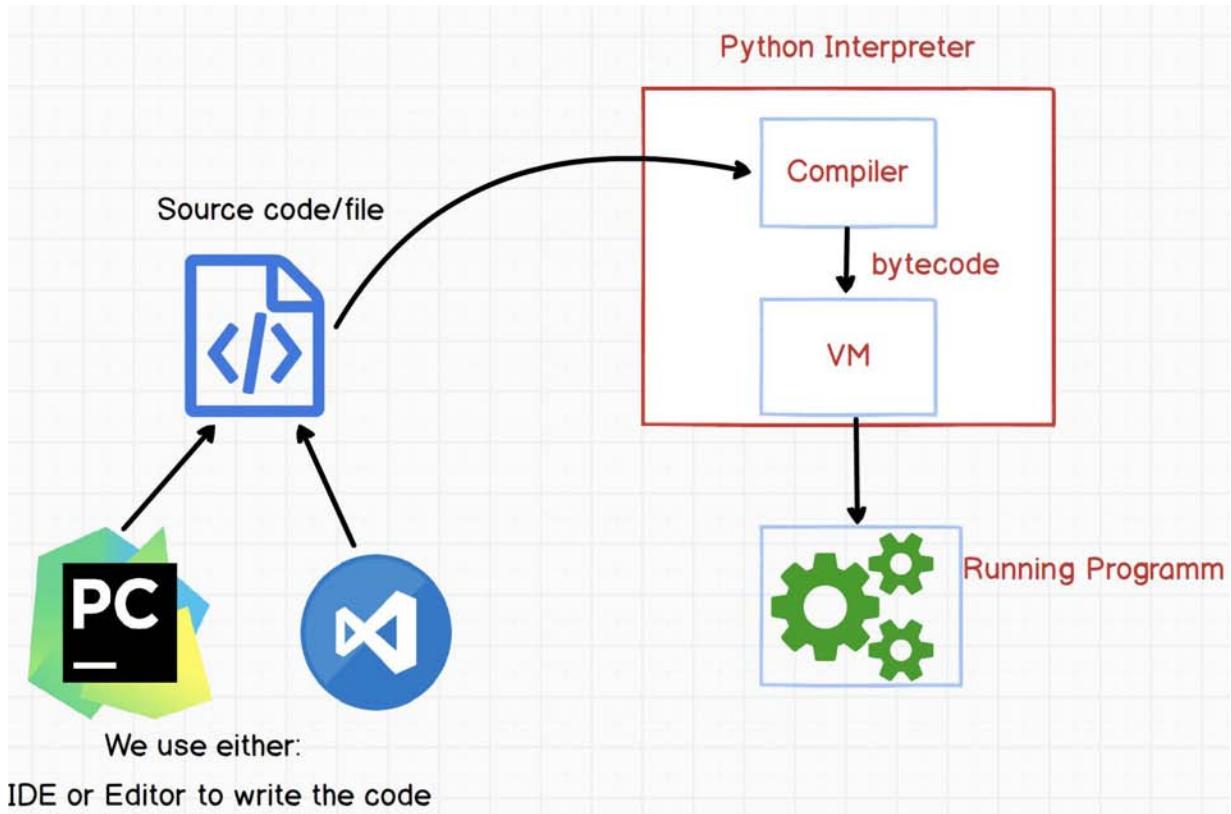
- Games development
- Web Development
- AI and Machine learning
- E-commerce
- Image processing and graphic design
- Scientific computing

Another benefit is that the Python code is interpreted, not compiled. This means that the code is interpreted line by line during the runtime and doesn't require pre-runtime compilation. Because the Python code uses an Interpreter, it executes the statements line by line and it's easier to debug and spot the errors. There are third-party tools that can be used to optimize the performance and to also to handle errors. One huge benefit is that Python code is portable and cross-platform functionality is a must in today's development. This means that applications created in Python can run on any operating system like Windows, macOS, Linus, and Unix without modifications. These are some of the features that make Python one of the most popular programming languages.

## Python Interpreter

As we know from the definition, Python is an **interpreted high-level** language. This means Python needs an interpreter to read the code from our file and execute the instructions from that file. Other **high-level** languages like **C++** require us to compile the code. The compilation process will turn the code that is humanly understandable into a code that only machines can

understand called **Machine code**. The machine code consists of basic-level instructions that are executed by the **CPU**. After the compilation process, an executable file is generated. On the other hand, **Python** is an interpreted language, meaning it will not translate the code into machine code but will translate it into **bytecode** which will create a file with an extension (**.pyc**). The Python interpreter comes when we install Python and we will see how this is done in chapter 3. This means when we install the official Python language, we have access to the Python interpreter as well. In short, the Python interpreter is a program that will read our source code and run it for us. So, what is the difference between the **bytecode** file and the **compilation file**? Well, the compilation file is a set of instructions that are executed on the **CPU** and the bytecode file instructions are executed on a Virtual Machine known as **CPython VM**. Because of this (CPython VM), the **bytecode** can be executed on any operating system or platform. In simple words, a developer writes the code in Python and then the code is compiled into Python **bytecode**. This will create a file with an extension (**.pyc**) and the **bytecode** is something that we as developers can't control because it happens in the background. So, the **bytecode** is a low-level representation of the code we write. The **VM (Python Virtual Machine)** is something we should not worry about as well because it is part of the Python system and is used to run our Python files/scripts. What is interesting is that the Python Interpreter is written and implemented as a C program and that is why it's called **CPython**. Here is the figure that explains the whole process of how the Interpreter works:



From the figure above, we can see that we write the Python code using different tools like IDEs and Code editors (we will install them in chapter 3) and the file is taken by the Interpreter that uses the compiler to create the **bytecode** file with the extension (.pyc). This file is taken and executed by the Virtual Machine that is platform-independent and gives us the option to use Python on any system we want. In chapter 3, we will see how we can install Python on our machines and how to use different tools to write and execute Python code professionally.

## Why so many programming languages?

The answer to why we have so many programming languages is very simple. Today we have different languages because there is no single language that can do all of the things we want to do. This means there is no single ultimate language that will be perfect for making games, websites, software applications, and much more. Today we have so many different application needs that make it impossible for one language to be able to support everything. This will make the language slower and will take a lot of time

and memory to run and operate. Every language has its positive and negative aspects, for example, Python as a language is slower than **C++** and **Java** but it's perfect for making applications that are used for machine learning or web servers and scripts that will process millions of files. That is why we have so many languages. We need to focus only on the best ones and Python is somewhere at the top.

## Getting Started – How to Run Python Code

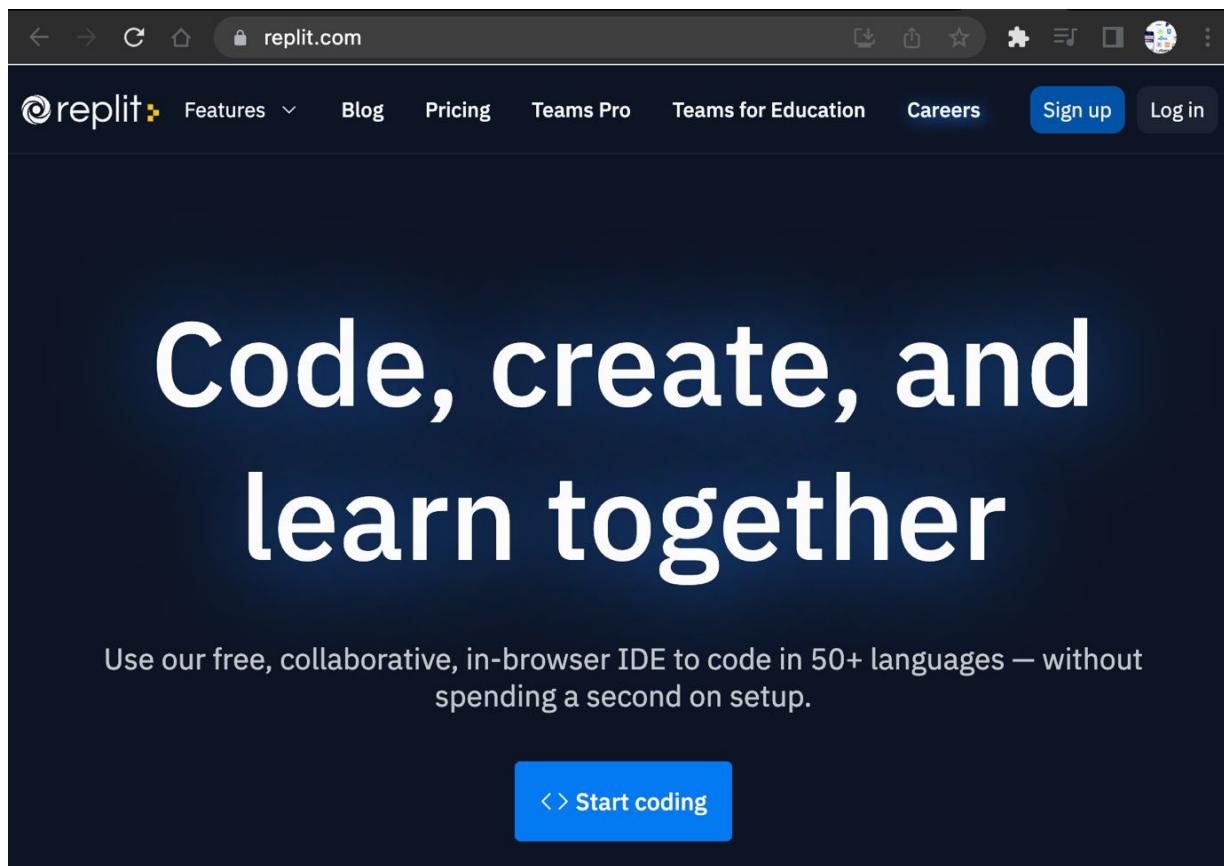
There are different ways we can run Python code. If you look at other books or courses, you will see that they start with Python installation and setting up tools in the first or second chapter. In this book, we will change that. We will not install Python and the tools until chapter 3, but why have I taken this approach? The reason is simple, I want you to learn the basics of Python without being stressed about why I need all of these different tools or why I need to install Python and use the terminal or command prompt to run some features. I promise you will learn a lot of ways to write, run, and execute Python code. Today we can run Python online from anywhere on this planet and all you need is to have access to the internet. This means you can write and execute Python code even from your mobile phones, tablets, and laptops without any software. In chapter 3, I will go over what professional web developers use to write and execute code, and instead of showing you only one way of doing this, I will include different tools and you can choose the ones you like. I promise there is no single book that will include all of them. So how we can run Python code? We can run Python code in many ways:

- We can run it on the terminal or command prompt (CMD)
- We can run Python code on different code/text editors like:
  - VS Code
  - Atom
  - Sublime
  - Vim
- We can run Python code on different IDEs like:
  - Spyder
  - PyCharm
  - Eclipse PyDev

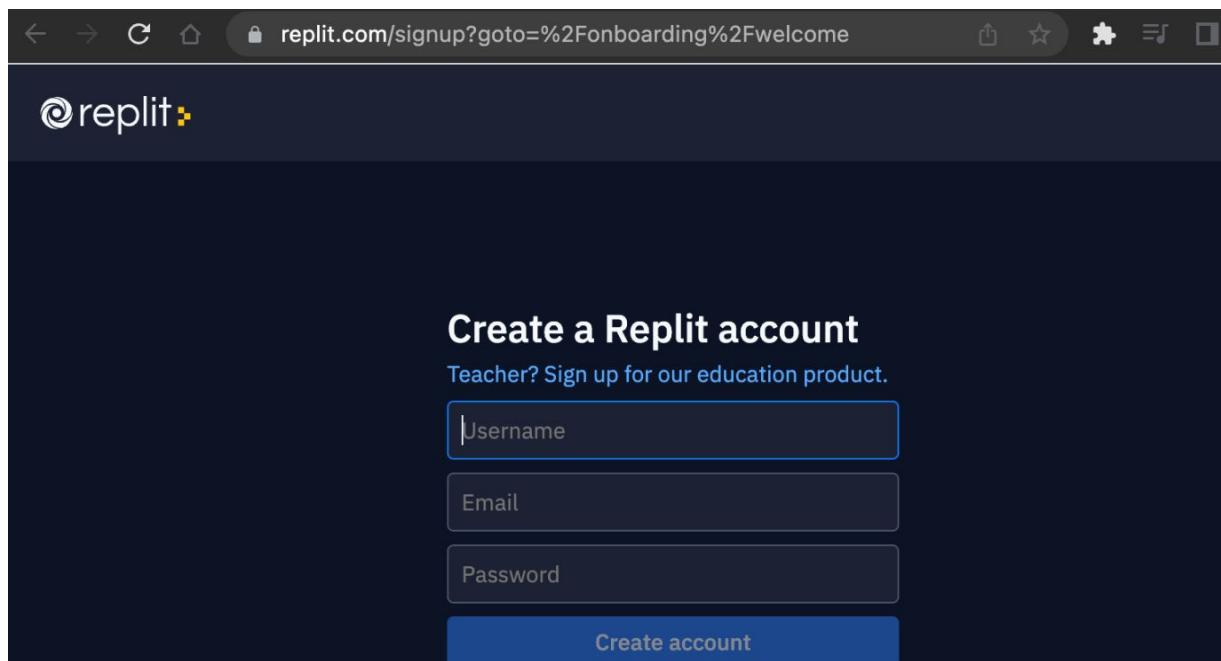
- IDLE
- Wing
- Notebooks like Jupiter

As you can see, there are different ways to write, run, and execute Python but let's start with the simplest one. There is a website called **replit** that we can use for free to write and run Python code. You can even finish my entire Python book using this website only. Here is a link to the official website:

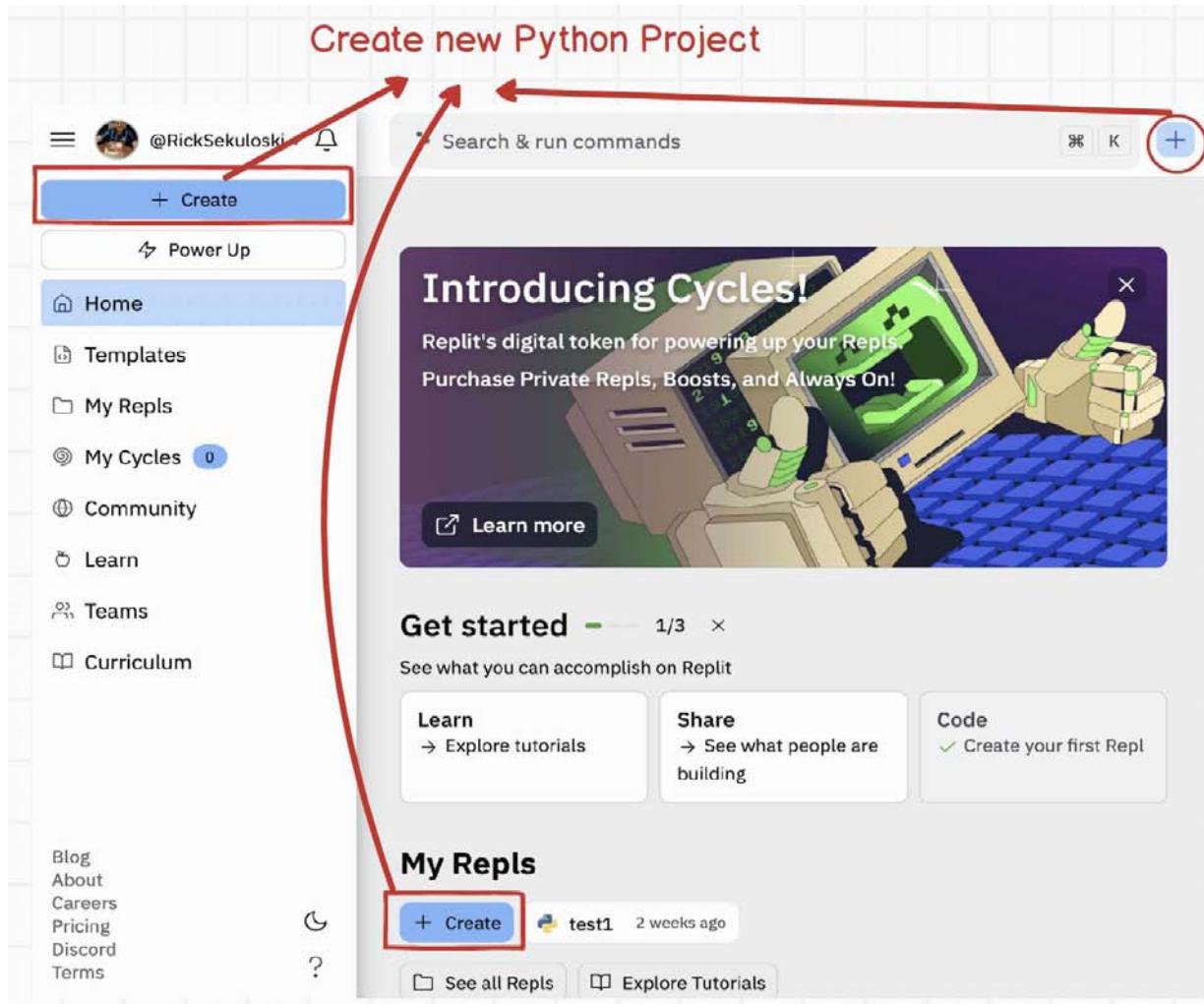
<https://replit.com/>



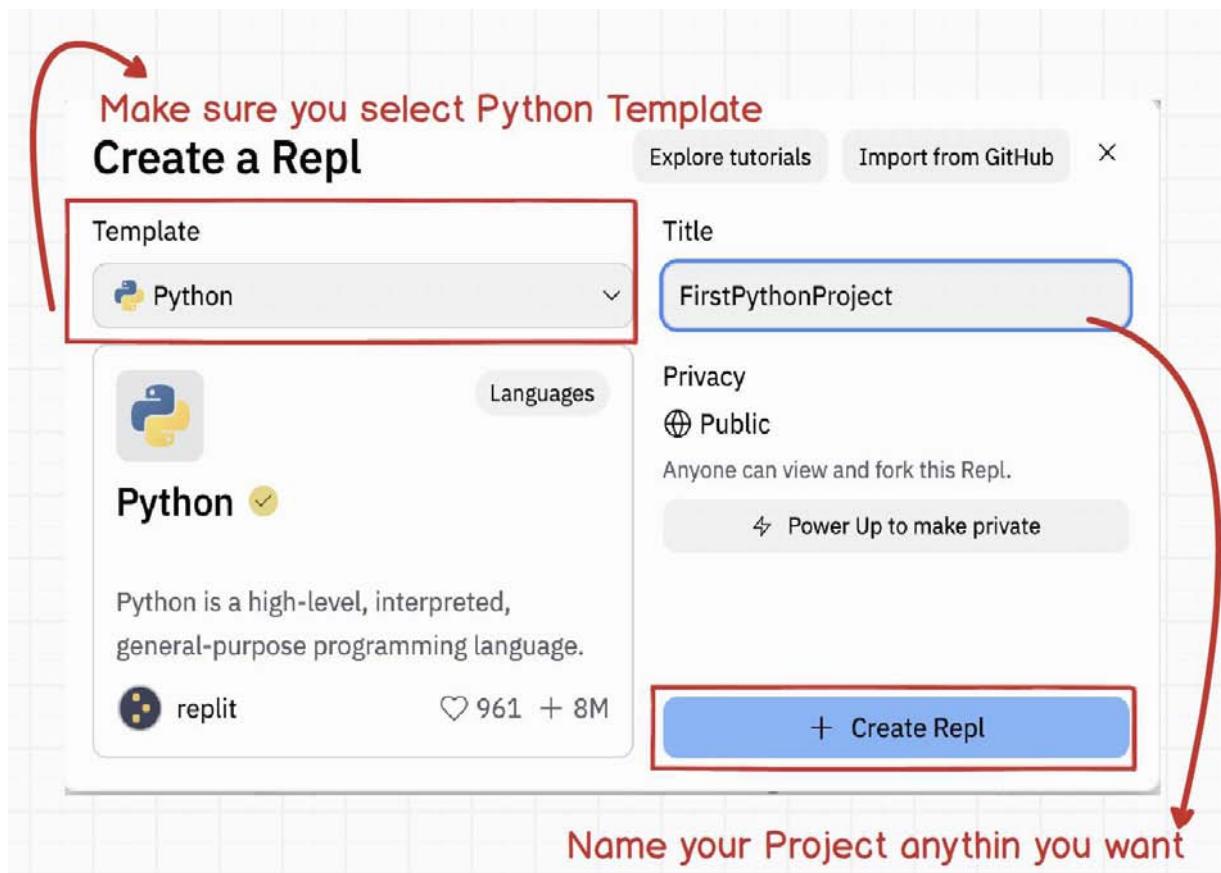
Using a website like **replit** is something I recommend for every beginner, and to be honest, it is simply perfect and easy to use. The one thing you need is to create an account, which is free:



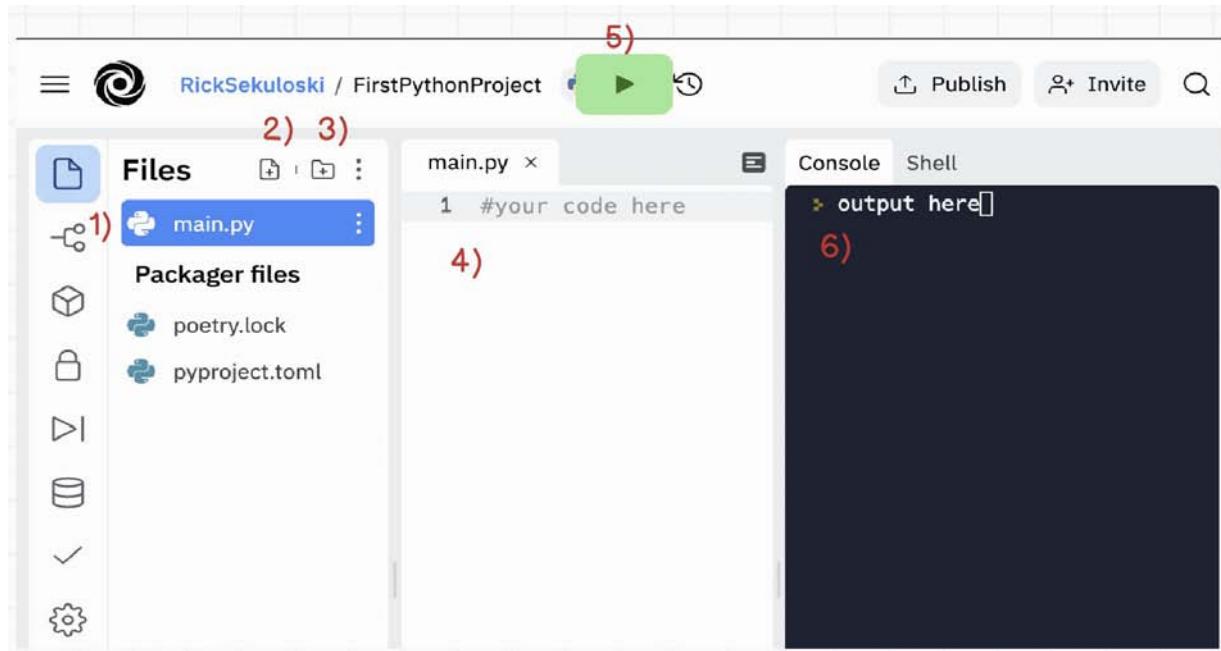
After you create your account, you can log in and create your first project or your first repl. There are different ways to create Python projects and in the following figure I highlight some of them:



In the future, the **replit** interface might slightly change but I believe it will be even easier to create a new project. After you click on the plus sign or on any of the icons from the figure above, you will have to name your first repl ( I have called the Project FirstPythonProject):



The last thing you need to do is to click on the (+ Create Repl) button and your project will be ready. After creating your first project, you should have a similar interface to the one in the figure below:

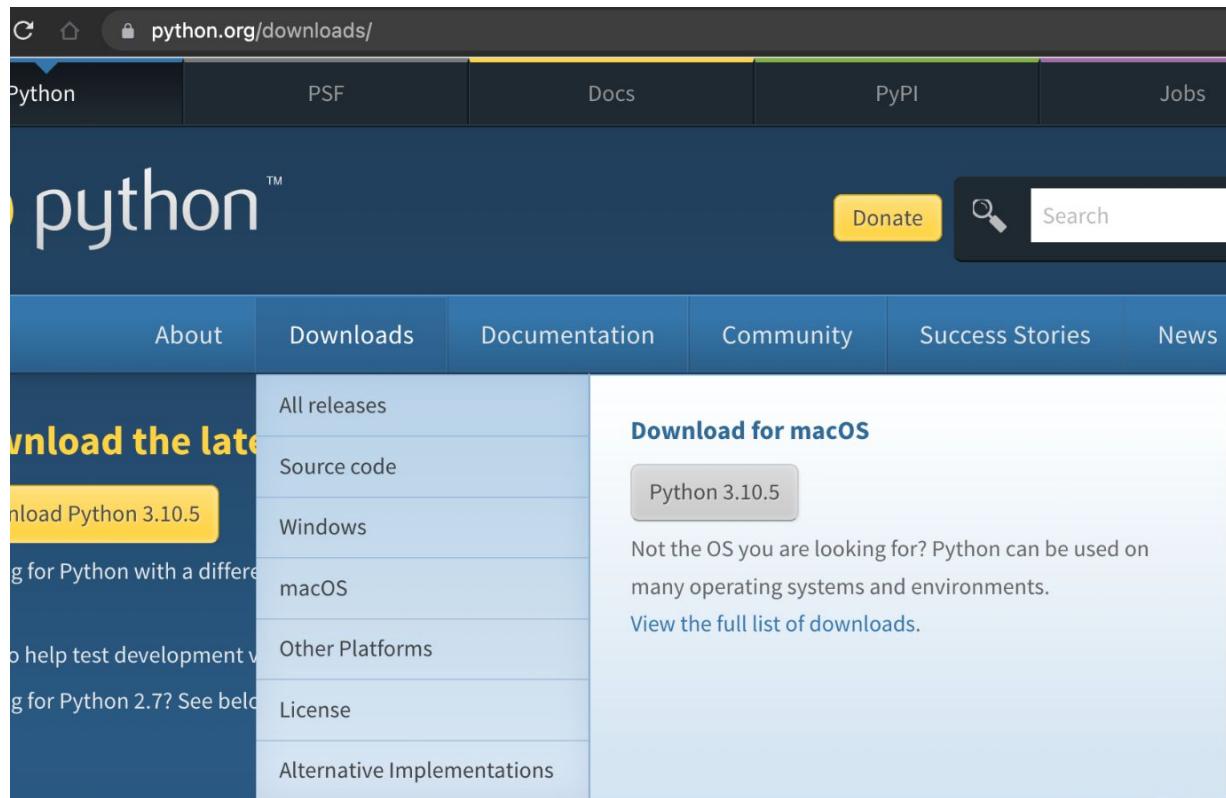


As you can see from the figure, I have marked the important items with numbers so let me explain what each of these numbers means:

- 1) The Number one is the **main.py** file. This is the file that **replit** creates for us when we create a new project. As you can see, we need to use the .py extension to indicate that we are going to write a Python code
- 2) The number two will let us add a new file to the existing project. Just don't forget to add .py extension when you name your file. For example, you can call or name your file **test.py**
- 3) The number 3 allows us to create a new folder where we can keep other Python files
- 4) The number 4 is where we write our actual Python code
- 5) The number 5 or the green play button is to execute the code we have written in the **main.py** file
- 6) The number 6 is where we will see the output after we execute the **main.py** file by clicking on the run button

The version number of Python in websites like replit is lower than the official Python that we download from the official website. For example, in chapter 3, we will install the latest Python version but I just wanted to show you that replit uses Python Interpreter as well. If we go to the official

Python.org website and click on the downloads tab from the menu, we will see something like this:



You can do the same if you visit the following link:

<https://www.python.org/downloads/>

As you can see from the figure above, the Python website recognizes my operating system automatically and gives me an option to download Python version 3.10.5 for macOS. In your case, depending on how far in the future you will read this book, you will have a different version. If we go back to our repl.it website and click on the shell tab and type: **python -V** or if that is not working try **python3 -V**, you will see that the Python version the website is using to run our code is not the same as the official Python page:

The screenshot shows a terminal window with two tabs: 'Console' and 'Shell'. The 'Shell' tab is selected and circled in red. The terminal output is as follows:

```
~/FirstPythonProject$ python -V
Python 3.8.12
~/FirstPythonProject$ python3 -V
Python 3.8.12
~/FirstPythonProject$
```

The version is 3.8.12, an older version of Python but as long as it starts with the same first number (3) as the official website, we are good to go. This is everything I wanted to explain about repl.it website. It is very easy and straightforward to use so in the next section we will write our first Python code.

## Write Our First Python Code

The simplest code we can write will be using the `print()` function. In order to write and run our first code, first, you need to open the repl.it website, and in the `main.py` file write the following code:

```
print('Hi everyone!')
```

After we've written the code, we can run or execute it if we press the green play button on top and make sure the 'Console' tab is selected:

The screenshot shows the repl.it interface. At the top, there's a project dropdown labeled 'FirstPythonProject' with a Python icon, a dropdown arrow, and a green play button. Below that is a code editor with a file named 'main.py' containing the code `print('Hi everyone!')`. To the right of the code editor is a terminal window. The 'Console' tab is selected and circled in red. The terminal output shows the result of the execution: `Hi everyone!`.

From the figure above, we can see that on the right side is where we have the output ‘Hi everyone!’ Don’t worry about the print function or the code we wrote because we will get to that part later in this chapter. One question that I usually get is, how does the **replit** website provide this output to us? In Python files like the one we are using at the moment **main.py**, we write the python code and it can be only one line or it can be thousands of lines of code. These lines are individual instructions and the Python Interpreter that runs in the **replit** will read the main.py file line by line and convert them to a **bytecode** so it can finally use the **CPython** Virtual machine to produce an output. Let’s write more code. It’s ok if you don’t understand the code at this stage but by doing this, we help you to get familiar with the Python language, therefore under the print() function, write the following code:

```
input('What is your name?')
```

If we click on the run button again, we will see the following output:



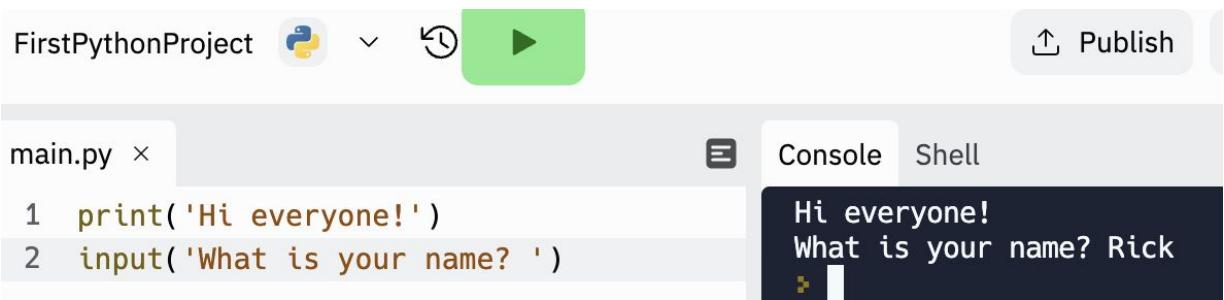
```
/ FirstPythonProject  Python  ⏪ Publish
```

```
main.py ×  Console Shell
```

```
1 print('Hi everyone!')
2 input('What is your name? ')
```

```
Hi everyone!
What is your name? Rick
```

The input function allows the user to enter text, and I have entered Rick after the question mark. As you can see, I wrote our first program asking the user to input something. In the console on the right, I have entered my name (you can enter your name if you want to) but I haven’t clicked on the return or enter on my keyboard and because of this the program still runs in the background and waits for us. We know the program waits because we don’t have the green play button on top, instead of that, we have a black square or stop button. This button will disappear as soon as we press return or enter on our keyboards:



```
FirstPythonProject Python ▾ ⏪ ▶ Publish
```

main.py ×

```
1 print('Hi everyone!')
2 input('What is your name? ')
```

Console Shell

```
Hi everyone!
What is your name? Rick
```

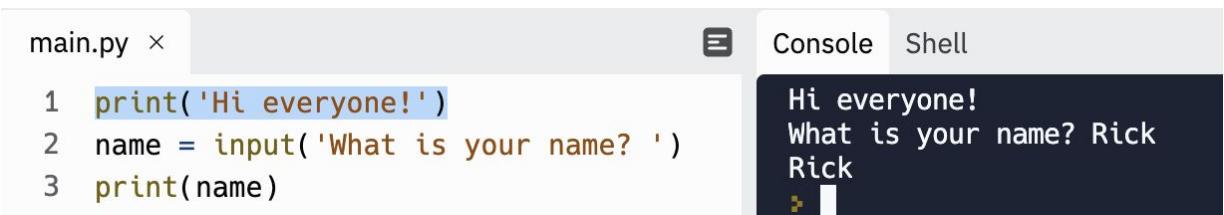
Before I explain something even more interesting, let's explain the print and input function. In Python, we have built-in functions that come when we install the Python language. In this case, Python is installed on the repl.it website and that is why we can use these functions. The `print()` function is used in Python to print out a specific message on the screen or console. The message in this case is a simple string but it can be much more complex. The second function we used is called `input()` and this just like the `print()` is a built-in function that Python allows us to use. The `input()` function allows users to input something.

## Variables in Python

If we want to make our simple program even more interesting, we can use variables. We will discuss variables in detail later in this section but you should know that variables in Python are like containers where we can store values. In our case, I want the name from the `input` function to be stored in a variable like this:

```
name = input('What is your name? ')
print(name)
```

If we click on the run button this will be the output:



```
main.py ×
```

```
1 print('Hi everyone!')
2 name = input('What is your name? ')
3 print(name)
```

Console Shell

```
Hi everyone!
What is your name? Rick
Rick
```

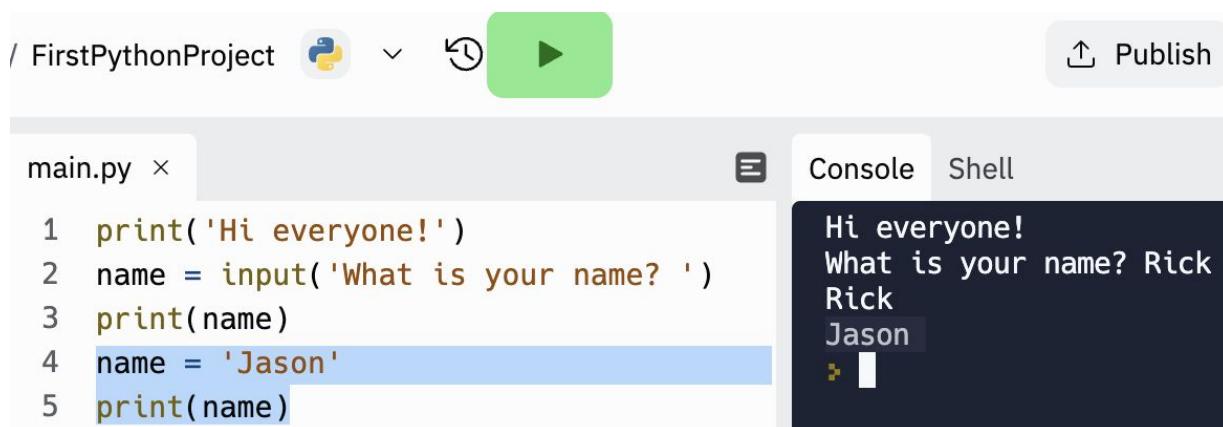
The variable, in this case, is called ‘name’ and we assigned the value from the input function. The input function after typing the name will return whatever we have entered in the console. In our case, I have typed ‘Rick’ and pressed return or enter and the function input() returns the ‘Rick’. This is the same as what I have done:

```
name = 'Rick'
```

On line three I’m using the print function to display the content of the variable name to the console and that is why I get the name ‘Rick’ in the output. To summarize, variables are names we give to the data we want to store. We need variables because we can store and manipulate different data. Our variable called name now can be reused again and again in our code. When we create a variable in the background, the program will allocate space in the memory where it will store the data. An important thing you need to know is that the data inside the variable can be changed with a different value in our program later. For example, after line number three, let’s write this code:

```
name = 'Jason'  
print(name)
```

If we run the following code, what do you think will be the output ‘Rick’ or ‘Jason’? Let’s run the file and observe the result:



The screenshot shows a Python code editor interface. At the top, there's a header with the project name "/ FirstPythonProject", a Python logo icon, a dropdown menu, a green play button icon, and a "Publish" button. Below the header, the code editor window has a tab labeled "main.py". The code in "main.py" is as follows:

```
1 print('Hi everyone!')  
2 name = input('What is your name? ')  
3 print(name)  
4 name = 'Jason'  
5 print(name)
```

The line "name = 'Jason'" is highlighted with a blue selection bar. To the right of the code editor is a terminal window titled "Console". It displays the following output:

```
Hi everyone!  
What is your name? Rick  
Rick  
Jason
```

As you can see, we can change the content of the variable called name to a different value. There are a few keywords I want you to remember from this

section:

- Variable declaration
- Initialization
- Initial value

The above keywords are not unique to Python only, they have been used in most programming languages.

Please consider the following statement:

```
my_age = 10
```

In this statement, we are defining a variable called my\_age and this is called variable declaration because we are giving the variable a name. We use the equal sign = to assign a value to the variable and this is the process of initialization. During the variable declaration, it is advised to give an initial value or data. In our case, this will be 10, but in Python, we can create variables without values as well but this is a topic for another section. We can create multiple variables in one line like this:

```
first, second = 1, 2
```

This is equivalent to:

```
first = 1  
second = 2
```

When we are creating variables and initialize them to a value, we always use the assignment operator ‘=’. Because of this operator (equal), we can achieve the binding process between a variable name and value. We will learn about some rules on how we can name the variables in Python but I think it is the right time to move and learn other Python features. This was our warming-up section and I hope you liked it.

## Comments in Python

We can write comments in our code if we use the # symbol in front. The comments are not going to be included in the final output and the Python Interpreter will ignore them. Comments are very useful in Python because they will make our code more readable not just for us but for any other developers that might read our code.

In Python, we can write the single line comment like this:

```
# this is single line comment
```

We can also write multiline comments if we use three single/double quotes like this:

```
""  
Hi,  
this is multi-  
line comment!  
""
```

The rule of thumb is that the comments we are using in our code must add valuable information. This information can help you if you read your own code in the future but can also help other developers understand what your code is about because not all of the features are easy to understand. Please do not add comments on every piece of code you write because you will clutter your code. The comments you write must be short yet concise.

## Data Types

Data types are values in Python. Each value in Python has a data type. There are various data types in Python. I will try to separate the data types in Python by their importance in different categories and I will start with listing the core or basic data types first:

- Int – integer
- Float – floating point number
- Complex – this is used beside int and float numbers
- Bool

- Str – string

There are more advanced data types in Python:

- List
- Tuple
- Set
- Dict – dictionary

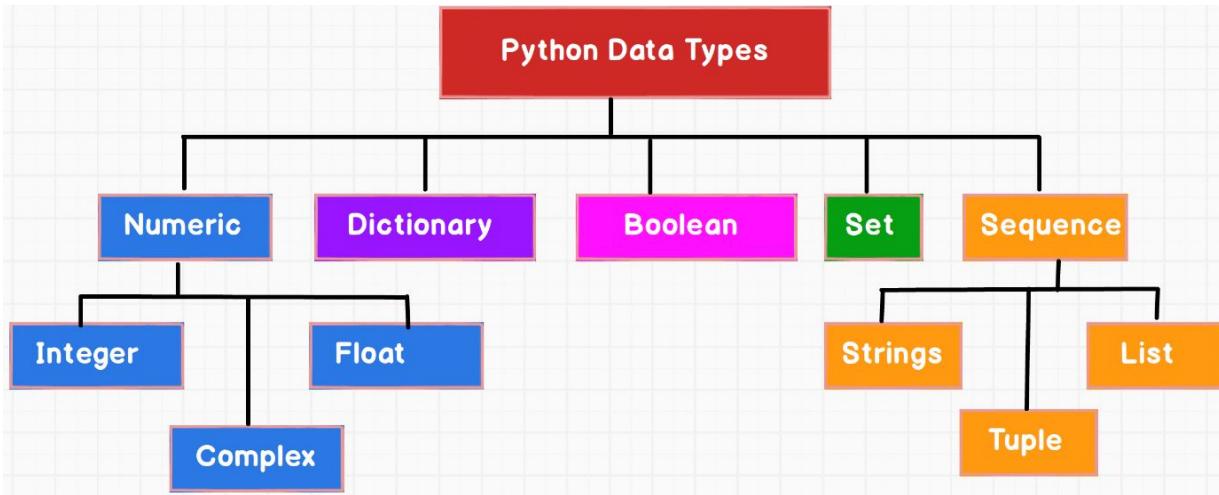
Most books will explain the above fundamental data types but in Python, some data types are beyond the core types. Python allows us to create our own data types using:

- Classes - there are used for creating custom types

In Python, we also have something called specialized data types. These data types are not built-in in python and they include:

- Packages
- Modules

These **packages** and **modules** can be used from different libraries and are used if the above data types are not sufficient for us. Another important data type is **None**. The **None** data type is used when we want to specify the absence of value, similarly to **Null** in JavaScript language. The final data type that I want to mention is called **Complex Numbers**. The complex numbers are represented by complex **Class**. In the following figure you can clearly see how the data types are divided into categories:



In this chapter, we will cover all of the Python data types but I will not be following the exact order as in the figure above. There are a lot of other Python features that are worth mentioning and I think including them as I explain the different data types will help you to understand Python. I promise the first couple of sections are the hardest as we can't move faster enough because we don't know the core features of Python. Therefore, I would like to suggest that you carefully read all of the sections from now onwards.

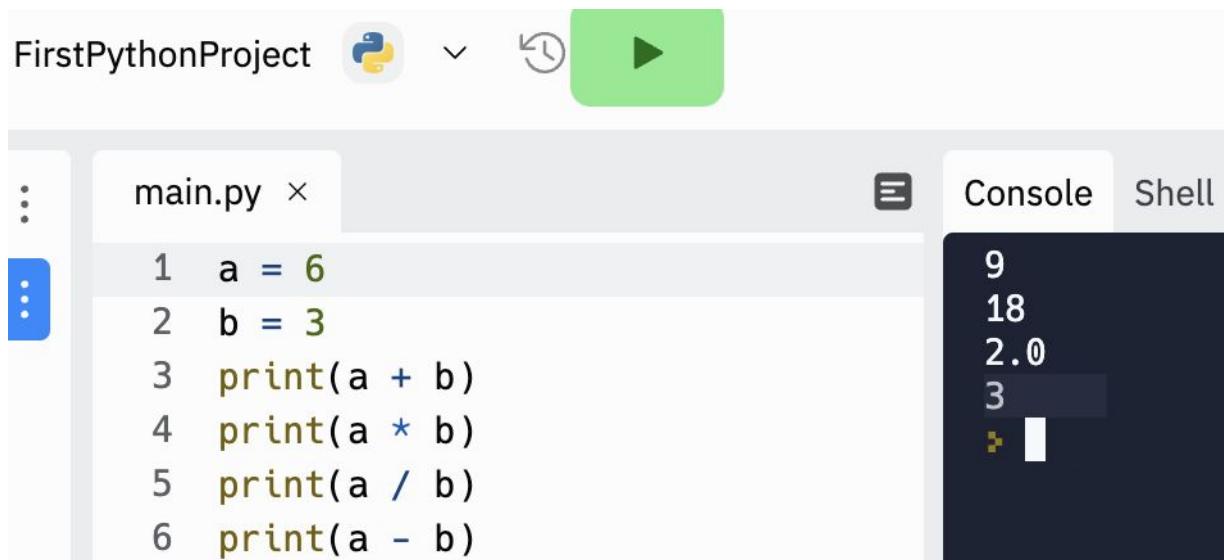
## Int – Integer (data type)

Python includes three numeric types (Integer, Float, and Complex) so we can represent the numbers. One of them is called Integer or Int. An Integer is a whole number, for example, it can be a zero (0), positive or negative whole number. Why do we need Integer numbers? Well, we use them to do mathematical operations using mathematical operators like +,-,/,\*. We can do some basic calculations like the following:

```
# Int
a = 6
b = 3
print(a + b)
print(a * b)
```

```
print(a / b)
print(a - b)
```

We can run the above code and check the output:



The screenshot shows the Replit IDE interface. At the top, there's a dark header bar with the text "print(a / b)" and "print(a - b)". Below this is a toolbar with icons for file operations and a green "Run" button. The main workspace is titled "FirstPythonProject" and contains a file named "main.py". The code in "main.py" is:

```
1 a = 6
2 b = 3
3 print(a + b)
4 print(a * b)
5 print(a / b)
6 print(a - b)
```

To the right of the workspace is a sidebar with tabs for "Console" and "Shell". The "Console" tab is selected, showing the following output:

```
9
18
2.0
3
```

If you have noticed, when we write functions in Python, we always use the () (parenthesis) or brackets. The brackets are a very important part when we try to access the function. Without them, the function will not return the result we expect. In Python, we can use the type() function to determine the data type. If you have done other programming languages such as JavaScript, you will notice that we have a lot of similarities between these two languages. In JavaScript, to determine the type, we use the typeof() function. From the figure above, we have two variables that have a value from type Integer. Let us check if this is true using the type() function:

```
print(type(a))
print(type(b))
```

If we run the above code in the replit, we will get the following output:

The screenshot shows a Python development environment with two tabs: 'Console' and 'Shell'. In the 'Console' tab, the following code is run:

```
main.py ×
1 a = 6
2 b = 3
3 print(a + b)
4 print(a * b)
5 print(a / b)
6 print(a - b)
7
8 print(type(a))
9 print(type(b))
```

The output in the 'Console' tab is:

```
9
18
2.0
3
<class 'int'>
<class 'int'>
> []
```

Both variables are from class ‘int’ and please don’t worry about what the class keyword means at this stage. To summarize, Integers are whole numbers without decimal parts.

## Float– Floating point number (data type)

The floating numbers are the numbers that have decimal parts such as 3.5, -9.7, 11.023. Here is one example of float data types:

The screenshot shows a Python development environment with two tabs: 'Console' and 'Shell'. In the 'Console' tab, the following code is run:

```
main.py ×
1 #Float
2 m = 1.5
3 n = 3.22
4 print(m * n)
5 print(type(m))
6 print(type(n))
```

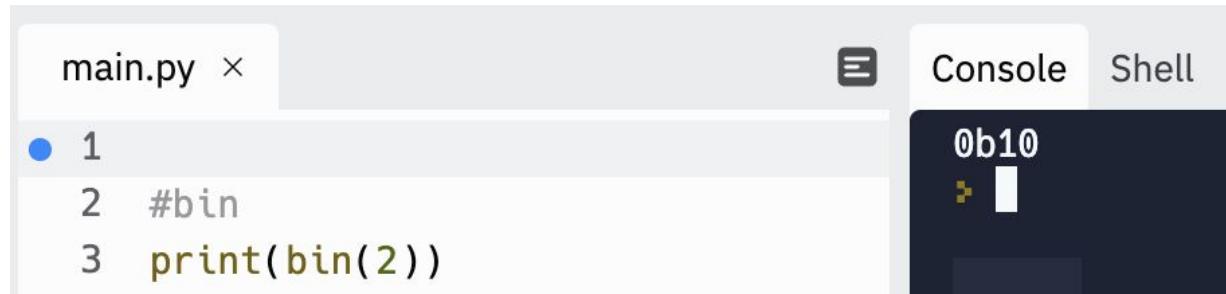
The output in the 'Console' tab is:

```
4.83
<class 'float'>
<class 'float'>
> []
```

## How Integer and Float are stored

We have seen what Integer and Float numbers are but is there any major difference between the float and integer numbers? Yes, there is a big difference because float numbers are stored in memory in different locations, therefore, they take up more space in the memory compared to integer numbers. Take as an example the float number 4.5. In the memory, it will be stored as 4 and 5 in two different locations and that is why the float decimal numbers take more memory space. The integer numbers are converted to binary numbers and then stored in the memory. The binary numbers are made up of 0s and 1s. There is a function we can use called bin() for binary and this function will return a binary representation of the integer number.

For example, let's try to run the following code in repl.it:



The screenshot shows a development environment with a code editor and a terminal window. The code editor has a file named 'main.py' with the following content:

```
1
2 #bin
3 print(bin(2))
```

The terminal window is titled 'Console' and shows the output: `0b10`. The 'Shell' tab is also visible.

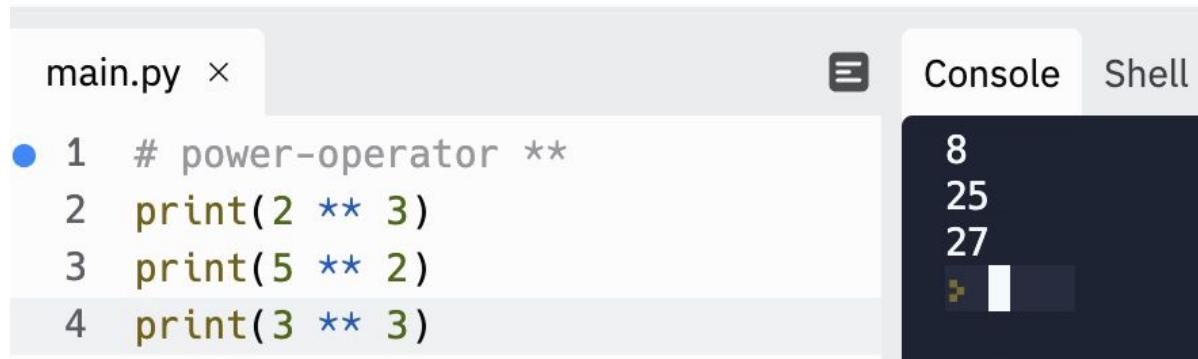
From the figure above, we can see that the output is 0b10, the first part '0b' is a prefix (zero b) to differentiate the binary numbers from numbers in base 10 representation, or in simple words, the prefix denotes a binary literal. The binary value for the actual number 2 is represented by binary 10. Now that we know these two data types (int and float), it is time to learn what mathematical operations we can perform on them.

## Python Power Operator (\*\*)

Python like most programming languages allows us to use integer and float numbers to perform basic or complex mathematical operations. For example, the double multiple `(**)` is used to raise a number in Python to the power of an exponent. This operator is known as power operator and it requires two values to perform a calculation. Here is an example of how we can use the power operator:

```
2 ** 3
```

In this expression, 2 is raised to the power of 3, therefore the result will be 8:



```
main.py ×
```

```
1 # power-operator **  
2 print(2 ** 3)  
3 print(5 ** 2)  
4 print(3 ** 3)
```

Console Shell

```
8  
25  
27
```

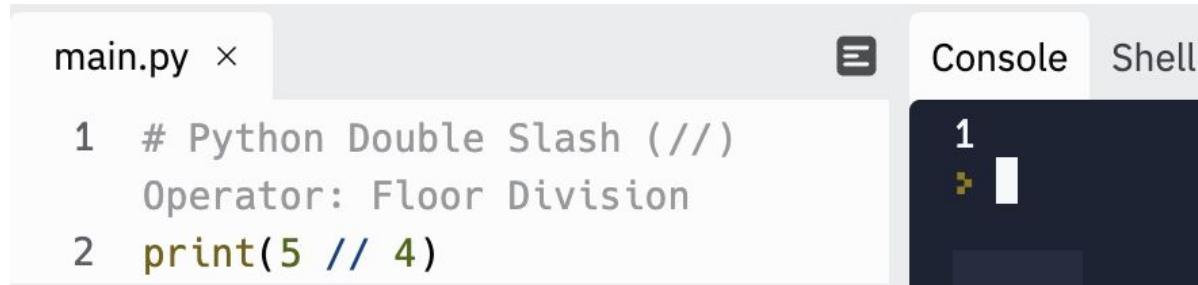
In the next section, we are going to discuss the floor division.

## Python Double Slash (//) Operator: Floor Division

In Python, we can use the double forward slashes (//) to perform floor division. The floor division is known as integer division and it uses the following syntax:

```
5 // 4
```

This operator will divide the first argument by the second argument and will round the result to the nearest whole number. In Python, there is a function called `math.floor()` that will do the exact same thing but in order to use this function, we need to learn how to import an external module and we are not going to cover importing modules in this chapter, therefore the double slash (//) operator is what we will use at this point. If we run the above example in the `main.py` file, we will get an output of 1:



```
main.py ×
```

```
1 # Python Double Slash (//)  
    Operator: Floor Division  
2 print(5 // 4)
```

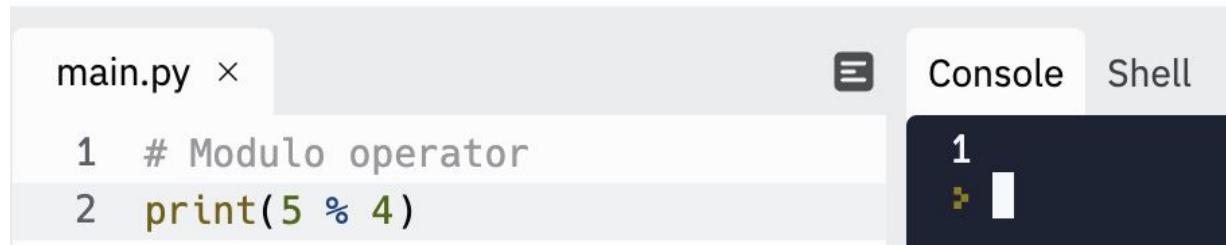
Console Shell

```
1
```

This operator will return an integer that is rounded down to the nearest whole number.

## Python Modulo Operator (%)

This operator is known as the Modulo operator and the symbol we need to use is (%). This operator is used to return the remainder when we are dividing the left operand by the right operand like in the following example:



The screenshot shows a Python development environment. On the left, there's a code editor window titled "main.py" containing two lines of code:  
1 # Modulo operator  
2 print(5 % 4)  
On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution: 1.

The result is 1 because the modulo operator will give us the remainder of this division. If you have read some of my JavaScript books you will see that we use the modulo operator in JavaScript as well. The modulo operator is considered an arithmetic operation along with these ones:

+, -, /, \*, \*\*, //

In the next section, we will cover what type of math functions we can use with floats and integer numbers.

## Math functions on Int and Float data types

These math functions we will learn in this section are built-in functions. This means that Python gives us access to these math functions without the need to import external libraries. The first function we will discuss is the round() function.

### round()

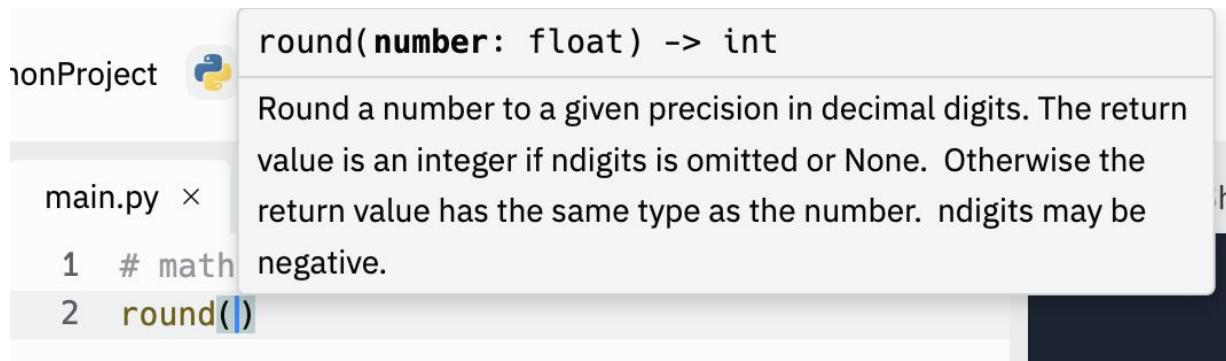
The round function rounds a float number to the nearest integer number. For example, if we have float number 4.3 and use the round() function, the output should be 4. Let's test a few more examples and see what the round() function will return:

The screenshot shows a Python script named `main.py` in a code editor. The script contains ten print statements using the `round` function on various floating-point numbers. The output in the console shows the results of each print statement.

```
1 # math functions
2 print(round(4.1))
3 print(round(4.2))
4 print(round(4.3))
5 print(round(4.4))
6 print(round(4.5))
7 print(round(4.6))
8 print(round(4.76000))
9 print(round(4.8))
10 print(round(4.9))
```

```
4
4
4
4
4
5
5
5
5
> []
```

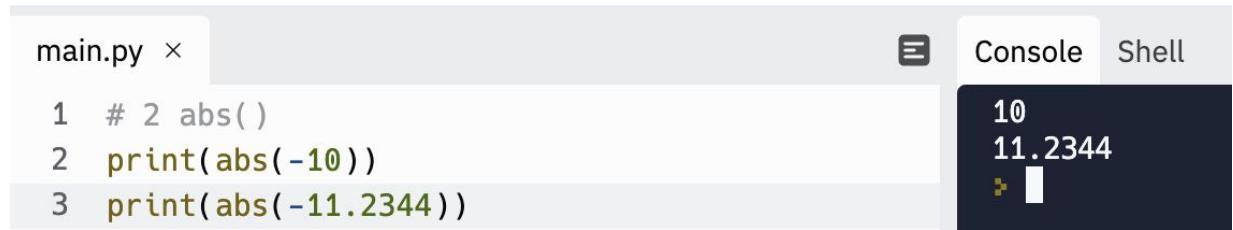
The replit website works similarly to having installed a code editor or as if we are using an IDE to write Python code. Let me show you why. If I type `round` and add the open and close bracket without pressing the run button the replit website will give me a short description of what this function does in Python.



This is the exact behavior we will get if we are using a code editor or IDE to run and execute Python code on our machines.

## abs()

The `abs()` function will return the absolute value of the argument. If you are not that good at maths, then the absolute function will never return a negative number:



The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 # 2 abs()
2 print(abs(-10))
3 print(abs(-11.2344))
```

On the right, a "Console" tab is active, displaying the output of the code:

```
10
11.2344
```

As we can see, the absolute value of negative ten will be positive ten. There are more math functions available on the Python official website and we will not cover all of them because the goal of this book is to teach you Python, not how to be good at math. If you open the link below, you will see a complete list of math functions that are included in the **math** module. But remember that in order to use them, you will need to learn what modules are and how to import them and this is something we will cover in-depth in future chapters. You can click on the link and just have a look at the functions that are available in Python:

<https://www.programiz.com/python-programming/modules/math>

## Operator Precedence

So far, we have learned that we can perform mathematical operations on the two Python data types (Integer and Float). In order to achieve this, we need to use operators like `+`, `-`, `*`, `/`. Most times, the mathematical expressions are complex and we use a combination of different operators to get the end result. This is where the operator precedence takes place. The operator precedence means that when different operators are combined in one expression, some of them will have precedence over others. The Python Interpreter knows these rules and always follows them.

For example, let's consider this expression:

`7 - 3 * 2`

From our math classes, we know that first, we need to perform multiplication (`3 * 2`) and then subtraction. The Interpreter knows this rule otherwise if we perform calculations based on the operator order, we will have a different result:

# Operator Precedence

**Correct Answer:**

1)  $7 - 3 * 2 =$

2)  $7 - 6 =$

3) 1

**Incorrect Answer:**

1)  $7 - 3 * 2 =$

2)  $4 * 2 =$

3) 8

Order of precedence:

- 1) () – brackets will have the highest precedence; we first calculate whatever is wrapped in the brackets
- 2) \*\* – (power of) is done only after the parenthesis ()
- 3) \*, / – then we perform multiplication and division
- 4) +, - The last precedence is the addition and subtraction

The order of precedence is the same for all programming languages but it is very important to know it. Without looking at the answer, please calculate the following expression:

$$(7 - 3) * 2 + 2^{**}2$$

The answer:

1<sup>st</sup>) whatever is in brackets (7-3) is done first because of the order of precedence. Therefore, the result after subtraction of 7-3 will be 4

2<sup>nd</sup>) will be the power operator and the result of  $2^{**}2$  will be equal to 4

3<sup>rd</sup>) will be a multiplication operator that will use the result of the 1<sup>st</sup> operation which is 4 and this will be multiplied by 2 which will give us a result of 8

4<sup>th</sup>) Finally, the addition operator takes place and the sum of 8 + 4 is 12

You can run the following code in repl.it. Just wrap it around a print function so you can see the output:

A screenshot of a Python IDE interface. On the left, a code editor window titled "main.py" contains the following code:

```
1 print((7 - 3) * 2 + 2**2)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
12
```

## Rules for Naming a Variable

We know that the variables are containers where we can store data or values. But in Python, we need to follow some rules when we name variables. For example, in Python, the variable name can only contain letters (a-z, A-Z), numbers, or underscores (\_). The first character cannot be a number or underscore. Here is an example with valid and invalid variable names (the values of the following variables are from a different data type that we will discuss later in this chapter):

A screenshot of a Python IDE interface. On the left, a code editor window titled "main.py" contains the following code:

```
1 #valid variable names
2 userEmail = 'rick@gmail.com'
3 user_email = 'rick@gmail.com'
4 user_email1 = 'rick@gmail.com'
5
6 #invalid variable names
7 _user_emails = 'rick@gmail.com'
8 1user_email = 'rick@gmail.com'
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution, which results in a SyntaxError:

```
File "main.py", line 6
    1user_email = 'rick@gmail.com'
               ^
SyntaxError: invalid syntax
```

When naming a variable, it is important to always come up with a good name that is descriptive so that everyone can understand what the variable is about. Python is case sensitive language meaning the variable `user_name` is different variable from the variable `User_Name`.

In Python, we can use two naming conventions:

- Camel case
- Snake case

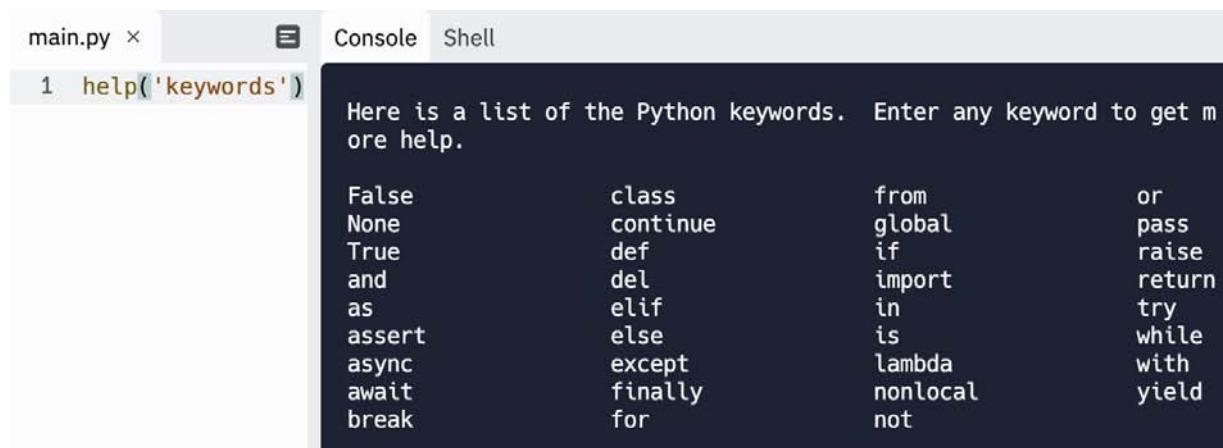
The camel case naming convention is noticeable when we are using compound words to create one variable name. For example, I would like to have a variable name that consists of these two words (first, name). The camel case rule is that the first word starts with a lowercase letter and then for every other word the first letter should be uppercase:

```
#camel case notation  
firstName = 'Jason'
```

The other convention I will use is the snake case where we use underscore to separate the words. Let us rewrite the camel case firstName variable to the snake case:

```
# snake case notation  
first_name = 'Jason'
```

You can use any of the naming notations you want but I would suggest picking only one and sticking to it. It is bad practice to use both naming conventions in one code. In Python, we have a predefined set of keywords that are reserved words, therefore we cannot use them to name variables. How we can see this predefined reserved keyword list? This is very easy in Python because we can use the help() function and type keywords like this: help('keywords'), and the output is the following:



```
main.py ×  Console Shell  
1 help('keywords')  
Here is a list of the Python keywords. Enter any keyword to get more help.  
False      class      from      or  
None       continue   global    pass  
True       def        if        raise  
and        del        import   return  
as         elif       in       try  
assert    else       is        while  
async     except    lambda  with  
await     finally   nonlocal not  
break
```

The built-in help() function is used to get documentation of variables, classes, functions, and modules. You will know you are using keywords in

your code because the keywords will be highlighted in a different color compared to the rest of the code. For example, in my case, the color is blue.

## Reassign variables to a new value

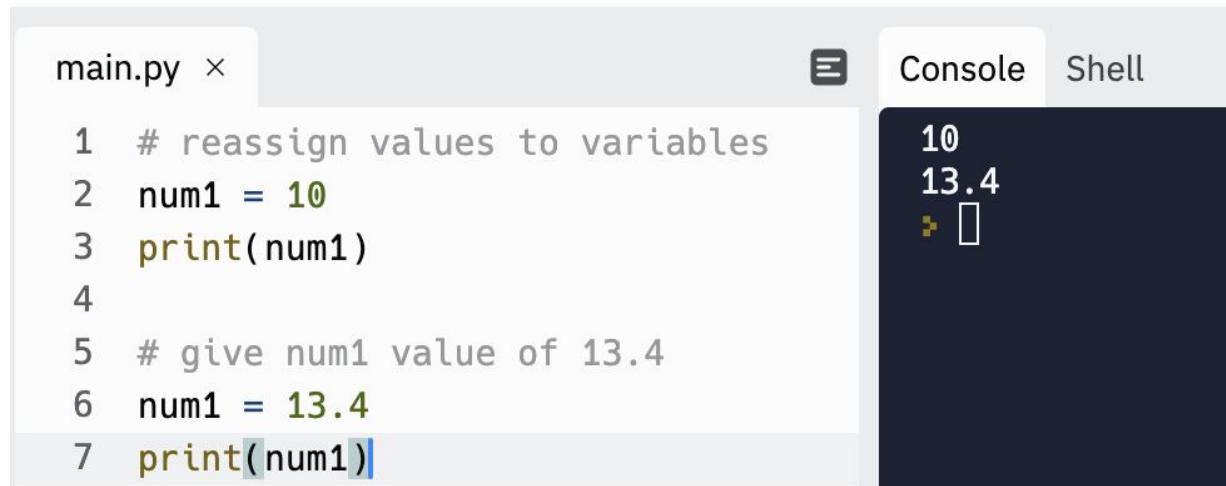
In Python, we can assign a new value to the same variable and this is called reassigning. For example, let's create a variable called num1 and assign a value of a random integer number:

```
# reassign values to variables
num1 = 10
print(num1)
```

Let's now assign a new value and this time it can be a float number:

```
# give num1 value of 13.4
num1 = 13.4
print(num1)
```

Let's check the output:



The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # reassign values to variables
2 num1 = 10
3 print(num1)
4
5 # give num1 value of 13.4
6 num1 = 13.4
7 print(num1)
```

On the right, there is a "Console" tab showing the output of the code execution:

```
10
13.4
> □
```

As you can see, we can reassign the value of the variable easily in Python. In Python, we can use the value of a variable and assign it as a value to a new variable. I know this is confusing, but hopefully, this example will clarify things:

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 # initialize a variable
2 my_weight = 78
3 # reasing
4 your_weight = my_weight - 10
5 print(your_weight)
```

On the right, a "Console" tab is active, displaying the output of the code: "68".

From the figure above, we have a variable called **my\_weight** that has a value of 78. On line 4, there is another variable called **your\_weight** but its value is based on the **my\_weight** value minus ten.

## Constants in Python

In Python, a constant is a type of variable whose value cannot be changed. But this is not entirely true because we still can change the value. In JavaScript for example, when we can create a constant variable, the value cannot be changed because it will throw an error if we try. When we name constants in Python, we should follow some new naming conventions. We need to use all capital letters and underscores if we have a compound name. For example:

```
# Constants
PI = 3.14
EARTH_GRAVITY = 9.807
```

But we can change its value although it's been declared as constant:

The screenshot shows a Python development environment. On the left, there's a code editor window titled "main.py" containing the following code:

```
1 # Constants
2 PI = 3.14
3 EARTH_GRAVITY = 9.807
4 print(PI)
5 print(EARTH_GRAVITY)
6
7 # Assign new value
8 PI = 3.1455
9 EARTH_GRAVITY = 9.8
10 print(PI)
11 print(EARTH_GRAVITY)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the following output:

```
3.14
9.807
3.1455
9.8
> █
```

As we can see, the constants just like the normal variable can be reassigned to new values. What is the point of having them? The rule for all developers is when you see a code that uses constants, you should never attempt to change their values because their values are meant to stay the same.

## Dunder Variables or Dunder Methods

The word dunder stands for ‘double underscore’ because these variables’ names are surrounded by two underscores:



\_dunder\_

In this chapter, I will just mention that there are variables called dunder and they are a special type that we should not try to change, modify, or reassign at this stage. We will learn more about them and what they can do in one of the following chapters.

## Expressions and Statements in Python

An Expression in Python is a sequence or combination of operators and operands that will produce value. It is important to know that an expression will always yield a value. An expression is a piece of code that will produce value. In most of the Python literature (Books or Online tutorials), the expressions and statements are explained in a complicated way.

In the following figure, the result/value of the `current_year - year_born` is the expression:

`current_year = 2023`

`year_born = 1990`

`user_age = current_year - year_born`

This is Python Expression

What are the statements? The statements are more complicated than expressions because they are the entire line of code:

```
current_year = 2023
```

```
year_born = 1990
```

```
user_age = current_year - year_born
```

This is Python Statement

The green line is a statement because it performs some sort of action. The action here is the assignment of a value to user\_age that is produced by the expression. That is why the whole line becomes a statement because it's an action. In the figure below, I have highlighted the two statements:

```
current_year = 2023
```

```
year_born = 1990
```

These two  
lines are  
Statements

```
user_age = current_year - year_born
```

This is Python Statement

## Augmented assignment operator

We know that we use the assignment operator (`=`) to assign value to a variable. Augmented assignment operators are a combination of the assignment operator and the arithmetic or bitwise operator.

Example:

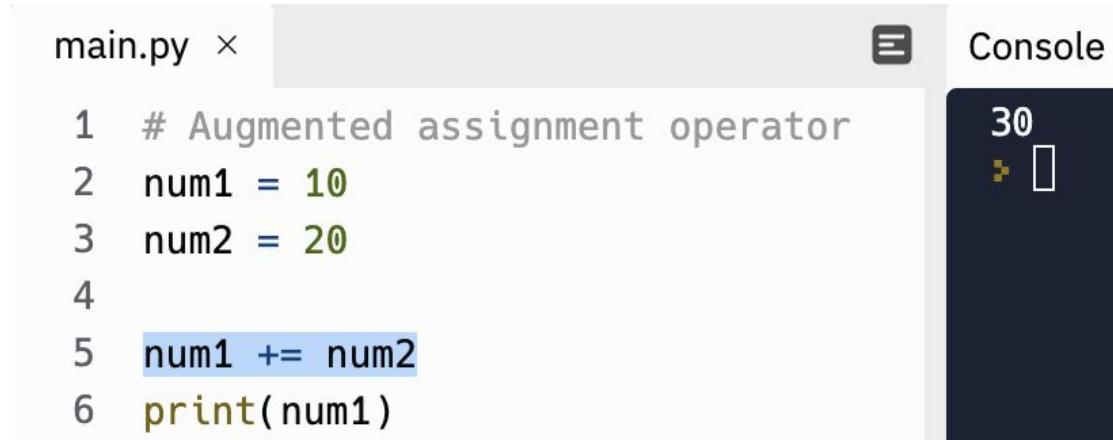
```
# Augmented assignment operator
num1 = 10
num2 = 20

num1 += num2
print(num1)
```

What does `num1 += num2` mean? The augmented assignment operator provides a short way to perform a binary operation and assign the result back to one of the operands. This is a short syntax because it is equivalent to:

```
num1 = num1 + num2
```

The result is:



A screenshot of a Python development environment. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # Augmented assignment operator
2 num1 = 10
3 num2 = 20
4
5 num1 += num2
6 print(num1)
```

To the right of the code editor is a "Console" window. It shows the output of the code execution:

```
30
> □
```

The above operator combines the functionality of arithmetic addition and assignment. In Python, we can use several different augmented assignment operators:

- `+=`, Addition and Assignment
- `-=`, Subtraction and Assignment
- `*=`, Multiplication and Assignment

- `/=`, Division and Assignment
- `//=`, Floor Division and Assignment
- `**=`, Power and Assignment
- `%=`, Modulo and Assignment

Examples:

```
# Augmented assignment operator
num1 = 30
num2 = 10
# Addition & Assignment
num1 += num2
print(num1)
# Output: 40
```

```
num1 = 30
num2 = 10
# Subtraction & Assignment
num1 -= num2
print(num1)
# Output: 20
```

```
num1 = 30
num2 = 10
# Multiplication & Assignment
num1 *= num2
print(num1)
# Output: 300
```

```
num1 = 30
num2 = 10
# Division & Assignment
num1 /= num2
print(num1)
```

```
# Output: 3.0
```

```
num1 = 30
num2 = 10
# Floor Division & Assignment
num1 //= num2
print(num1)
# Output: 3
```

```
num1 = 30
num2 = 10
# Power & Assignment
num1 **= num2
print(num1)
# Output: 5904900000000000
```

```
num1 = 30
num2 = 10
# Modulo & Assignment
num1 %= num2
print(num1)
# Output: 0
```

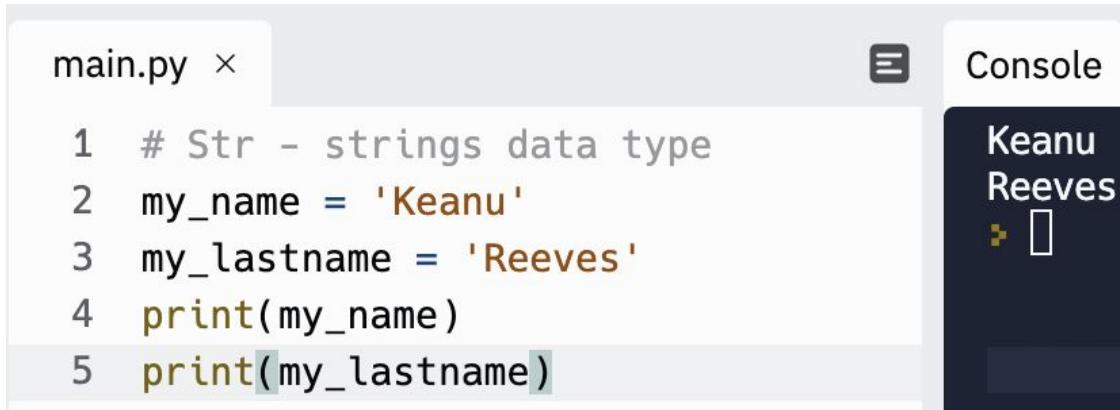
## Str – String (data type)

Strings are another data type in Python. String or **str** in Python is a text that is surrounded by single or double quotes.

String example:

```
# Str - strings data type
my_name = 'Keanu'
my_lastname = "Reeves"
print(my_name)
print(my_lastname)
```

From the example above, we can see that we can use both single and double quotes to wrap the text, and according to the editor you're using, the text within the quotes will be in a different color like green or blue or orange like in repl.it. If we run the same code on our replit, we will see the following output:



The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 # Str - strings data type
2 my_name = 'Keanu'
3 my_lastname = 'Reeves'
4 print(my_name)
5 print(my_lastname)
```

On the right, a "Console" window displays the output of the code execution:

```
Keanu
Reeves
```

We can check the data type like this:

```
#check the data type
print(type(my_name))
print(type(my_lastname))
```

There is another way to write strings in Python and this method is preferred when we have a long string that stretches into multiple lines. For example, when we have a long piece of text that will stretch into multiple lines, we can then use the following syntax:

```
# Long string
text = """
first line
second line
third line
"""

print(text)
```

As you can see, we need to use the three single quotes in a row and then we can write the string in different lines. If we run the same piece of code, this will be the output:

The screenshot shows a Python IDE interface. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 # Long string
2 text = """
3 first line
4 second line
5 third line
6 """
7 print(text)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution:

```
first line
second line
third line
```

The output is in three lines or the same as the original string, and this cannot be achieved with a single quote because it will give us a syntax error:

The screenshot shows a Python IDE interface. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 # Long string
2 text = 'first line
3 second line
4 third line
5 '
6 print(text)
```

On the right, the "Console" tab is active and shows a syntax error message:

```
File "main.py", line 2
    text = 'first line
               ^
SyntaxError: EOL while scanning string literal
```

## String Concatenation

A string concatenation simply means adding strings together using the addition operator '+'. The + operator will join the different strings. From the example above we have two variables `my_name` and `my_lastname` that have strings as data type so with the string concatenation, we are able to achieve the following:

A screenshot of a Python code editor. The code in the main.py file is:

```
1 # String Concatenation
2 my_name = 'Keanu'
3 my_lastname = "Reeves"
4 full_name = my_name + my_lastname
5 print(full_name)
```

The output in the Console tab shows:

```
Keanu Reeves
```

Great but there is no space between the first and last name and this is because there is no white space in the my\_lastname variable so there are two ways how we can add space between the two strings. The first way is not something I would suggest you do but it does the job:

A screenshot of a Python code editor. The code in the main.py file is:

```
1 # String Concatenation
2 my_name = 'Keanu'
3 my_lastname = " Reeves"
4 full_name = my_name + my_lastname
5 print(full_name)
```

The output in the Console tab shows:

```
Keanu Reeves
```

As you can see, I have added a space before the first letter 'R' in the my\_lastname variable. The output is exactly what we need but there is another more elegant way to do this and that is by adding a space in between both variables during the concatenation:

A screenshot of a Python code editor. The code in the main.py file is:

```
1 # String Concatenation
2 my_name = 'Keanu'
3 my_lastname = "Reeves"
4 full_name = my_name + ' ' + my_lastname
5 print(full_name)
```

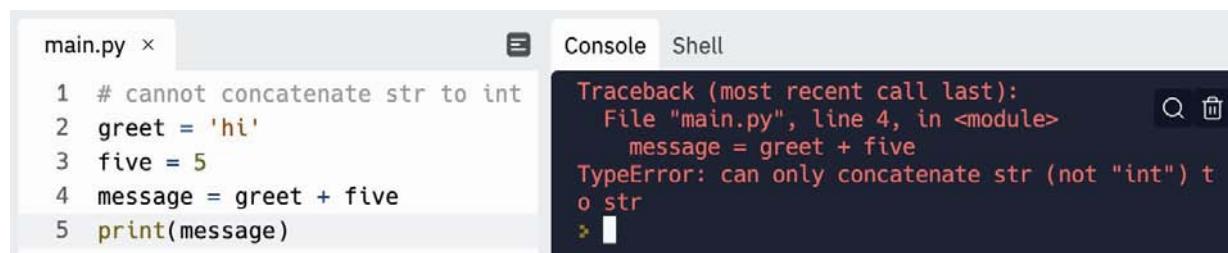
The output in the Console tab shows:

```
Keanu Reeves
```

Great! Now you know that we can use the plus operator to combine multiple strings. There are some interesting scenarios that you need to be careful of when doing string concatenation.

Example:

```
# can't concatenate str to int
greet = 'hi'
five = 5
message = greet + five
print(message)
```



```
main.py x
1 # cannot concatenate str to int
2 greet = 'hi'
3 five = 5
4 message = greet + five
5 print(message)

Traceback (most recent call last):
  File "main.py", line 4, in <module>
    message = greet + five
TypeError: can only concatenate str (not "int") to str
> |
```

From the figure, it is clear that we can only concatenate strings together otherwise we will get typeError. We will explain this in the following section called type conversion.

## Type Conversion or Type Casting

In Python, the strings are written as **str** and this is a built-in function in Python, but if **str** is a function, can we use brackets as with the other built-in functions? Let's use the **str** function to pass an integer number as a value:

```
# type conversion
print(str(250))
```

If we run this in the browser, the output will be 250, but is 250 an integer number or string? Just to make things clear, let us rewrite the above code and store the result from the function **str** into a variable and then check the type of the variable:

The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # type conversion
2 result = str(250)
3 print(result)
4 print(type(result))
```

To the right of the code editor is a "Console" tab which is selected. The console output is displayed in a dark-themed terminal window:

```
250
<class 'str'>
> □
```

As you can see, the result or the value 250 is from type string. This means that the str() function converts the 250 integer number we initially passed into the str function to string. Because the value of the result variable is string, we cannot do normal mathematical operations like the following:

```
# can only concatenate str (not "int") to str
print(result + 10)
```

The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # type conversion
2 result = str(250)
3 print(result)
4 print(type(result))
5 # error
6 print(result + 10)
```

To the right of the code editor is a "Console" tab which is selected. The console output is displayed in a dark-themed terminal window, showing a traceback:

```
250
<class 'str'>
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    print(result + 10)
TypeError: can only concatenate str (not "int") to str
> □
```

This is called explicit type conversion, where users/developers like us try to convert the variable value from one data type to a required data type. We can do this type conversion with the built-in functions like int(), float(), str(),etc. Let's try to do a type conversion from a string to integer:

```
# type conversion from string to int
num0 = str(5)
print(type(num0))
num2 = 10
num1 = int(num0)
print(type(num1))
result1 = num1 + num2
```

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
print(result1)
1 # type conversion from string
  to int
2 num0 = str(5)
3 print(type(num0))
4 num2 = 10
5 num1 = int(num0)
6 print(type(num1))
7 result1 = num1+ num2
8 print(result1)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the following output:

```
<class 'str'>
<class 'int'>
15
> █
```

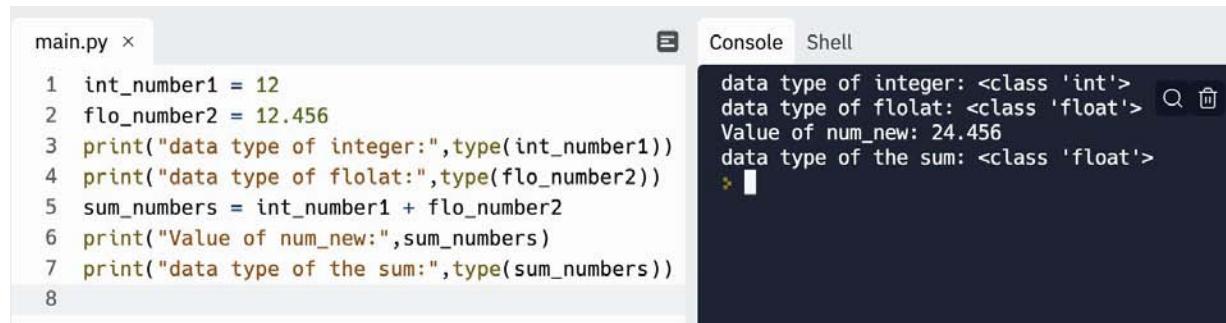
To summarize, type conversion is when we convert one data type to another data type.

## Implicit Type Conversion

We have seen that we can do explicit type conversion or type casting if we use some of the built-in Python functions, but there is another conversion called **implicit type** conversion. This conversion doesn't require us to get involved and use built-in Python functions. The implicit type conversion is automatically done by Python behind the scenes. The Interpreter will do some background checks and calculations just to avoid data loss. Here is one example where Python will automatically convert the integer to float number just to avoid data loss:

```
int_number1 = 12
flo_number2 = 12.456
print("data type of integer:", type(int_number1))
print("data type of flolat:", type(flo_number2))
sum_numbers = int_number1 + flo_number2
print("Value of num_new:", sum_numbers)
print("data type of the sum:", type(sum_numbers))
```

If we run this example:



The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 int_number1 = 12
2 flo_number2 = 12.456
3 print("data type of integer:", type(int_number1))
4 print("data type of float:", type(flo_number2))
5 sum_numbers = int_number1 + flo_number2
6 print("Value of num_new:", sum_numbers)
7 print("data type of the sum:", type(sum_numbers))
```

On the right, a "Console" tab is active, showing the output of the code execution:

```
data type of integer: <class 'int'>
data type of float: <class 'float'>
Value of num_new: 24.456
data type of the sum: <class 'float'>
```

## Escape Sequence

We have discussed that we can create strings if we wrap the text in single and double quotes, but what we didn't discuss is that there are special characters that require us to take some actions. Some of the characters are illegal and we need to use an escape character. For example, if we want to add an apostrophe, we need to use an escape character (\) followed by the character we want to insert (apostrophe).

Let's have a look at the following example:

```
text = 'I'm Rick'
```

The example above is clearly not working, therefore we need to escape it using the backslash:

```
text = 'I\'m Rick'
print(text)
```



The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 text = 'I\'m Rick'
2 print(text)
```

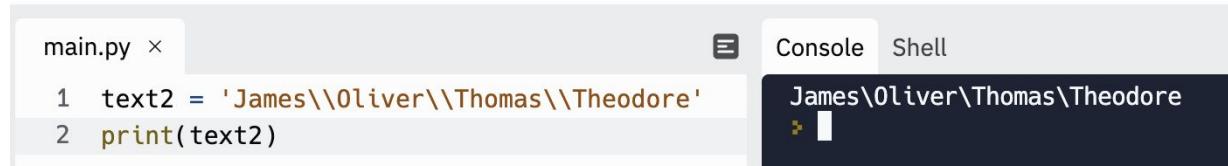
On the right, a "Console" tab is active, showing the output of the code execution:

```
I'm Rick
```

If we have used double quotes to wrap the text, then we can use the apostrophe:

```
text1 = "I'm Rick"  
print(text1)
```

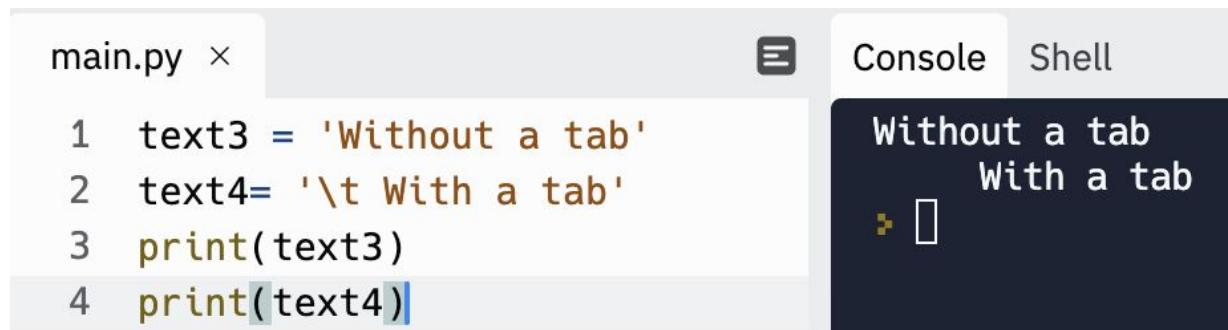
If case we need to include the backslash in our string, then we can escape it like this:



```
main.py ×  
1 text2 = 'James\\Oliver\\Thomas\\Theodore'  
2 print(text2)
```

Console Shell  
James\Oliver\Thomas\Theodore  
▶

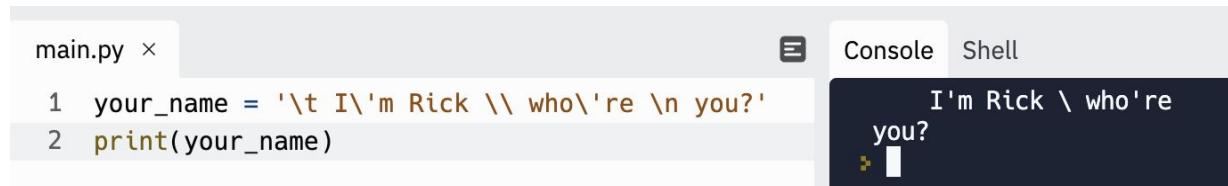
If we want to add a **tab** in front of the string and make an indentation, we can use backslash t (\t)



```
main.py ×  
1 text3 = 'Without a tab'  
2 text4= '\t With a tab'  
3 print(text3)  
4 print(text4)
```

Console Shell  
Without a tab  
With a tab  
▶

If we want to add a **new line** in our strings, we need to use backslash \n:



```
main.py ×  
1 your_name = '\t I'm Rick \\ who're \n you?'  
2 print(your_name)
```

Console Shell  
I'm Rick \ who're  
you?  
▶

If we want to use double quotes inside a string that has already been surrounded by double quotes, then you need to use the escape character \"



```
main.py ×  
1 text5 ="He said \"This is hard\", and  
I believe him"  
2 print(text5)
```

Console Shell  
He said "This is hard", and I believe him  
▶

These are the most important escape characters used in Python:

## Code

\'

## Result

Single Quote

\\\	Backslash
\n	New line
\t	Tab
\r	Carriage Return
\b	Backspace
\f	From Feed
\xhh	Hex Value

## Formatted Strings

If we want to format the strings, we can use the f-strings. They are called f-springs because of the **f** leading character preceding the string literal. We only need to add the letter **f** before the string and that will tell the Python interpreter that we want our string to be formatted. Imagine we are collecting the user details from the registration form:

```
# f-strings to format the strings
first_name = 'James'
last_name = 'Bond'
age = 55
```

```
print(f'Dear {first_name} {last_name}, thank you for being 007 for so many
years!')
```

```
main.py ×
1 # f-strings to format the strings
2 first_name = 'James'
3 last_name = 'Bond'
4 age = 55
5
6 print(f'Dear {first_name} {last_name}, thank
you for being 007 for so many years!')

```

Console Shell

```
Dear James Bond, thank you for being 007 for so many years!
> █
```

In the f-string, we can grab the values from the variables if we put the variable names in the curly braces `{}`. The same can be achieved using the string concatenation, but if you remember, we need to manually add the space between the names and this will be much more complex and longer to write:

```
print('Dear ' + first_name + ' ' + last_name + ', thank you for being 007 for so many years!')
```

## How Strings are Stored?

So far, we learned how integer and float numbers are stored in memory. The strings are stored in memory as a sequence of individual characters. This means that a single character is stored as a string with a length of 1 and can be accessed by its index. In order to understand how strings are stored, we need to look at the following example:

```
my_name = 'Rick S'
```

As you can see, we have a variable `my_name` with type string data. We know the strings are stored as ordered sequence of characters and the first character is 'R' and it will be stored somewhere in memory at location/index zero (0). The following figure explains how the string is stored in memory and each of the characters has its own index starting from zero:

```
my_name = 'Rick S'  
012345
```

Indexes:

This is very powerful because the way the strings are stored makes it easy for us to retrieve a specific character based on its location. How we can get an individual character from a string? We can use the name of the variable followed by the square brackets where we pass the index value. Remember the strings start from position zero, not 1. Let us get the first character of the string above using the following syntax:

```
# String Indexes  
my_name = 'Rick S'
```

```
print(my_name[0])
```

The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # String Indexes
2 my_name = 'Rick S'
3 print(my_name[0])
```

On the right, there is a "Console" tab showing the output of the code execution. The output is:

```
R
```

As you can see, we only need to know the index position to retrieve any character. What will happen if we try to run this code:

```
print(my_name[4])
```

It looks like nothing but we actually got the white space back because even those are stored in memory. On the right side of the following figure, I have highlighted where the empty space is actually printed:

The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # String Indexes
2 my_name = 'Rick S'
3 print(my_name[0])
4 print(my_name[4])
```

On the right, there is a "Console" tab showing the output of the code execution. The output is:

```
R  
  S
```

What will happen if you use an index that is not in the string range? If we do this, we will get an IndexError and this means that the character we try to access by that index simply does not exist:

The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # String Indexes
2 my_name = 'Rick S'
3 print(my_name[0])
4 print(my_name[4])
5 print(my_name[5])
6 print(my_name[6])
```

On the right, there is a "Console" tab showing the output of the code execution. The output is:

```
R  
  S  
Traceback (most recent call last):  
  File "main.py", line 6, in <module>  
    print(my_name[6])  
IndexError: string index out of range
```

This happens very often. Beginners usually make these mistakes because they keep forgetting that the first character is stored at index zero (0)

## Slicing Strings – get a substring of a string

We've learned that we can extract individual characters based on their indexes, now it is time to learn how to return multiple characters based on a specified range. This is called string slicing or extracting a substring of a string and we only need to use the following syntax:

[start:stop]

The first index is called the **start index** and it is separated by a colon from the second index called **stop**. Let's try the following example:

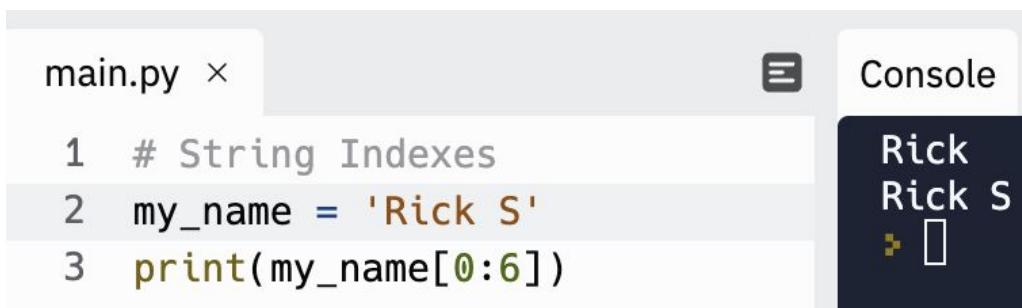


```
main.py ×
1 # String Indexes
2 my_name = 'Rick S'
3 print(my_name[0:4])
```

Console  
Rick

You should know that the character at the stop index will not be included in the output. In the above example, that would be the white space. If we want to get the full string, we need to write:

`print(my_name[0:6])`



```
main.py ×
1 # String Indexes
2 my_name = 'Rick S'
3 print(my_name[0:6])
```

Console  
Rick  
Rick S

From the picture above, we can see that the stop or the end index is 6 but the indexes for these particular string end at position 5. This will not throw an error because the end index will never be used in the output.

## String-slicing with stepover

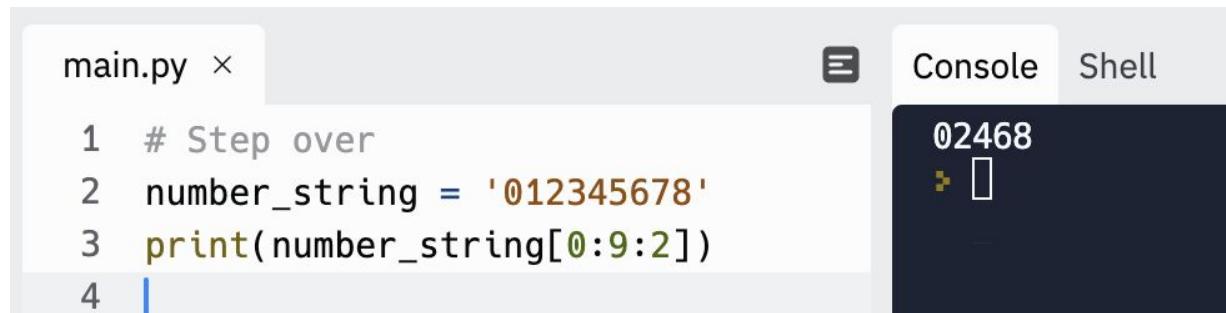
Here is the stepover syntax:

[start:stop:stepover]

In the previous section, we have learned how to slice a string based on the start and stop index but now we can include a third value which is called stepover. The default value of the stepover is 1 and we don't have to write it explicitly. The step over indicates the step size and as I mentioned, the default value is 1, meaning we are going character by character but if we increase the size of the step, then we will start skipping and the output will be different. Let's take a look at the following example:

```
# Step over
number_string = '012345678'
print(number_string[0:9:2])
```

From the example, we can see that now the step value is 2 and it will start from the first position, then step over 2 times and print 2 then step over 2 more times and print 4, and so on. Here is our output:



The screenshot shows a Python development environment. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # Step over
2 number_string = '012345678'
3 print(number_string[0:9:2])
4 |
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the program:

```
02468
```

Working with string slicing is very interesting because there are different scenarios that we should be aware of. For example, if we only include the start index and omit the end index, for example like this:

number\_string[ 1 : ]

Can you guess what will happen? Here is the output:

The screenshot shows a Python development environment. On the left, there's a code editor window titled "main.py" containing the following code:

```
1 # Step over
2 number_string = '012345678'
3 print(number_string[1:])
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the printed string:

```
12345678
> █
```

This basically means we need to start at position 1 and then because there is no stop index it will go until the end. This is perfect when we have a long string and we don't know what the last index number is. We use this because we know it will go on until the end. Another interesting scenario is when we omit the start index but include the index of the end/stop position like this:

`number_string[:4]`

Can you guess the output? If not, here is the screenshot:

The screenshot shows a Python development environment. On the left, there's a code editor window titled "main.py" containing the following code:

```
1 # Step over
2 number_string = '012345678'
3 print(number_string[:4])
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the printed string:

```
0123
> █
```

This means starting from zero although the value is missing and going until index 4 but not including the value at the last fourth position. What will happen if we omit the first stop and the last end index and only include the stepover like this:

`number_string[::1]`

This will print the entire string because it starts at the default position zero, ends where the string ends, and stepping over will be the default value 1. Here is another example of when the two (start, stop) indexes are missing:

```
main.py ×
```

```
1 # Step over
2 number_string = '012345678'
3 print(number_string[::-1])
4 print(number_string[::2])
```

```
Console Shell
```

```
012345678
02468
> █
```

Great! We have learned a lot about string slicing, so let's see what will happen if we include negative values as well:

```
main.py ×
```

```
1 # Step over
2 number_string = '012345678'
3 print(number_string[-1])
```

```
Console Shell
```

```
8
> █
```

The output is 8 because in Python the negative index means start at the end of the string instead of position zero. One of the most common operations with negative indexes is this one:

```
main.py ×
```

```
1 # Step over
2 number_string = '012345678'
3 print(number_string[-1])
4 print(number_string[::-1])
```

```
Console Shell
```

```
8
876543210
> █
```

This is a perfect example of when we need a string to be reversed. The negative values can even be included for the skip value.

## String immutability

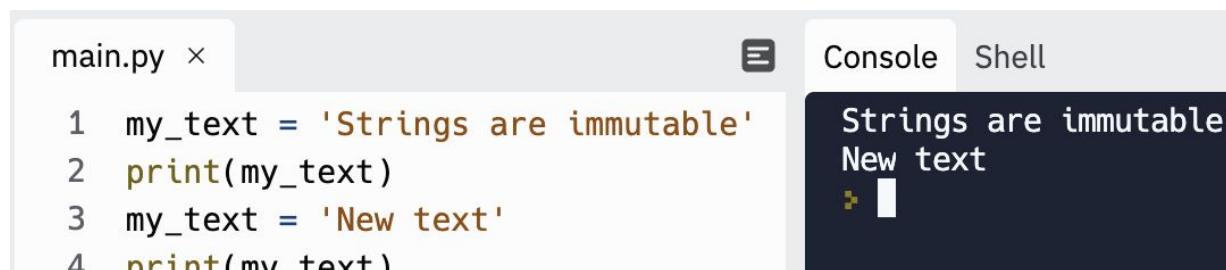
In the previous sections, we have learned a lot about strings - how we can create them, how they are stored in memory, and how we can access the individual characters. String slicing is even more useful because we can slice the string however we like and can get a specific substring if we specify the correct range. In this section, we are going to learn about an important concept for programming called **immutability**. What does **immutability** mean? Once the Strings in Python are created, they are immutable. You should not confuse mutability with value reassignment. For example, we can create a string with a simple text like this:

```
my_text = 'Strings are immutable'  
print(my_text)
```

We know we can reassign a new value to a variable and it will work:

```
my_text = 'Strings are immutable'  
print(my_text)  
my_text = 'New text'  
print(my_text)
```

If we run this on repl.it, we will get two different outputs:



The screenshot shows a development environment with a code editor and a terminal window. The code editor contains a file named 'main.py' with the following content:

```
1 my_text = 'Strings are immutable'  
2 print(my_text)  
3 my_text = 'New text'  
4 print(my_text)
```

The terminal window, labeled 'Console', shows the output of the code execution:

```
Strings are immutable  
New text
```

But the strings are immutable, meaning we cannot change the individual characters from the strings. We know that in Python the strings can be accessed by their indexes, for example, if I want to access the first letter of the `my_text`, I need to use the index zero (0) like this:

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1
2 my_text = 'New text'
3 print(my_text)
4 print(my_text[0])
-
```

On the right, a "Console" tab is active, displaying the output of the code execution:

```
New text
N
```

Great! We got the expected result but can I change the first character to a new character like this:

```
my_text[0] = 'K'
print(my_text)
```

If we run this code, we will get a `TypeError` saying we cannot assign a new value to string item:

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code, with line 6 highlighted:

```
1
2 my_text = 'New text'
3 print(my_text)
4 print(my_text[0])
5 #this will not work
6 my_text[0] = 'K'
7 print(my_text)
```

On the right, a "Console" tab is active, displaying the output of the code execution, which includes a traceback:

```
New text
N
Traceback (most recent call last):
  File "main.py", line 6, in <module>
    my_text[0] = 'K'
TypeError: 'str' object does not support item assignment
> |
```

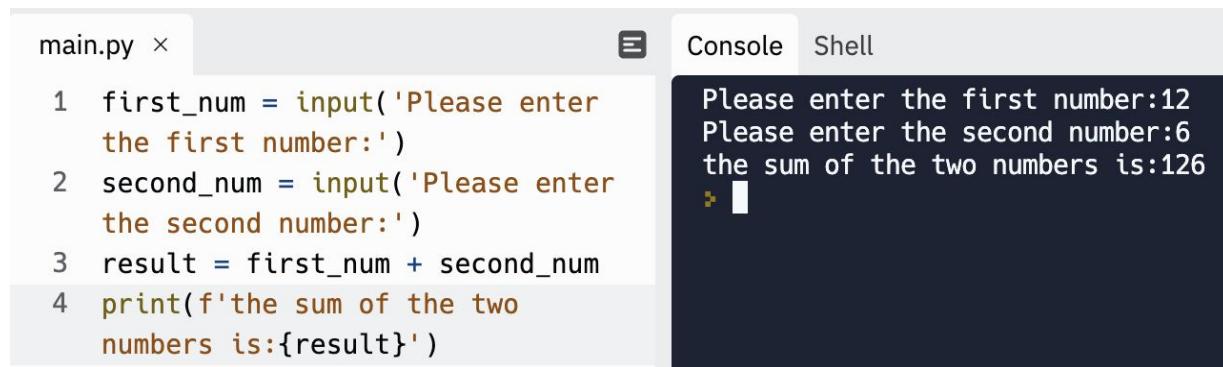
As we can see from the figure above, string immutability is not possible because we cannot change the value of individual characters once created. The individual characters are assigned with indexes and stored in the memory. The only way we can do this is to completely reassign a new value to the `my_text` variable. The Python interpreter will remove the old content from the memory and add new content and that is why reassigning is possible. To summarize, we cannot reassign part of the string like individual characters because the Strings in Python are immutable, but we can assign a new value to the existing string because Python completely deletes the old string from memory.

## Practice time

So far, we have covered a lot of the basic Python features but we still have a lot of ground to cover and I think you are ready to write your first mini-Python program. Imagine you want to create a small program that will ask the user to input two numbers and calculate the sum of the numbers. How can we write this program? In order to prompt the user to enter values, we need to use the built-in function called `input()`. The way the `input()` function works is that it waits for the user to input or in our case to write the numbers. Here is the entire code:

```
first_num = input('Please enter the first number:')  
second_num = input('Please enter the second number:')  
result = first_num + second_num  
print(f'the sum of the two numbers is:{result}')
```

Let's run the above code and enter 12 and 6 as numbers:



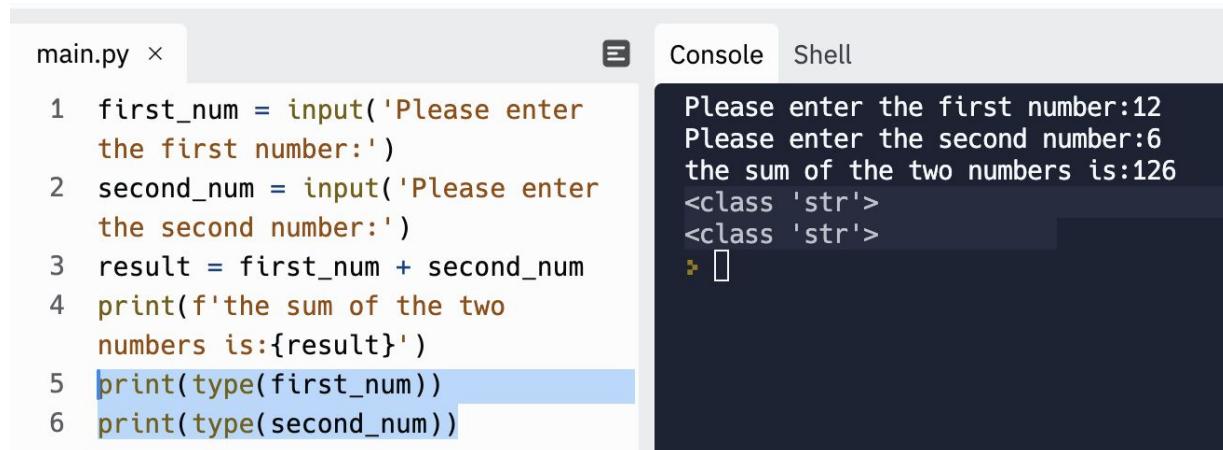
The screenshot shows a Python development environment with a code editor and a terminal window. The code editor contains the following Python script:

```
main.py ×  
1 first_num = input('Please enter  
the first number:')  
2 second_num = input('Please enter  
the second number:')  
3 result = first_num + second_num  
4 print(f'the sum of the two  
numbers is:{result}')
```

The terminal window (Console tab) shows the output of running the script:

```
Please enter the first number:12  
Please enter the second number:6  
the sum of the two numbers is:126  
▶
```

If we look at the code above, the sum is 126 but should be 18, right? What is happening? If you are not sure why is this happening, we can always check the type of the variables (`first_num`, `second_num`):



The screenshot shows the same Python development environment. The code editor has been modified to include two additional `print` statements to display the types of the variables:

```
main.py ×  
1 first_num = input('Please enter  
the first number:')  
2 second_num = input('Please enter  
the second number:')  
3 result = first_num + second_num  
4 print(f'the sum of the two  
numbers is:{result}')  
5 print(type(first_num))  
6 print(type(second_num))
```

The terminal window (Console tab) shows the output:

```
Please enter the first number:12  
Please enter the second number:6  
the sum of the two numbers is:126  
<class 'str'>  
<class 'str'>  
▶
```

The values of the two variables are from type string and when we use the ‘+’ operator, the Python interpreter will think we are trying to make String concatenation. In order to fix this simple problem, we need to do a type conversion that is explicit using the built-in int() function:

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 first_num = input('Please enter  
the first number:')  
2 second_num = input('Please enter  
the second number:')  
3 result = int(first_num) +  
    int(second_num)  
4 print(f'the sum of the two  
numbers is:{result}')  
5 print(type(first_num))  
6 print(type(second_num))
```

The line "3 result = int(first\_num) + int(second\_num)" is highlighted with a blue selection bar.

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the following output:

```
Please enter the first number:12  
Please enter the second number:6  
the sum of the two numbers is:18  
<class 'str'>  
<class 'str'>
```

From the figure above, the result is 18 and you should know that the values coming from the input() function are strings, not numbers, therefore we need to convert them to the right data type.

## Functions and Methods – What differentiates them?

In this chapter, we saw few built-in functions like str(), int(), round(), abs(), bin(), print(), and there are more built-in functions in Python. Here is the complete list of the built-in functions:

## Built-in Functions

<b>A</b>	<b>E</b>	<b>L</b>	<b>R</b>
abs() aiter() all() any() anext() ascii()	enumerate() eval() exec()	len() list() locals()	range() repr() reversed() round()
<b>B</b>	<b>F</b>	<b>M</b>	<b>S</b>
bin() bool() breakpoint() bytearray() bytes()	format() frozenset()	map() max() memoryview() min()	set() setattr() slice() sorted() staticmethod() str() sum() super()
<b>C</b>	<b>H</b>	<b>O</b>	<b>T</b>
callable() chr() classmethod() compile() complex()	hasattr() hash() help() hex()	object() oct() open() ord()	tuple() type()
<b>D</b>	<b>I</b>	<b>P</b>	<b>V</b>
delattr() dict() dir() divmod()	id() input() int() isinstance() issubclass() iter()	pow() print() property()	vars()
			<b>Z</b>
			zip()
			—
			__import__()

<https://docs.python.org/3/library/functions.html>

Let's explore some of the very useful built-in functions you will use as a developer. The first one is called `len()`, which stands for length:

**len()**

This built-in function will return the length or the number of items of an object. We still haven't covered what Objects are in Python but they are the most important thing because basically everything is an Object in Python. The argument that we pass inside the len() function can be a sequence like a String. Because we learned a lot about String data type, we can use this function and pass the string as an argument like this:

```
#1) len() - returns the length
```

```
text = 'A string'
```

```
print(len(text));
```

If we run the code, the output will be 8:



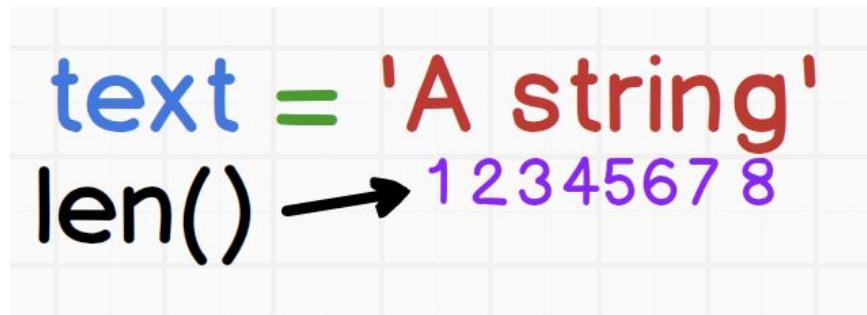
The screenshot shows a Python development environment. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 #1) len() - returns the lenght
2 text = 'A string'
3 print(len(text))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
8
```

This means that the len() function will give us the actual length starting the count from 1 not from zero like with the indexes.

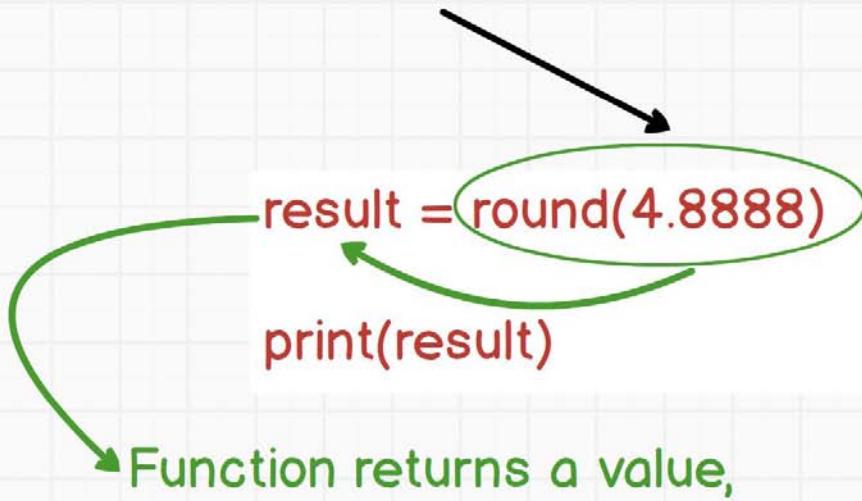


As you can see, the function we are using have only one purpose and that is to return some value. The `len()` function is used to return the actual length of an Object like the String. We have used the term functions but I have not explained what functions are. You can think of functions as a unit that includes one or more lines of code or statements. The purpose of a function is to run a piece of code with the data we provide. The data we provide is called arguments, and we are passing them directly into the function. As we know, functions have parenthesis like this `()`. Inside the parenthesis is where we pass the arguments or the data. We have seen a few built-in functions but before we start passing arguments, we need to know what that function accepts because we cannot pass any data we can think of. We can create our own functions as well and then we know exactly what data we need to pass but for the built-in functions we need to read the documentation first and only then we can start using them with confidence. We will learn how to create our own functions, how to define them, and how to call them, but at this stage, we just need to know what they are. To summarize, when we use a function, we need to know what arguments/data we should provide, for example, here is the `round` function:

`round(number[, ndigits])`

This `round()` function accepts two arguments (`number`, `ndigits`) and if you read the Python documentation, you will know that the second argument (`ndigits`) can be omitted or is an optional argument. A function call is when we use the function name with the arguments:

## A Function Call



and that value is stored in the variable result

The functions that are created by us are not built-in functions, they are called user-defined functions. Let's talk about methods. What are methods in Python? Methods in Python are the exact same thing as functions, there is no difference between them. The only difference is how they are called.

Methods, in simple words, belong to something or someone. This something is actually the Objects that we will cover in the future. For example, Strings have their own built-in methods but these methods can only be called directly by strings and not as we call the print or input function. So how do strings call these built-in methods? Strings can call the methods using the ‘dot syntax.’ Let's go over a few handy String methods starting with the method called upper.

### Upper()

This method will convert the given string to upper case.

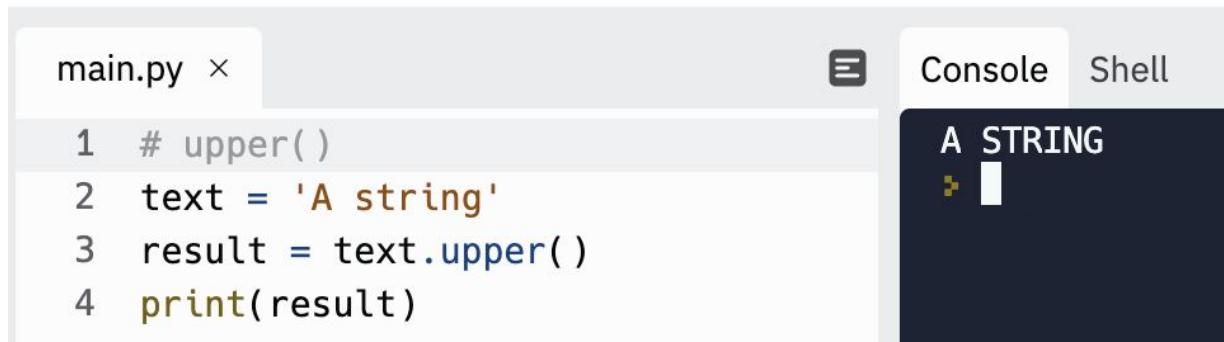
Here is one example:

```
# upper()
text = 'A string'
result = text.upper()
print(result)
```

As you can see, we use the String that is stored in the variable called text to call/invoke the method upper(). This can be also written like this:

```
print('A string'.upper())
```

The output is:



A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # upper()
2 text = 'A string'
3 result = text.upper()
4 print(result)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is selected and shows the output of the code execution:

```
A STRING
> █
```

Now you know the methods are functions but they are called by a specific Object like the strings. This means we cannot call the upper() method like this:

```
upper()
```

The methods for strings will not work for other data types, they belong to strings only.

## Capitalize()

The next interesting method that is used a lot with strings is the capitalize method. This method will turn the first character into a capital letter:

```
#capitalize()
print('hi there'.capitalize())
```



A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 #capitalize()
2 print('hi there'.capitalize())
```

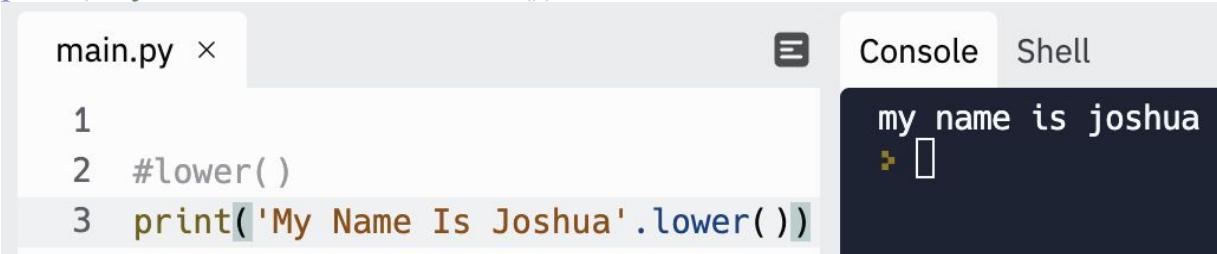
On the right, there are two tabs: "Console" and "Shell". The "Console" tab is selected and shows the output of the code execution:

```
Hi there
> █
```

## lower()

This method will convert the given string into a lower case:

```
#lower()  
print('My Name Is Joshua'.lower())
```



```
main.py × 1  
2 #lower()  
3 print('My Name Is Joshua'.lower())
```

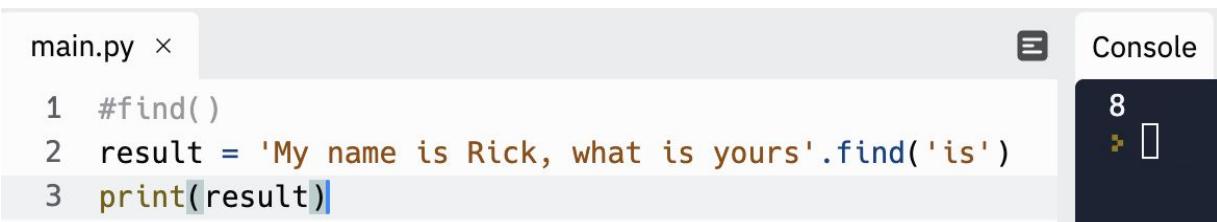
Console Shell

```
my name is joshua  
▶ □
```

## find()

This method will find the first occurrence(index) of the specified substring. This, in simple words, means that this method will check if a certain substring exists in a given string:

```
#find()  
result = 'My name is Rick, what is yours'.find('is')  
print(result)
```



```
main.py × 1  
2 #find()  
3 result = 'My name is Rick, what is yours'.find('is')  
3 print(result)
```

Console

```
8  
▶ □
```

## replace()

We use this method when we want to replace a part of the string with a new value. This method returns the entire string. The syntax of this method is more complex and requires passing two values:

```
string.replace(oldvalue, newvalue)
```

The first argument is the substring/value you want to replace and the second argument is the new value. For example, let's replace 'Rick' with 'Mick':

```
#replace()  
result = 'My name is Rick, what is yours'.replace('Rick', 'Mick')  
print(result)
```

The screenshot shows a Python IDE interface. On the left, a code editor window titled "main.py" contains the following code:

```
1 #replace()
2 result = 'My name is Rick, what is
3         yours'.replace('Rick', 'Mick')
4 print(result)
```

On the right, a "Console" tab is active, showing the output of the code execution:

```
My name is Mick, what is yours
> █
```

Great! It works, but wait, we said strings are immutable but it seems this method changes the part of a string. What is going on? This is confusing because of the way I have written the code to show you that the original string cannot be changed. Let's re-write the code like this:

```
#replace()
original_string = 'My name is Rick, what is yours'
result = original_string.replace('Rick', 'Mick')
print(original_string)
print(result)
```

The screenshot shows a Python IDE interface. On the left, a code editor window titled "main.py" contains the following code:

```
1 #replace()
2 original_string = 'My name is Rick, what is yours'
3 result = original_string.replace('Rick', 'Mick')
4 print(original_string)
5 print(result)
```

On the right, a "Console" tab is active, showing the output of the code execution:

```
My name is Rick, what is yours
My name is Mick, what is yours
> █
```

As you can see, the original\_string is not changed. It is important to understand that the replace() method returns a value that is modified so we can store that value in a variable (result) but the replace method would not affect the original string.

## join()

This method will join or concatenate a list of strings together and create a new string with the desired delimiter. Let me show you what this means in the following example:

```
#join()
result = "-".join(["This", "book", "is", "awesome!"])
print(result)
```

The screenshot shows a Python development environment. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 #join()
2 result = "-".join(["This", "book", "is", "awesome!"])
3 print(result)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution:

```
This-book-is-awesome!
```

The join method accepts a List of strings and we will learn about Lists as a separate data type. There are other useful methods you can find in the Python documentation, but for now, these are the essential methods you should know.

## Python IN Keyword

Just like the built-in functions and methods in Python, we have reserved keywords as well. These keywords are known as identifiers and cannot be used as ordinary identifiers. The keywords must be spelled exactly as they are. In this section, we will talk about the keyword 'in' which is very useful because it has many applications. The 'in' reserved keyword can be used with Strings like this:

```
# In Keyword with Strings
long_string = 'Andy is 5 years old!'
print('5' in long_string)
```

The screenshot shows a Python development environment. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # In Keyword with Strings
2 long_string = 'Andy is 5 years old!'
3 print('5' in long_string)
4
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution:

```
True
```

This keyword has many purposes and one of them is to check if the part of a string is present in the sequence. We are trying to find if '5' is present in the long\_string sequence of characters. As we can see, the value '5' is present and therefore it returns the output True. This value 'True' belongs to another data type in Python called Boolean. We will cover this data type in the next section. Here is a list of reserved keywords in Python:

## 2.3.1 Keywords

The following identifiers are used as reserved words, or *keywords* of the

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

<https://docs.python.org/2.5/ref/keywords.html>

## Booleans in Python (data type)

Boolean is another data type in Python. The Boolean in Python represents only one of the two possible values: True or False. The Boolean is not unique to Python, we have Boolean in almost all programming languages. This allows us to set a variable to be True or False like in the following example:

```
# Boolean True or False  
is_true = True  
print(is_true)
```

```
is_false = False  
print(is_false)
```

```
main.py ×
```

```
1 # Boolean True or False
2 is_true = True
3 print(is_true)
4
5 is_false = False
6 print(is_false)
```

```
Console Shell
```

```
True
False
> █
```

The power of Boolean will be seen when we learn how to make decisions in Python. The decision statements like the if-statements have a condition and that condition will always be evaluated to Boolean True or False. We can also evaluate an expression to Boolean like in this example:

```
#compare values
print(2 > 1)
print (10 == 10)
print(10 < 6)
```

```
main.py ×
```

```
1 # Boolean True or False
2 print(2 > 1)
3 print (10 == 10)
4 print(10 < 6)
```

```
Console Shell
```

```
True
True
False
> █
```

This topic will be explored more in one of the future sections. Let's run this example and see the outputs it creates:

```
print(bool(0))
print(bool(1))
```

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 # Boolean True or False
2 print(bool(0))
3 print(bool(1))
```

On the right, a "Console" tab displays the output of the code:

```
False
True
```

Boolean zero(0) will return False and Boolean 1 will return True. The Boolean function will convert the integer values to either True or False and it will be discussed in more detail in the conditional logic section.

## Lists in Python (Data type)

As you can see, we are flying through the data types, and in between, we are learning so many new and fundamental Python features. That is why I decided to create this guide in this particular order because it's more natural for us to learn. I could have listed all of the data types in the beginning and then explain the rest of the features but I find this way much easier to learn. Another important data type in Python is Lists. In Python, lists are ordered sequence of objects that can be from any data type. If you have done any other programming language, you will notice that what I'm about to teach you is the same as the arrays in other programming languages so let's create a variable called my\_list:

```
# Lists - data type
my_list = [1,2,3,4,5,6]
my_list1 = ['a','b','c','d']
my_list2 = ['a',2,'c',3]

print(my_list)
print(my_list1)
print(my_list2)
```

The screenshot shows a Python development environment. On the left, the code file `main.py` contains the following Python script:

```
1 # Lists - data type
2 my_list = [1,2,3,4,5,6]
3 my_list1 = ['a','b','c','d']
4 my_list2 = ['a',2,'c',3]
5
6 print(my_list)
7 print(my_list1)
8 print(my_list2)
```

The right side of the interface has two panes: `Console` and `Shell`. The `Console` pane displays the output of the `print` statements:

```
[1, 2, 3, 4, 5, 6]
['a', 'b', 'c', 'd']
['a', 2, 'c', 3]
```

As I mentioned, the lists can hold an ordered sequence of data. So far, we have used the variables to store only a single value but now the lists are making the variables a complex container where we store multiple data of different types. As you can see, the variable called `my_list` can hold an ordered sequence of integer values. The second variable called `my_list1` is an ordered sequence of values that belong to the String data type. The last variable has mixed data type and this is normal behavior for storing different types of data in one single container. When we create lists, we need to use square brackets. The data inside the brackets are called items or elements. The items or elements are separated by a comma. This data type (lists) is exactly the same as the arrays in JavaScript. Lists are a collection of items or ordered sequence of objects. Lists are also our first **Data Structure**. The Data structure allows us to store the data in an organized container and that is why we need to use those square brackets because we need to contain the data in one single unit. Data Structure is a very important concept because it allows us to contain the information in an organized manner in one container or box. If we use the `print` function and pass the list variable, we can see that it will print all of the items in square brackets but how can we access these individual items/elements from the list? Well, similar to Strings, we can use the index position. The List indexes start from position zero, just like Strings:

```
#individual items
print(my_list[0])
```

```
print(my_list1[1])
print(my_list2[2])
```

A screenshot of a Python IDE interface. On the left, the code editor shows a file named 'main.py' with the following content:

```
1 # Lists - data type
2 my_list = [1,2,3,4,5,6]
3 my_list1 = ['a','b','c','d']
4 my_list2 = ['a',2,'c',3]
5
6 #individual items
7 print(my_list[0])
8 print(my_list1[1])
9 print(my_list2[2])
```

The 'Console' tab is selected at the top right. The output window shows the following results:

```
1
b
c
> 
```

If we use an index that does not match the number of the list items, we will end up getting an **IndexError**:

A screenshot of a Python IDE interface. On the left, the code editor shows a file named 'main.py' with the following content:

```
1 # Lists - data type
2 my_list = [1,2,3,
3 my_list1 = ['a',
4 my_list2 = ['a',2
5
6 #individual items
7 print(my_list[0])
8 print(my_list1[1])
9 print(my_list2[2])
10 print(my_list2[4])
```

The 'Console' tab is selected at the top right. The output window shows the following results and an error message:

```
1
b
c
Traceback (most recent call last):
  File "main.py", line 10, in <module>
    print(my_list2[4])
IndexError: list index out of range
> 
```

The **IndexError** indicates that the index we used is out of range. The way the list items are stored in memory is in a sequence or one after another. We can check the type of the list using the `type` function like this:

```
#list type
print(type(my_list))
```

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 # Lists - data type
2 my_list = [1,2,3,4,5,6]
3 my_list1 = ['a','b','c','d']
4 my_list2 = ['a',2,'c',3]
5
6 #type
7 print(type(my_list))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution:

```
<class 'list'>
> []
```

If we want to know the actual number of items/elements we have in a list, we can use the `len()` built-in function. The `len()` function will return the actual number of items and don't confuse them with indexes because they start from zero and `len()` starts counting from 1:

```
#length of list
print(len(my_list))
```

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 # Lists - data type
2 my_list = [1,2,3,4,5,6]
3 #length of list
4 print(len(my_list))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution:

```
6
> []
```

## List Slicing

We already saw how slicing works on Strings and we can do the same with List items. We can define where the slicing should start and where it should end. Same as with Strings, the last item will not be included in the output:

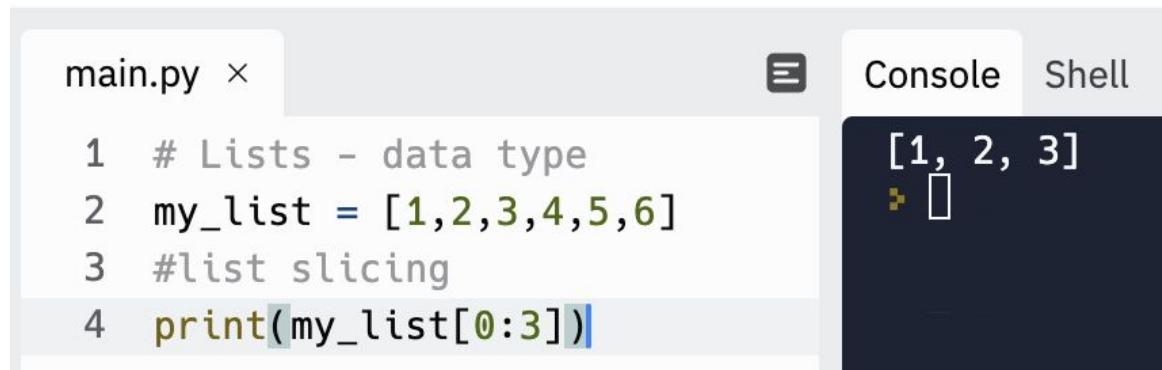
Syntax:

```
#list slicing:  
listname[start:stop]
```

For example, let's slice the first list and get the first 3 items:

```
#list slicing  
print(my_list[0:3])
```

The output will be:



```
main.py ×
```

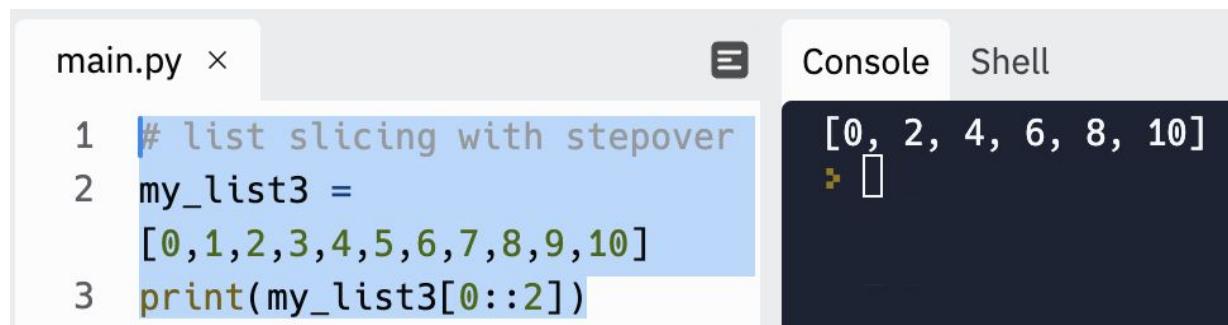
```
1 # Lists - data type  
2 my_list = [1,2,3,4,5,6]  
3 #list slicing  
4 print(my_list[0:3])
```

```
Console Shell
```

```
[1, 2, 3]  
▶ □
```

We can also include the step size if we want. Let's try an example where the step size is two:

```
# list slicing with stepover  
my_list3 = [0,1,2,3,4,5,6,7,8,9,10]  
print(my_list3[0::2])
```



```
main.py ×
```

```
1 # list slicing with stepover  
2 my_list3 =  
3 [0,1,2,3,4,5,6,7,8,9,10]  
3 print(my_list3[0::2])
```

```
Console Shell
```

```
[0, 2, 4, 6, 8, 10]  
▶ □
```

The example above means the slicing should start from position zero, and because the stop index is omitted, we are basically saying go till the end with a step size of 2. Are Strings different from the Lists in Python? Yes, they are different but they also share a lot of similarities. The biggest difference between both is that Strings are immutable, this means that once a String is created, we cannot change individual characters. Lists, however, are mutable. This means that we can change the List individual items like this:

```
#Lists are mutable  
my_list3[0] = -1  
print(my_list3)
```

A screenshot of a Python development environment. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # list slicing with stepover  
2 my_list3 =  
3     [0,1,2,3,4,5,6,7,8,9,10]  
4  
5 #Lists are mutable  
6 my_list3[0] = -1  
7 print(my_list3)
```

On the right, there is a "Console" tab showing the output of the code execution:

```
[-1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
▶ []
```

Same as with Strings, List slicing returns a result that we can store in a variable and the slicing does not affect the original list. This is good when we want to create a copy of the original list:

```
#copy of a list  
string_list = ['one', 'two', 'three']  
string_list_copy = string_list[0:]  
print(string_list_copy)
```

A screenshot of a Python development environment. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 #copy of a list  
2 string_list =  
3     ['one', 'two', 'three']  
4 string_list_copy =  
5     string_list[0:]  
6 print(string_list_copy)
```

On the right, there is a "Console" tab showing the output of the code execution:

```
['one', 'two', 'three']  
▶ []
```

As we can see, we have copied the first list into the new list successfully. This means that after the copy is created, we can mutate some of the elements in the `string_list_copy` without affecting the original list:

The screenshot shows a Python development environment with two tabs: "main.py" and "Console". The code in "main.py" is as follows:

```
1 # add new element to the copied list
2 string_list = ['one','two','three']
3 string_list_copy = string_list[0:]
4 string_list_copy[2] = 'four'
5 print(string_list)
6 print(string_list_copy)
```

The "Console" tab shows the output:

```
['one', 'two', 'three']
['one', 'two', 'four']
```

Great! The two lists are independent of each other; this means they are stored in different locations in the memory. Most beginner developers will try creating a copy of a list using the assignment operator like this:

```
#create a copy using assignment operator
string_list1 = ['one','two','three']
string_list_copy1 = string_list1
print(string_list_copy1)
```

The screenshot shows a Python development environment with two tabs: "main.py" and "Console". The code in "main.py" is as follows:

```
1 #create a copy using assignment operator
2 string_list1 = ['one','two','three']
3 string_list_copy1 = string_list1
4 print(string_list_copy1)
```

The "Console" tab shows the output:

```
['one', 'two', 'three']
```

Well, this looks like it's working but it's not. Now let's modify the last element of the newly copied list and print both lists at the same time:

```
#create a copy using the assignment operator
string_list1 = ['one','two','three']
string_list_copy1 = string_list1
string_list_copy1[2] = 'four'
print(string_list_copy1)
print(string_list1)
```

```

main.py ×

1 #create a copy using assignment operator
2 string_list1 = ['one','two','three']
3 string_list_copy1 = string_list1
4 string_list_copy1[2] = 'four'
5 print(string_list_copy1)
6 print(string_list1)

```

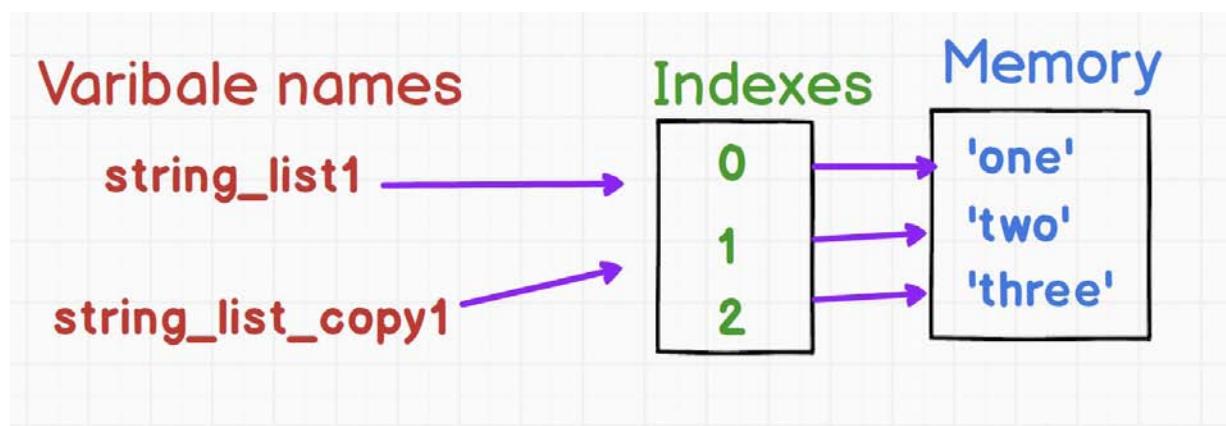
Console Shell

```

['one', 'two', 'four']
['one', 'two', 'four']
> 

```

What is happening here? We have changed the `string_list_copy1` and the change affected the original list as well. This is happening because the `string_list_copy1` is not a copy of the `string_list1` but a reference to the `string_list1`. The variables are just a pointer to the values stored in the memory. This means the `string_list1` and `string_list_copy1` point to the same values in the memory:



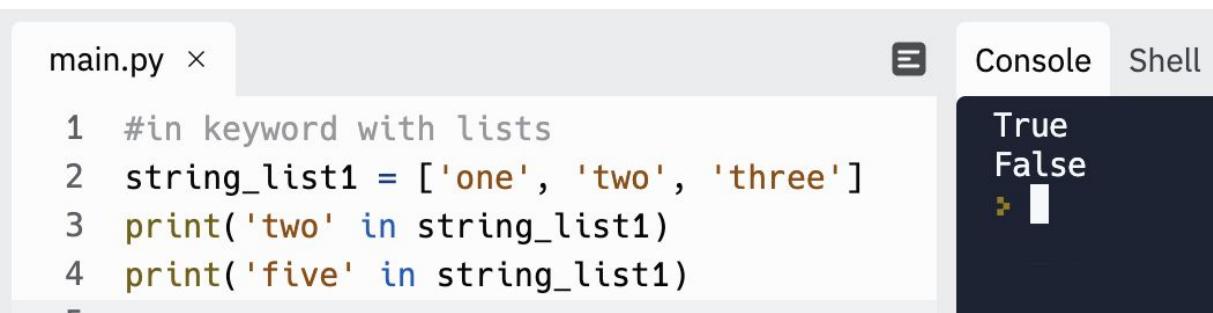
## Python IN Keyword with Lists

Same as in Strings, we can use the ‘in’ keyword to find if an item is present in the list. The output will be Boolean value True or False:

```

#in keyword with lists
string_list1 = ['one', 'two', 'three']
print('two' in string_list1)
print('five' in string_list1)

```



```
main.py ×
```

```
1 #in keyword with lists
2 string_list1 = ['one', 'two', 'three']
3 print('two' in string_list1)
4 print('five' in string_list1)
```

```
Console Shell
```

```
True
False
> █
```

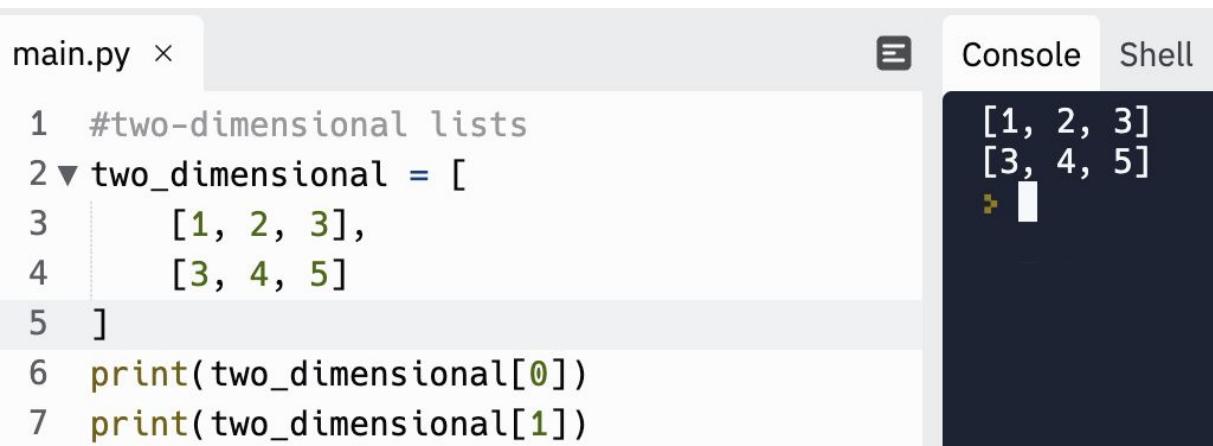
This is a very useful keyword because we can find if the item is present in a very long list.

## Multi-Dimensional Lists

The list examples we have covered so far are one-dimensional. Multi-dimensional lists are lists within lists and there are other data types that are a better choice when it comes to creating multi-dimensional levels:

```
# two_dimensional lists
two_dimensional = [
    [1, 2, 3],
    [3, 4, 5]
]
```

The example above is a two-dimensional list also known as matrix. It can be used in different applications like image processing or machine learning. How can we access the elements inside this list? Here is an example:



```
main.py ×
```

```
1 #two-dimensional lists
2 ▼ two_dimensional = [
3     [1, 2, 3],
4     [3, 4, 5]
5 ]
6 print(two_dimensional[0])
7 print(two_dimensional[1])
```

```
Console Shell
```

```
[1, 2, 3]
[3, 4, 5]
> █
```

If we want to get an element from some of the inner lists, we can use the double square brackets `[] []`. The first bracket will receive an index that will indicate which of the inner list we are trying to access and in the second bracket, we pass the index of the individual list item:

```
#first inner list elements
print('First Inner List Items')
print(two_dimensional[0][0])
print(two_dimensional[0][1])
print(two_dimensional[0][2])
print('Second Inner List Items')
#second inner list elements
print(two_dimensional[1][0])
print(two_dimensional[1][1])
print(two_dimensional[1][2])
```

The screenshot shows a Python development environment with two panes. On the left, the code editor displays `main.py` with the following content:

```
1 #two-dimensional lists
2 two_dimensional = [[1, 2, 3], [3, 4, 5]]
3 #the two lists
4 print(two_dimensional[0])
5 print(two_dimensional[1])
6 #first inner list elements
7 print('First Inner List Items')
8 print(two_dimensional[0][0])
9 print(two_dimensional[0][1])
10 print(two_dimensional[0][2])
11 print('Second Inner List Items')
12 #second inner list elements
13 print(two_dimensional[1][0])
14 print(two_dimensional[1][1])
15 print(two_dimensional[1][2])
```

On the right, the terminal window (Console tab) shows the execution results:

```
[1, 2, 3]
[3, 4, 5]
First Inner List Items
1
2
3
Second Inner List Items
3
4
5
> []
```

As you can see, we have accessed the inner list items with the indexes we passed in the second pair of square brackets. If we use a three-dimensional list, we need to use another pair of square brackets if we want to access the inner list items.

# Lists Methods

Same as the String data type, Lists have its own methods and we need to use the dot notation in order to call these methods. There are a few Lists methods that we can use and here is the link to where you can read all about them:

<https://docs.python.org/3/tutorial/datastructures.html>

We will cover the most used ones. Let's start with the first one called `append()`

## append() method

We simply use this method to add an element at the end of the list:

```
# 1 append
my_list = ['one', 'two', 'three'];
my_list.append('four')
print(my_list); ['one', 'two', 'three', 'four']
```



The screenshot shows a Python development environment. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # 1 append
2 my_list = ['one', 'two', 'three']
3 my_list.append('four')
4 print(my_list)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution:

```
['one', 'two', 'three', 'four']
```

It is important to know that this method will modify the existing array by adding the new element at the end.

## insert() method

This method will insert a new element into the list at the specified location. This means we need to provide the position/index where we want the new item to be inserted. For example, let's add the element 'five' at the end of `my_list`:

```
# 2 insert
my_list.insert(4,'five')
```

```
print(my_list)
```

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # 1 append
2 my_list = ['one', 'two', 'three'];
3 my_list.append('four')
4 # 2 insert
5 my_list.insert(4, 'five')
6 print(my_list)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
['one', 'two', 'three', 'four', 'five']
```

Insert method, just like the append method, will add a new element to the list at a specified index and will modify the original list.

## extend() method

This method takes an iterable as an argument. An iterable means something we can iterate or loop over. Extend method works in a way that it appends all of the items from the iterable to the items in the original list. I know it sounds confusing but here is an example:

```
# 3 extend
my_list.extend(['six','seven'])
print(my_list)
```

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # 1 append
2 my_list = ['one', 'two', 'three'];
3 my_list.append('four')
4 print(my_list)
5 # 2 insert
6 my_list.insert(4, 'five')
7 print(my_list)
8 # 3 extend
9 my_list.extend(['six','seven'])
10 print(my_list)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
['one', 'two', 'three', 'four']
['one', 'two', 'three', 'four', 'five']
['one', 'two', 'three', 'four', 'five', 'six', 'seven']
```

As we can see in our case, the iterable is a List of two items and it is called iterable because we can loop or iterate over this List using loops that will be covered in the later sections of this chapter.

## pop() method

This method will remove an item from the end of the List and if we do this on our existing list, it will pop/remove the item ‘seven’

```
# 4 pop  
my_list.pop()  
print(my_list)
```

The screenshot shows a Python development environment with two tabs: 'main.py' and 'Console'. The 'main.py' tab contains the following code:

```
1 # 1 append  
2 my_list = ['one', 'two', 'three'];  
3 my_list.append('four')  
4 print(my_list)  
5 # 2 insert  
6 my_list.insert(4, 'five')  
7 print(my_list)  
8 # 3 extend  
9 my_list.extend(['six', 'seven'])  
10 print(my_list)  
11 # 4 pop  
12 my_list.pop()  
13 print(my_list)
```

The 'Console' tab shows the output of the code execution:

```
['one', 'two', 'three', 'four']  
['one', 'two', 'three', 'four', 'five']  
['one', 'two', 'three', 'four', 'five', 'six', 'seven']  
['one', 'two', 'three', 'four', 'five', 'six']  
> 
```

If we want to remove a specific item from a list, we can call the `pop()` method with an index of the item we want to remove, for example, let's say we want to remove the first item and we know this item index is going to be zero:

```
# remove the first item  
my_list.pop(0)  
print(my_list)
```

The screenshot shows a Python development environment with two panes. The left pane, titled 'main.py x', contains the following code:

```
1 # 1 append
2 my_list = ['one','two','three'];
3 my_list.append('four')
4 print(my_list)
5 # 2 insert
6 my_list.insert(4,'five')
7 print(my_list)
8 # 3 extend
9 my_list.extend(['six','seven'])
10 print(my_list)
11 # 4 pop
12 my_list.pop()
13 print(my_list)
14 # remove the first item
15 my_list.pop(0)
16 print(my_list)
```

The line 'my\_list.pop(0)' is highlighted with a blue selection bar. The right pane, titled 'Console', shows the output of the code execution:

```
['one', 'two', 'three', 'four']
['one', 'two', 'three', 'four', 'five']
['one', 'two', 'three', 'four', 'five', 'six', 'seven']
['one', 'two', 'three', 'four', 'five', 'six']
> ['two', 'three', 'four', 'five', 'six']
```

## remove() method

This method is used when we want to remove an item from a list based on the item value. For example, let's say we want to remove the item with the value 'four':

```
#5 remove
my_list.remove('four')
print(my_list)
```

The screenshot shows a Python development environment with two panes. The left pane is titled 'main.py' and contains the following code:

```
4 print(my_list)
5 # 2 insert
6 my_list.insert(4,'five')
7 print(my_list)
8 # 3 extend
9 my_list.extend(['six','seven'])
10 print(my_list)
11 # 4 pop
12 my_list.pop()
13 print(my_list)
14 # remove the first item
15 my_list.pop(0)
16 print(my_list)
17
18 #5 remove    remove(value: _T, /) ->
19 my_list.remove('four')
20 print(my_list)
21
```

The right pane is titled 'Console' and shows the output of the code execution:

```
['one', 'two', 'three', 'four']
['one', 'two', 'three', 'four', 'five']
['one', 'two', 'three', 'four', 'five', 'six', 'seven']
['one', 'two', 'three', 'four', 'five', 'six']
['two', 'three', 'four', 'five', 'six']
['two', 'three', 'five', 'six']
> |
```

This method like the previous ones will modify the existing list but they don't return anything else after the operation is done. We can check this if we wrap the remove method in a print function like this:

The screenshot shows a Python development environment with two panes. The left pane is titled 'main.py' and contains the following code:

```
4 print(my_list)
5 # 2 insert
6 my_list.insert(4,'five')
7 print(my_list)
8 # 3 extend
9 my_list.extend(['six', 'seven'])
10 print(my_list)
11 # 4 pop
12 my_list.pop()
13 print(my_list)
14 # remove the first item
15 my_list.pop(0)
16 print(my_list)
17
18 #5 remove
19 print(my_list.remove('four'))
20 print(my_list)
```

The right pane is titled 'Console' and shows the output of the code execution:

```
['one', 'two', 'three', 'four']
['one', 'two', 'three', 'four', 'five']
['one', 'two', 'three', 'four', 'five', 'six', 'seven']
['one', 'two', 'three', 'four', 'five', 'six']
['two', 'three', 'four', 'five', 'six']
None
['two', 'three', 'five', 'six']
> |
```

As we can see from the figure above, this method returns None but it modifies the original list. If we try to print the other methods, we will see that they don't return any value as well but they change the original list.

## clear() method

This method will clear or remove all of the items in the specified list

```
# 6 clear  
my_list.clear()  
print(my_list)
```

The screenshot shows a Python development environment with two panes. The left pane is titled 'main.py' and contains the following code:

```
9 my_list.extend(['six','seven'])  
10 print(my_list)  
11 # 4 pop  
12 my_list.pop()  
13 print(my_list)  
14 # pop the first item  
15 my_list.pop(0)  
16 print(my_list)  
17  
18 #5 remove  
19 print(my_list.remove('four'))  
20 print(my_list)  
21  
22 # 6 clear  
23 my_list.clear()  
24 print(my_list)
```

The right pane has tabs for 'Console' and 'Shell'. The 'Console' tab is active and displays the following output:

```
['one', 'two', 'three', 'four']  
['one', 'two', 'three', 'four', 'five']  
['one', 'two', 'three', 'four', 'five', 'six', 'seven']  
['one', 'two', 'three', 'four', 'five', 'six']  
['two', 'three', 'four', 'five', 'six']  
None  
['two', 'three', 'five', 'six']  
[]
```

## index() method

This method will return the position or index of the value we pass into the method as an argument for:

```
#new list  
new_list = ['one','two','three','four','five']  
# 7 index  
print(new_list.index('three'))
```

A screenshot of a Python IDE interface. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 #new list
2 new_list =
3     ['one','two','three','four','five']
4 # 7 index
5 print(new_list.index('three'))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
2
> [
```

This simply means finding the item 'three' in the list and returning the index where this item is. This method takes two additional/optional arguments. These arguments are used when we want to speed up the searching process because we are saying start the searching from this index zero and stop at index 3:

```
# index method with 3 params
print(new_list.index('three',0,3))
```

A screenshot of a Python IDE interface. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 #new list
2 new_list =
3     ['one','two','three','four','five']
4 # 7 index
5 print(new_list.index('three'))
6 # index method with 3 params
7 print(new_list.index('three',0,3))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
2
2
> [
```

As we can see from the figure above, the result will be the same but it will be a lot faster because we are not searching the entire List to get the item position. What will happen if the item we are looking for is not in the range we specified? Well, it will throw a Value Error saying the item we are looking for is not in the range:

```
print(new_list.index('three',0,2))
```

The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 #new list
2 new_list =
3     ['one','two','three','four','five']
4 # 7 index
5 print(new_list.index('three'))
6 # index method with 3 params
7 print(new_list.index('three',0,3))
8 print(new_list.index('three',0,2))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the following output:

```
2
2
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    print(new_list.index('three',0,2))
ValueError: 'three' is not in list
> []
```

## count() method

The count method will count how many times the same value occurs in the list:

```
# 8 count
my_list1 = ['one','two','three','one','four']
print(my_list1.count('one'))
```

The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # 8 count
2 my_list1 =
3     ['one','two','three','one','four']
4 print(my_list1.count('one'))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the following output:

```
2
> []
```

In the figure above, the count for the value 'two' is two because item two can be found at position zero and 3 in the **my\_list1**.

## sort() method

In order to explain this method, let us create a List with items that are random integer numbers:

```
#9 sort
numbers_list = [3,6,7,1,5,2,4]
```

```
numbers_list.sort()  
print(numbers_list)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code contains four lines: 1. #9 sort, 2. numbers\_list = [3,6,7,1,5,2,4], 3. numbers\_list.sort(), and 4. print(numbers\_list). To the right of the editor is a terminal window titled 'Console' which displays the output: [1, 2, 3, 4, 5, 6, 7].

```
main.py ×  
1 #9 sort  
2 numbers_list = [3,6,7,1,5,2,4]  
3 numbers_list.sort()  
4 print(numbers_list)  
Console Shell  
[1, 2, 3, 4, 5, 6, 7]  
▶
```

From the figure above, we can see that the un-ordered list is now sorted. In Python, we also have the built-in function called sorted() and this function produces a new array called the sorted array, but it will not affect the original one like with the sort() method:

```
# sorted() function not a method  
numbers_list1 = [3,6,7,1,5,2,4]  
store_sorted =sorted(numbers_list1)  
print(numbers_list1)  
print(store_sorted)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code contains five lines: 1. # sorted() function not a method, 2. numbers\_list1 = [3,6,7,1,5,2,4], 3. store\_sorted =sorted(numbers\_list1), 4. print(numbers\_list1), and 5. print(store\_sorted). The lines 3, 4, and 5 are highlighted in blue. To the right of the editor is a terminal window titled 'Console' which displays two lists: [3, 6, 7, 1, 5, 2, 4] and [1, 2, 3, 4, 5, 6, 7].

```
main.py ×  
1 # sorted() function not a method  
2 numbers_list1 = [3,6,7,1,5,2,4]  
3 store_sorted =sorted(numbers_list1)  
4 print(numbers_list1)  
5 print(store_sorted)  
Console Shell  
[3, 6, 7, 1, 5, 2, 4]  
[1, 2, 3, 4, 5, 6, 7]  
▶
```

This proves that the built-in function does not modify the existing list.

## copy() method

We can use this method when we need a copy of some list:

```
#10 copy  
numbers_list = [3, 6, 7, 1, 5, 2, 4]  
new_list = numbers_list.copy()  
print(new_list)
```

```
main.py ×
1 #10 copy
2 numbers_list = [3, 6, 7, 1, 5, 2, 4]
3 new_list = numbers_list.copy()
4 print(new_list)
```

[3, 6, 7, 1, 5, 2, 4]

The **new\_list** will be a copy from the **numbers\_list** and if we want to add insert or remove elements from the **new\_list**, then this will not affect the original **numbers\_list**

### reverse() method

This method will reverse the entire list without sorting it:

```
#11 reverse
numbers_list2 = [3, 6, 7, 1, 5, 2, 4]
numbers_list2.reverse()
print(numbers_list2)
```

```
main.py ×
1 #11 reverse
2 numbers_list2 = [3, 6, 7, 1, 5, 2, 4]
3 numbers_list2.reverse()
4 print(numbers_list2)
```

[4, 2, 5, 1, 7, 6, 3]

This is the last method I wanted to show you but before we start learning some new Python features, I would like to mention that you can combine different List methods in your projects, for example, if you need a reversed list but sorted you can call the `sort()` method first and then the `reverse()` method:

```
# combining different list methods
numbers_list3 = [3, 6, 7, 1, 5, 2, 4]
numbers_list3.sort()
numbers_list3.reverse()
print(numbers_list3)
```

## Useful tips and tricks used on Lists

These are common tips and tricks that developers use on Python Lists, and once you know them, you will use them as well. We have already seen the `len()` function that returns the number of items in a list. We have also seen how we can reverse the String but the same can be done with Lists as well

### Reverse the list using list slicing

```
# reverse the list using slicing  
num_list = [3,6,7,1,5,2,4]  
print(num_list[::-1])
```

The screenshot shows a Python code editor with a file named `main.py`. The code uses list slicing to reverse a list of numbers. The `Console` tab shows the output: `[4, 2, 5, 1, 7, 6, 3]`.

```
main.py ×  
1 # reverse the list using slicing  
2 num_list = [3,6,7,1,5,2,4]  
3 print(num_list[::-1])
```

Console	Shell
[4, 2, 5, 1, 7, 6, 3]	> █

### Copy of the list using the slicing method

```
# copy the list using the slicing:  
num_list1 = [3,6,7,1,5,2,4]  
new_list = num_list1[:]  
print(new_list)  
print(num_list1)
```

The screenshot shows a Python code editor with a file named `main.py`. The code creates a copy of a list using slicing. The `Console` tab shows two identical lists: `[3, 6, 7, 1, 5, 2, 4]`.

```
main.py ×  
1 num_list1 = [3,6,7,1,5,2,4]  
2 new_list = num_list1[:]  
3 print(new_list)  
4 print(num_list1)  
5
```

Console	Shell
[3, 6, 7, 1, 5, 2, 4] [3, 6, 7, 1, 5, 2, 4]	> █

### Generate new list using range() function

We can generate and populate a numbered list using the `range` function. For example, I want to create a numbered list from a range of 0 to 50:

```
#create and populate numbered list
num_list2 = list(range(0,50))
print(num_list2)
```

```
main.py ×
```

```
1 #create and populate numbered list
2 num_list2 = list(range(0,50))
3 print(num_list2)
4
```

```
Console Shell
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49]
```

## .join()

This method will take all of the items from a list (iterable) and join the values into one string. We also need to specify the separator as well:

```
#.join()
usernames = ['andy','carol','steve','jason']
joined_usernames = ', '.join(usernames)
print(joined_usernames);
```

```
main.py ×
```

```
1 #.join()
2 usernames =
3     ['andy','carol','steve','jason']
4 joined_usernames = ', '
5     '.join(usernames)
6 print(joined_usernames);
```

```
Console Shell
```

```
andy, carol, steve, jason
```

## List Unpacking

In JavaScript, this is known as the concept of destructuring, but in Python, this is known as list unpacking and it's very useful when we want the values/items from the List to be stored in different variables:

```
# list unpacking
andy,carol,steve,jason = ['andy','carol','steve','jason']
print(andy)
print(carol)
```

```
print(steve)  
print(jason)
```

The screenshot shows a code editor window titled "main.py" with the following content:

```
1 # list unpacking  
2 andy,carol,steve,jason =  
3     ['andy','carol','steve','jason']  
4 print(andy)  
5 print(carol)  
6 print(steve)  
7 print(jason)
```

To the right of the code editor is a terminal window with tabs "Console" and "Shell". The "Console" tab is selected, showing the output of the script:

```
andy  
carol  
steve  
jason
```

As you can see now, the variables on the left side of the '=' operator will have the values from the list items. This is how we can unpack the list items into variables. But we can unpack only a few values from a List and keep the rest of the items stored in a single variable name as a list:

```
andy,carol,*rest = ['andy','carol','steve','jason']  
print(andy)  
print(carol)  
print(rest)
```

The screenshot shows a code editor window titled "main.py" with the following content:

```
1 andy,carol, *rest =  
2     ['andy','carol','steve','jason']  
3 print(andy)  
4 print(carol)  
5 print(rest)
```

To the right of the code editor is a terminal window with tabs "Console" and "Shell". The "Console" tab is selected, showing the output of the script:

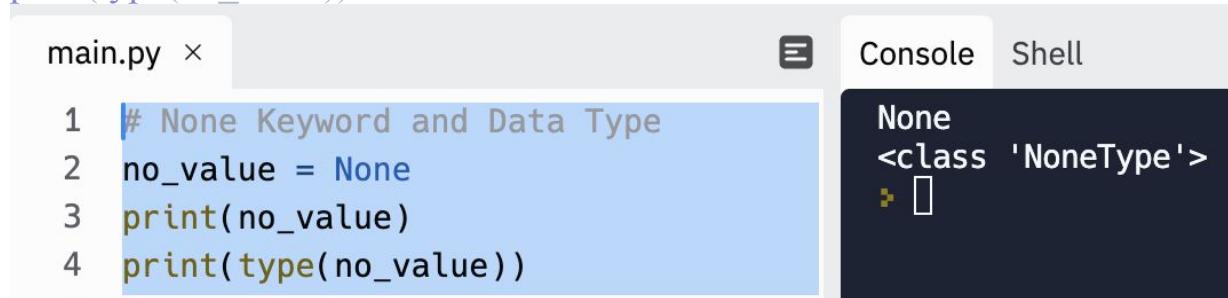
```
andy  
carol  
['steve', 'jason']
```

I think we have covered a lot of the basic and some intermediate features of Python in this section. Let's learn the rest of the data types in Python.

## None in Python (data type)

The **None** Python is a keyword and is used to define a **null** value or no value at all. The **None** is also a data type from the class **NoneType**. The **None** simply means absence of value, **null** value, or no value at all. In **JavaScript**, we have **Null** and we have **None**. Some developers still don't understand that this **None** is not the same as 0 (zero) or empty string and it can be very useful in some scenarios. For example, you want to create a program and at the beginning of that program, you need a variable that is only declared but without any value. We can assign the variables to None like this:

```
# None Keyword and Data Type
no_value = None
print(no_value)
print(type(no_value))
```



A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # None Keyword and Data Type
2 no_value = None
3 print(no_value)
4 print(type(no_value))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
None
<class 'NoneType'>
> □
```

Before we move to the next section, I want you to remember this about the **None** data type:

None is not the same as 0 (zero)

None is not the same as False

None is not an empty String ''

Comparing None to None will return True

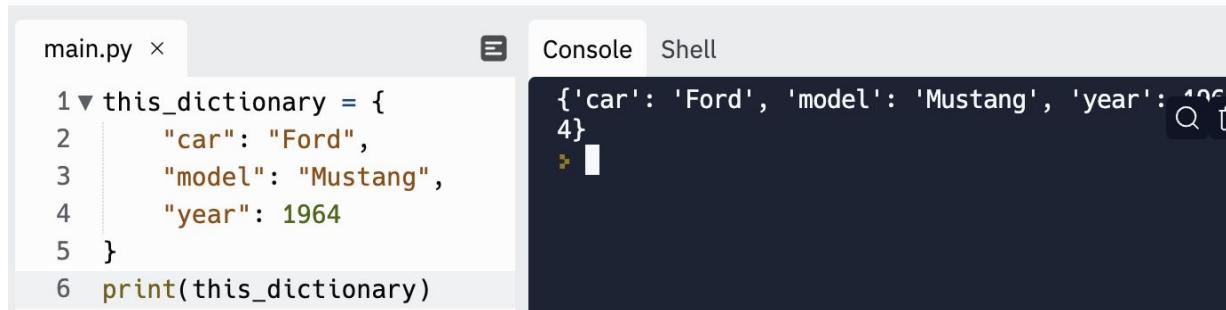
Comparing None to anything else will always return False

## Dictionary in Python (data structure)

Same as the Python Lists, the Dictionary is a data type and data structure. This is another way for us to organize and access the data. The dictionary stores the data as **key:value** pairs. Before the dictionaries had an unordered data type, but after version 3.7 of Python, they are now ordered. Dictionaries can be written with curly brackets and can have one or more key/value pairs:

```
this_dictionary = {  
    "car": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}
```

We can print the dictionary like this:

A screenshot of a Python development environment. On the left, there is a code editor window titled 'main.py' containing the following Python code:

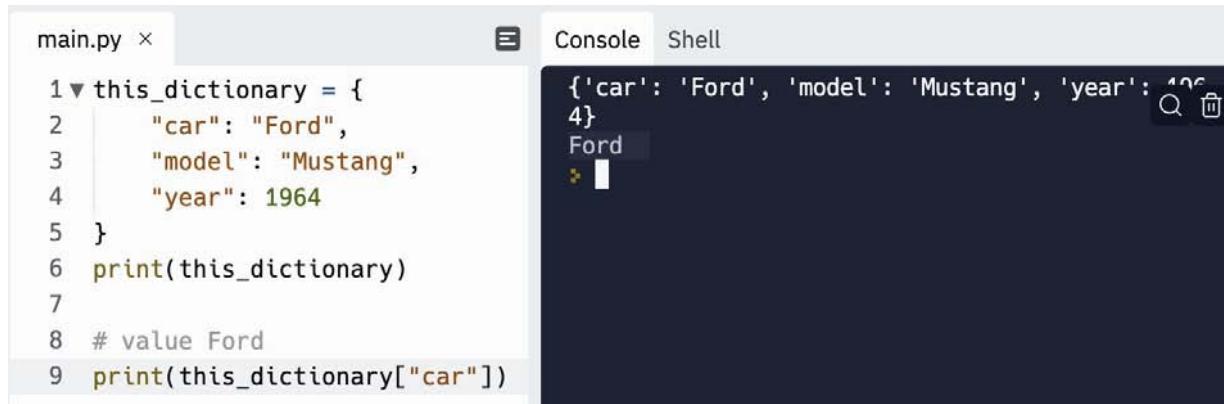
```
1 this_dictionary = {  
2     "car": "Ford",  
3     "model": "Mustang",  
4     "year": 1964  
5 }  
6 print(this_dictionary)
```

On the right, there is a terminal window titled 'Console' showing the output of the code:

```
{'car': 'Ford', 'model': 'Mustang', 'year': 1964}
```

From the dictionary definition, we can see that the keys are on the left side and values are on the right side of the colon. For example, 'car' is key and 'Ford' is value in the example above. So how we can access the values from the dictionaries? Remember that in Lists we used indexes but here we need to use keys:

```
# value Ford  
print(this_dictionary["car"])
```



A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

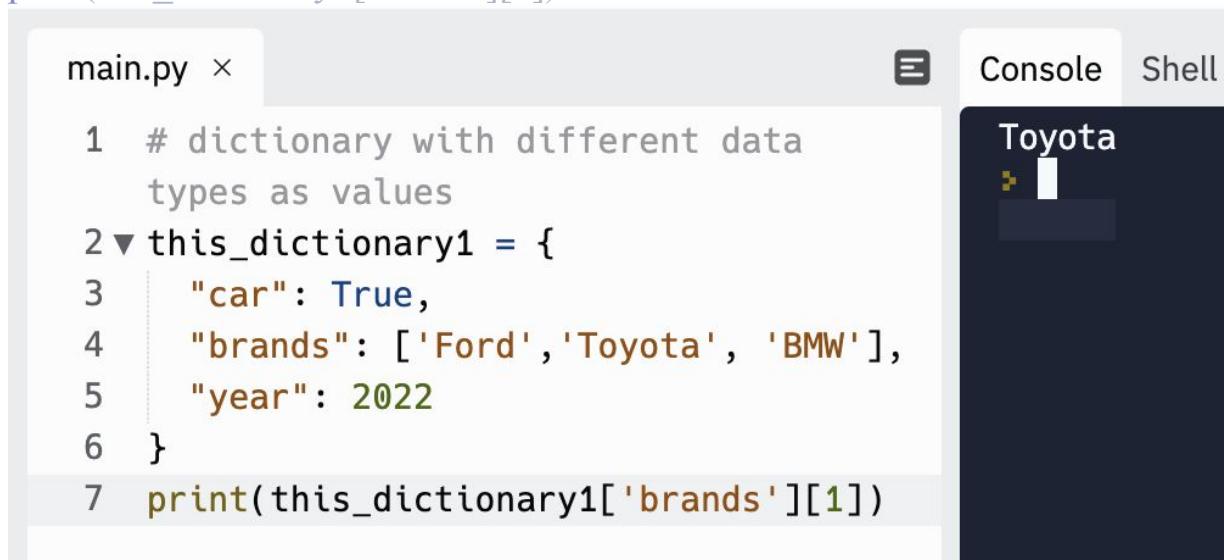
```
1▼ this_dictionary = {  
2     "car": "Ford",  
3     "model": "Mustang",  
4     "year": 1964  
5 }  
6 print(this_dictionary)  
7  
8 # value Ford  
9 print(this_dictionary["car"])
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
{'car': 'Ford', 'model': 'Mustang', 'year': 1964}  
Ford
```

The dictionary is very similar to Objects in JavaScript. The values can be from any data type, for example, the values can be a combination of integers, floats, strings, lists, etc:

```
# dictionary with different data types as values  
this_dictionary1 = {  
    "car": True,  
    "brands": ['Ford', 'Toyota', 'BMW'],  
    "year": 2022  
}  
print(this_dictionary1['brands'][1])
```



A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # dictionary with different data  
    types as values  
2▼ this_dictionary1 = {  
3     "car": True,  
4     "brands": ['Ford', 'Toyota', 'BMW'],  
5     "year": 2022  
6 }  
7 print(this_dictionary1['brands'][1])
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
Toyota
```

Now that we know about this new data type, we can go back to Python Lists because I have something important to discuss. The lists can have values that are from different data types, including the dictionaries:

```
# lists with dictionary as data type
new_list = [
    {
        "car": True,
        "brands": ['Ford', 'Toyota', 'BMW'],
        "year": 2022
    },
    {
        "car": False,
        "brands": ['Ranger', 'Kluger', 'X5'],
        "year": 2022
    }
]
```

If we want to access the first element of this list, we need to write this query:

```
print(new_list[0])
```

```
main.py ×
```

```
1 # lists with dictionary as data type
2 new_list = [
3     {
4         "car": True,
5         "brands": ['Ford', 'Toyota',
6             'BMW'],
7         "year": 2022
8     },
9     {
10         "car": False,
11         "brands": ['Ranger',
12             'Kluger', 'X5'],
13         "year": 2022
14     }
15 ]
16 print(new_list[0])
```

Console Shell

```
{'car': True, 'brands': ['Ford', 'Toyota', 'BMW'], 'year': 2022}
```

If we want to find what values the key 'car' have, we can do this:

```
print(new_list[0]['car'])
```

The screenshot shows a Python code editor with a file named `main.py`. The code defines a list of two dictionaries, `new_list`, where each dictionary has keys `'car'`, `'brands'`, and `'year'`. The first dictionary has `'car': True`, `'brands': ['Ford', 'Toyota', 'BMW']`, and `'year': 2022`. The second dictionary has `'car': False`, `'brands': ['Ranger', 'Kluger', 'X5']`, and `'year': 2022`. Below the list, there are two `print` statements: one for the entire list and one for the value of `'car'` at index 0.

```
main.py x
1 # lists with dictionary as data type
2 new_list = [
3     {
4         "car": True,
5         "brands": ["Ford", "Toyota", "BMW"],
6         "year": 2022
7     },
8     {
9         "car": False,
10        "brands": ["Ranger", "Kluger", "X5"],
11        "year": 2022
12    }
13 ]
14 print(new_list[0])
15 print(new_list[0]['car'])
```

The right side of the interface is a dark-themed terminal window titled "Console". It shows the output of the code execution:

```
{'car': True, 'brands': ['Ford', 'Toyota', 'BMW'], 'year': 2022}
True
```

We can also get the car brands:

```
print(new_list[0]['brands'])
```

The screenshot shows the same `main.py` file as before, but now it includes an additional `print` statement at the end of the code block, printing the value of `'brands'` at index 0.

```
main.py x
1 # lists with dictionary as data type
2 new_list = [
3     {
4         "car": True,
5         "brands": ["Ford", "Toyota", "BMW"],
6         "year": 2022
7     },
8     {
9         "car": False,
10        "brands": ["Ranger", "Kluger", "X5"],
11        "year": 2022
12    }
13 ]
14 print(new_list[0])
15 print(new_list[0]['car'])
16 print(new_list[0]['brands'])
```

The terminal window shows the output:

```
{'car': True, 'brands': ['Ford', 'Toyota', 'BMW'], 'year': 2022}
True
['Ford', 'Toyota', 'BMW']
```

And from this result, I want to get the 'BMW' brand using the index number 2

```
print(new_list[0]['brands'][2])
```

I know this is getting complex but I can assure you that if you spend some time thinking about Lists and Dictionaries, you will see that it is very easy. Another important thing I want to discuss is the dictionary values. The dictionary values can be from any data type but dictionary keys cannot be mutable. That is why we have used String as key because we know strings are immutable. We can also use numbers and Boolean because they are immutable as well but Python Lists cannot be used as dictionary keys because they are mutable. For this reason, we cannot use Lists as keys in dictionaries. The keys are mostly going to be strings that are always descriptive. The key in the dictionary has to be unique, no duplication of keys should be allowed like in this example:

```
this_dictionary1 = {  
    "car": True,  
    "brands": ['Ford', 'Toyota', 'BMW'],  
    "car": 2022  
}
```

As we can see, the ‘car’ is the key and it’s used in two different places. We want to avoid getting errors like this because they will crush the program. There is another way we can create a dictionary using the following syntax:

```
#second way of creating new dictionary  
new_dict = dict(car = 'Tesla')  
print(new_dict)
```

In the above example, we are using the built-in Python function called `dict()` to create a new dictionary and this syntax is not that common.

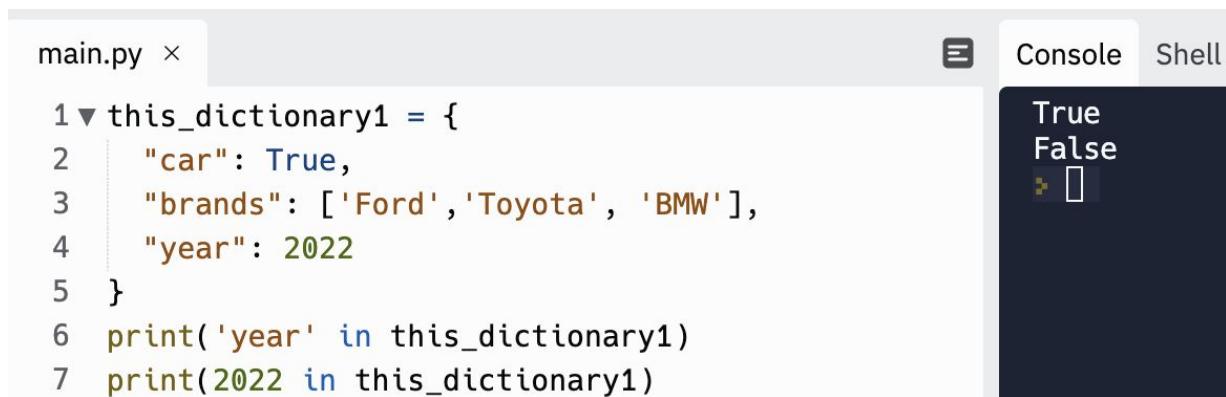
## Dictionary – methods

In this section, we will go through some methods that we can use on dictionaries. Before we delve into the dictionary methods, let’s first talk about how can we use the ‘in’ keyword. The ‘in’ keyword can be used to check the existence of a particular key in the dictionary. The ‘in’ keyword is used to check the key not the value:

```

this_dictionary1 = {
    "car": True,
    "brands": ['Ford', 'Toyota', 'BMW'],
    "year": 2022
}
print('year' in this_dictionary1)
print(2022 in this_dictionary1)

```



```

main.py ×
1 ▼ this_dictionary1 = {
2     "car": True,
3     "brands": ['Ford', 'Toyota', 'BMW'],
4     "year": 2022
5 }
6 print('year' in this_dictionary1)
7 print(2022 in this_dictionary1)

```

Console Shell

```

True
False
▶ []

```

If the key doesn't exist, it will return False like in the figure above because I have tried to find the year 2022 but there is no 'key' 2022. Let's start with some useful dictionary methods.

### **get() method**

The get method will take one parameter and this parameter will be the 'key' from a dictionary. The get method will return the value of that key:

```
print(this_dictionary1.get('car'))
```

The screenshot shows a Python code editor with a file named 'main.py'. The code defines a dictionary 'this\_dictionary1' with keys 'car' (value: True), 'brands' (value: ['Ford', 'Toyota', 'BMW']), and 'year' (value: 2022). It then prints the value of 'car'. To the right, the 'Console' tab is active, displaying the output 'True'.

```
1 ▼ this_dictionary1 = {
2     "car": True,
3     "brands": ['Ford', 'Toyota', 'BMW'],
4     "year": 2022
5 }
6 print(this_dictionary1.get('car'))
```

If the key we are trying to get does not exist, we will get None. The get method can accept a second parameter called value:

```
dictionary.get(keyname, value)
```

The value parameter is optional and we usually specify it as a backup because if there is no key found, what value do we want this method to return? Let's create an example where the key doesn't exist but we specify the values we want that method to return:

```
print(this_dictionary1.get('cars', ['ford', 'audi']))
```

From the example above, the 'cars' key does not exist in the dictionary but will return a list of two items (Ford and Audi) because we provided the second parameter:

The screenshot shows a Python code editor with a file named 'main.py'. The code defines a dictionary 'this\_dictionary1' with keys 'car' (value: True), 'brands' (value: ['Ford', 'Toyota', 'BMW']), and 'year' (value: 2022). It then prints the value of 'cars' using the get method with a second parameter of ['ford', 'audi']. To the right, the 'Console' tab is active, displaying the output '['ford', 'audi']'.

```
1 ▼ this_dictionary1 = {
2     "car": True,
3     "brands": ['Ford', 'Toyota', 'BMW'],
4     "year": 2022
5 }
6 # get method with second parameter known as value
7 print(this_dictionary1.get('cars',
8     ['ford', 'audi']))
```

**keys()**

This dictionary method is used when we want to get all of the keys that exist in the dictionary. This method will return a view object. This object will contain all of the keys in the dictionary:

```
#keys method  
print(this_dictionary1.keys())
```



The screenshot shows a Python code editor with a file named 'main.py'. The code defines a dictionary 'this\_dictionary1' with keys 'car', 'brands', and 'year'. It then prints the keys of this dictionary. To the right, a terminal window titled 'Console' shows the output of the command 'dict\_keys(['car', 'brands', 'year'])', which returns a list of keys: ['car', 'brands', 'year'].

```
main.py x  
1 ▼ this_dictionary1 = {  
2     "car": True,  
3     "brands": ['Ford', 'Toyota', 'BMW'],  
4     "year": 2022  
5 }  
6 #keys method  
7 print(this_dictionary1.keys())
```

```
Console Shell  
dict_keys(['car', 'brands', 'year'])  
['car', 'brands', 'year']
```

We can combine the keys() method with the 'in' keyword and this will return Boolean True and False:

```
# find a key from the view object  
print('car' in this_dictionary1.keys())
```

## values() method

This method will return a view object as a List with all of the values in the dictionary:

```
# values method  
print(this_dictionary1.values())
```



The screenshot shows a Python code editor with a file named 'main.py'. The code defines a dictionary 'this\_dictionary1' with keys 'car', 'brands', and 'year'. It then prints the values of this dictionary. To the right, a terminal window titled 'Console' shows the output of the command 'dict\_values([True, ['Ford', 'Toyota', 'BMW'], 2022])', which returns a list of values: [True, ['Ford', 'Toyota', 'BMW'], 2022].

```
main.py x  
1 ▼ this_dictionary1 = {  
2     "car": True,  
3     "brands": ['Ford', 'Toyota', 'BMW'],  
4     "year": 2022  
5 }  
6 # values method  
7 print(this_dictionary1.values())
```

```
Console Shell  
dict_values([True, ['Ford', 'Toyota', 'BMW'], 2022])  
[True, ['Ford', 'Toyota', 'BMW'], 2022]
```

Same as with the keys, the values method can be combined with the 'in' keyword to check if any of the values exists in the dictionary.

```
print(2022 in this_dictionary1.values())
```

## items() method

The items method is a little bit more complicated to understand because we haven't used tuples (another data type) yet. This method will return a view object that contains a key-values pair of the dictionary in a form of Tuples. We will talk about tuples soon:

```
#items method  
print(this_dictionary1.items())
```

```
main.py ×          Console Shell  
1 ▼ this_dictionary1 = {  
2     "car": True,  
3     "brands": ['Ford', 'Toyota',  
4                  'BMW'],  
5     "year": 2022  
6 }  
6 #items method  
7 print(this_dictionary1.items())
```

```
dict_items([('car', True), ('brands', ['Ford', 'Toyota', 'BMW']), ('year', 2022)])
```

## clear()

This method will remove all of the key/value pairs from the dictionary. The clear method does not require anything to be passed as an argument and it will return the empty dictionary:

```
# clear method  
print(this_dictionary1.clear())  
print(this_dictionary1)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code defines a dictionary 'this\_dictionary1' with keys 'car', 'brands', and 'year'. It then calls the 'clear' method on 'this\_dictionary1' and prints the dictionary again. The output in the 'Console' tab shows that after calling 'clear', the dictionary is empty (None), and then it prints an empty dictionary {}.

```
main.py ×
1 ▼ this_dictionary1 = {
2     "car": True,
3     "brands": ['Ford', 'Toyota',
4                 'BMW'],
4     "year": 2022
5 }
6 # clear method
7 print(this_dictionary1.clear())
8 print(this_dictionary1)
```

```
None
{}
> █
```

## copy()

You can use this method to create a copy of a dictionary and the copied dictionary will be independent of the original dictionary:

```
#copy method
this_dictionary2 = {
    "car": True,
    "brands": ['Ford', 'Toyota', 'BMW'],
    "year": 2022
}
dict_copy = this_dictionary2.copy()
print(dict_copy)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code defines a dictionary 'this\_dictionary2' with keys 'car', 'brands', and 'year'. It then creates a copy of 'this\_dictionary2' using the 'copy' method and prints the copied dictionary. The output in the 'Console' tab shows the copied dictionary, which has the same structure as the original but is a separate object.

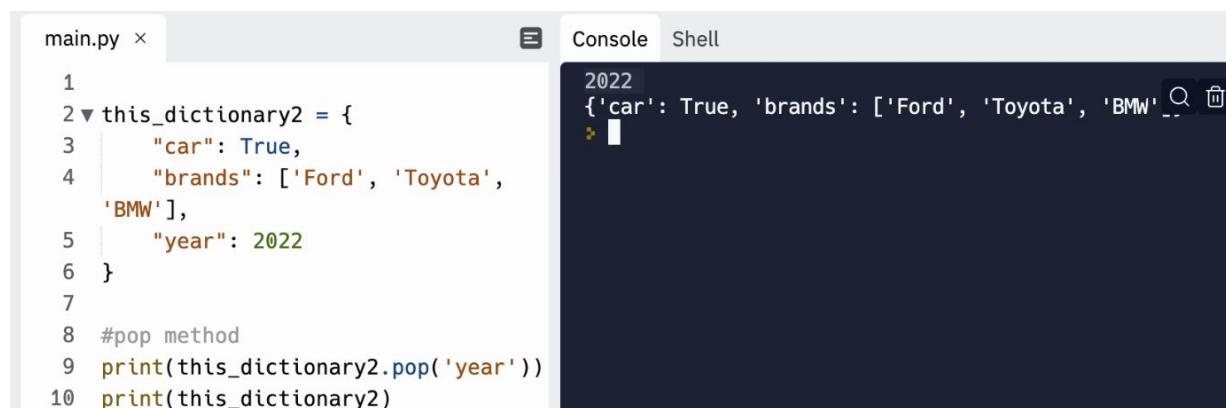
```
main.py ×
1 #copy method
2 ▼ this_dictionary2 = {
3     "car": True,
4     "brands": ['Ford', 'Toyota',
5                 'BMW'],
5     "year": 2022
6 }
7 dict_copy = this_dictionary2.copy()
8 print(dict_copy)
```

```
{'car': True, 'brands': ['Ford', 'Toyota', 'BMW'], 'year': 2022}
> █
```

## pop()

This method takes one argument and that is a dictionary key. The pop() method will remove a value from the dictionary based on the key. If we try to print this method, it will return the item it was removed from in the dictionary:

```
#pop method
print(this_dictionary2.pop('year'))
print(this_dictionary2)
```



```
main.py x
1
2 ▼ this_dictionary2 = {
3     "car": True,
4     "brands": ['Ford', 'Toyota', 'BMW'],
5     "year": 2022
6 }
7
8 #pop method
9 print(this_dictionary2.pop('year'))
10 print(this_dictionary2)
```

Console Shell

```
2022
{'car': True, 'brands': ['Ford', 'Toyota', 'BMW']}
> █
```

## popitem()

The popitem() is used to pop or remove the last **key:value** pair from the dictionary. In the previous Python versions, this method removed a random **key:value** pair but now it will remove the last.

```
this_dictionary2 = {
    "car": True,
    "brands": ['Ford', 'Toyota', 'BMW'],
    "year": 2022
}
#popitem
this_dictionary2.popitem()
print(this_dictionary2)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code defines a dictionary 'this\_dictionary2' with keys 'car', 'brands', and 'year'. The 'brands' value is a list of three car brands. The 'year' value is set to 2022. A call to 'popitem()' removes the last item from the dictionary. The code then prints the dictionary. In the 'Console' tab, the output is shown as a dictionary with 'car': True, 'brands': ['Ford', 'Toyota', 'BMW'], and 'year': 2022.

```
1
2 ▼ this_dictionary2 = {
3     "car": True,
4     "brands": ['Ford', 'Toyota',
5     'BMW'],
6     "year": 2022
7 }
8 #popitem
9 print(this_dictionary2)
```

```
{'car': True, 'brands': ['Ford', 'Toyota', 'BMW'], 'year': 2022}
```

## update()

This method is very useful if we want to update a value based on a specified key. Let's update the year to 2023:

```
#update
print(this_dictionary2.update({'year':2023}))
print(this_dictionary2)
```

The screenshot shows a Python code editor with the same 'main.py' file. After the previous code, there is a comment '#update' followed by a call to 'update()' with a parameter of {'year': 2023}. Finally, 'print(this\_dictionary2)' is executed. The 'Console' tab shows the updated dictionary where the 'year' key now has the value 2023.

```
1
2 ▼ this_dictionary2 = {
3     "car": True,
4     "brands": ['Ford', 'Toyota', 'BMW'],
5     "year": 2022
6 }
7 #update
8 print(this_dictionary2.update({'year':2023}))
9 print(this_dictionary2)
```

```
None
{'car': True, 'brands': ['Ford', 'Toyota', 'BMW'], 'year': 2023}
```

What will happen if we try to update a value with a wrong key that doesn't exist in the original dictionary? You need to be careful in this case because the update method will look into the dictionary and if it cannot find the key, it will think we want to insert a new key/value pair. Let's say we want to update the 'car' key to a new value 'False' but we make a spelling mistake and instead of the key 'car' we typed 'cars', this will happen:

```
print(this_dictionary2.update({'cars':False}))
```

```
main.py ×
1
2 ▼ this_dictionary2 = {
3     "car": True,
4     "brands": ['Ford', 'Toyota', 'BMW'],
5     "year": 2022
6 }
7 #update
8 print(this_dictionary2.update({'cars':False}))
9 print(this_dictionary2)
```

Console Shell

```
None
{'car': True, 'brands': ['Ford', 'Toyota', 'BMW'], 'year': 2022, 'cars': False}
> █
```

As we can see, we have the new key/value pair 'car':False added at the end of the dictionary and this is not the result we wanted. That is why you need to be careful when trying to use the update method.

### setdefault() method

The setdefault () method will return the value of a key if that key is in the dictionary. If the key cannot be found, this method will insert new key/value pair

```
#setdefault
year = this_dictionary2.setdefault('year')
print(year)
```

```
main.py ×
1
2 ▼ this_dictionary2 = {
3     "car": True,
4     "brands": ['Ford', 'Toyota', 'BMW'],
5     "year": 2022
6 }
7 #setdefault
8 year = this_dictionary2.setdefault('year')
9 print(year)
```

Console Shell

```
2022
> █
```

## Tuples in Python (data type)

Similarly like the Lists and Dictionaries, we have Tuples that can be used to store multiple values in a single variable. Tuples are one of the four most important built-in data types in Python used to store collections of data.

There is one more data type we are going to learn and it's called Sets. The Tuples are similar to Lists but the major difference is that we cannot modify them so they are immutable, similarly to Strings. In short, you can imagine that working with Tuples is the same as working with immutable Lists. Let us create our first tuple:

```
# tuples  
new_tuple = (1, 2, 3)  
print(new_tuple)
```



The screenshot shows a Python development environment. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # tuples  
2 new_tuple = (1, 2, 3)  
3 print(new_tuple)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution:

```
(1, 2, 3)
```

In tuples, we use the round brackets instead of the curly brackets we used in dictionaries {}. A tuple is a collection of data that is ordered and we cannot change the order of items once the tuple is created. We have already printed the tuple but if we want to access the individual items, we need to use their indexes. Same as with Lists, the indexes start from position zero. Same as lists, to access the items, we need to use the square brackets where we pass the index position:

```
#access items  
print(new_tuple[2])
```

The screenshot shows a Python development environment with two tabs: 'main.py' and 'Console'. The code in 'main.py' is:

```
1 # tuples
2 new_tuple = (1, 2, 3)
3
4 #access items
5 print(new_tuple[2])
```

The 'Console' tab shows the output:

```
3
> [
```

Let's check if the tuples are indeed immutable. In the following example, let's modify the element in the position/index 2:

```
# modify tuples
new_tuple[2] = 4
print(new_tuple)
```

The screenshot shows a Python development environment with two tabs: 'main.py' and 'Console'. The code in 'main.py' is:

```
1 # tuples
2 new_tuple = (1, 2, 3)
3
4 #modify tuples
5 new_tuple[2] = 4
6 print(new_tuple)
```

The 'Console' tab shows the error output:

```
3
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    new_tuple[2]= 4
TypeError: 'tuple' object does not support item assignment
> [
```

As we can see, we cannot assign a new value at position 2 because the Python Interpreter will throw a `TypeError` ('tuple' object does not support item assignment). We can get how many items we have in the Tuple using the `len()` or `length` built-in function:

```
# length
print(len(new_tuple))
```

The screenshot shows a Python development environment. On the left, there's a code editor window titled "main.py" containing the following Python code:

```
1 # tuples
2 new_tuple = (1, 2, 3)
3
4 # length
5 print(len(new_tuple))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution: "3".

Why do we need to use another data type like Tuple when everything can be achieved using Lists? Using Tuples have advantages over Lists and the first one is that Tuples are safer to use compared to Lists because of the ability to stay unchanged once created. When we are working in a team, we can use Tuples instead of Lists so we can indirectly tell others that Tuples are used because the data must remain the same. The other advantage is that tuples are processed much faster compared to lists. But not everything is so pink and good about tuples. Here are a few disadvantages:

- Cannot mutate the Tuples once created
- Cannot sort the Tuples
- Cannot reverse the Tuples
- No flexibility whatsoever

If you remember, from the dictionaries section, we mentioned Tuples when we used one of its methods called **items()**. This method will return a view object that consists of all the key-value pairs.

Here is the code that will bring back your memory:

```
car = {
    'brand': 'toyota',
    'generations': [1,2,3,4],
    'year': 2023
}
print(car.items())
```

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 ▼ car = {  
2     'brand': 'toyota',  
3     'generations': [1,2,3,4],  
4     'year': 2023  
5 }  
6 print(car.items())
```

On the right, a "Console" tab is active, displaying the output of the code execution:

```
dict_items([('brand', 'toyota'), ('generations', 2, 3, 4), ('year', 2023)])  
▶
```

I also mentioned that the keys in the Dictionary must be unique and we cannot change them, so no mutation is allowed. Therefore, we can have Tuples, numbers, and strings as a key but not Lists because they are mutable. Let's create a Dictionary with a key as Tuple:

```
car = {  
    'brand': 'toyota',  
    'generations': [1,2,3,4],  
    'year': 2023,  
    ('car','suv'): '2.5L engine'  
}
```

This is how we can access the values from a dictionary when we have Tuple as a key:

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 ▼ car = {  
2     'brand': 'toyota',  
3     'generations': [1,2,3,4],  
4     'year': 2023,  
5     ('car','suv'): '2.5L engine'  
6 }  
7 print(car['car', 'suv'])
```

On the right, a "Console" tab is active, displaying the output of the code execution:

```
2.5L engine  
▶
```

Similarly to Lists, we can use slicing to create a new tuple:

```
# slicing tuples  
new_tuple = (1,2,3,4)
```

```
new_tuple1 = new_tuple[0:2]
print(new_tuple1)
```

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # slicing tuples
2 new_tuple = (1,2,3,4)
3 new_tuple1 = new_tuple[0:2]
4 print(new_tuple1)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is selected and shows the output of the code execution:

```
(1, 2)
> █
```

We can unpack the Tuples values into variables:

```
#tuples unpacking
new_tuple = (1,2,3,4,6)
a,b,c, *rest = new_tuple
```

```
print(a)
print(b)
print(c)
print(rest)
```

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 #tuples unpacking
2 new_tuple = (1,2,3,4,6)
3 a,b,c, *rest = new_tuple
4
5 print(a)
6 print(b)
7 print(c)
8 print(rest)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is selected and shows the output of the code execution:

```
1
2
3
[4, 6]
> █
```

From the figure above, we can see that the last item (rest) we print is not a Tuple, it's a List and we can confirm this if we try to mutate one of its elements like the last value 6 to a new value 7:

```
#modify the last rest item because it is a list  
rest[1]= 7  
print(rest)
```

The screenshot shows a Python development environment with two panes. On the left, the code editor displays `main.py` with the following content:

```
1 #tuples unpacking  
2 new_tuple = (1,2,3,4,6)  
3 a,b,c, *rest = new_tuple  
4  
5 print(a)  
6 print(b)  
7 print(c)  
8 print(rest)  
9  
10 #modify the last rest item  
    because it is a list  
11 rest[1]= 7  
12 print(rest)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
1  
2  
3  
[4, 6]  
[4, 7]  
:> □
```

## Tuple methods

Let's demonstrate some of the tuple methods. For now, we have only two (count and index).

### Count()

It is a very easy method because it will return how many times the same value occurs in the Tuple collection.

```
#count  
new_tuple = (1,2,3,4,5,6,1)  
print(new_tuple.count(1))
```

```
main.py ×
```

```
1 #count
2 new_tuple = (1,2,3,4,5,6,1)
3 print(new_tuple.count(1))
```

```
Console Shell
```

```
2
> █
```

## index()

The index method will return the index or position of the element:

```
#index
print(new_tuple.index(3))
```

```
main.py ×
```

```
1
2 new_tuple = (1,2,3,4,5,6,1)
3 #index
4 print(new_tuple.index(3))
```

```
Console Shell
```

```
2
> █
```

The element 3 index is two, but what will happen if we look for a value that occurs multiple times in the Tuple? What index will be returned?

```
main.py ×
```

```
1
2 new_tuple = (1,2,3,4,5,6,1)
3 #index
4 print(new_tuple.index(3))
5 print(new_tuple.index(1))
```

```
Console Shell
```

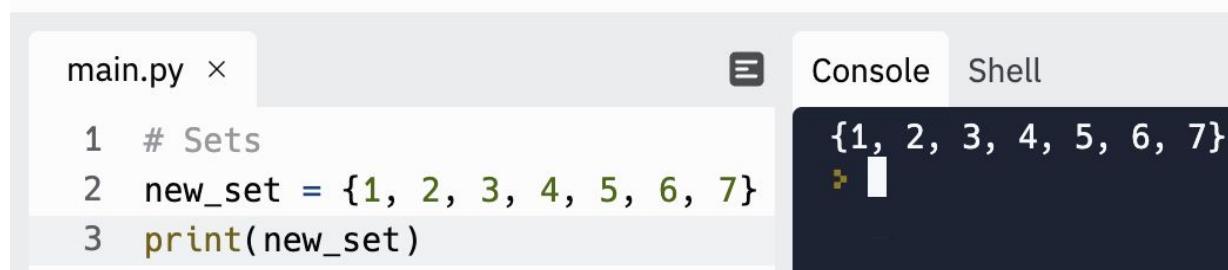
```
2
0
> █
```

When we try index(1), the index will be 0. The same element is in index six but the index method will return index zero because that is the first occurrence of the value.

## Sets in Python (data type and data structure)

The last data structure in Python is Set. Sets in Python are an unordered collection of unique elements/objects. Sets are iterable collection that does not contain duplicate elements. We can create Sets using the curly brackets just like dictionaries:

```
# Sets  
new_set = {1, 2, 3, 4, 5, 6, 7}  
print(new_set)
```



```
main.py ×  Console Shell  
1 # Sets  
2 new_set = {1, 2, 3, 4, 5, 6, 7}  
3 print(new_set)  
  
{1, 2, 3, 4, 5, 6, 7}
```

This is an unordered collection so you cannot be sure in which order the items will appear. It is also a collection of unique items so any duplicates will be discarded:

```
# Duplicates will be removed  
new_set = {1, 2, 3, 4, 5, 6, 7, 1}  
print(new_set)
```



```
main.py ×  Console Shell  
1 # duplicates will be removed  
2 new_set = {1, 2, 3, 4, 5, 6, 7, 1}  
3 print(new_set)  
  
{1, 2, 3, 4, 5, 6, 7}
```

We can add new items to the collection using the `add()` method:

```
new_set = {1, 2, 3, 4, 5, 6, 7}  
new_set.add(8)  
print(new_set)
```

The screenshot shows a Python development environment. On the left, the code editor displays a file named 'main.py' with the following content:

```
1 # add method
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 new_set.add(8)
4 print(new_set)
```

On the right, the terminal window (Console tab) shows the output of running the script:

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

The values in the Sets are located in the memory without any specific order but that will not cause problems when we access them because Sets do not contain duplicate values.

## Accessing Items in Sets

We know that Sets are an unordered collection with unique values but how can we access its elements? Can we do it like this:

```
# Accessing Set Items
new_set = {1, 2, 3, 4, 5, 6, 7, 6}
print(new_set[0])
```

If we run this code on the repl.it website, we will get this error:

The screenshot shows a Python development environment. On the left, the code editor displays a file named 'main.py' with the following content:

```
1 # accessing Set Items
2 new_set = {1, 2, 3, 4, 5, 6, 7, 6}
3 print(new_set[0])
4
```

On the right, the terminal window (Console tab) shows a traceback error:

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print(new_set[0])
TypeError: 'set' object is not subscriptable
```

As we can see, the Set items cannot be accessed by referring to an index since the Sets are an unordered collection of items. But there is another way we can access the Set items and it requires the use of loops so it loops or iterates over its elements. Because we haven't covered loops in Python, we will use the 'In' keyword and ask if a specified value is present in a Set. If the item is there, we will get Boolean True but if the item we are asking for is not there, we will get False:

```
# ask for item using the 'in' Python keyword
print(7 in new_set)
print(0 in new_set)
```

The screenshot shows a Python development environment. On the left, there's a code editor window titled "main.py" containing the following Python code:

```
1 # accessing Set Items
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 # ask for item using the 'in'
4     Python keyword
5 print(7 in new_set)
6 print(0 in new_set)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the following output:

```
True
False
> █
```

## Check the Set length

We can use the `len()` built-in function to get the exact number of items in the Set:

```
print(len(new_set))
```

The screenshot shows a Python development environment. On the left, there's a code editor window titled "main.py" containing the following Python code:

```
1 # accessing Set Items
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 # ask for item using the 'in'
4     Python keyword
5 print(len(new_set))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the following output:

```
7
> █
```

If for example we have duplicate values in the Set, the `len()` function will give us the length of the items without the duplicates because it will count only the unique values.

## Create a Set from a List

We know a List can contain items from any data type and they don't have to be unique, so duplicates are allowed. The question here is how we can convert a List to a Set and remove all of the duplicates. All of this is taken care of by one function called set() and this is one of the Python built-in functions:

```
#create a Set from a List using the set() function  
new_list = [1,2,3,1,5,4,5,6,7,6]  
new_set = set(new_list)  
print(new_set)
```



```
main.py ×
```

```
1 #create a Set from a List using  
2     the set() function  
3 new_list = [1,2,3,1,5,4,5,6,7,6]  
4 new_set = set(new_list)  
5 print(new_set)
```

```
Console Shell
```

```
{1, 2, 3, 4, 5, 6, 7}
```

## Convert a Set to a List

Python has a built-in function called list() that we can use to convert the existing Set to a new List:

```
# convert Set to a List  
new_set = {1, 2, 3, 4, 5, 6, 7, 7}  
new_list = list(new_set)  
print(new_list)
```



```
main.py ×
```

```
1 # convert Set to a List  
2 new_set = {1, 2, 3, 4, 5, 6, 7, 7}  
3 new_list = list(new_set)  
4 print(new_list)
```

```
Console Shell
```

```
[1, 2, 3, 4, 5, 6, 7]
```

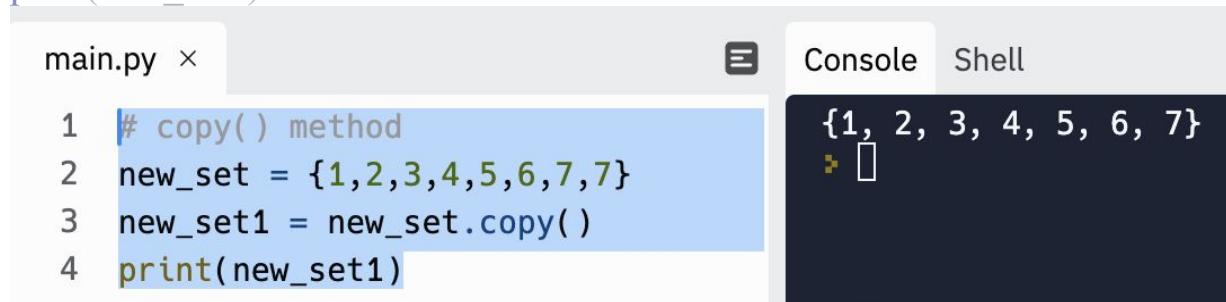
## Python Set Methods

Compared to Tuples, Sets have many methods and we have already used the add() method to insert new items in a Set. Let's start learning new methods in the following section.

## copy() method

We can create a Set copy using the copy method:

```
# copy() method
new_set = {1, 2, 3, 4, 5, 6, 7, 7}
new_set1 = new_set.copy()
print(new_set1)
```



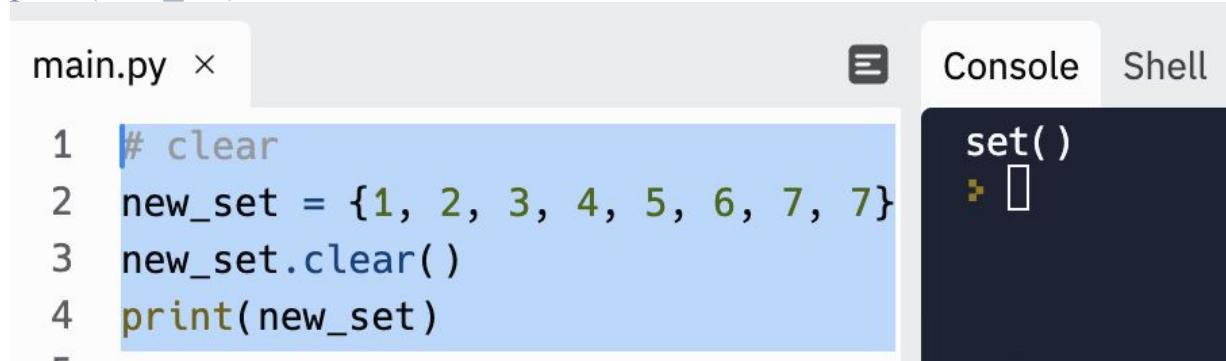
```
main.py ×
1 # copy() method
2 new_set = {1, 2, 3, 4, 5, 6, 7, 7}
3 new_set1 = new_set.copy()
4 print(new_set1)

Console Shell
{1, 2, 3, 4, 5, 6, 7}
> []
```

## clear() method

This method will remove all of the elements from a Set

```
# clear
new_set = {1, 2, 3, 4, 5, 6, 7, 7}
new_set.clear()
print(new_set)
```



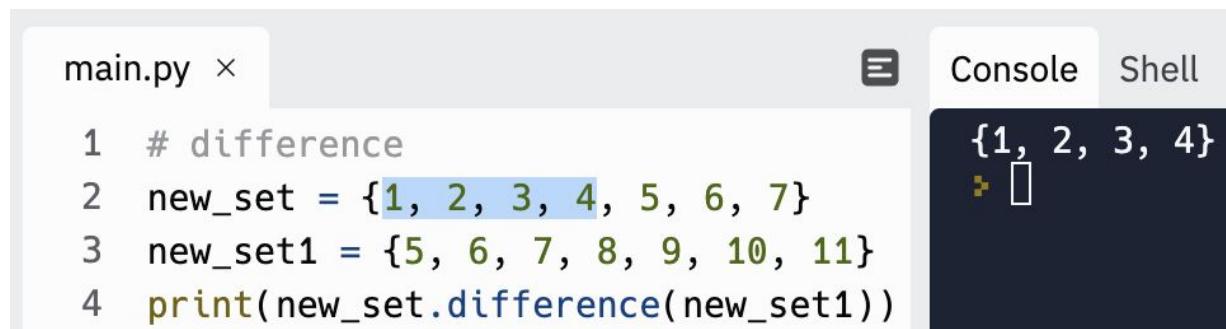
```
main.py ×
1 # clear
2 new_set = {1, 2, 3, 4, 5, 6, 7, 7}
3 new_set.clear()
4 print(new_set)

Console Shell
set()
> []
```

## difference() method

This is a very useful method that returns the difference between two or more sets as a new set. This does not mean it will include all of the different items from both sets. It means it will return a new set with items that exist in the first/original set:

```
# difference
new_set = {1, 2, 3, 4, 5, 6, 7}
new_set1 = {5, 6, 7, 8, 9, 10, 11}
print(new_set.difference(new_set1))
```



The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following code:

```
1 # difference
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 new_set1 = {5, 6, 7, 8, 9, 10, 11}
4 print(new_set.difference(new_set1))
```

To the right of the code editor are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution:

```
{1, 2, 3, 4}
```

As you can see, it will return a new set that includes the different items from the new set and will not include the duplicate items for both Sets.

### **difference\_update() method**

This method will modify the existing set. We use this method when we want to remove the items that exist in both sets. The difference() method returned a new set without the unwanted items and the difference\_update method will only remove the unwanted items from the original set

```
# difference_update() method
new_set = {1, 2, 3, 4, 5, 6, 7}
new_set1 = {5, 6, 7, 8, 9, 10, 11}
new_set.difference_update(new_set1)
print(new_set)
```

```
main.py ×
```

```
1 # difference_update() method
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 new_set1 = {5, 6, 7, 8, 9, 10, 11}
4 new_set.difference_update(new_set1)
5 print(new_set)
```

```
Console Shell
```

```
{1, 2, 3, 4}
> █
```

## discard() method

This method will remove or discard the specified item from the Set:

```
# discard() method
new_set = {1, 2, 3, 4, 5, 6, 7}
new_set.discard(7)
print(new_set)
```

```
main.py ×
```

```
1 # discard() method
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 new_set.discard(7)
4 print(new_set)
```

```
Console Shell
```

```
{1, 2, 3, 4, 5, 6}
> █
```

## intersection () method

Returns a new Set of items that contain the items that are the same between two or more sets. The new set will have items that exist in both sets:

```
# intersection method
new_set = {1, 2, 3, 4, 5, 6, 7}
new_set1 = {5, 6, 7, 8, 9, 10, 11}
print(new_set.intersection(new_set1))
```

```
main.py ×
```

```
1 # intersection method
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 new_set1 = {5, 6, 7, 8, 9, 10, 11}
4 print(new_set.intersection(new_set1))
```

```
☰
```

```
Console Shell
```

```
{5, 6, 7}
> □
```

If we remove the intersection method and use this shorthand symbol ‘&’ the result will be the same:

```
print(new_set & new_set1)
```

```
main.py ×
```

```
1 # intersection method
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 new_set1 = {5, 6, 7, 8, 9, 10, 11}
4 print(new_set & new_set1)
```

```
☰
```

```
Console Shell
```

```
{5, 6, 7}
> □
```

## intersection\_update() method

This method will remove the items from the first Set that are not present in the second Set and will update the original first set:

```
# intersection_update method
new_set = {1, 2, 3, 4, 5, 6, 7}
new_set1 = {5, 6, 7, 8, 9, 10, 11}
new_set.intersection_update(new_set1)
print(new_set)
```

```
main.py ×
```

```
☰
```

```
Console Shell
```

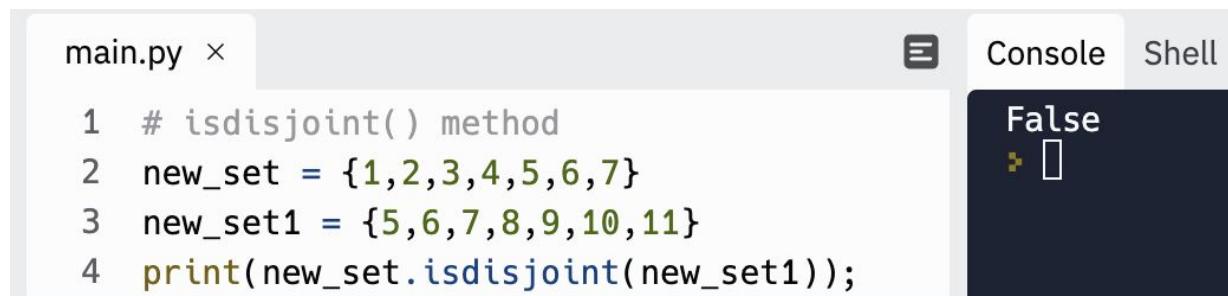
```
1 # intersection_update method
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 new_set1 = {5, 6, 7, 8, 9, 10, 11}
4 new_set.intersection_update(new_set1)
5 print(new_set)
```

```
{5, 6, 7}
> □
```

## isdisjoint() method

This method will return either Boolean True or False as a result. It will return a True only if the Sets we are comparing have different items and it will return False if at least one of the items from one Set can be found in the second Set.

```
# isdisjoint() method
new_set = {1, 2, 3, 4, 5, 6, 7}
new_set1 = {5, 6, 7, 8, 9, 10, 11}
print(new_set.isdisjoint(new_set1))
```



```
main.py ×
1 # isdisjoint() method
2 new_set = {1,2,3,4,5,6,7}
3 new_set1 = {5,6,7,8,9,10,11}
4 print(new_set.isdisjoint(new_set1));
```

Console Shell

```
False
> □
```

It returned False because 5,6,7 are the items that are present in both sets.

```
# isdisjoint() method
new_set = {1, 2, 3, 4}
new_set1 = {5, 6, 7, 8, 9, 10, 11}
print(new_set.isdisjoint(new_set1))
```



```
main.py ×
1 # isdisjoint() method
2 new_set = {1, 2, 3, 4}
3 new_set1 = {5, 6, 7, 8, 9, 10, 11}
4 print(new_set.isdisjoint(new_set1))
```

Console Shell

```
True
> □
```

As we can see, the **isdisjoint** method returns True because none of the items in both Sets are the same.

## Union()

A very easy method to understand because it returns a new Set that contains all of the items from the original and specified set. Union of all items.

```
# union method
new_set = {1, 2, 3, 4, 5, 6, 7}
new_set1 = {5, 6, 7, 8, 9, 10, 11}
print(new_set.union(new_set1))
```

```
main.py ×
1 # union method
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 new_set1 = {5, 6, 7, 8, 9, 10, 11}
4 print(new_set.union(new_set1))

Console Shell
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

As we can see, the union method creates a new Set with all of the items from both Sets but it excludes the duplicates. The union method can be replaced with the straight line | that we can write if we press shift + backslash and the result will be exactly the same:

```
print(new_set | new_set1)
```

## issubset()

This method returns Boolean True or False as a result. It will return True if all of the items that are in the first/original Set are included in the second Set:

```
# issubset()
new_set = {5, 6, 7}
new_set1 = {5, 6, 7, 8, 9, 10, 11}
print(new_set.issubset(new_set1))
```

```
main.py ×
1 # issubset()
2 new_set = {5, 6, 7}
3 new_set1 = {5, 6, 7, 8, 9, 10, 11}
4 print(new_set.issubset(new_set1))

Console Shell
True
```

Here is an example where the same method returns False:

```
# issubset() that returns false
new_set = {1, 2, 3, 4, 5, 6, 7}
new_set1 = {5, 6, 7, 8, 9, 10, 11}
print(new_set.issubset(new_set1))
```

The screenshot shows a Python code editor with a file named 'main.py'. The code is identical to the one above. To the right is a 'Console' tab showing the output: 'False' followed by a copy icon.

```
main.py ×
1 # issubset() that returns false
2 new_set = {1, 2, 3, 4, 5, 6, 7}
3 new_set1 = {5, 6, 7, 8, 9, 10, 11}
4 print(new_set.issubset(new_set1))
```

```
Console Shell
False ➤ └
```

## Issuperset()

This method will return Boolean True if all of the items from the specified/second Set exist in the original/first Set, otherwise it will return Boolean False.

```
# issuperset()
new_set = {5, 6, 7}
new_set1 = {5, 6, 7, 8, 9, 10, 11}
print(new_set.issuperset(new_set1))
```

The screenshot shows a Python code editor with a file named 'main.py'. The code is identical to the one above. To the right is a 'Console' tab showing the output: 'False' followed by a copy icon.

```
main.py ×
1 # issuperset()
2 new_set = {5, 6, 7}
3 new_set1 = {5, 6, 7, 8, 9, 10, 11}
4 print(new_set.issuperset(new_set1))
```

```
Console Shell
False ➤ └
```

Let's try an example where this method returns True:

```
# issuperset() true
new_set = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

```
new_set1 = {5, 6, 7, 8, 9, 10, 11}  
print(new_set.issuperset(new_set1))
```

main.py ×



Console

Shell

```
1 # issuperset() true  
2 new_set = {1, 2, 3, 4, 5, 6, 7, 8,  
3   9, 10, 11}  
4 new_set1 = {5, 6, 7, 8, 9, 10, 11}  
5 print(new_set.issuperset(new_set1))
```

True



## **Summary**

Congratulations! This was one of the most important chapters because you have covered a lot of the basic but core features of Python. We have covered many built-in features like functions, methods, and Python reserved keywords. The Python data types are so important because without them we can't move forward and learn the new features. I promise that the next chapter will be more challenging and more interesting.

# Chapter 2 – Python Conditional, Logical Operators and loops

So far, the way we executed the Python code was line by line, or as most developers call it, sequential code execution. In this chapter, we will learn how to write a code that based on a given condition will skip some statements and execute other statements. This will make our program smarter because it will be capable of making choices and decisions. In the previous chapter, we mentioned that using the loops we can iterate or loop over iterable such as Strings and Lists, therefore, we will use loops to execute a series of statements several times, repetitively. Let's start with learning the control structures first.

## Control Structures

The control structures will direct the order of execution of the statements in our program. This means that we can break the current flow of executing the statements. In the previous chapter, we learned about the Boolean data types (True and False) and in this chapter, we will see how useful they are for the control structures. The first control structure we will discuss is called if-statement.

### If-statement

This statement is one of the most common statements we can use to control the flow. To start using the if-statement, we need to use the keyword ‘if’, and here is the if-statement syntax:

```
if <expr>:  
    <statement>
```

As we can see from the syntax above, after the ‘if’ keyword, there is an expression or a condition. The condition is evaluated to Boolean True or False and based on the condition appropriate statement/statements are executed. Let us create a variable called **is\_developer** which will have a Boolean value of True:

```
is_developer = True
```

This variable will always be evaluated to Boolean True so we can write out the first if-statement like this:

```
if is_developer:  
    print('Wow this must be super exciting!')
```

Make sure that after the condition/expression, you always add the colon ( : ). The print function, as you can see, is indented and this means it belongs to the if-statement body. In the body, we can have one or more lines of code or statements. Make sure the line after the **if is\_developer** is always indented because the Python Interpreter relies on indentation to determine the scope of the code or where the code belongs. If we don’t have a proper indentation, it will throw an error. In JavaScript, the same if-statement can be written like this:

```
If(is_developer){  
    console.log('Wow this must be super exciting!');  
}
```

In JavaScript, we use curly brackets to define the scope or the body but if we have only one line of code, the curly brackets can be omitted. Let's get back to Python if-statement. How does this statement actually work? Basically, after the 'if' keyword we need to set the condition and it can be very simple like in our example, or much more complex. The condition must be evaluated to True in order to execute the statement/statements in the body. The body is the statement/statements that go right below the if-statement and it must be indented to indicate that the body belongs to the above if-statement. Now when the interpreter reads the first line with the if keyword, it will check if the variable `is_developer` is True, and in our case, the `is_developer` will always be True because we initialized the variable before the if-statement. Because the condition is true, the interpreter will know that next it should go into the body and execute the statement. In our case, we have only one `print()` function in the body but as I mentioned, we can have multiple lines of code/statements. When the interpreter executes the statement/statements in the body, after that, it will exit the if-block. Let us add one more line after the if-block but this time without indentation and run this code in:

```
# If statements
is_developer = True
if is_developer:
    print('Wow this must be super exciting!')
print('New line')
```



The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 is_developer = True
2 ▼ if is_developer:
3     print('Wow this must be super exciting!')
4 print('New line')
```

On the right, there is a "Console" tab showing the output of the code execution:

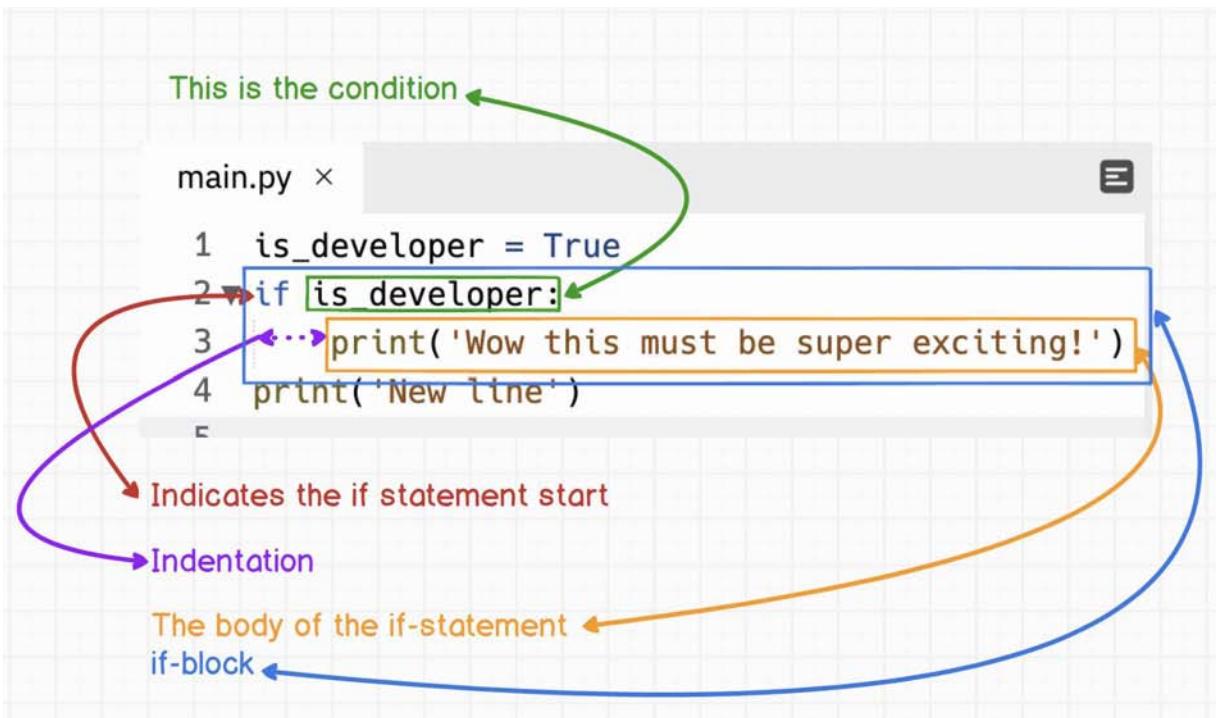
```
Wow this must be super exciting!
New line
```

The second `print()` does not have an indentation and this will tell the Python interpreter that the second `print()` function does not belong in the if code block. Notice that in line two from the figure above, next to the `if` keyword there is a black triangle facing down so we can click on it and observe what will happen:

```
main.py ×
```

```
1 is_developer = True
2 ► if is_developer: ...|
4 print('New line')
5
```

As you can see now, the body is not visible anymore because it belongs to this if-block and if we click it again, we will get the body of the if-statement. These are small things that you should know and they are very important. In the figure below, you will see all of the keywords that we learned so far:



If we run the following code, the output will be:

```
main.py ×
```

```
1 is_developer = True
2 ▼ if is_developer:
3   print('Wow this must be super exciting!')
4   print('New line')
5
```

```
Console Shell
```

```
Wow this must be super exciting!
New line
> █
```

Because the if-statement condition is evaluated to true, the if-block will be executed. When the last statement in the if-block is executed, the Interpreter will go out of the if-block and execute the next print(). But what if we change the **is\_developer** variable to have a Boolean False value, what will happen?

The screenshot shows a code editor window titled "main.py". The code contains an if-block where the condition is set to False:

```
1 is_developer = False
2 ▼ if is_developer:
3     print('Wow this must be super exciting!')
4 print(['New line'])
```

Below the code editor is a terminal window with tabs "Console" and "Shell". The "Console" tab is active, showing the output of the run command:

```
New line
```

As we can see, the output is different. The if-statement condition will be evaluated to False, therefore the body of the if-block will never be executed. This means the statement/statements in the if-block will never run if the condition is evaluated to False.

## if-else statement

The if-else statement is when we want to set an alternative path, for example, if the condition is evaluated to true, execute the statement/statements inside the body or if the condition is evaluated to False, execute the statements in the else clause. The syntax of the if-else statement is:

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

Let's rewrite the previous if-block example to include the else clause, but this time we need to make the condition be evaluated to False:

```
# if-else
is_developer = False
```

```
if is_developer:  
    print('Wow this must be super exciting!')  
else:  
    print('What is your profession then?')
```

If we run the new code:

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following code:

```
1 is_developer = False  
2 if is_developer:  
3     print('Wow this must be super exciting!')  
4 else:  
5     print('What is your profession then?')
```

A green "Run" button is visible above the code editor. To the right, a terminal window titled "Console" shows the output of the code execution:

```
What is your profession then?  
> █
```

We can see that the else clause statement/statements are executed because the if condition was evaluated to False. Now we have two options based on one condition. If the condition is evaluated to True, run the statements in the body or if the condition is evaluated to False, run the statements in the else clause. In both cases, the indentation and colon are very important.

## elif

The if-else statement is great but it only covers two scenarios - if the condition is true, do this and if the condition is false, execute the else clause statements. We can do much more than just these two operations because there is a syntax that allows us to define more than two options. The name of this is elif conditional statement, a shorthand for else-if. The syntax for elif is:

```
if <expr>:  
    <statement(s)>  
elif <expr>:  
    <statement(s)>  
elif <expr>:  
    <statement(s)>  
...  
...
```

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

Maybe the syntax does look a bit confusing at first but it is not that hard in reality. The **Python** interpreter will evaluate the **if-expression** first, and if it's **True**, it will execute the code below the **if-statement**, and if the condition is evaluated to **False**, it will go to the first **elif** statement where we have another condition. The Interpreter will evaluate the condition (in the **elif** statement) to **True** or **False** and if it's **True**, then it will execute the statements inside the **elif** block and if the condition is **False**, it will jump out and go to the next **elif** statement where there is another condition. Finally, if every condition is evaluated to **False**, the final **else-clause** statement/statements will be executed.

Here is one example:

```
# elif  
is_developer = False  
is_employed = True  
if is_developer:  
    print('Wow this must be super exciting!')  
elif is_employed:  
    print('Great at least you have a job :)')  
else:  
    print('What is your profession then?')
```

The screenshot shows a Python development environment with a code editor and a terminal window. The code editor has a file named 'main.py' with the following content:

```
1 # elif  
2 is_developer = False  
3 is_employed = True  
4 ▼ if is_developer:  
5     print('Wow this must be super exciting!')  
6 ▼ elif is_employed:  
7     print('Great at least you have a job :)')  
8 ▼ else:  
9     print('What is your profession then?')
```

The terminal window is titled 'Console' and shows the output of the program:

```
Great at least you have a job :)
```

Let's explain how the above example is working. The if-condition, as we can see, will be evaluated to False because the `is_developer` is set to False and then it will go to the next elif-statement where the condition will be evaluated to True. Because the condition in the elif is True, the statement inside the elif-block will be executed. Inside the elif block, we have only one line of code or a simple message that gets printed. If we set both of the if-statement and elif-statement conditions to False and run this code again, we have:

The screenshot shows a Python code editor with a file named `main.py`. The code contains the following:

```
1 is_developer = False
2 is_employed = False
3▼ if is_developer:
4    print('Wow this must be super exciting!')
5▼ elif is_employed:
6    print('Great at least you have a job :)')
7▼ else:
8    print('What is your profession then?')
```

Below the code editor is a terminal window titled "Console". It displays the message:

```
What is your profession then?
> █
```

The else clause statement will be executed. So far in the **if** and **elif** conditions, we have used simple Boolean values that are either True or False, and this is okay and normal but the condition itself can be a much more complex expression like this:

```
x = 10

if x > 11:
    print('The value of x is smaller then 11')
elif x > 5:
    print('The value of x is bigger than 5')
else:
    print('The value of x is undefined')
```

```
main.py x
1 x = 10
2
3▼ if x > 11:
4     print('The value of x is smaller then 11')
5▼ elif x > 5:
6     print('The value of x is bigger than 5')
7▼ else:
8     print('The value of x is undefined')

Console Shell
The value of x is bigger than 5
```

## Understanding Python Indentation

Indentation in Python is very important and it refers to the spaces at the beginning of the code. If you are a JavaScript developer, then you know the spacing is not that big of a deal but we (developers) do it anyway just to style our code. Proper indentation makes the code much more readable and easier to understand. In Python, the indentation will indicate that the code is different from the one above, belongs to a different block, and have its own scope. You will get an error if you avoid indentation. The number of empty spaces is not important as long as there is at least one. If you use more than one space, make sure you apply the same number of spaces throughout the entire file, otherwise, you will get an error called **IndentationError** or unexpected indent:

```
▶ Run Publish Invite
main.py x
1 x = 10
2
3▼ if x > 11:
4     print('The value of x is smaller then
      11')
5▼ elif x > 5:
6     print('The value of x is bigger than 5')
7 else:
8     print('The value of x is undefined')

Console Shell
File "main.py", line 8
    print('The value of x is undefined')
          ^
IndentationError: expected an indented block
```

As you can see from the figure above, the code below the last else-clause is without indentation and the Interpreter will underline it in red and throw an error if we try to run the code. Another interesting thing is that developers prefer making the indentation by pressing the tab key or simply by adding multiple spaces. The number of spaces they put is usually four but so far in repl.it.com, there are two spaces added for us. This can be changed in the settings, where you can choose the indent type (spaces or tabs) and set the indent size:

The screenshot shows the repl.it Python editor interface. On the left is a sidebar with the following settings:

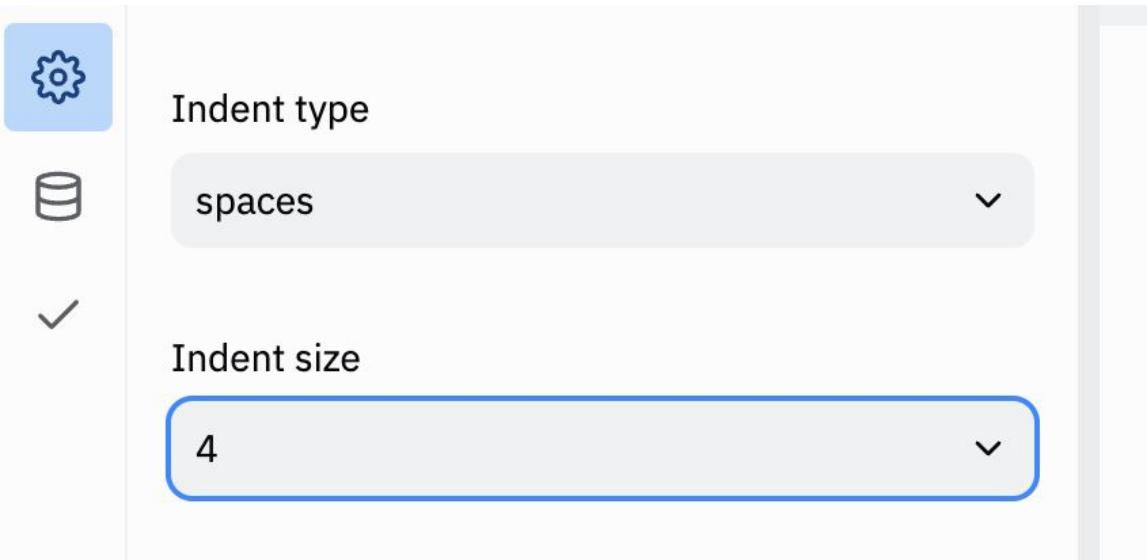
- Layout: side-by-side
- Font size: normal
- Indent type: spaces (highlighted with a blue border)
- Indent size: 2

The main area contains a code editor titled "main.py" with the following Python code:

```
1 x = 10
2
3▼ if x > 11:
4 | print('The value of x is smaller than
5 | 11')
5▼ elif x > 5:
6 | print('The value of x is bigger than 5')
7 else:
8 | print('The value of x is undefined')
```

The line "print('The value of x is undefined')" is underlined with a red wavy line, indicating an error.

If you want four spaces to be added, then set the indent type to spaces and increase the indent size from two to four like in the following figure:



As you can see, indentation is very important in Python.

## Python Operators

Operators are very important in all programming languages. Python classifies the operators into different groups:

- Arithmetic
- Assignment
- Comparison
- Logical
- Identity
- Membership
- Bitwise

## Arithmetic Operators

The arithmetic operators are used with numbers so we can do some mathematical calculations:

Operator	Name	Example
----------	------	---------

+	Addition	$a + b$
-	Subtraction	$a - b$
*	Multiplication	$a * b$
/	Division	$a / b$
%	Modulus	$a \% b$
**	Exponential	$a ** b$
//	Floor division	$b // b$

We have explained the arithmetic operators in the first chapter so I just wanted you to have the table as a reference. We have also used the ‘+’ operator to concatenate different Strings.

## Assignment Operators

As we know, assignment operators are used when we want to assign a value to a variable. We have covered the basic assignment operators and augmented assignment operators in the previous chapter because they play a very special role in Python programming. The function of the augmented assignment operator is to combine the functionality of the arithmetic and bitwise operator with the assignment operator. Augmented assignment operators are nothing but short syntax so we can perform a binary operation and assign the results to one of the operands.

For example:

```
a += 7
```

The above expression is the same as the following one:

```
a = a + 7
```

Operator	Description	Example	Same as
=	Assign; The value of the right side of the equal ‘=’ will be	$a = 3$	$a = 3$

	assigned to the left side operand		
<code>+=</code>	Add and Assign; This will add the right-side operand with the left-side operand and will then assign value to the left operand	<code>a += 2</code>	<code>a = a + 2</code>
<code>-=</code>	Subtract and Assign; This will subtract the right-side operand from the left-side operand and then assign the value to the left operand.	<code>a -= 2</code>	<code>a = a - 2</code>
<code>*=</code>	Multiply AND; Multiply the right-side operand with the left-side operand and then assign the value to the left operand	<code>a *=4</code>	<code>a = a * 4</code>
<code>/=</code>	Divide AND;	<code>a /= 3</code>	<code>a = a / 3</code>

	Divide the left-side operand with the operand on the right-side and then assign the value to the left operand		
%=	Modulus AND; This will take modulus from the left and right operands and assign the result to the left operand	a %= 6	a = a % 6
//=	Divide floor; Divide the left operand with the right operand and then assign the floor value to the left operand	a // 4	a = a // 4
**=	Exponent; Calculate the exponent using operands and assign the value to the left operand	a **= 4	a = a ** 4
&=	This operator will perform	a &= 2	a = a & 2

	bitwise AND on operands and assign the value to the left operands		
$ =$	This operator will perform bitwise OR on operands and assign the value to the left operand	$a  = 3$	$a = a   3$
$^=$	Perform bitwise xOR on operands and assign the value to the left operand	$a ^= 3$	$a = a ^ 3$
$>>=$	Perform bitwise right shift on operands and assign value to the left operand	$a >>= 3$	$a = a >> 3$
$<<=$	Perform bitwise left shift on operands and assign value to the left operand	$a <<= 3$	$a = a << 3$

Here are a few examples:

### Assign operator ‘=’

```
m = 3  
n = 4  
x = m + n  
print(x)
```

```
#Output: 7
```

### Add and Assign +=

```
m = 3  
n = 4  
m += n  
print(m)  
#Output: 7
```

### subtract and assign -=

```
m = 5  
n = 3  
m -= n  
print(m)  
#Output: 2
```

### multiply and assign \*=

```
m = 5  
n = 3  
m *= n  
print(m)  
# Output: 15
```

### divide and assign /=

```
m = 5  
n = 3
```

```
m /= n  
print(m)  
# Output: 1.666666666666667
```

### **modulus and assign %=**

```
m = 5  
n = 3  
m %= n  
print(m)  
# Output: 2
```

### **divide(floor) and assign // =**

```
m = 5  
n = 3  
m //= n  
print(m)  
# Output: 1
```

### **exponent and assign \*\*=**

```
m = 5  
n = 3  
m **= n  
print(m)  
# Output: 125
```

### **bitwise AND &=**

```
m = 5  
n = 3  
m &= n  
print(m)  
# Output: 1
```

### **bitwise OR |=**

```
m = 5  
n = 3  
m |= n  
print(m)
```

### bitwise XOR ^=

```
m = 5  
n = 3  
m ^= n  
print(m)  
# Output:6
```

### bitwise right shift and assign

```
m = 5  
n = 3  
m >>= n  
print(m)  
# Output: 0
```

### bitwise left shift and assign

```
m = 5  
n = 3  
m <<= n  
print(m)  
# Output: 40
```

## Comparison Operators

We can use the comparison operators when we need to compare two values:

Operator	Name	Example	Output
<code>==</code>	Equal	<code>x = 3 y = 4 print(x == y)</code>	False; because 3 is not equal to 4
<code>&gt;</code>	Greater than	<code>x = 4</code>	True; because 4 is greater than 3

		y = 3 print(x > y)	
<	Less than	x = 4 y = 3 print(x < y)	False; because 4 is not less than 3
>=	Greater than or equal to	x = 4 y = 3 print(x >= y)	True; because 4 is greater or equal to 3
<=	Less than or equal to	x = 3 y = 3 print(x <= y)	True; because 3 is greater or equal to 3
!=	Not equal	x = 4 y = 3 print(x != y)	True; because 4 is not equal to 3

The Code:

```
# Comparison Operators
x = 4
y = 3
```

```
# == equal
print(x == y)
# Output False
```

```
# > greater than
print(x > y)
# Output True
```

```
# < less than
print(x < y)
# Output False
```

```
# >= greater than or equal to
print(x >= y)
# Output True
```

```
# >= less than or equal to
```

```
print(x <= y)
```

```
# Output False
```

```
# != Not Equal
```

```
print(x != y)
```

```
# Output True
```

main.py ×

```
1 # Comparison Operators
2 x = 4
3 y = 3
4
5 # == equal
6 print(x == y)
7 # Output False
8
9 # > greater than
10 print(x > y)
11 # Output True
12
13 # < less than
14 print(x < y)
15 # Output False
16
17 # >= greater than or eqaul to
18 print(x >= y)
19 # Output True
20
21
22 # >= less than or eqaul to
23 print(x <= y)
24 # Output False
25
26 # != Not Equal
27 print(x != y)
28 # Output True
```

False  
True  
False  
True  
False  
True  
▶

# Logical operators

The logical operators are very useful because they give us the option to check multiple conditions at the same time. The logical operators are used to combine conditional statements:

<b>Operator</b>	<b>Description</b>	<b>Example</b>	<b>Output</b>
and	If both statements are true, it will return true	<code>a = 4 print(a &gt; 3 and a &lt;10)</code>	True; because 4 is greater than 3 AND 4 is less than 10
or	Only one of the statements needs to be true to return True	<code>a = 4 print(a &gt; 3 and a &gt;10)</code>	True; because we only need one of the statements to be true and 4 is greater than 3 is evaluated to True, therefore, the whole expression is evaluated to True
not	Reverse the result, if it's True will convert to False and vice versa	<code>a = 4 print(not(a &gt; 3 and a &gt;10))</code>	False; The inner expression within the 'not' keyword is evaluated to True but the 'not' keyword will reverse the result, therefore, the final return will be False

I know the table is self-explanatory but let's see how we can use the above operators with conditional statements like the if-statement. Let's create a program that will tell us if we can legally drive a car based on our age. For example, let's say we have a driver's license and we are 16 years of age so we are allowed to drive:

```
my_age = 16
is_licenced = True
if is_licenced and my_age >= 16:
    print('Hop on for a ride')
else:
    print('Sorry buddy, I\'m driving today!')
```



The screenshot shows a Python code editor interface. At the top, there is a toolbar with icons for file operations and a green 'Run' button. Below the toolbar, the file 'main.py' is open. The code in the editor is identical to the one shown above. To the right of the code editor is a 'Console' tab which displays the output of the run command: 'Hop on for a ride'. There is also a 'Shell' tab next to it.

As you can see from the code above, I have used the logical 'and' operator in conjunction with the comparison operator to evaluate the if-condition. Both sides (left and right) of the 'and' logical operator have to be true so the entire expression can be evaluated to True. We know that **is\_licenced** will be evaluated to True immediately but for the second part of the expression, we have used the comparison operator to check whether the driver's age is greater than or equal to 16 years. I hope you can see the power of the operators. If the logical operator in the middle is changed to the 'or' operator, then only one of the two statements (left or right) will need to be evaluated to True for the entire condition to be evaluated to true. Let's change just the variable **my\_age** to have a value of 15:

The screenshot shows a Python code editor with a file named 'main.py'. The code contains an if statement that prints 'Hop on for a ride' if the condition 'is\_licensed or my\_age >= 16' is True. The output window shows the result 'Hop on for a ride'.

```
1 my_age = 16
2 is_licensed = True
3 if is_licensed or my_age>=16:
4     print('Hop on for a ride')
5 else:
6     print('Sorry buddy, I\'m driving today!')
```

Hop on for a ride

As we can see, because the **is\_licensed** is evaluated to True, the entire condition will be evaluated as True and therefore the code in the if-block will be executed.

## Truthy vs Falsy values

I think we've had enough of the operators for now, we will cover more of them in this chapter but let's shift our focus to something more interesting in Python. This new topic is called truthy and falsy values. Up until this point, we have covered the two most common Boolean values (True and False), but Python has truthy and falsy values as well. The Boolean values are very important when we want to make our program smart and make decisions based on the condition we set. For example, let's create a very simple if-statement with a simple condition:

```
# Truthy vs Falsey
if 6 > 4:
    print("True")
else:
    print("False")
```

The output will be True because 6 is greater than 4. But can you guess the output of this example?

```
m = 10
if m:
    print("True")
```

The screenshot shows a Python code editor with a file named 'main.py'. The code contains an if-else statement where the condition is a single variable 'm' assigned the value 10. The 'else' block prints 'False'. In the 'Console' tab, the output is 'True', which is unexpected given the code. The tabs at the top are 'main.py ×', 'Console', and 'Shell'.

```
else:  
    print("False")  
  
main.py ×  
1 m = 10  
2 ▼ if m:  
3     print("True")  
4 ▼ else:  
5     print("False")  
6 |  
  
Console Shell  
True  
▶
```

The output is True, but why? As you can see in the condition, we don't actually have an expression so it can be evaluated to either True or False, just a single variable 'm' which has a value of 10 assigned before the if-statement. The question is, why is this variable in the condition evaluated to True? The answer lies in the Truthy and Falsy values. In Python, in order for a value to be evaluated to True or False, it does not have to be part of an expression and this is based on some rules that are built-in in the language, and of course, the Interpreter knows these rules. In the following section, I will discuss the Boolean Context.

## Boolean Context

The Boolean context is very important because it requires a value to be in one of the two states, True and False, therefore, any statement or a value we have in the if-condition must be converted before in the background to True or False. I know this can be confusing but in Python, by default, all of the values are considered truthy except for the following list of values which are always 'falsy':

- Constant False
- Constant None
- Zero (0) of any numeric type
- Float 0.0
- Complex Obj
- Empty list []

- Empty tuples ()
- Empty dictionaries {}
- Empty Sets sets()
- Empty strings “”
- Empty range(0)

Let's run the above list of values and check if they are actually considered as falsy:

The screenshot shows a Python development environment. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # falsy
2 n = 0
3 s = ''
4 print(bool(n))
5 print(bool(s))
6 print(bool(False))
7 print(bool(None))
8 print(bool([]))
9 print(bool({}))
10 print(bool(()))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is selected and displays the output of the code, which consists of ten "False" values, each preceded by a yellow arrowhead indicating a new line.

Line	Value
1	False
2	False
3	False
4	False
5	False
6	False
7	False
8	False
9	False
10	False

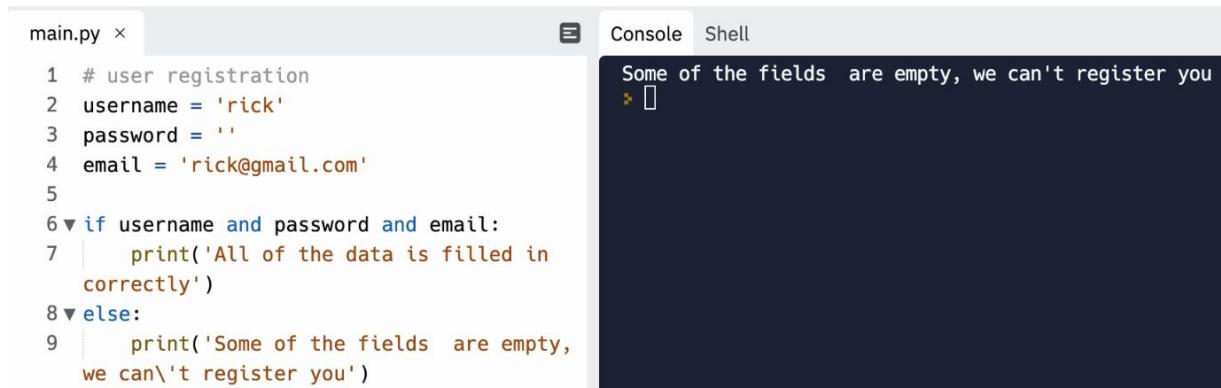
## The built-in bool() function

We can check if the value is truthy or falsy using the `bool()` function which according to the Python documentation will always return a Boolean value (True or False). The function takes a parameter/value but we need to use the `print` function to get the result. As you can see from the figure above, I have tested some of the falsy values using this `bool` function. The results are printed on the right side and they are all `False` values. I hope you understand why Python needs to have these values. I will provide a very simple example that will help you to understand this new concept. For example, let's say we have a subscription website and we want the users during the registration process to fill in all of the data correctly. Imagine the registration form consists of three fields that the user needs to fill in like the name, email, and password. Most of the time, we check in the background if the user enters the data in the right format but because of the truthy and falsy values, we can now escape at least one of the steps or when the user misses one of the fields:

```
# user registration
username = 'rick'
password = ""
email = 'rick@gmail.com'

if username and password and email:
    print('All of the data is filled in correctly')
else:
    print('Some of the fields are empty, we can\'t register you')
```

As we can see from the code above, the `password` field is an empty string and empty strings in Python are evaluated as falsy values which will not allow the code in the `if`-block to be executed and to store, for example, this user 'rick' will never be stored in the database without a password:



```
main.py ×
1 # user registration
2 username = 'rick'
3 password = ''
4 email = 'rick@gmail.com'
5
6▼ if username and password and email:
7     print('All of the data is filled in
correctly')
8▼ else:
9     print('Some of the fields are empty,
we can\'t register you')

Console Shell
Some of the fields are empty, we can't register you
> □
```

# Ternary Operator

So far, we have used the if, else, and elif conditional statements, and based on the condition, either True or False, one set of statement/statements will be executed. This makes our program branch the code. But there is another way we can achieve the conditional logic using a shorthand syntax called **ternary operator**. In some literature, this is called **conditional expression**. This will make your code cleaner but I admit it can look strange. Remember statements are different from expressions. Expressions are operations that will evaluate something based on a condition. Expressions are a representation of value. In Python, expressions are relatively new, meaning they existed since Python version 2.5.

The syntax of Ternary Operator:

```
[condition_if_true] if [expression] else [condtition_if_false]
```

Let's create an example::

```
is_old = True  
is_allowed = "You can drink tonight" if is_old else "You cannot drink  
tonight coz of your age"  
print(is_allowed)  
on] else [condtition_if_false]
```

If we run the above code, the output will be ‘You can drink tonight.’ Let’s start by explaining how this operator works. We can see that the variable **is\_old** has the Boolean value True and the variable **is\_allowed** will be the variable that will hold the result of what the ternary operator returns so its value at the end will be either ‘You can drink tonight’ or ‘You cannot drink tonight coz of your age.’ Because the condition **is\_old** is evaluated to True, the **is\_allowed** will be assigned the value ‘You can drink tonight.’ If the condition was evaluated to false, the message after the else-clause will be the one to be assigned to the variable **is\_allowed**. In the following figure, I explain what section of the ternary operator will execute when the condition is True and what section when the condition is False:

```

main.py ×
1 is_old = True
2 is_allowed = "You can drink tonight" if is_old else "You cannot drink tonight coz of your age"
3 print(is_allowed)
4

```

Here are the screenshots for both scenarios:

```

test1 🐍 ⏱
▶ Run
main.py ×
1 is_old = True
2 is_allowed = "You can drink tonight" if is_old else
   "You cannot drink tonight coz of your age"
3 print(is_allowed)

```

Console Shell

```
You can drink tonight
> []
```

The above figure shows how the ternary operator is used to assign a value to the **is\_allowed** variable based on a given condition (**is\_old**).

```

i / test1 🐍 ⏱
▶ Run
↑ Publish
main.py ×
1 is_old = False
2 is_allowed = "You can drink tonight" if is_old else
   "You cannot drink tonight coz of your age"
3 print(is_allowed)

```

Console Shell

```
You cannot drink tonight coz of your age
> []
```

The above figure is when the **is\_old** variable is set to Boolean False and that will make the if-condition be evaluated to False, therefore the **else-clause** message will be assigned as a value to the variable **is\_allowed**. As we can see, ternary operators is not that scary, but it takes time to get used to this syntax.

## Short-Circuiting

In order to explain how short-circuiting works in Python, I would use one of the previous examples with the driver's license.

The screenshot shows a Python code editor window titled "test1". The file "main.py" contains the following code:

```
1 my_age = 16
2 is_licensed = True
3 if is_licensed or my_age>=16:
4     print('Hop on for a ride')
5 else:
6     print('Sorry buddy, I\'m driving today!')
```

The output window shows the result of running the code: "Hop on for a ride".

The if condition will be evaluated to True if only one of **is\_licensed** or **my\_age >=16** statements are evaluated to true. Short-circuiting is essential because it will save time and increase our program performance. Therefore, when the Python interpreter evaluates the expression that involves the 'or' operator, it will stop immediately one of the operands returns True. Let's go line by line and explain how the Python interpreter works:

- 1) **my\_age = 16;** The interpreter will assign a value of 16 to the variable **my\_age** in the first statement
- 2) The second statement is assigning the value True to the **is\_licensed** variable
- 3) The third statement is the if-block:

Now the interpreter will evaluate what is after the if keyword and the first statement is the **is\_licensed**. Obviously, this one will immediately be evaluated to True and the Interpreter will go and evaluate what is next but when it sees the 'or' operator, the Interpreter will stop because the first operand already returned true.

- 4) Then it will execute the `print()` function below the if block

This is how short-circuiting works in Python, it saves time and increases the performance because it can skip the evaluation of the next expression because at least one returned True.

## Python Identity Operators

The identity operators will compare if the Objects are the same and if they point to the same memory location. We haven't learned what Objects are but I know you will understand the examples because I will use Lists:

Operator	Description	Example	Output
Is	Will return true if both of the variables are the same object	a = ["apple"] b = ["apple"] c = a print (a is c)	True; because c is the same object as a
Is not	Will return true if both variables are not the same object	a = ["apple"] b = ["apple"] c = a print (a is not b)	True; because a is not the same object as b, although they have the same values

Here are a few more examples of the 'is' operator:

```
# is operator

a = ["apple"]
b = ["apple"]
c = a
print (a is c)
# this returns true because a is same object as c
```

```
main.py ×
```



Console

Shell

```
1 # is operator
2
3 a = ["apple"]
4 b = ["apple"]
5 c = a
6 print (a is c)
7 # this returns true because a
    is same object as c
```

```
True
```



If we try to check if the list/object ‘a’ is equal to object ‘b’ then the result will be false because ‘a’ is not the same object as ‘b’:

```
print (a is b)
```

```
main.py ×
```



Console

Shell

```
1 # is operator
2
3 a = ["apple"]
4 b = ["apple"]
5 c = a
6 print (a is b)
7 # this returns false because
    a is not same as b
```

```
False
```



What will happen if we use the comparison operator == to compare the object ‘a’ and the object ‘b’? The result will be True because the content of the object ‘a’ is equal to the content of ‘b’

```
print (a == b)
```

The screenshot shows a Python development environment with two tabs: 'Console' and 'Shell'. The 'Console' tab is active, displaying the following output:

```
True  
False  
True  
▶ []
```

To the left of the console, there is a code editor window titled 'main.py' containing the following Python code:

```
1 # is  
2 a = ["apple"]  
3 b = ["apple"]  
4 c = a  
5 print(a is c)  
6 # this returns true because a is  
# same object as c  
7 print(a is b)  
8 # this returns false because a is  
# not same as object b  
9 print(a == b)  
10 # this returns true because the  
# content of a is same as the  
# content of object b
```

Here are a few more examples of the ‘is not’ operator:

```
# is not  
a = ["apple"]  
b = ["apple"]  
c = a  
  
print(a is not c)  
#this returns false because a is the same as object c
```

```
print(a is not b)  
# this will return true because object a is not same as the object as b even if  
# they have the same content/values
```

```
print(a != b)
# The comparison operator != will return false because a is equal to b
```

The screenshot shows a Jupyter Notebook interface. On the left, there is a code cell with the following Python code:

```
main.py ×
1 # is not
2 a = ["apple"]
3 b = ["apple"]
4 c = a
5
6 print(a is not c)
7 #this returns false because a is the
8     same as object c
9
10 print(a is not b)
11 #this will return true because object a
12     is not same as the object as b even if
13     they have the same content/values
14
15 print(a != b)
16 #The comparison operator != will return
17     false because a is equal to b
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is selected, showing the output of the code:

```
False
True
False
> []
```

## The ‘is’ operator versus the ‘==’

Although I have included the difference between the ‘is’ and ‘==’ in the previous section, I think you deserve to know a little bit more. Can you guess the output from the following expression?

```
print(True == 1)
```

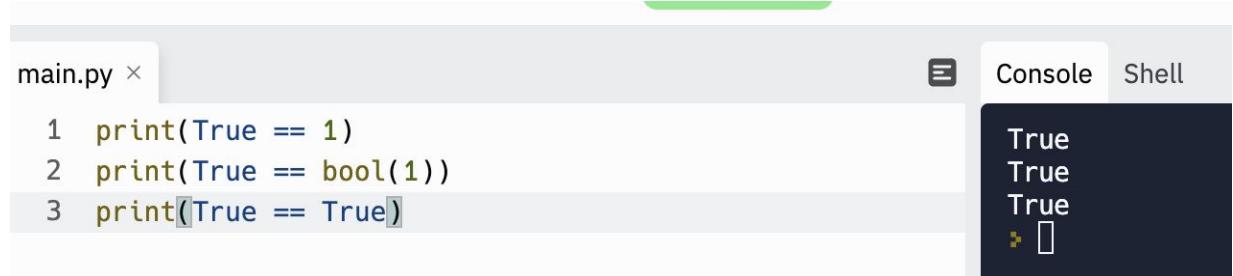
The output will be True, but why? Well, the equality operator will check if both of the operands are equal to each other. But here we are evaluating two different Data types, Boolean True on the left and Integer number on the right. Because they are two different types, the Interpreter will try to convert them to the same type. Therefore the integer will be converted to Boolean using the `bool()` function, and `bool(1)` is evaluated to True:

```
print(True == bool(1))
```

The above code is exactly the same as the following one:

```
print(True == True)
```

Now you know why `True == 1` is evaluated to True:

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing three lines of Python code: `1 print(True == 1)`, `2 print(True == bool(1))`, and `3 print(True == True)`. On the right, there is a "Console" tab showing the output of the code: "True", "True", and "True" respectively. A small yellow arrow icon is visible next to the last output line.

```
main.py ×
1 print(True == 1)
2 print(True == bool(1))
3 print(True == True)
```

Console Shell

```
True
True
True
> □
```

Let's check the equality of this code:

```
#2 checking for equality
print('' == 1)
```

The output will be False because the empty Strings in Python are considered as Falsy value or Boolean False and Integer 1 will be evaluated as True, so we are trying to check for equality between (`False == True`) and that is why we got False in the first place:

```
#2 checking for equality
print('' == 1)
print(bool('') == bool(1))
print(False == True)
```

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following code:

```
1
2 #2 checking for equality
3 print('' == 1)
4 print(bool('') == bool(1))
5 print(False == True)
6
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is selected and displays the following output:

```
False
False
False
> █
```

When we use double equality, the best scenario is to compare values that belong to the same data type but if they are different, then let's hope that Python using the truthy and falsy values rules will solve the problem for us. Now you know that the equality operator will check for the equality of the values. On the other hand, the 'is' operator will check if the location in memory where the value is stored is exactly the same between both of the objects. The 'is' operator will not perform type conversion using the `bool()` function. Let's go over a few more examples and see how the 'is' is working compared to `==`:

```
# is vs ==
print(True is True)
print([1, 2] is [1, 2])
```

And the output is:

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following code:

```
1
2 print(True is True)
3
4 print([1,2] is [1,2])
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is selected and displays the following output:

```
True
False
> █
```

The expression (True is True) will be evaluated as True because Boolean True values will always have only one memory location and it could never be changed regardless of what we do in our code. The second print will give us False output, but why? Both Lists look the same. Those two Lists have the exact same values but every time a new List is created, the values are stored somewhere in the memory in a new location. Both Lists are in different locations in memory and although they have the same values, they are not the same. Remember Lists are not only data type, they are data structures as well. This means more memory is required to store a data structure like Lists compared to other data types like Strings, Numbers, and Boolean values.

Please consider the following example:

```
list1 = [1,2]
list2 = list1
print(list1 is list2)
```

The output will be True. The list1 is created and the values are stored somewhere in the memory in some location. In the next line, we are saying that list2 is equal to list1. This simply means that list2 will not create a new memory to store the same values because they are already stored by list1. Both list1 and list2 will point to the same values stored in the memory. Now if we change the list2 and add a new item, list1 will also be affected because they both points to the same values stored in the memory:

```
list1 = [1,2]
list2 = list1
print(list1 is list2)
list2.append(3)
print(list1)
print(list1)
```

```

main.py ×
1 list1 = [1,2]
2 list2 = list1
3 print(list1 is list2)
4 list2.append(3)
5 print(list1)
6 print(list2)

```

True  
[1, 2, 3]  
[1, 2, 3]

I hope you now understand the difference between the ‘is’ operator and the ‘==’ operator.

## Membership Operators

In Python, we use the membership operator to test if an item is present in the object. We have used this ‘in’ keyword in our examples but now you know they are called membership operators:

Operator	Description	Example	Output
in	This will return true only if the sequence with the specified value is found in the object	#in a = ["first", "second"] print("second" in a)	True, because the ‘second’ is present in the list
not in	This will return true only if the sequence with the specified value is not found in the object	#not in a = ["first", "second"] print("third" not in a)	True, because the item ‘three’ is not found or present in the list

## Python Loops

Here comes one of the most interesting features of any programming language in general. So far, we have covered control structures and operators that will direct the order of execution of the statements. Loops are very powerful because based on a specified condition we can make a program to execute a block of code over and over as long as the condition is True. This is very powerful because we can run these loops as many times as we need, and the machines are not going to be tired of repeating the same instructions because they are not humans. Loops are one of the most essential features you should know. One of the loops we are going to learn is called a ‘for’ loop and in order to create one, we need to use the ‘for’ keyword.

Here is the for-loop syntax:

```
for val in sequence:  
    loop body
```

In the syntax above, we have used the ‘for’ keyword which is a must and after the for-keyword, we have a variable name. The variable name can be any name you want but should be appropriate for the loop. Then we have the ‘in’ keyword and sequence of items. The variable ‘val’ which can be called ‘item’ will be created for each item in the sequence. Remember we mentioned in the previous chapter that Strings or Lists are iterable sequences. This means we can iterate over each item inside that sequence. Let’s do a very basic example where we iterate over a simple sequence of characters in the String:

The screenshot shows a Python code editor with a file named 'main.py'. The code contains a for loop that iterates over the characters in the string 'Your name is?'. The output window on the right shows the characters being printed one by one: 'Y', 'o', 'u', 'r', ' ', 'n', 'a', 'm', 'e', ' ', 'i', 's', '?'. A small yellow arrow icon is visible at the bottom right of the output window.

```
1 ▼ for char in 'Your name is?':
2     print(char)
```

As you can see, in the above example, I have used ‘char’ as the variable name. I decided to name it ‘char’ because it symbolizes each of the individual characters in the sequence ‘Your name is?’ And it will print each character in the iterable sequence in order. We can loop Lists as well using the for-loop:

```
# Iterate a List
list1 = [1, 3, 5, 7, 9]
for item in list1:
    print(item)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code contains a for loop that iterates over the elements of the list [1, 3, 5, 7, 9]. The output window on the right shows the numbers 1, 3, 5, 7, and 9 being printed sequentially. A small yellow arrow icon is visible at the bottom right of the output window.

```
1 list1 = [1,3,5,7,9]
2 ▼ for item in list1:
3     print(item)
```

Let’s check another example where we loop over the items in a Set:

```
# Iterate over a Set
set1 = {"apple", "banana", "cherry"}
```

```
for fruit in set1:  
    print(fruit)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code is:

```
1 # Iterate over a Set  
2 set1 = {"apple", "banana", "cherry"}  
3 ▶ for fruit in set1:  
4     print(fruit)
```

To the right of the code editor is a terminal window titled 'Console' which displays the output of the script:

```
apple  
cherry  
banana  
▶
```

We can loop over the Tuples as well:

```
tuple1 = ("apple", "banana", "cherry")  
for fruit in tuple1:  
    print(fruit)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code is:

```
1 tuple1 = ("apple", "banana", "cherry")  
2 ▶ for fruit in tuple1:  
3     print(fruit)
```

To the right of the code editor is a terminal window titled 'Console' which displays the output of the script:

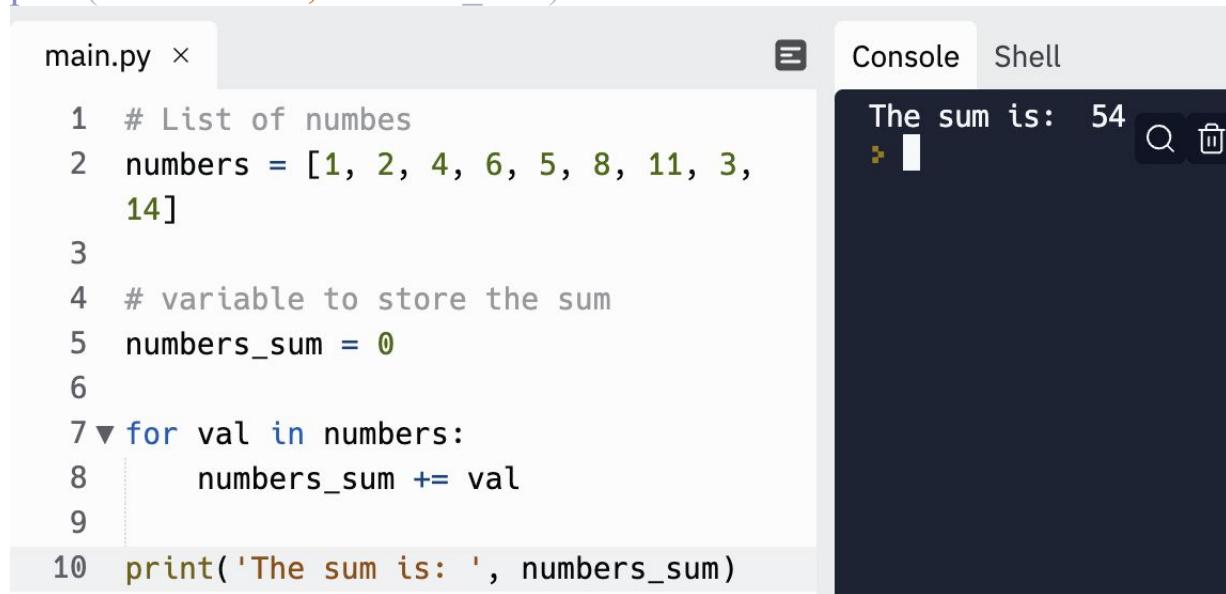
```
apple  
banana  
cherry  
▶
```

We can loop over the dictionaries as well but we will do this in some of the next sections. As you can see, the for-loops syntax is similar to the if-statements. With the for-loops, we can loop over a collection of data like lists, strings, tuples, sets, etc. Same as the if-statements, we need to use colon ':' after the collection and the code needs to be properly indented. The indentation is important because it will indicate to the Interpreter that the following code belongs to the for-block. It is important to know that when the last item from the collection is reached, the loop will terminate and the Interpreter will jump out of the loop and will start executing the code that comes after the for-loop. Let's use the for-loop to calculate the sum of all of the items from a list:

```
# List of numbers  
numbers = [1, 2, 4, 6, 5, 8, 11, 3, 14]
```

```
# variable to store the sum  
numbers_sum = 0
```

```
for val in numbers:  
    numbers_sum += val  
  
print('The sum is: ', numbers_sum)
```



The screenshot shows a code editor with a script named `main.py`. The code is as follows:

```
1 # List of numbers  
2 numbers = [1, 2, 4, 6, 5, 8, 11, 3,  
14]  
3  
4 # variable to store the sum  
5 numbers_sum = 0  
6  
7 ▼ for val in numbers:  
8     numbers_sum += val  
9  
10 print('The sum is: ', numbers_sum)
```

The output window shows the result of running the script:

```
The sum is: 54
```

## Nesting loops

Nested loops are very important because they allow us to nest a loop within another loop as long as there is a proper indentation. Here is one example of nesting two for-loops:

```
# nested for-loops

numbers1 = [1, 2, 3]
letters1 = ['x', 'l']

# iterate over the two lists

for val in numbers1:
    print(val)
    for letter in letters1:
        print(letter)
```

The screenshot shows a code editor window with a tab labeled "main.py". The code in the editor is identical to the one above. To the right of the editor is a "Console" window showing the execution of the script. The output in the console is:

```
1
x
l
2
x
l
3
x
l
> █
```

The output illustrates that in the first iteration of the outer loop, the value that is printed is 1, which then triggers the inner for-loop and this one will print the letters 'x' and 'l' consecutively. Once the last letters of the inner loop are completed the program will go back to the top of the outer loop and it will print 2. After printing the value 2, the inner loop runs again and will print the letters 'x', 'l,' and so on. We can also nest if-statements like this:

```
# nested if statements
grade = 80
```

```
if grade >= 60:  
    if grade >= 90:  
        print("A")  
  
    elif grade >= 80:  
        print("B")  
  
    elif grade >= 70:  
        print("C")  
  
    elif grade >= 60:  
        print("D")  
  
else:  
    print("Your grade is F")
```

The screenshot shows a code editor window titled "main.py" with the following Python code:

```
1 # nested if statements
2 grade = 80
3▼ if grade >= 60:
4▼   if grade >= 90:
5     print("A")
6
7▼   elif grade >= 80:
8     print("B")
9
10▼   elif grade >= 70:
11     print("C")
12
13▼   elif grade >= 60:
14     print("D")
15
16▼ else:
17   print("Your grade is F")
```

To the right of the code editor is a "Console" tab which is active, showing the output of the code:

```
B
```

## While Loop

In Python, we have two primitive loops, one is the for-loop we have already discussed and the second one is the while loop. With the while loop, we can execute a set of statements as long as the condition we set in the while loop is evaluated to True.

Syntax:

```
while test_expression:  
    Body of while
```

The syntax of the while loop is very basic and if you understood how the for-loop works, the while loop will be much easier to understand. So why do we need the while loop when we have the for-loop? Well, we can use this loop when we don't know the exact number of times we need to iterate or loop. The condition in the while loop is evaluated at the beginning before the loop can start. The body of the while loop will be executed only if the condition is True. It is important to know that after each iteration, the test expression (condition) is checked again. This loop will stop when the expression evaluates to False. The body of the while-loop in Python is determined by indentation and in other programming languages, this can be done using curly brackets like in JavaScript. Let's go over this example:

```
# while loop

# calc sum up to this number
the_number = 10
# initialize sum and counter
the_sum = 0
i = 1

while i <= the_number:
    the_sum = the_sum + i
    i = i + 1 # update/increment the counter

# print the sum
print('The sum is: ', the_sum)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code implements a simple while loop to calculate the sum of numbers from 1 to 10. The 'Console' tab shows the output: 'The sum is: 55'. The code is as follows:

```
1 # while loop
2
3 # calc sum up to this number
4 the_number = 10
5 # initialize sum and counter
6 the_sum = 0
7 i = 1
8
9▼while i <= the_number:
10    the_sum = the_sum + i
11    i = i + 1 # update/increment the
12    counter
13 # print the sum
14 print('The sum is: ', the_sum)
```

From the figure above, we can see that test expression `i <= the_number` will be True as long as the counter variable ‘`i`’ is less than or equal to `the_number` (10 in our case). We need to find a way to increase the counter after each iteration because if we don’t do this, the test expression will always be true and we will create an infinite loop that will block our machine (please be careful). Therefore, after the sum calculation, in the body of the while loop, we increase the value of the counter variable ‘`i`’ by one. When the value of ‘`i`’ becomes 11, then the test expression will be evaluated to False and that will terminate the while loop. Another interesting feature that I want to share is the ‘`break`’ statement. This is our way of stopping the while loop and forcing the loop to terminate or stop its execution. We can break from the while loop before the while condition is evaluated to False like this:

The screenshot shows a Python code editor with a file named 'main.py'. The code contains a while loop that prints numbers from 0 to 7, then exits the loop when i reaches 7 due to the break statement. The output window shows the printed numbers: 0, 1, 2, 3, 4, 5, 6, 7.

```
1 # break statement
2 i = 0
3 ▼ while i <= 10:
4     print(i)
5 ▼     if (i == 7):
6         break
7     i += 1
8
```

0
1
2
3
4
5
6
7

As you can see, when the counter ‘i’ reaches the number 7, the if condition inside the while loop will be evaluated to True and the break statement will terminate the loop. That is why the rest of the numbers (8,9,10) will never be printed. I have also used the augmented operator ( $+=$ ) syntax or shorter syntax to increase the counter by one. The while loop can be combined with an else clause if we want the else clause to be executed at the end when the condition in the while loop becomes False:

```
# else clause statement
j = 0
while j <= 10:
    print(j)
    j += 1
else:
    print('This means the while condition is false')

print('This line is outside the while loop')
```

The screenshot shows a Python development environment with two panes. The left pane is titled 'main.py' and contains the following Python code:

```
1 # else clause statement
2 j = 0
3 while j <= 10:
4     print(j)
5     j += 1
6 else:
7     print('This means the while
8 condition is false')
9 print('This line is outside the while
10 loop')
```

The right pane is titled 'Console' and shows the output of running the script. It displays the numbers from 0 to 10, followed by the message 'This means the while condition is false', and finally 'This line is outside the while loop'.

As you can see, when the while condition becomes False, the else-clause will be executed. Inside the else clause, we have a very simple print simple message. But if we try the previous example where we used the ‘break’ statement then the else-clause will never be executed because we forced the loop to exit at a certain point:

```
# break statement
i = 0
while i <= 10:
    print(i)
    if(i == 7):
        break
    i += 1
else:
    print('This means the while condition is false')

print('This line is outside the while loop')
```

The screenshot shows a Python code editor with a script named `main.py`. The code contains a `while` loop that prints integers from 0 to 7, then breaks, and then prints two additional messages outside the loop. The output window shows the numbers 0 through 7, followed by a message indicating the code is outside the loop.

```
main.py ×
1 # break statement
2 i = 0
3▼ while i <= 10:
4     print(i)
5▼     if(i == 7):
6         break
7     i += 1
8▼ else:
9     print('This means the while
condition is false')
10
11 print('This line is outside the while
loop')

Console Shell
0
1
2
3
4
5
6
7
This line is outside the while loop
> █
```

## Iterables

Iterables in Python are very important because it is an Object or a collection of items that can be iterated over. In Python, Iterables are Tuples, Sets, Dictionaries, Lists, and Strings. Another key term is ‘iterated’ or ‘looped over’ and this simply means that the collection can be visited for each of its items one by one. This feature can be found in other programming languages like JavaScript. To summarize, the iterable is the collection and ‘iterate’ is the action we perform on that collection. We have covered how we can iterate most of the collections in Python but we didn’t include an example of the Dictionary. Let us create one example where we iterate over a dictionary collection:

```
# looping over a dictionary
# car dictionary
ford = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 2023,
    "transmission": "Automatic"
}
```

```
for item in ford:  
    print(item)
```

The screenshot shows a Python development environment with two panes. On the left, the code editor pane displays a file named 'main.py' with the following content:

```
1 # looping over a dictionary  
2 # car dictionary  
3 ▼ ford = {  
4     "brand": "Ford",  
5     "model": "Mustang",  
6     "year": 2023,  
7     "transmission": "Automatic"  
8 }  
9  
10 ▼ for item in ford:  
11     print(item)
```

On the right, the 'Console' tab is active, showing the output of the executed code. The output lists the keys of the dictionary: 'brand', 'model', 'year', and 'transmission'. Each key is followed by a small yellow triangle icon indicating it is a list item.

When looping over a dictionary, the values that are returned will be the keys from the dictionary data type. But we can use other methods that will return both keys and values. These methods are popular and used by the developers:

### items() method

This method will get the key/value pairs and return them as a Tuple:

```
# items method  
for item in ford.items():  
    print(item)
```

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # looping over a dictionary
2 # car dictionary
3 ▼ ford = {
4     "brand": "Ford",
5     "model": "Mustang",
6     "year": 2023,
7     "transmission": "Automatic"
8 }
9
10 #items method
11 ▼ for item in ford.items():
12     print(item)
```

To the right of the code editor is a terminal window with tabs labeled "Console" and "Shell". The "Console" tab is active, showing the output of the code execution:

```
('brand', 'Ford')
('model', 'Mustang')
('year', 2023)
('transmission', 'Automatic')
```

## values() method

This method will give us only the values from the dictionary, not the keys:

```
# values method
for item in ford.values():
    print(item)
```

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 # looping over a dictionary
2 # car dictionary
3 ▼ ford = {
4     "brand": "Ford",
5     "model": "Mustang",
6     "year": 2023,
7     "transmission": "Automatic"
8 }
9
10 # values method
11 ▼ for item in ford.values():
12     print(item)
```

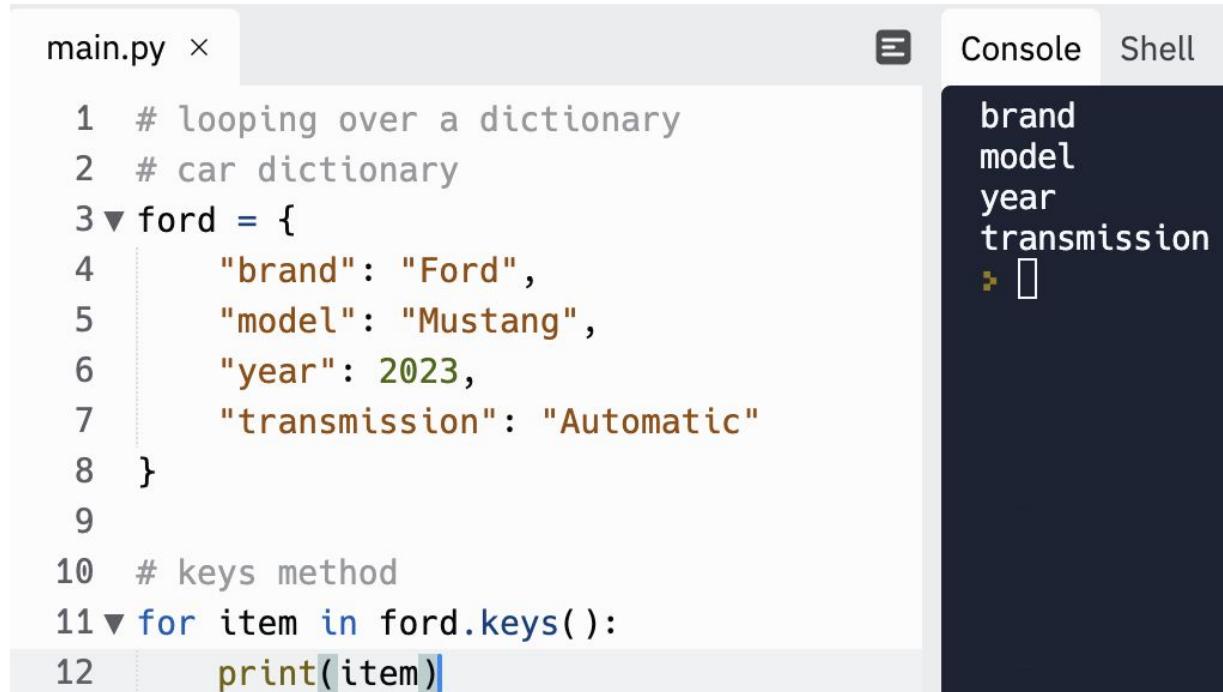
To the right of the code editor is a terminal window with tabs labeled "Console" and "Shell". The "Console" tab is active, showing the output of the code execution:

```
Ford
Mustang
2023
Automatic
```

## keys() method

The normal for-loop does the same job as this method, meaning it will only return the keys of a dictionary but I guess it's more descriptive like this:

```
# keys method
for item in ford.keys():
    print(item)
```



A screenshot of a Python development environment. On the left, the code editor shows a file named 'main.py' with the following content:

```
1 # looping over a dictionary
2 # car dictionary
3 ▼ ford = {
4     "brand": "Ford",
5     "model": "Mustang",
6     "year": 2023,
7     "transmission": "Automatic"
8 }
9
10 # keys method
11 ▼ for item in ford.keys():
12     print(item)
```

On the right, there are two panes: 'Console' and 'Shell'. The 'Console' pane displays the output of the code execution, showing the keys of the 'ford' dictionary:

```
brand
model
year
transmission
> []
```

But what if we want to get the key/value pairs without getting a Tuple back? There is a simple trick we can do using the power of unpacking but we need to use the items() method because it will give us access to both keys and values:

```
# unpack the values and keys
for item in ford.items():
    key,value = item
    print(item)
```

```
main.py x
1 # looping over a dictionary
2 # car dictionary
3 ▼ ford = {
4     "brand": "Ford",
5     "model": "Mustang",
6     "year": 2023,
7     "transmission": "Automatic"
8 }
9
10 # unpack the values and keys
11 ▼ for item in ford.items():
12     key,value = item
13     print(key, value)
```

Console Shell

```
brand Ford
model Mustang
year 2023
transmission Automatic
> █
```

Another way to get the keys and values from the dictionary is to change the syntax of the for loop a bit:

```
# we can get keys & values like this
for key,value in ford.items():
    print(key,value)
```

```
main.py x
1 #car dictionary
2 ▼ ford = {
3     "brand": "Ford",
4     "model": "Mustang",
5     "year": 2022,
6     "transmition": "Automatic"
7 }
8
9 ▼ for key,value in ford.items():
10     print(key,value)
```

Console Shell

```
brand Ford
model Mustang
year 2022
transmition Automatic
> █
```

The last syntax is very common among developers. Before we wrap up how to loop the dictionaries, I would like to mention that naming the variables (key, value) is up to you. I have used something that is descriptive but even this code is perfectly okay and executable:

```
for k,v in ford.items():
    print(k,v)
```

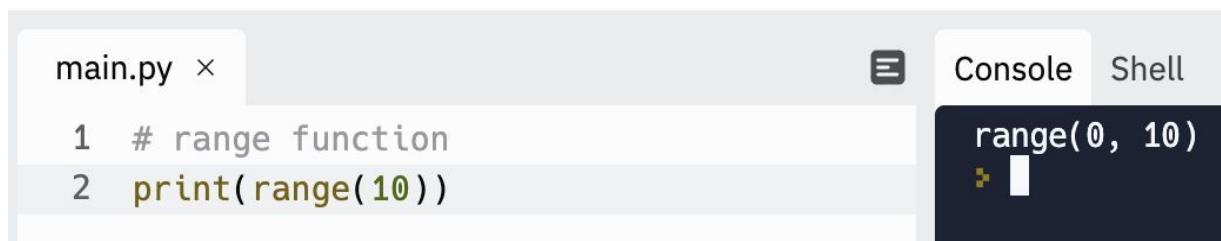
## The built-in Range() function

The range() function is one of the built-in functions in Python that allows us to iterate over a sequence of numbers. The range() function will return a sequence of numbers that will start from zero (0) by default and will increment by 1 until it reaches the end (a specified number we pass into the range function). The range function has the following syntax:

```
range(start, stop, step)
```

As you can see, the range can take 3 parameters, the first one is called start and is optional. This would be an integer number to specify the start position. If omitted, the default will be zero (0). The second parameter is the ‘stop,’ and this one is required and will be an integer number that will specify the end. The final parameter is called ‘step’ and is the same as the ‘start’ parameter, this one is optional as well and it will specify the step size. The default step size is one. We know that from the parameters the one in the middle called stop is required so let’s print what this function does when we supply only one parameter:

```
# range function
print(range(10))
```



A screenshot of a Python development environment. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 # range function
2 print(range(10))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and shows the output of the code execution:

```
range(0, 10)
```

From the figure above we now have a range that starts from zero and ends at ten and although we haven’t specified the start, it was added for us. The range() is very popular when it’s used in conjunction with loops:

```
# range() with for-loop
for number in range(0, 10):
```

```
print(number)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code contains a for-loop that iterates over a range from 0 to 9, printing each number. The 'Console' tab on the right shows the output: the numbers 0 through 9, each on a new line.

```
main.py ×
```

```
1 # range()
2 ▼ for number in range(0,10):
3     print(number)
```

```
Console
```

```
0
1
2
3
4
5
6
7
8
9
```

From the figure above, we can see that we have used the range function so the for-loop can run 10 times. Instead of specifying the variable name like ‘number,’ we can use ‘\_’ underscore and it will still be valid:

The screenshot shows a Python code editor with a file named 'main.py'. The code contains a for-loop that iterates over a range from 0 to 9, printing each number. The variable name is replaced by an underscore. The 'Console' tab on the right shows the output: the numbers 0 through 9, each on a new line.

```
main.py ×
```

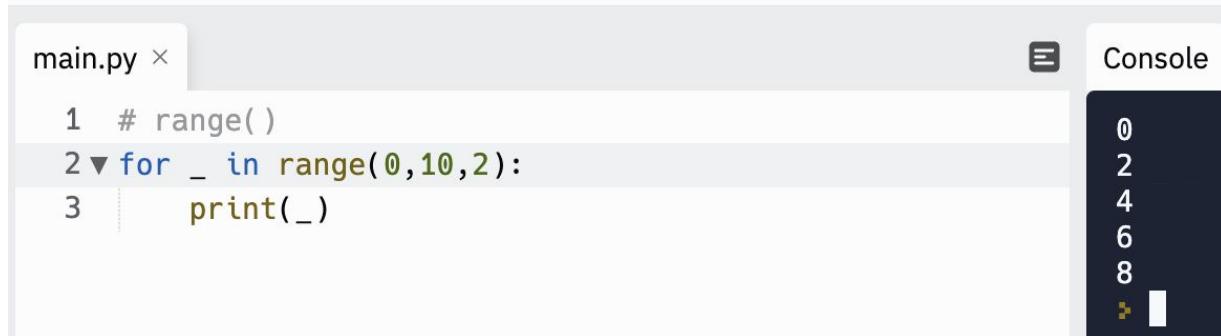
```
1 # range()
2 ▼ for _ in range(0,10):
3     print(_)
```

```
Console
```

```
0
1
2
3
4
5
6
7
8
9
```

This means we don’t care about naming a variable but we need the loop to run a specific number of times specified in the range function. We can even specify the stepover parameter:

```
# stepover
for _ in range(0, 10, 2):
    print(_)
```



```
main.py ×
1 # range()
2 ▼ for _ in range(0,10,2):
3     print(_)

Console
0
2
4
6
8
> █
```

## Enumerate ()

Similar to the range () function, we have a function called enumerate(). The enumerate () function will take a collection like a Tuple and will return it as enumerate object. This function will also add a counter/index as the key to each item in the set. This means that the function allows us to iterate through a sequence but it will keep track of both element and index. For example, let's pass a Tuple collection into this new function:

```
# enumerate ()
x = ('first', 'second', 'third')
y = enumerate(x)

for item in y:
    print(item)
```



```
main.py ×
1 # enumarate()
2 x = ('first', 'second', 'third')
3 y = enumerate(x)
4
5 ▼ for item in y:
6     print(item)

Console
(0, 'first')
(1, 'second')
(2, 'third')
> █
```

As we can see, the enumerate() function took an iterable as a parameter and it returned an object with an index which is the key, and now the Tuples have access to indexes just like the Lists. Instead of printing just the variable ‘item,’ we can do the following:

```
for key, item in y:  
    print(key, item)
```

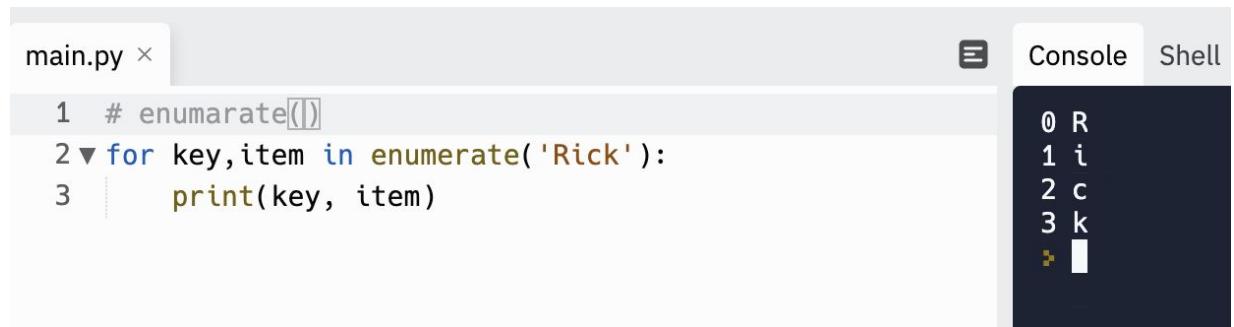


```
main.py ×  
1 # enumerate()  
2 x = ('first', 'second', 'third')  
3 y = enumerate(x)  
4  
5 ▼ for key, item in y:  
6     print(key, item)
```

Console	Shell
0 first 1 second 2 third ▶ □	

Excellent! Now we got back not just the value of the collection but the index as well. We can achieve the same thing with Strings and Lists:

```
# Strings  
for key, item in enumerate('Rick'):  
    print(key, item)  
  
# Lists  
for key, item in enumerate([1, 2, 3]):  
    print(key, item)
```



```
main.py ×  
1 # enumerate()  
2 ▼ for key, item in enumerate('Rick'):  
3     print(key, item)
```

Console	Shell
0 R 1 i 2 c 3 k ▶ □	

The screenshot shows a Python code editor with a file named 'main.py'. The code uses the built-in function 'enumerate()' to iterate over a list of numbers [1, 2, 3]. For each item, it prints the index (key) and the value (item). The output window to the right shows the results: index 0 corresponds to value 1, index 1 corresponds to value 2, and index 2 corresponds to value 3.

```
main.py x
1 # enumerate()
2 ▼ for key,item in enumerate([1,2,3]):
3     print(key, item)
Console
0 1
1 2
2 3
> □
```

As we can see, the `enumerate()` function is very useful because we can iterate over the collection plus we have access to the index and the value.

## Break, continue and pass statements

We saw that the `break` statement will stop a normal execution of a loop and will terminate that loop. In that example, we have used the `break` statement inside the `while` loop but we can use the `break` statement in any other loops if we want. The `break` statement can be used to stop the loop before it has finished looping all of the items:

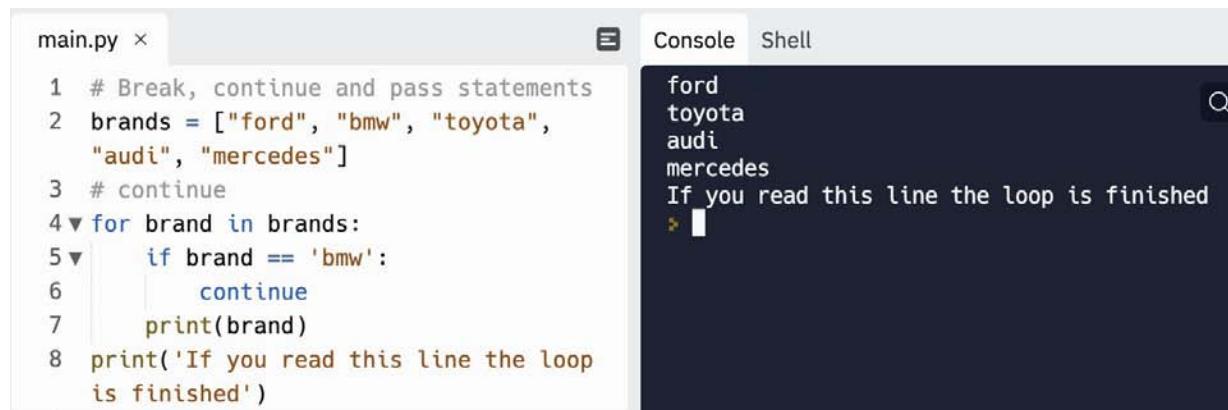
```
# Break, continue and pass statements
brands = ["ford", "bmw", "toyota", "audi", "mercedes"]
for brand in brands:
    print(brand)
    if brand == 'toyota':
        break
print('If you read this line the loop is finished')
```

The screenshot shows a Python code editor with a file named 'main.py'. The code uses a `for` loop to iterate over a list of car brands. It prints each brand name. When it reaches the brand 'toyota', it executes the `if` condition and immediately exits the loop due to the `break` statement. After the loop, it prints a message indicating that the loop has finished.

```
main.py x
1 # Break, continue and pass statements
2 brands = ["ford", "bmw", "toyota",
3           "audi", "mercedes"]
4 ▼ for brand in brands:
5     print(brand)
6     if brand == 'toyota':
7         break
8 print('If you read this line the loop
      is finished')
Console Shell
ford
bmw
toyota
If you read this line the loop is finished
> □
```

Another statement we can use in our loops is the ‘continue’ statement. When we use this statement, we will not exit the loop the way we did with the ‘break’ statement but we will stop the loop once for the current iteration not the entire loop like the ‘break statements.’ Let’s use the continue statement to skip the iteration when the car brand is equal to ‘bmw’ but we need to continue to iterate over the rest of the brands:

```
# continue
for brand in brands:
    if brand == 'bmw':
        continue
    print(brand)
print('If you read this line the loop is finished')
```



The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the provided Python code. On the right, there is a "Console" tab showing the output of the code execution. The output in the console is:

```
ford
toyota
audi
mercedes
If you read this line the loop is finished
```

The third statement is called ‘pass statement’ and it is rarely used because it doesn’t do anything in particular. The pass statement will be used as a placeholder for future code. For example, when the pass statement is executed, it will not do anything but you can avoid getting an error because an empty code is not allowed. I know this is confusing but let’s check the following example:

```
#pass
is_old = True
if is_old:
    #no code
print('This will give us an error')
```

The screenshot shows a Python code editor with a file named 'main.py'. The code contains an if-statement where the body is a single 'pass' statement. The last line is a print statement. The console output shows an IndentationError: expected an indented block.

```
main.py ×
1 #pass
2 is_old = True
3▼ if is_old:
4     #no code
5 print('This will give us an error')

File "main.py", line 5
    print('This will give us an error')
^
IndentationError: expected an indented block
```

As you can see, we got IndentationError because there is no code after the if-statement and the Interpreter expects us to finish the if-statement and write some code. The Interpreter will look into the next line (5), see that there is no indentation, and will throw an error because the Interpreter is confused and thinks the print message should go in the body of the if-statement. Therefore, when we don't know what to write in the if-block, we can use the pass statement:

```
#pass
is_old = True
if is_old:
    #no code
    pass
print('This will give us an error')
```

The screenshot shows the same code as before, but now it runs successfully. The console output shows the string 'This will give us an error'.

```
main.py ×
1 #pass
2 is_old = True
3▼ if is_old:
4     #no code
5     pass
6 print('This will give us an error')

This will give us an error
```

As we can see, we can run the code without getting errors.

## Practice Time

Now is the right time to practice some of the things we have covered in this chapter and this includes the loops, conditionals, and methods. The exercise will be to clean a dirty. A dirty list is a list with duplicate values and our program should create a clean list without any duplicates. Before you see my solution, I would like you to think, and even attempt to write your own solution. Here is the dirty list:

```
#chapter-2 exercise  
dirty_list = [1, 2, 3, 2, 4, 1, 5, 6, 5, 5, 7, 8, 1, 9, 10, 9, 8, 3];
```

### Solution:

As we can see from the list above, a few items are repeated so we need to create a new list and call it **clean\_list**. Hint, there are many different ways to solve this problem but I suggest using one of the membership operators to make our life easier.

Here is my solution (code only)

```
# chapter-2 exercise  
dirty_list = [1, 2, 3, 2, 4, 1, 5, 6, 5, 5, 7, 8, 1, 9, 10, 9, 8, 3]  
# empty list  
clean_list = []  
# loop over the items  
for item in dirty_list:  
    # check if the item doesn't exist in the list  
    if item not in clean_list:  
        # if item is not found then add it  
        clean_list.append(item)  
  
print(clean_list)
```

If we run the code, this will be the output:

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 #chapter-2 exercise
2 dirty_list = [1, 2, 3, 2, 4, 1, 5, 6, 5, 5, 7, 8,
1, 9, 10, 9, 8, 3];
3 #empty list
4 clean_list = []
5 #loop over the items
6 for item in dirty_list:
7     #check if the item doesn't exist in the list
8     if item not in clean_list:
9         #if item is not found then add it
10        clean_list.append(item)
11
12 print(clean_list)
```

On the right, there is a "Console" tab showing the output of the code execution:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

## Python Functions

We have already seen a few of the built-in functions in Python but we can create our own functions as well. Functions are a core feature of Python. How we can create a function? Well, we can start using the keyword 'def' which means define a function. After that, we need to name the function:

```
# functions
def hi_function():
    print('Hi there')
```

After we name the function, we need to use these brackets () and colon : that will tell the Interpreter that the function body will be in the following line. But if we run the code above, we will not see the print message, why? The functions need to be called in order to be executed so we can call the function like this:

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 def hi_function():
2     print('Hi there')
3
4 hi_function()
```

On the right, there is a "Console" tab showing the output of the code execution:

```
Hi there
```

We make a function call outside the body using the function name together with the brackets (). The functions are really useful because once we create them, we can call them as many times as we want. This is very handy because each function we write has a purpose and each function must return something. Once the function is created, we don't have to repeat and write the same code over and over again. This is called the DRY principle and it means 'Do Not Repeat Yourself', so no unnecessary code or duplicates. Another important thing you should know is that we cannot call a function before it has been created or defined. Let's try to call a function before it has been declared:

```
main.py x
1 hi_function()
2 def hi_function():
3     print('Hi there')
4
```

Console Shell

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    hi_function()
NameError: name 'hi_function' is not defined
```

This will throw a `NameError` because the function is not defined. The Python Interpreter will throw the above error because it tries to call a function that does not exist in the memory. Anytime we write a function name without the brackets, we use that name to point to a place where that function is stored in the memory. We can prove this if we print the function without the brackets:

```
main.py x
1 def hi_function():
2     print('Hi there')
3
4 print(hi_function)
5
```

Console Shell

```
<function hi_function at 0x7fd9e7b0280>
```

As we can see, the function called `hi_function` is stored somewhere in the memory at this location: `0x7fd9e7b0280`. But when we write the `hi_function()` with brackets, then the interpreter knows that we are trying to execute the function body.

## Function Arguments and Parameters

The functions we have created so far were very easy because they only had one message to print. But functions exist to do much more complex things. The `hi_function()` is very simple because it does not take any parameters. We can add parameters inside the function brackets. The name of these parameters can be anything and we can pass one or more parameters but they need to be separated by a comma. Let us add one parameter inside the function:

```
def hi_function1(name):  
    print('Hi there')
```

This means that the `hi_function` can now receive one parameter called `name`. We can add multiple parameters if we separate them with a comma:

```
def hi_function2(name, lastname):  
    print('Hi there')
```

These two parameters can now be used in the function body as local variables:

```
def hi_function3(name,lastname):  
    print(f'Hi there {name} {lastname}')
```

Now we are using the formatted ‘f’ string to print the values of these local variables. But where will the values of these two parameters/variables come from? We can pass them into the function during the function call. Here is a function call without any parameters:

```
hi_function()
```

Let’s pass the two arguments:

```
hi_function3('Rick','Sekuloski')
```

Why have I used two different terms (parameters and arguments) for the same thing? In the function definition, they are called parameters, and arguments are called when we call/invoke the function. They are the same thing and developers always mix them. But this is not a big deal. Let’s run the function and see the output:

The screenshot shows a Python development environment. On the left, the code editor displays a file named 'main.py' with the following content:

```
1 def hi_function3(name, lastname):
2     print(f'Hi there {name} {lastname}')
3
4 hi_function3('Rick', 'Sekuloski')
```

On the right, the terminal window shows the output of running the script:

```
Hi there Rick Sekuloski
```

So the arguments are the actual values we send to the function and the parameters are the variables we use to access those values in the function body. Parameters are used during the function declaration and arguments during the function call. You will read somewhere that the function call is called function invocation. So these are the terms I want you to learn from this section:

- Function declaration
- Function call
- Function invocation or function call
- Function arguments
- Function parameters
- Defining a function

I hope you now realize that the functions can do some calculations based on the arguments we supply. We can call the same function with different arguments and the output will always be different:

The screenshot shows a Python development environment. On the left, the code editor displays a file named 'main.py' with the following content:

```
1 def hi_function3(name, lastname):
2     print(f'Hi there {name} {lastname}')
3
4 hi_function3('Rick', 'Sekuloski')
5 hi_function3('Joseph', 'Smith')
6 hi_function3('Abraham', 'Lincoln')
7 hi_function3('Morgan', 'Freeman')
```

On the right, the terminal window shows the output of running the script:

```
Hi there Rick Sekuloski
Hi there Joseph Smith
Hi there Abraham Lincoln
Hi there Morgan Freeman
```

## Positional and Keyword Arguments

In this section, we will discuss the importance of positional and keyword arguments in Python. The positional arguments as the name suggest are arguments that need to be written in the right order or position. This means that the argument/arguments must be provided in the correct position and order if we want to have the desired function output. Let's look at the previous function example again and modify it slightly so it can take 3 arguments and 3 parameters:

```
def hi_function4(name, lastname, age):
    print(f'Hi there {name} {lastname}. Are you {age} old?')

hi_function4('Rick', 'Sekuloski', 33)
```

The screenshot shows a Python code editor with a file named 'main.py'. The code defines a function 'hi\_function4' that takes three parameters: 'name', 'lastname', and 'age'. It prints a formatted string with these variables. Below the code, a call to 'hi\_function4' with the arguments 'Rick', 'Sekuloski', and '33' is shown. To the right, a terminal window titled 'Console' shows the output: 'Hi there Rick Sekuloski. Are you 33 old?'.

```
main.py ×
1 def hi_function4(name, lastname, age):
2     print(f'Hi there {name} {lastname}. Are
3         you {age} old?')
4 hi_function4('Rick', 'Sekuloski', 33)

Console Shell
Hi there Rick Sekuloski. Are
you 33 old?
> 
```

As you can see from the figure above, we have a function that takes 3 values or arguments and the last one is the person's age. So in this case the order of the arguments matters, for example, let us switch the lastname and age:

The screenshot shows the same 'main.py' file. The code is identical to the previous one, but the arguments in the call to 'hi\_function4' are swapped: 'Rick', '33', and 'Sekuloski'. The terminal window shows the output: 'Hi there Rick 33. Are you Sek
uloski old?'.

```
main.py ×
1 def hi_function4(name, lastname, age):
2     print(f'Hi there {name} {lastname}. Are
3         you {age} old?')
4 hi_function4('Rick', 33, 'Sekuloski')

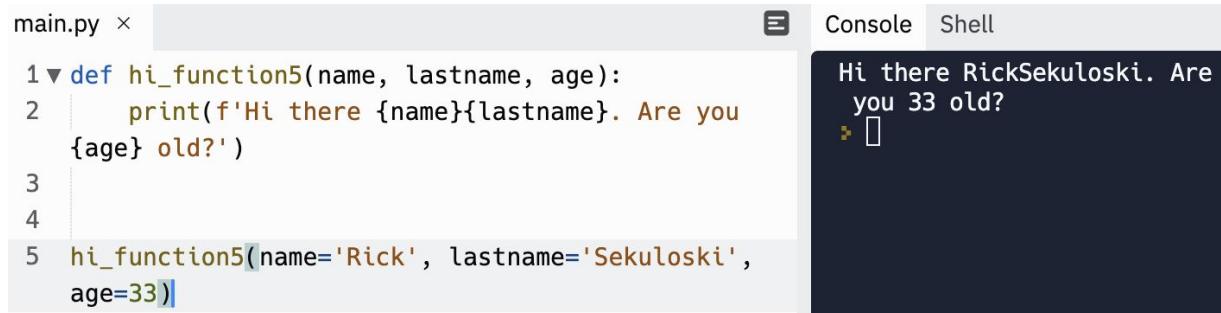
Console Shell
Hi there Rick 33. Are you Sek
uloski old?
> 
```

So we have invoked the function with the name first, then age, and finally, the lastname. The function will still work but the output will not make any sense. Therefore, positional arguments are very important because their order or position matters. What are the keyword arguments in Python? Keyword arguments don't care about the position because the arguments are labeled with a keyword. Let's create an example where we use keyword arguments:

```
def hi_function5(name, lastname, age):
    print(f'Hi there {name} {lastname}. Are you {age} old?')
```

```
hi_function5(name='Rick', lastname='Sekuloski', age=33)
```

I have to give the functions different names because I save them in a file. Inside the file, you cannot have functions with the same name because they will throw an error. So let's run the function:



```
main.py ×
```

```
1▼ def hi_function5(name, lastname, age):
2     print(f'Hi there {name}{lastname}. Are you
3         {age} old?')
4
5 hi_function5(name='Rick', lastname='Sekuloski',
    age=33)|
```

```
Console Shell
```

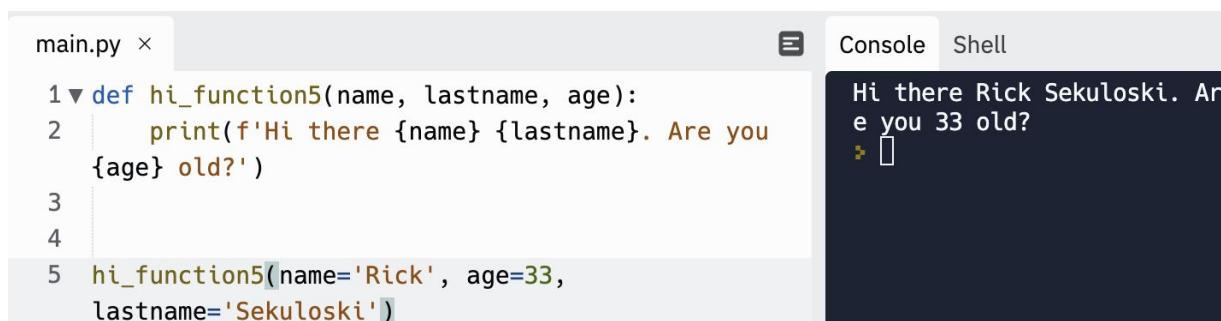
```
Hi there RickSekuloski. Are
you 33 old?
> |
```

Let us focus on the function invocation:

```
hi_function5(name='Rick', lastname='Sekuloski', age=33)
```

In the function call, we have a keyword that you can consider as a variable name and we assigned a value to that keyword. If we look at the output, we can see that the order is correct, but let's change the order of the keywords in the function call and run it again:

```
hi_function5(name='Rick', age=33, lastname='Sekuloski')
```



```
main.py ×
```

```
1▼ def hi_function5(name, lastname, age):
2     print(f'Hi there {name}{lastname}. Are you
3         {age} old?')
4
5 hi_function5(name='Rick',
    lastname='Sekuloski', age=33)|
```

```
Console Shell
```

```
Hi there Rick Sekuloski. Ar
e you 33 old?
> |
```

The output is correct again because of the keyword arguments. Keyword arguments are very good because they will match the parameters in the function definition. Personally, this is not the best practice, our function call looks messy. The function call was so much cleaner without these keywords. This is everything you should know about positional and keyword arguments in Python.

## Default Arguments and Parameters

In this section, we will talk about what default parameters and arguments are. Default parameters will allow us to give default values right in the function declaration. Please consider this example:

```
# name, lastname are parameters

def hi_function6(name='John', lastname='Doe', age=30):
    print(f'Hi there {name} {lastname}. Are you {age} old?')
```

If we look closely, we will notice that this looks like the keyword arguments we wrote in the function call, right? Yes, that is correct and this is another reason why I don't recommend using the keyword arguments because most of the time developers mix them with default parameters. The default parameters are useful because now the parameters have default or starting value. This is great because if we miss adding a value in the function call, there will be still an output. Let's call the function with 3 arguments first:

The screenshot shows a code editor with a Python file named `main.py`. The code defines a function `hi_function6` with three parameters: `name`, `lastname`, and `age`, each with a default value. It then prints a message using an f-string. Below the code editor is a terminal window titled "Console". The terminal shows the output of running the script with arguments `'Novak'`, `'Djokovic'`, and `40`.

```
main.py ×
1 # name, lastname are parameters
2
3
4 def hi_function6(name='John', lastname='Doe', age=30):
5     print(f'Hi there {name} {lastname}. Are you {age} old?')
6
7
8 hi_function6('Novak', 'Djokovic', 40)
9 |
```

```
Console Shell
Hi there Novak Djokovic. Are you 40 old?
> |
```

Did you expect the output from above? If the answer is yes, then congratulations because this means you understood how default parameters work in the background. When we call the function with all of the arguments, the default parameter values will be overwritten by the arguments we pass into the function call. But let's now make a function call with two values only:



The screenshot shows a Python code editor with a file named 'main.py'. The code defines a function 'hi\_function6' with three parameters: 'name', 'lastname', and 'age', each with a default value. It then prints a message using an f-string. Below the code, a call to 'hi\_function6' with arguments 'Novak' and 'Djokovic' is shown. To the right, a terminal window titled 'Console' shows the output: 'Hi there Novak Djokovic. Are you 30 old?'.

```
main.py ×
1 # name, lastname are parameters
2
3
4 def hi_function6(name='John', lastname='Doe', age=30):
5     print(f'Hi there {name} {lastname}. Are you {age} old?')
6
7
8 hi_function6('Novak', 'Djokovic')
```

```
Console Shell
Hi there Novak Djokovic. Are you 30 old?
> 
```

As we can see from the figure above, the function call doesn't have all of the arguments but in the output, the argument that was missing was replaced by the default value. That is why the default arguments are so important because now we called the function without any argument and it still produced a result at the end:



The screenshot shows a Python code editor with a file named 'main.py'. The code defines a function 'hi\_function6' with three parameters: 'name', 'lastname', and 'age', each with a default value. It then prints a message using an f-string. Below the code, a call to 'hi\_function6' without arguments is shown. To the right, a terminal window titled 'Console' shows the output: 'Hi there John Doe. Are you 30 old?'.

```
main.py ×
1 # name, lastname are parameters
2
3
4 def hi_function6(name='John', lastname='Doe', age=30):
5     print(f'Hi there {name} {lastname}. Are you {age} old?')
6
7
8 hi_function6()
```

```
Console Shell
Hi there John Doe. Are you 30 old?
> 
```

## Function Return

The ‘return’ is a special keyword in Python and is very useful and common in functions. The idea is that the functions always return a value and we can use that value to do whatever we want. Let’s create a function to calculate the sum of two integer numbers:

```
# sum function with return keyword
```

```
def calc_fun(num1, num2):  
    num1 + num2
```

```
calc_fun(3, 4)
```

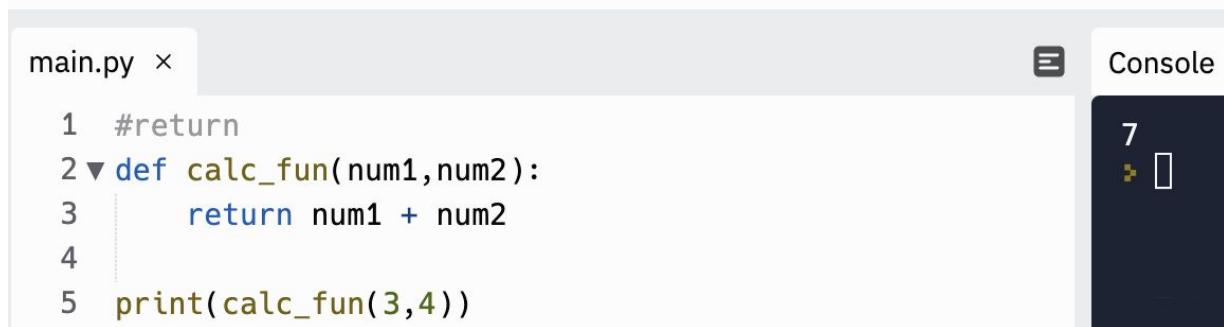
The function above will calculate the sum of the two arguments we pass in the function call but when we run it, the output is not there. I know you are thinking we didn't use the print function so we can print the result. We don't always have to use the print function to get the value, instead of printing messages, the function goal is to return a value. We can return the result like this:

```
# sum function with return keyword
```

```
def calc_fun(num1, num2):  
    return num1 + num2
```

```
calc_fun(3, 4)
```

Good! Now we can wrap the function call into a print function so we can get the result:



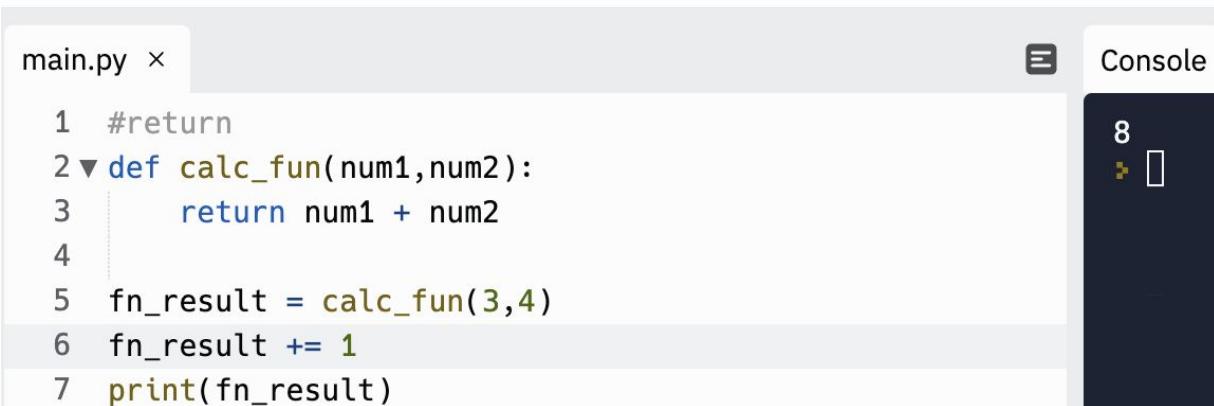
The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following code:

```
1 #return  
2 ▼ def calc_fun(num1,num2):  
3     return num1 + num2  
4  
5 print(calc_fun(3,4))
```

On the right, a "Console" window displays the output of the code execution:

```
7
```

We can also store the return value into a variable and use that variable later in our code:



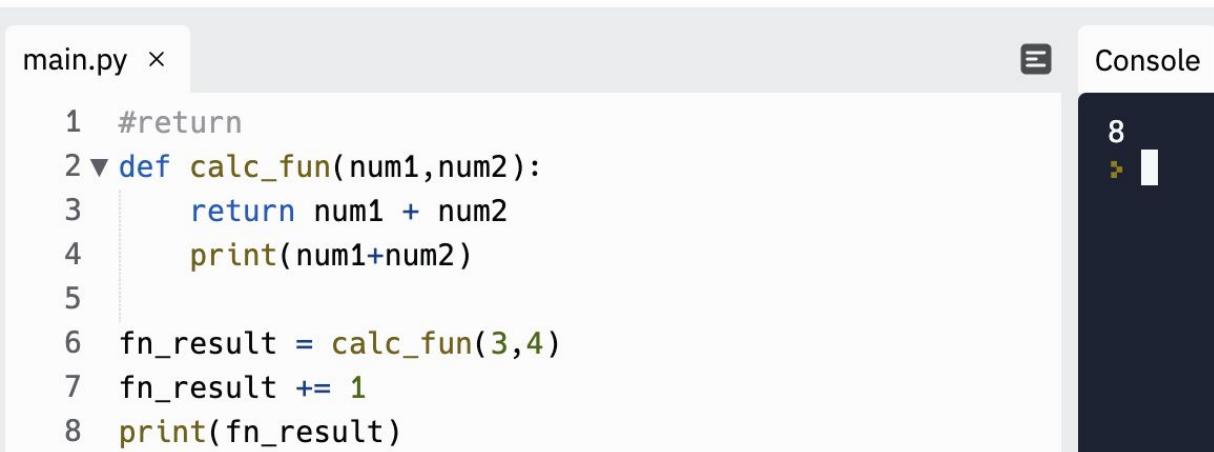
The screenshot shows a Python IDE interface. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 #return
2 ▼ def calc_fun(num1,num2):
3     return num1 + num2
4
5 fn_result = calc_fun(3,4)
6 fn_result += 1
7 print(fn_result)
```

On the right, a "Console" window displays the output of the code execution:

```
8
```

Now the value is 8 because we can manipulate the value of the variable. What you need to know is that when the Interpreter sees the return keyword in the body of the function, the function will return that value and it will be terminated. You need to be careful about placing code after the keyword return in the function body because that code will never be executed, like the following example:



The screenshot shows a Python IDE interface. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 #return
2 ▼ def calc_fun(num1,num2):
3     return num1 + num2
4     print(num1+num2)
5
6 fn_result = calc_fun(3,4)
7 fn_result += 1
8 print(fn_result)
```

On the right, a "Console" window displays the output of the code execution:

```
8
```

Line 4 will never be executed because of the return keyword on line 3.

## Nested functions

The nested functions are very interesting because we can have a function that is defined within another function. Instead of having one function, we will have an inner/nested function that can access the variables defined in the enclosing or outer function. I know this might be confusing but let's create one simple example so you can understand how nesting works.

```
# nested function
def outer_fn(num1, num2):
    def inner_fn(n1, n2):
        return n1 + n2
    total = inner_fn(num1, num2)
    return total

result = outer_fn(2, 3)

print(result)
```

The output will be 5 and here is the screenshot:

```
main.py ×
```

```
1 #nested function
2 ▼ def outer_fn(num1,num2):
3 ▼   def inner_fn(n1, n2):
4     return n1 + n2
5   total = inner_fn(num1,num2)
6   return total
7
8 result = outer_fn(2,3)
9
10 print(result)
```

Console Shell

5 > []

From the figure above, we have two functions, one is **outer\_fn** which takes two parameters **num1** and **num2** and the second function is **inner\_fn**. On line 8, we call the function with two arguments, and whatever is returned is stored in the variable called **result**. When we call that function(**outer\_fn**), the Interpreter goes to line 2, and now the two function parameters **num1** and **num2** will have access to the values 2 and 3 respectively. Then it goes into the body of the **outer\_fn** which will be in line 3. In line 3, we have another function declaration that takes two new parameters **n1** and **n2**. This function will be skipped because no one called it yet. The interpreter goes to line 5 where we have the actual call to the **inner\_fn** with two arguments which are the parameters from the **outer\_fn**. This is the same as if we wrote it like this:

```
total = inner_fn(2,3)
```

Next, the interpreter goes to line 3 again but now the parameters n1 and n2 will have the values 2 and 3 and inside the function body we are returning the sum of these 2 parameters. The function returns and goes to line number 6 where it returns the output with the result from the **inner\_fn**. In line 8, the result variable will have the value 5 that is returned from **outer\_fn**, and next, it will go to line 10 where it will print the result. That is how the **inner\_fn** can use and have access to the variables/parameters from the **outer\_fn**. Remember the key part here played the ‘return’ keyword otherwise we would never have gotten this result back.

## Docstrings in Python

The docstrings in Python are string literals that will appear after the definition of function, class, method, or even a module. First, we need to see how we can write a string literal:

```
''' This is string literal '''
```

We write the string literals inside triple quotation marks. Let’s now create a function that will calculate the square of a number:

```
# docstrings
def num_square(num):
    """This function takes a number and returns the square of that number"""
    return num**2

result = num_square(4)
print(result)
```

If we run the code above, this will be the output:

A screenshot of a Python code editor showing a file named 'main.py'. The code contains a function definition with a docstring:`1 #docstrings
2 ▼def num_square(num):
3 '''This function takes a number and returns the
4 square of that number'''
5
6 result = num_square(4)
7 print(result)`The output window on the right shows the result of running the code: '16'.

As you can see, we wrote the string literal in triple quotation marks after the function declaration. And if we run this function, we will not see the string literal. If you read the comment in the template literal you will see that it is a short description or info on what this function does. If we hover with our mouse over the function call, we will see the string literal message:

A screenshot of a Python code editor showing a file named 'main.py'. The code contains a function definition with a docstring:`1 #docstrings
2 ▼def num_square(num):
3 '''This fu
4 square of that
5 return num
6 result = num_square(4)
7 print(result)`A tooltip is displayed over the call to 'num\_square(4)', showing the docstring: 'This function takes a number and returns the square of that number'.

This is called a docstring, and it is basically a comment inside the function that explains what the function does. This is very good if we are working on a large application and if other developers want to join so they can quickly read about the functions without going through the entire documentation. Another way to find out what the function does is by using the `help()` function. This is another built-in Python function that is used by developers to understand what some of the functions are:

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:`1 #docstrings
2 def num_square(num):
3 '''This function takes a number and
4 returns the square of that number'''
5 return num**2
6
7 result = num_square(4)
8 help(num_square)`On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active, showing the output of the command "help(num\_square)". The output is:`Help on function num_square in module __main__:
num_square(num)
 This function takes a number and returns the square of
 that number
> []`

We can even do this:

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the command "help(len)". On the right, the "Console" tab is active, showing the output of the command "help(len)". The output is:`Help on built-in function len in module builtins:
len(obj, /)
 Return the number of items in a container.`

As you can see, the `len()` function will return the number of items in a container. There are other ways to see what the function does, for example, we can use the magic or dunder methods. The magic or dunder methods in Python are the special methods that start and end with the double underscores:

A screenshot of a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:`1 #docstrings
2 def num_square(num):
3 '''This function takes a
4 number and returns the square of
5 that number'''
6 return num**2
7
8 result = num_square(4)
9 print(num_square.__doc__)`On the right, the "Console" tab is active, showing the output of the command "print(num\_square.\_\_doc\_\_)". The output is:`This function takes a number and returns the square of that number
> []`

To summarize, the docstrings are the comments that we want others to see. These comments should be meaningful and explain what the function does.

## Python \*args and \*\*kwargs

In this section, we will learn about ‘args’ and ‘kwargs’ and how they can be used. We know the functions we defined are reusable pieces of code that can be called or invoked multiple times. During our function call, we pass specific values known as arguments. Let’s define a function that will calculate how old you are based on your birth year and current year:

```
# args and kwargs
def years_old(curr, dob):
    return curr - dob

age = years_old(2022, 1999)
print(age)
```

The screenshot shows a Python development environment. On the left, the code editor window titled "main.py" contains the provided Python script. On the right, the "Console" tab is active, displaying the output of the script's execution. The output shows the value 23, indicating the age calculated by the function.

```
main.py ×
1
2 # args and kwargs
3 ▼ def years_old(curr, dob):
4     return curr - dob
5
6
7 age = years_old(2022, 1999)
8 print(age)

23
```

As we can see from the figure above, the current year is the first argument and the year of birth is the second argument. What will happen if we pass three arguments in the function call instead of the required two? Let us find out:

The screenshot shows a Python development environment. The code editor window is identical to the one in the previous screenshot. However, the "Console" tab now displays a traceback error message. The error message indicates that a `TypeError` occurred because the function `years_old()` was called with 3 positional arguments, but it only expects 2. The specific line of code causing the error is highlighted in red.

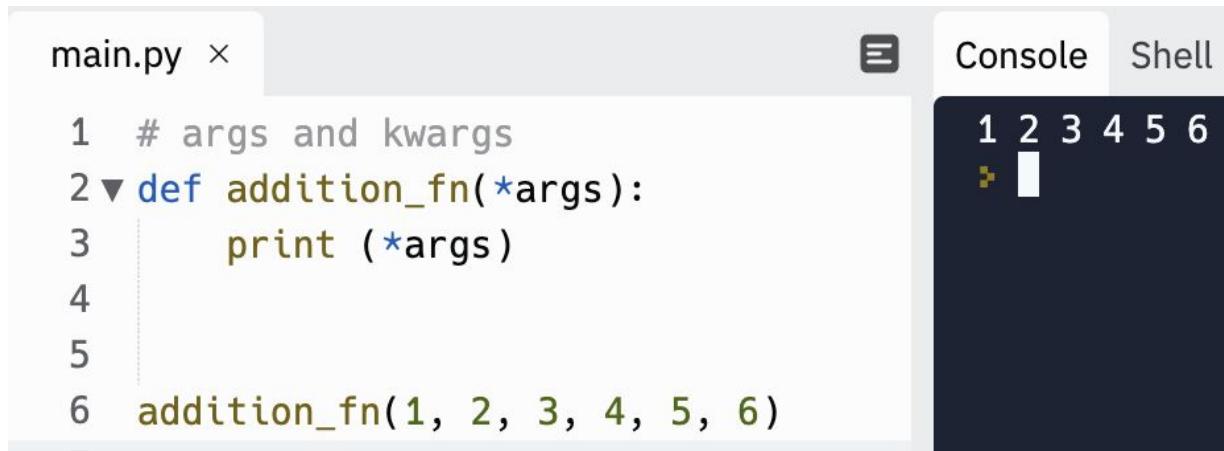
```
main.py ×
1 # args and kwargs
2 ▼ def years_old(curr, dob):
3     return curr - dob
4
5
6 age = years_old(2022, 1999)
7 age = years_old(2022, 1999, 2019)
8 print(age)

Traceback (most recent call last):
  File "main.py", line 7, in <module>
    age = years_old(2022, 1999, 2019)
TypeError: years_old() takes 2 positional arguments but 3 were given
```

We get an error saying we have given 3 arguments and the function `years_old()` takes only two positional arguments. This is excellent because it will help me explain what `*args` and `**kwargs` are in Python. The `*args` syntax stands for ‘Non-Keyword Argument’ and `**kwargs` stands for ‘Keyword Arguments’ but why are they useful? Well, we can use `*args` as arguments when we don’t know the exact number of arguments we need to pass in the function. This will prevent the program from crashing like in the figure above, so they are very useful when we don’t know how many arguments should be passed to the function. Let’s change the function declaration by using star `*args`. This will mean we can use any number of arguments. Let’s write a new function that will calculate the sum of numbers:

```
# args and kwargs
def addition_fn(*args):
    print (*args)

addition_fn(1, 2, 3, 4, 5, 6)
```



The screenshot shows a Python development environment with a code editor and a terminal window. The code editor has a file named `main.py` with the following content:

```
1 # args and kwargs
2 def addition_fn(*args):
3     print (*args)
4
5
6 addition_fn(1, 2, 3, 4, 5, 6)
```

The terminal window, titled "Console", shows the output of the code execution:

```
1 2 3 4 5 6
> 
```

As you can see, the `addition_fn` has 6 arguments, which are numbers from 1 to 6 but in the function declaration we have only used `*args` as a single parameter. On the right side, we can see that the arguments we passed in the `addition_fn` are accessed through `*args`. This gives us the ability to call the built-in Python function called `sum()`. This function will return the sum of all of the items from an iterable collection like a Tuple. But `*args` is not a Tuple. One way to use `*args` is without the star in front like this:

The screenshot shows a Python development environment. On the left, there's a code editor window titled "main.py" with the following content:

```
1 # args and kwargs
2 def addition_fn(*args):
3     print(args)
4
5
6 addition_fn(1, 2, 3, 4, 5, 6)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution:

```
(1, 2, 3, 4, 5, 6)
```

Great! Now we have a Tuple as an output so we can call the sum() function without any problems. Let's do this and check the output:

The screenshot shows the same Python development environment. The code in "main.py" has been modified:

```
1 # args and kwargs
2 def addition_fn(*args):
3     return sum(args)
4
5
6 result = addition_fn(1, 2, 3, 4, 5, 6)
7 print(result)
```

The "Console" tab shows the output:

```
21
```

As we can see, our logic works and we will get the sum of 21. To summarize, we use \*args when we are not sure how many arguments the function will take. We can use a different name for \*args like \*nums. Let's focus on \*kwargs. Kwargs is a dictionary of keyword arguments and the double '\*\*' allows us to pass any number of keyword arguments. We can add keywords in the function call and access them in the function body in form of dictionaries:

```
# args and kwargs
def addition_fn(*args, **kwargs):
    print(kwargs)
    return sum(args)

result1 = addition_fn(1, 2, 3, 4, 5, 6, n7=7, n8=8)
print(result1)
```

The screenshot shows a Python development environment with a code editor and a terminal window. The code in the editor is:

```
main.py ×
1
2 # args and kwargs
3 def addition_fn1(*args, **kwargs):
4     print(kwargs)
5     return sum(args)
6
7
8 result1 = addition_fn1(1, 2, 3, 4, 5,
9 n7=7, n8=8)
10
11 print(result1)
```

The terminal window shows the output of the code execution:

```
{'n7': 7, 'n8': 8}
21
> █
```

But how can we access the items and their values from `kwargs`? Well we can use a loop to iterate over the dictionary items and from there we can grab the values using the `.values()` method:

The screenshot shows a Python development environment with a code editor and a terminal window. The code in the editor is:

```
main.py ×
1
2 # args and kwargs
3 def addition_fn1(*args, **kwargs):
4     result = 0
5     for item in kwargs.values():
6         result += item
7     return sum(args) + result
8
9
10 result1 = addition_fn1(1, 2, 3, 4,
11 5, 6, n7=7, n8=8)
12
13 print(result1)
```

The terminal window shows the output of the code execution:

```
36
> █
```

As you can see, the function call can take as many positional arguments as we want using the `*args` syntax but it can also take named keywords arguments. But this is not everything you should know because there is one nice rule you need to follow when it comes to using the arguments. The rule is that if you use all of them in the same function, then their order is very important. The highest order has the parameters, then the `*args`, then the default parameters and `**kwargs` at the end. It is finally time to summarize what we can do with `*args` and `**kwargs`:

- 1) \*args and \*kwargs are used to make function argument list flexible and they are special keywords that allow the function to take any number of arguments
  - 2) \*args means non-keyword arguments and they occur before the \*\*kwargs in the function definition
  - 3) \*\*kwargs are keyword arguments and they are used in the function like a Dictionary
  - 4) You can use different names if you want; args and kwargs are a convention, not a requirement

## Scope

Scope is important in Python. Scope refers to the access we have to the variables. When we try to access a variable before it is even created, we will get this error:

The screenshot shows a code editor with a file named `main.py`. The code contains two lines:

```
1 #scope
2 print(result)
```

The word `result` is underlined with a red squiggle, indicating a syntax error. To the right, a terminal window titled "Console" shows the following traceback:

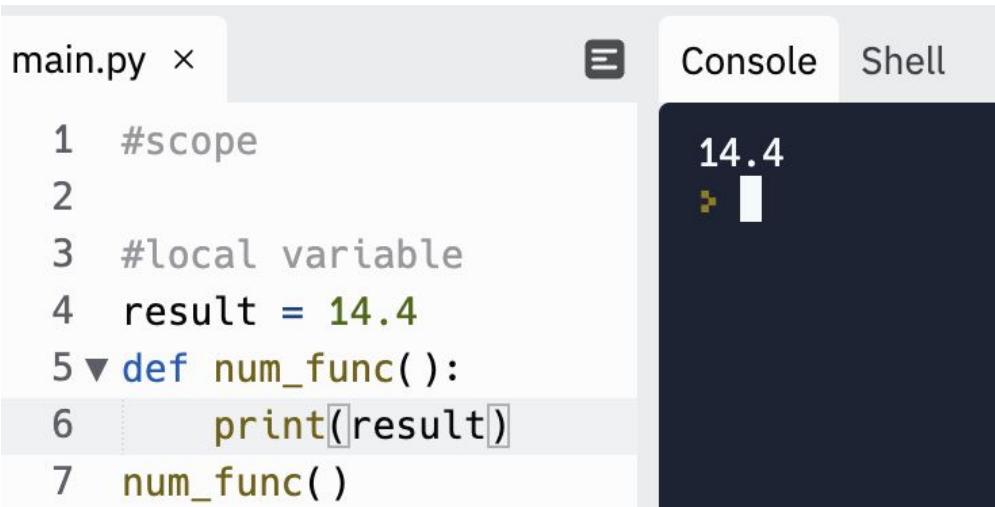
```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    print(result)
NameError: name 'result' is not defined
```

This is a **NameError** because the result variable is not even defined and we are trying to access it. This is happening because the variable does not live in that scope. That is why we need to learn about the existence of different scopes. If we declare and initialize a variable in Python outside of any function, this means the variable belongs to the global scope. The global scope is the scope that will provide access to this variable. Anyone can access a variable declared in the global scope. Let's declare a variable in the global scope and also create a function that will print that variable:

```
# local variable  
result = 14.4
```

```
def num_func():  
    print(result)
```

```
num_func()
```



The screenshot shows a Python development environment with two panes. On the left, the code editor displays a file named 'main.py' with the following content:

```
1 #scope  
2  
3 #local variable  
4 result = 14.4  
5 def num_func():  
6     print(result)  
7 num_func()
```

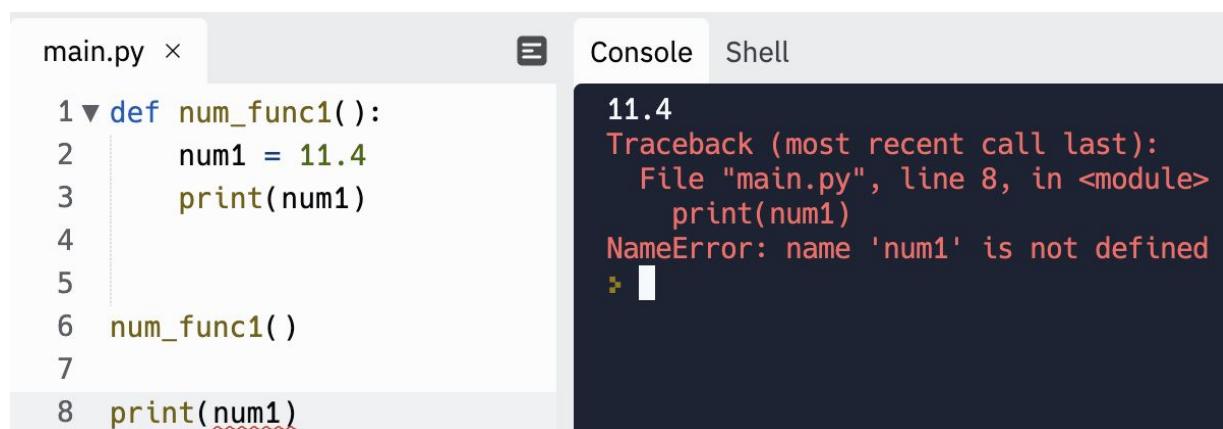
On the right, there are two tabs: 'Console' and 'Shell'. The 'Console' tab is active and shows the output of the code execution:

```
14.4  
▶
```

As we can see, the variable called **result** belongs to the global scope where it's been declared and bound to a value of 14.4. Because of the global scope, we can safely use this variable inside the function body. But the functions in Python have their own block scope. The function scope means whatever we have defined in the function body can be accessed and used in that function only. If we have variables declared in the function and we try to print them outside of the function body, we will get an error. Here is one example:

```
def num_func1():
    num1 = 11.4
    print(num1)
```

```
num_func1()
print(num1)
```



The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following code:

```
1 def num_func1():
2     num1 = 11.4
3     print(num1)
4
5
6 num_func1()
7
8 print(num1)
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is selected and shows the following output:

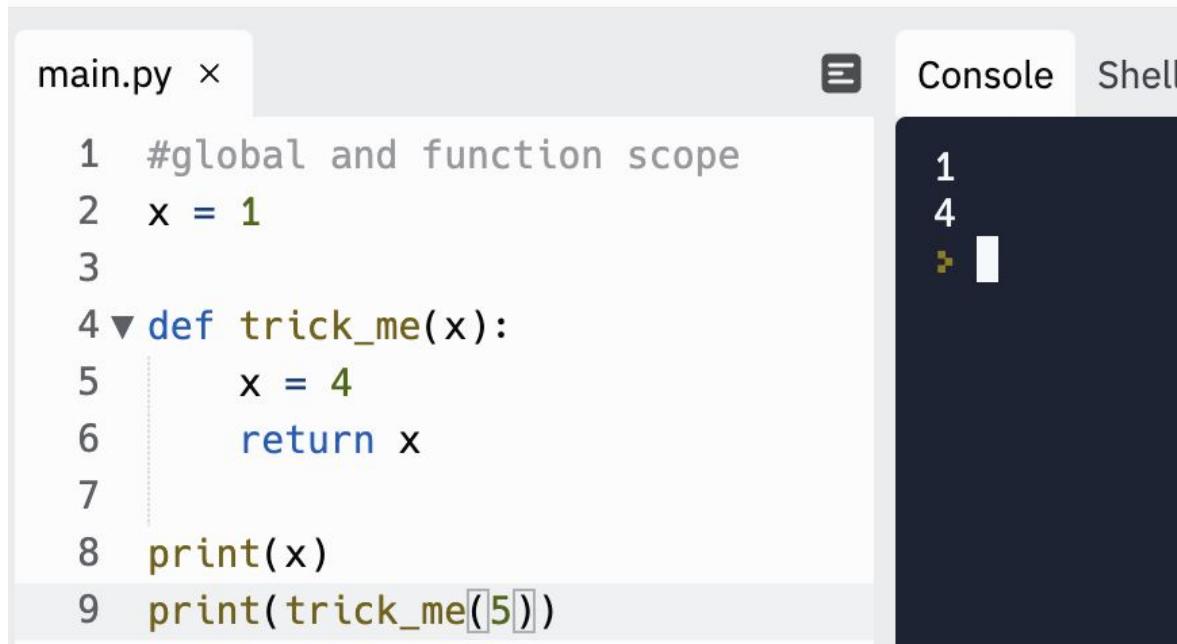
```
11.4
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    print(num1)
NameError: name 'num1' is not defined
> █
```

Line 8 is where we try to print a variable declared inside the function called `num_func1()`. This will cause the Interpreter to underline with red. If we run this code, we will get `NameError` because `num1` is not defined in the global scope. Can you please guess the output of the following code:

```
#guess the output
x = 1
def trick_me(x):
    x = 4
    return x
```

```
print(x)
print(trick_me(5))
```

If you couldn't figure it out here is the output:



The screenshot shows a Python development environment. On the left, there's a code editor window titled "main.py" containing the following Python code:

```
1 #global and function scope
2 x = 1
3
4 def trick_me(x):
5     x = 4
6     return x
7
8 print(x)
9 print(trick_me(5))
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is selected and shows the following output:

```
1
4
> |
```

As you can see on line eight, we are printing the value of `x` which is declared on line two. Do not get confused with the parameter `x` we passed into the function because we could have named this parameter differently. On line nine, I print the value of the `trick_me` where I pass an argument with a value of five, but this argument will not make any difference because line five will create a local variable for the function and re-assign the variable to a new value of four. I'm saying local variable because that variable is local for the function. On line six, we return the last value of the `x` which will be four. That is why the output of the function `trick_me` will be four.

## More about Scopes

In the previous section, I mentioned that the function parameters are local variables for that function. They belong to the function where they are defined and not to the global scope:

```
main.py ×

1 #global and function scope
2 x = 1
3
4 def trick_me(x):
5     #local variable
6     x = 4
7     return x
8
9 print(x)
10 print(trick_me(5))
```

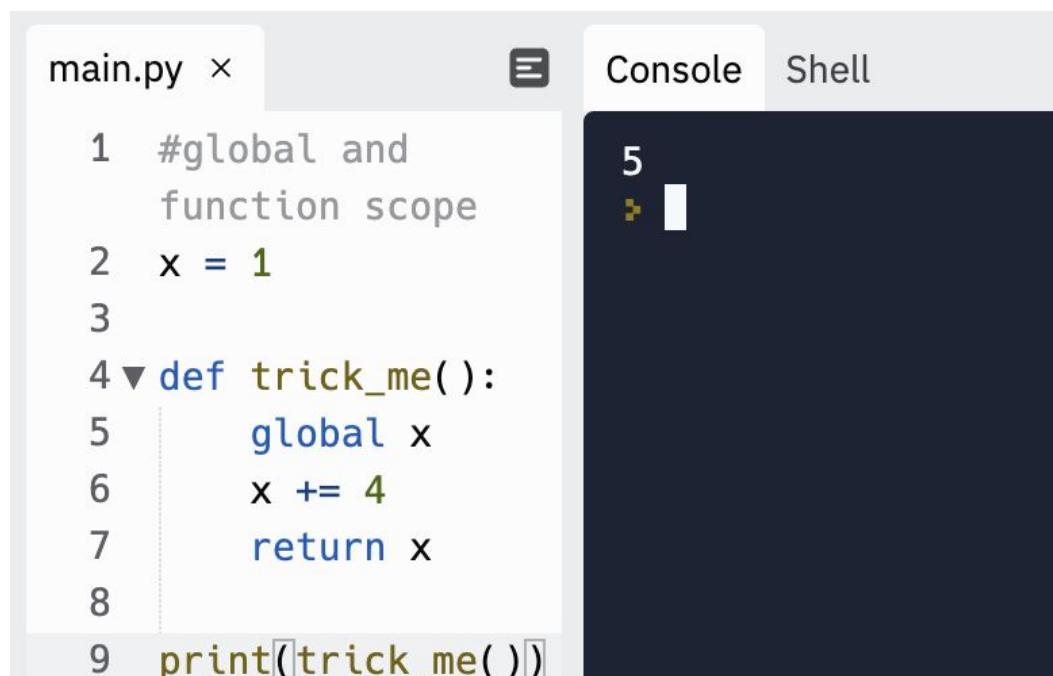
Another interesting error you will get if you are a beginner is the `UnboundLocalError`. What does this error mean? Let me explain it with this example:

```
main.py ×
1 #global and
2 function scope
3
4 def trick_me():
5     x += 4
6     return x
7
8 print(trick_me())
```

Console Shell

```
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    print(trick_me())
  File "main.py", line 5, in trick_me
    x += 4
UnboundLocalError: local variable 'x' referenced
before assignment
> []
```

As you can see from the figure above, we have created a variable **x** on line two and assigned a value of one. We have used that same global variable in our `trick_me` function and we incremented its value by four. Please note that the function `trick_me()` does not accept any parameters at this stage. On line eight, we call the `print` function and we get the `UnboundLocalError` saying the local variable ‘**x**’ is referenced before assignment. But this is not true because on line two we declared and assigned the variable ‘**x**’ to a value of one. But the function `trick_me` doesn’t know about what is being created outside the function scope. There must be a way the function can use the global variable **x**, right? Yes, luckily for us we can use the ‘`global`’ keyword inside the function body to tell the function that we want to use the variable that is defined in the global scope like our variable ‘**x**’:



The screenshot shows a Python IDE interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 #global and
  function scope
2 x = 1
3
4 def trick_me():
5     global x
6     x += 4
7     return x
8
9 print(trick_me())
```

On the right, there are two tabs: "Console" and "Shell". The "Console" tab is active and displays the output of the code execution:

```
5
```

But there is better way of accessing variables that are declared outside the function and we know this syntax because we have used it before:

The screenshot shows a Python development environment. On the left, a code editor window titled "main.py" contains the following Python code:

```
1 #global and function scope
2 x = 1
3
4 def trick_me(x):
5     x += 4
6     return x
7
8 print(trick_me(x))
```

On the right, a "Console" window displays the output of the code execution:

```
5
```

This is what we were doing so far, we passed the global variable `x` as an argument on line 8 and we can use it in the function body through parameter `x`. This is much better than using the `global` keyword inside the function, right?

## Python nonlocal Keyword

The non-local keyword was added in Python 3 and can be explained if we use nested functions, therefore we need to go over the following example:

```
# nonlocal
def outer_fn():
    x = 'John'

    def inner_fn():
        nonlocal x
        x = 'hello'
    inner_fn()
    return x

print(outer_fn())
```

The screenshot shows a Python IDE interface. On the left, the code editor window titled "main.py" contains the following Python code:

```
1 #nonlocal
2 ▼ def outer_fn( ):
3     x = "John"
4 ▼     def inner_fn( ):
5         nonlocal x
6         x = "hello"
7     inner_fn( )
8     return x
9
10 print(outer_fn( ))
```

On the right, the "Console" tab is active, showing the output of the code execution:

```
hello
> █
```

We can use the non-local keyword like the one on line five so we can work with variables that are declared in the outer functions. We use the nonlocal keyword to declare that the variable is not local for that function, meaning it belongs to the functions that are one level up. When the Interpreter sees the nonlocal 'x', it will automatically know that we don't want to create another local variable for that function but to use the variable that is declared in the outer function. The outer function is the parent of the inner function. On line five, the non-local variable value will be 'John' which comes from line three of the **outer\_fn** body. Line six is where we overwrite the value of x from 'John' to 'hello.' This will modify the value of x for the **outer\_fn** and when we return the x, the value will be 'hello.' What do you think will happen if we remove line number five? It will print John because the x will become a local variable for the **inner\_fn()** that will never be used anywhere. The yellow underline in Python occurs when we assign a variable but never use it, it's like a soft warning.

The screenshot shows a Python development environment with two tabs: "Console" and "Shell". The "Console" tab is active and displays the output "John". The "Shell" tab is also visible. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 #nonlocal
2 ▼ def outer_fn():
3     x = "John"
4 ▼   def inner_fn():
5       # nonlocal x
6       x = "hello"
7       inner_fn()
8   return x
9
10 print(outer_fn())
```

So what are the advantages of having non-local variables in our code? Well, the first one is that we can now access the variables that are declared in the upper scope. When we reuse the same variable in the inner function, we don't create another variable somewhere in memory so we save memory space and extra referencing. The disadvantage is that this keyword works only inside nested structures and we can't use it to reference global or local variables.



# Chapter 3 Python Installation

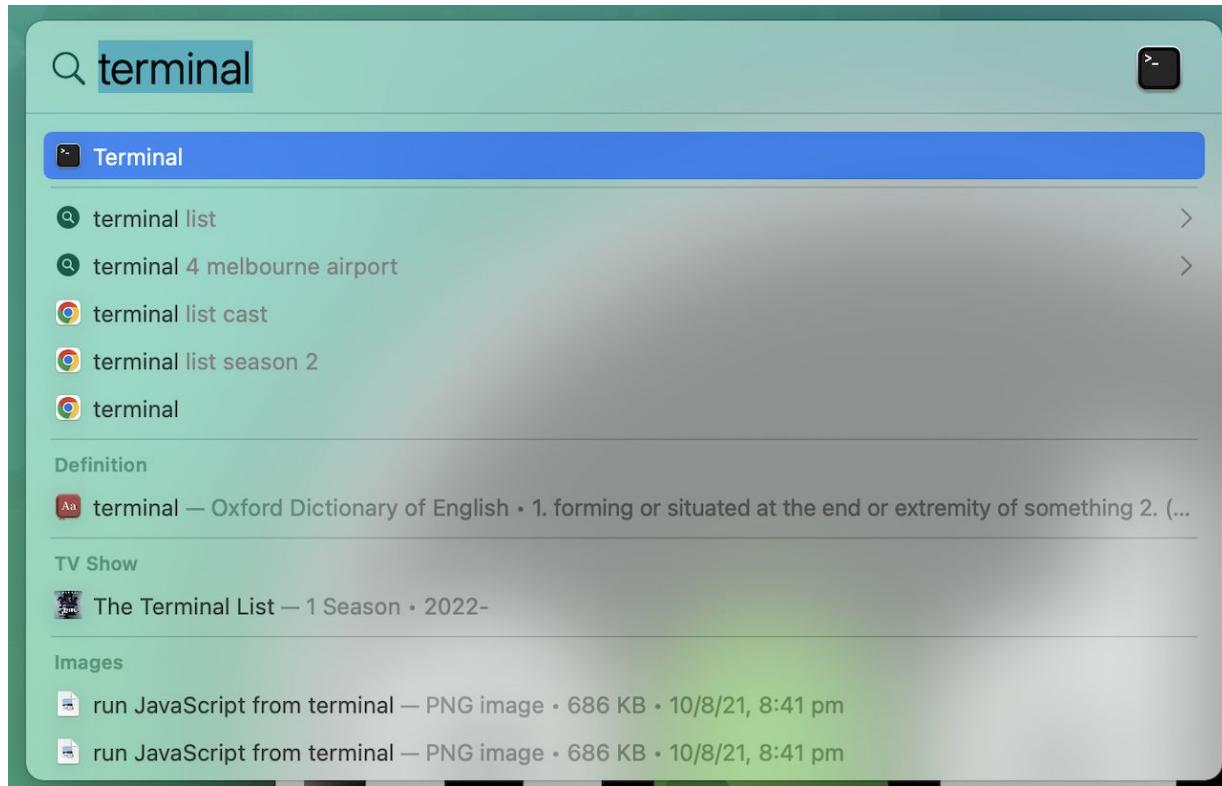
In this section, we will learn how to install Python on different operating systems like Windows, macOS, and Linux. We will also learn the most important Terminal and Command Prompt commands. Finally, we will install a few tools (VS Code and PyCharm) that you can use to write and execute Python code. You don't have to install both of them because that is not the point. You will make a decision on which of the tools suits you the most. PyCharm is one of the best IDEs on the market and I think it's best for writing Python because it's built for this language. Among the code editors, the best one is the VS Code. The editors are popular because you can use them to write and execute different programming languages. I decided to give you the best tools available on the market.

## Python installation

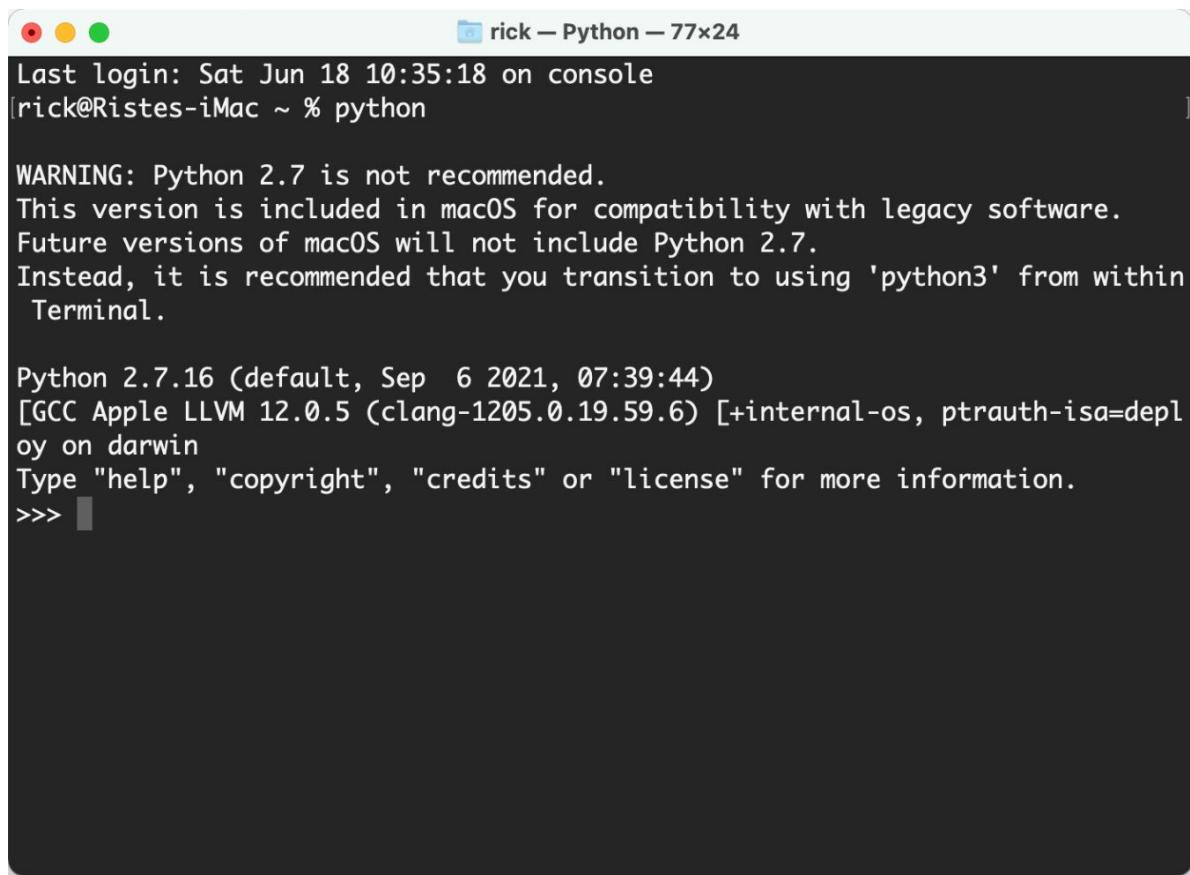
So far we have used the repl.it.com website where we wrote and executed the Python code, but this is not the solution if you are serious about Python programming. We can install Python on our machines and we should know that the installation is different for each of the operating systems. If you are using a macOS or Linux operating system, then the possibility of having Python pre-installed is big. I will guide you on how you can install it on macOS and Windows operating systems because these are the two most used systems by developers and I also will provide an additional link for Linux users so they can read and install it on their machines as well. Let's start with macOS Python installation. If you are a Windows user, you can just skim this section.

### macOS Python installation

We can check if Python is installed on your Mac machine if you open the terminal, so press command + space and this will open up the Search bar where you type 'terminal' and press enter/return like in the figure below:



When the terminal is opened, you can type ‘python’ and press return/enter and this is one of the outputs that you might get:



Last login: Sat Jun 18 10:35:18 on console  
[rick@Ristes-iMac ~ % python

WARNING: Python 2.7 is not recommended.  
This version is included in macOS for compatibility with legacy software.  
Future versions of macOS will not include Python 2.7.  
Instead, it is recommended that you transition to using 'python3' from within Terminal.

Python 2.7.16 (default, Sep 6 2021, 07:39:44)  
[GCC Apple LLVM 12.0.5 (clang-1205.0.19.59.6) [+internal-os, ptrauth-is-a=deploy on darwin]  
Type "help", "copyright", "credits" or "license" for more information.  
>>> █

I already have Python version 2.7 but it says I need to transition to Python3. I mentioned that you might not get this message and instead you will be inside something called repl or read-eval-print loop. Let's see if we can write Python code:

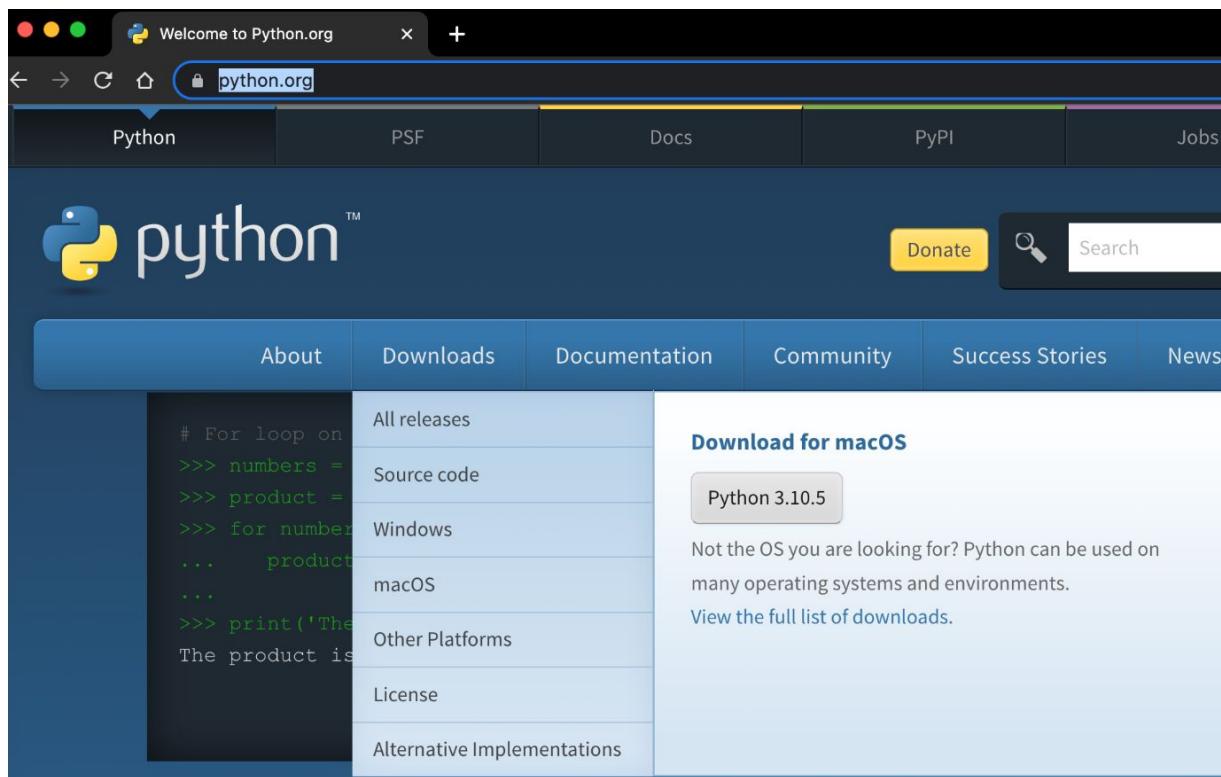
```
Last login: Sat Jun 18 10:35:18 on console
[rick@Ristes-iMac ~ % python

WARNING: Python 2.7 is not recommended.
This version is included in macOS for compatibility with legacy software.
Future versions of macOS will not include Python 2.7.
Instead, it is recommended that you transition to using 'python3' from within
Terminal.

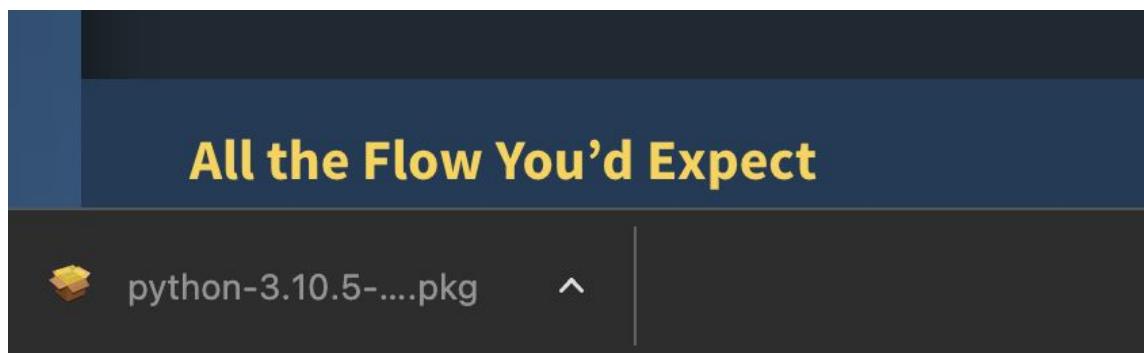
Python 2.7.16 (default, Sep  6 2021, 07:39:44)
[GCC Apple LLVM 12.0.5 (clang-1205.0.19.59.6) [+internal-os, ptrauth-is-a=deploy on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello')
hello
>>>
```

As you can see, it's working just like in repl.it.com but my current version of Python is very old and it's not even Python3 so let us go to python.org:

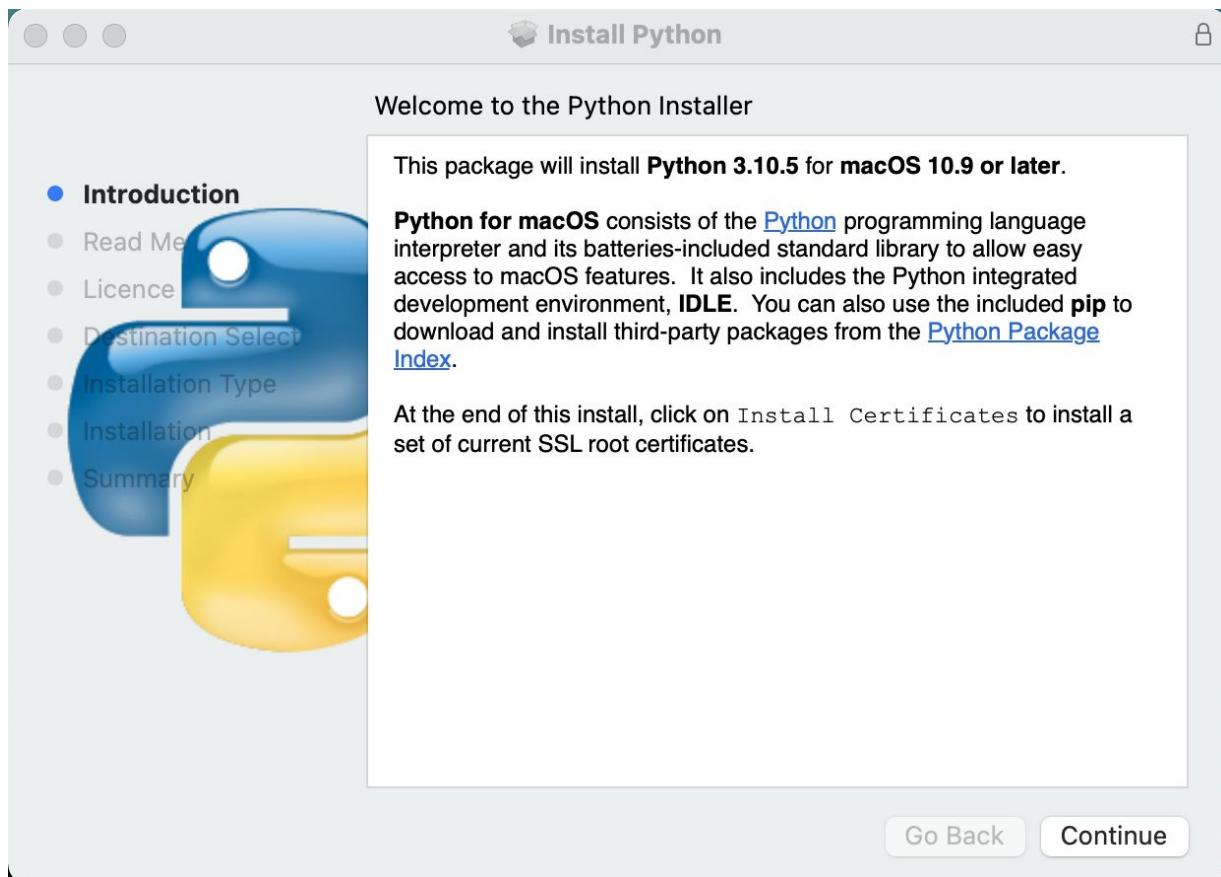
<https://www.python.org/>



From this website, you need to go to the downloads tab and download the version for your operating system. As you can see, it automatically recognized that my operating system is macOS and the current Python version that I need to download is 3.10.5. This version will definitely be different when you read this book but that will not be a problem because the installation will be the same. So click install and it will start downloading the package:



After the packages have been downloaded, you can click on it so it will open the installation wizard:



After opening the package, you need to press the continue button in the right bottom corner, and on the next page should be the ‘Read Me’ section where you should scroll till the end (it is a good idea to read the text but it’s not going to be a big deal if you don’t):



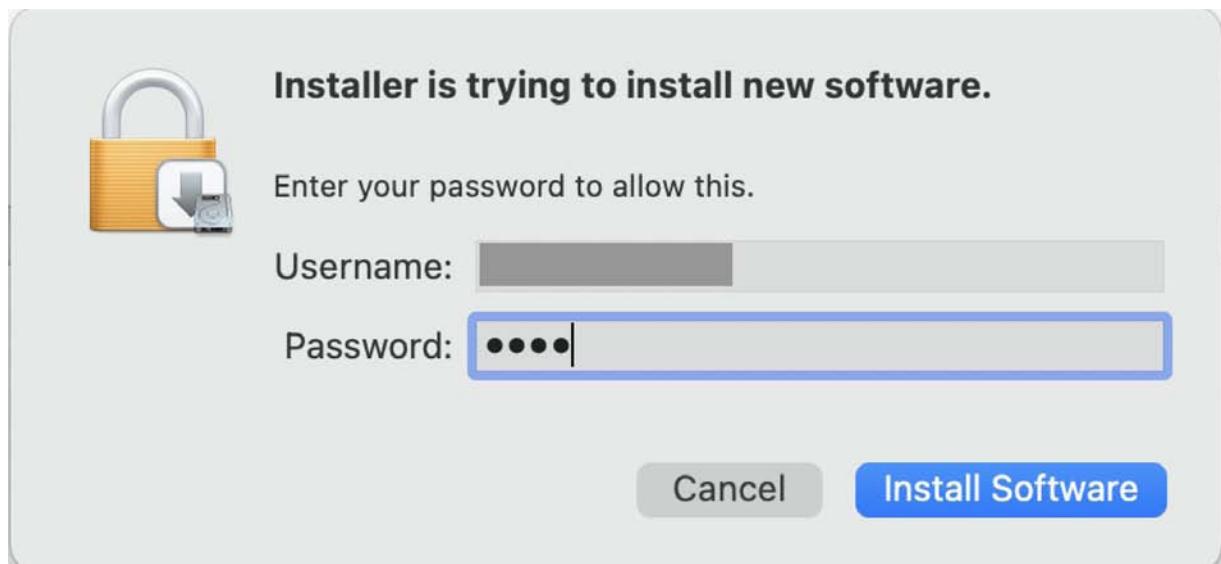
Continue by clicking on the ‘Continue’ button and ‘Agree’ with the license, then select the disk where you want the installation to happen:



And then it will finally give you the option to install Python:



It will ask for your password or fingerprint so provide your details and you are done with this process:



Finally, after a while if the installation is successful, you will see this window:



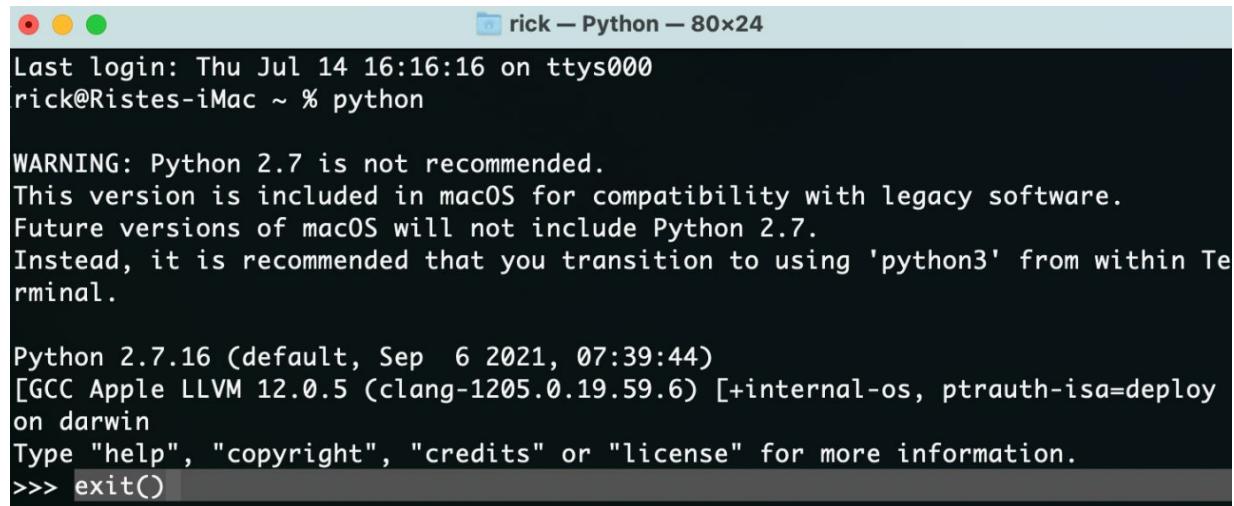
Click the ‘close’ button and you have now installed the latest version of Python on your macOS. If the installation was successful, we can close the previous terminal window or we can even type `exit()` and press return. Next is to check the latest version by typing ‘`python`’:

```
rick — Python — 80x24
Last login: Thu Jul 14 16:16:16 on ttys000
rick@Ristes-iMac ~ % python

WARNING: Python 2.7 is not recommended.
This version is included in macOS for compatibility with legacy software.
Future versions of macOS will not include Python 2.7.
Instead, it is recommended that you transition to using 'python3' from within Terminal.

Python 2.7.16 (default, Sep 6 2021, 07:39:44)
[GCC Apple LLVM 12.0.5 (clang-1205.0.19.59.6) [+internal-os, ptrauth-isa=deploy
on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

This was probably not the window you expected to see but this is normal when it comes to installing Python, the installations are tricky, so type exit():

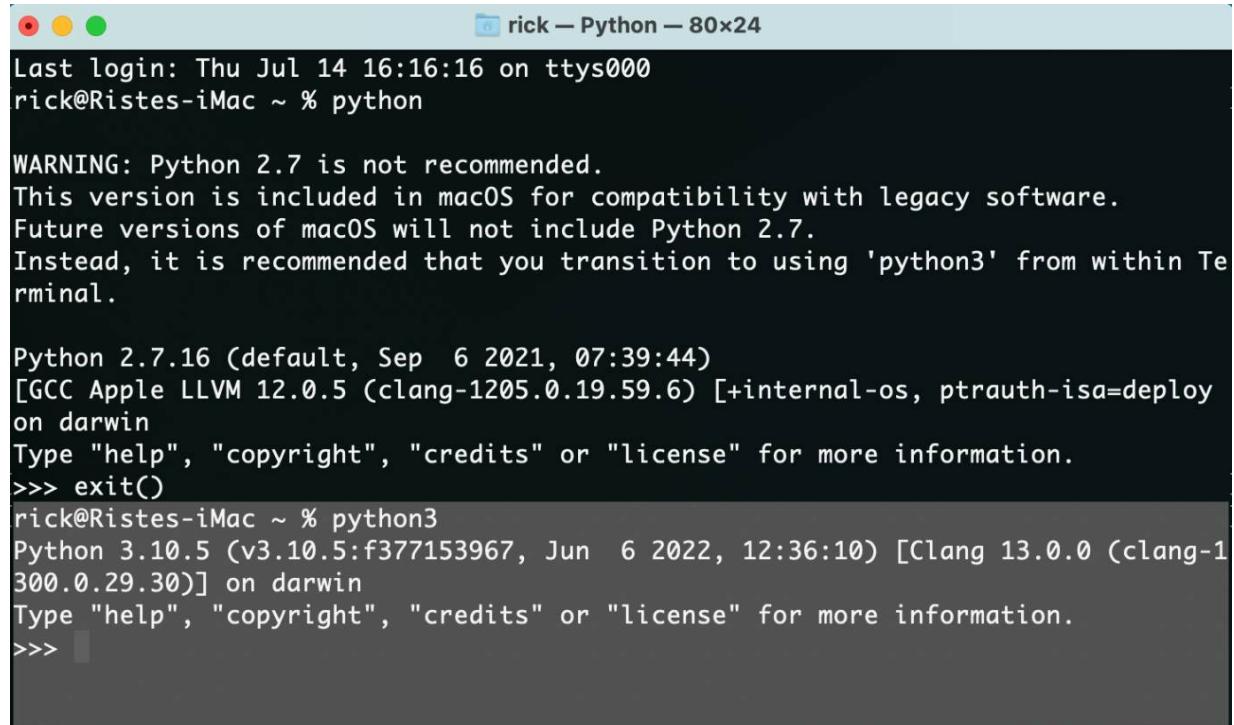


```
Last login: Thu Jul 14 16:16:16 on ttys000
rick@Ristes-iMac ~ % python

WARNING: Python 2.7 is not recommended.
This version is included in macOS for compatibility with legacy software.
Future versions of macOS will not include Python 2.7.
Instead, it is recommended that you transition to using 'python3' from within Terminal.

Python 2.7.16 (default, Sep 6 2021, 07:39:44)
[GCC Apple LLVM 12.0.5 (clang-1205.0.19.59.6) [+internal-os, ptrauth-is-a=deploy
on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

Finally, we can try again but this time type python3 instead of python and hit return or enter:



```
Last login: Thu Jul 14 16:16:16 on ttys000
rick@Ristes-iMac ~ % python

WARNING: Python 2.7 is not recommended.
This version is included in macOS for compatibility with legacy software.
Future versions of macOS will not include Python 2.7.
Instead, it is recommended that you transition to using 'python3' from within Terminal.

Python 2.7.16 (default, Sep 6 2021, 07:39:44)
[GCC Apple LLVM 12.0.5 (clang-1205.0.19.59.6) [+internal-os, ptrauth-is-a=deploy
on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
rick@Ristes-iMac ~ % python3
Python 3.10.5 (v3.10.5:f377153967, Jun 6 2022, 12:36:10) [Clang 13.0.0 (clang-1
300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Great! After providing the version number of Python, we have success. In the future, we might even need to type python4 in order to see if the latest version was successfully installed. If for some reason you still cannot install Python, then I suggest opening the following link:

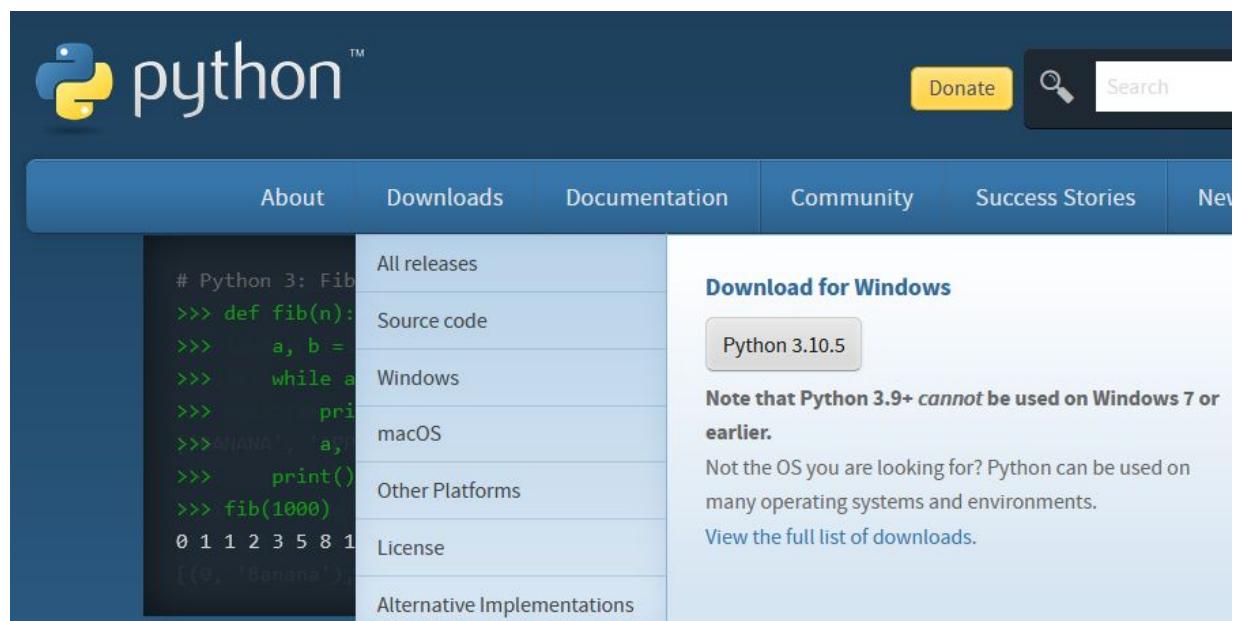
<https://osxdaily.com/2018/06/13/how-install-update-python-3x-mac/>

## Windows OS Python installation

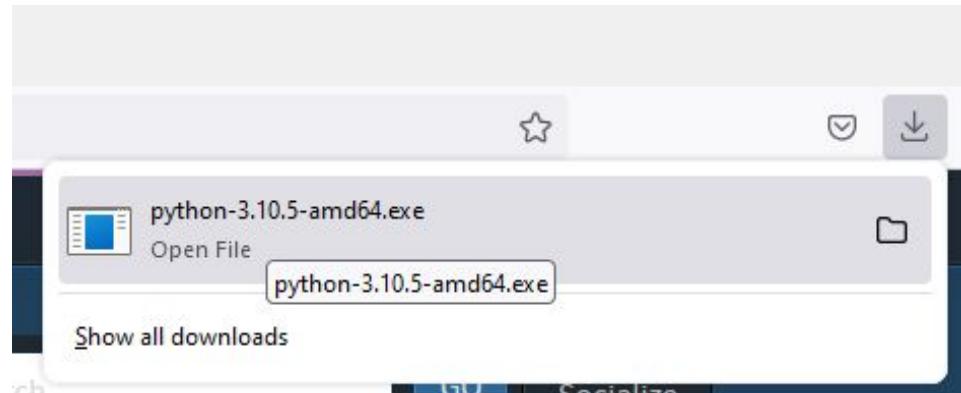
To start the installation process, you need to open the python.org website on your computer:

<https://www.python.org/>

Select the downloads tab and click on Python version 3.10.5 so it can start downloading the Python file. It should recognize your machine operating system version. In my case, it's a 64-bit system but you might have another version like 32-bit. If there is an option to choose, make sure the system version you select is the right one based on your computer configuration. Important, Python 3.10 version cannot be used on Windows 7 or earlier versions. The Python version number is currently 3.10.5 but when you read this book the numbers might be different and this is normal as new versions tend to pop up pretty fast. You should not worry because it will not change the installation process:



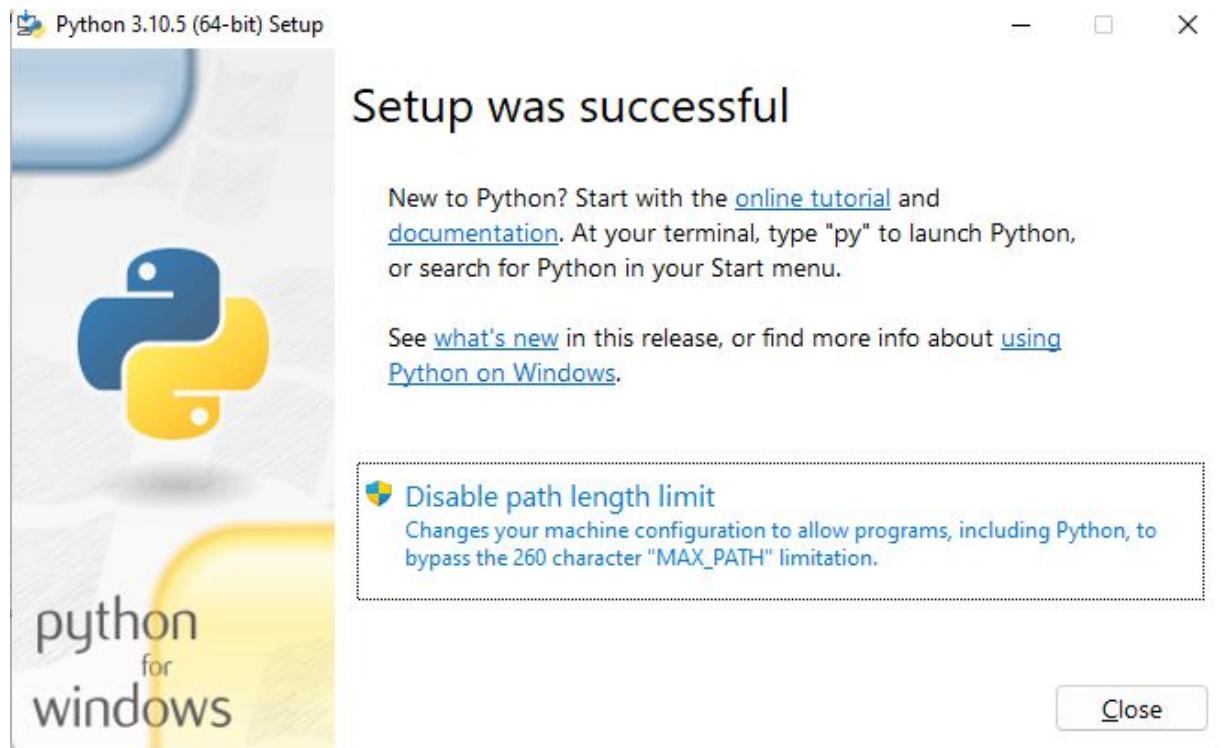
After downloading the file, you can find it in your downloads folder so click on the file so you can start the installation:



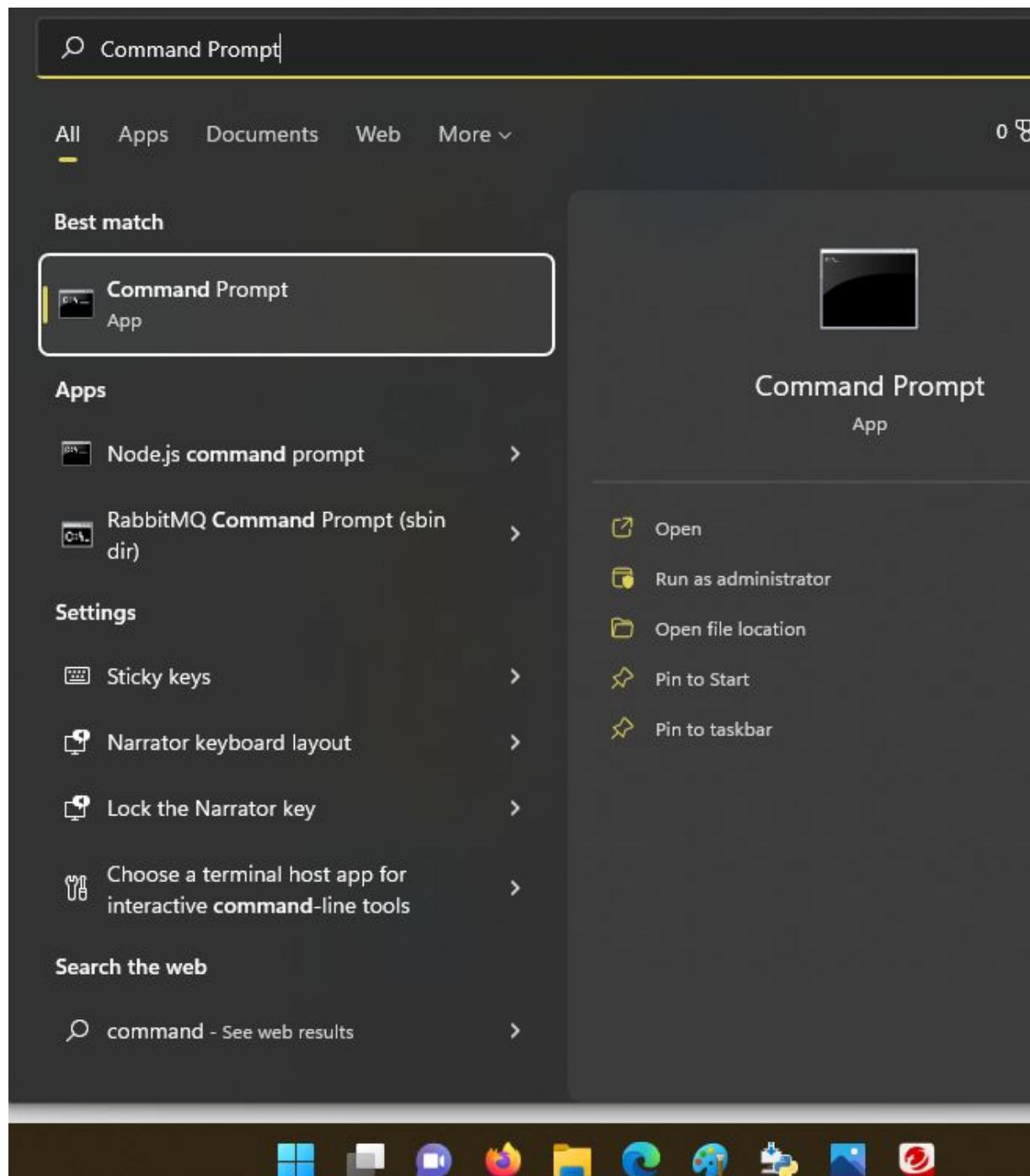
Make sure you select the last two options, the first one should be preselected but the second one saying ‘Add Python 3.10 to PATH.’ It is very important to select this before you start the installation. Click the install now and follow the setup wizard:



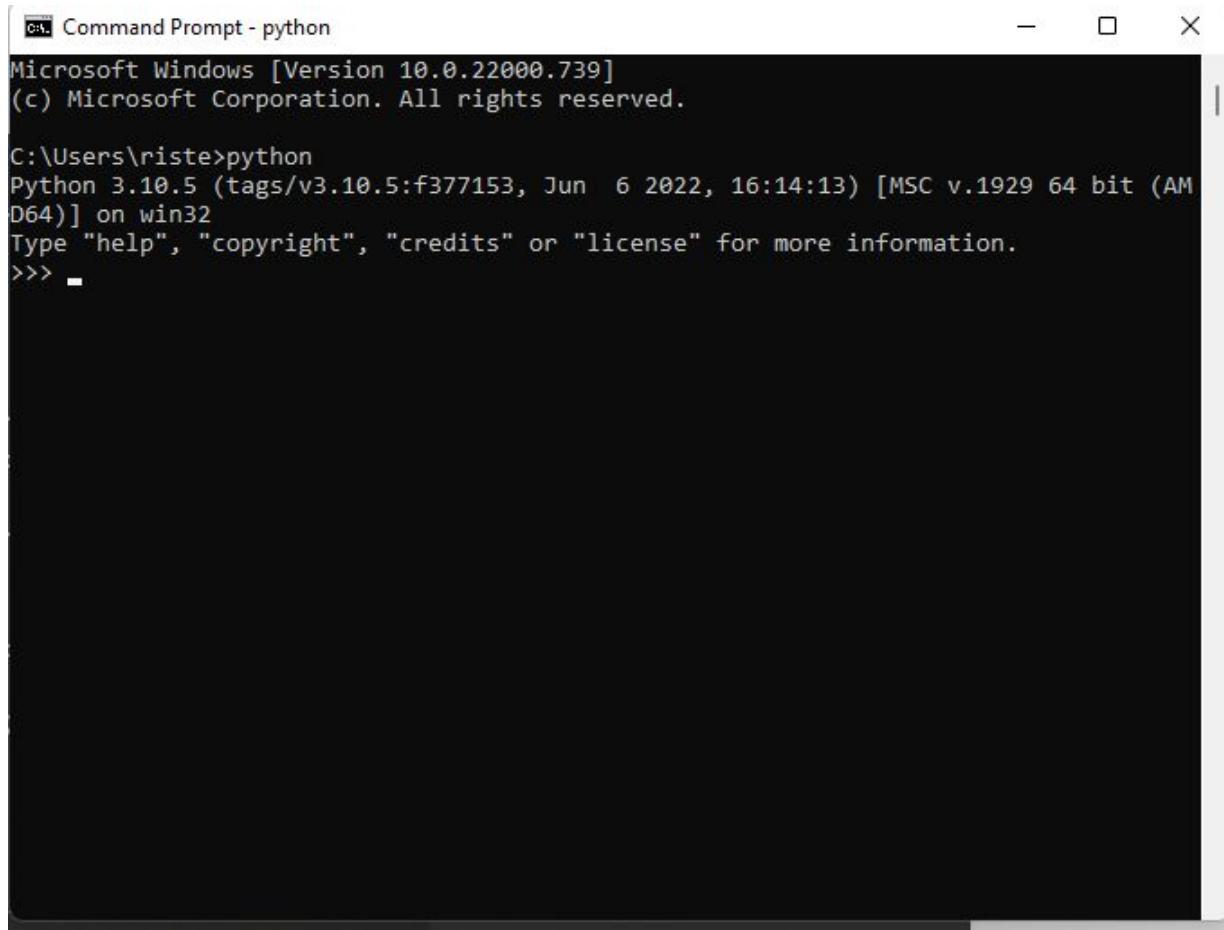
Make sure you allow the app to make changes to your device so click yes and the installation should start immediately; after a while, you should see the following window:



Click on the close button and find the command prompt. If you have an older version of Windows, you can search for it or if you have the latest version like me, you can click on the start Windows icon and type ‘Command Prompt’ in the search bar:



This will open the command prompt where you should type 'python' and press enter:

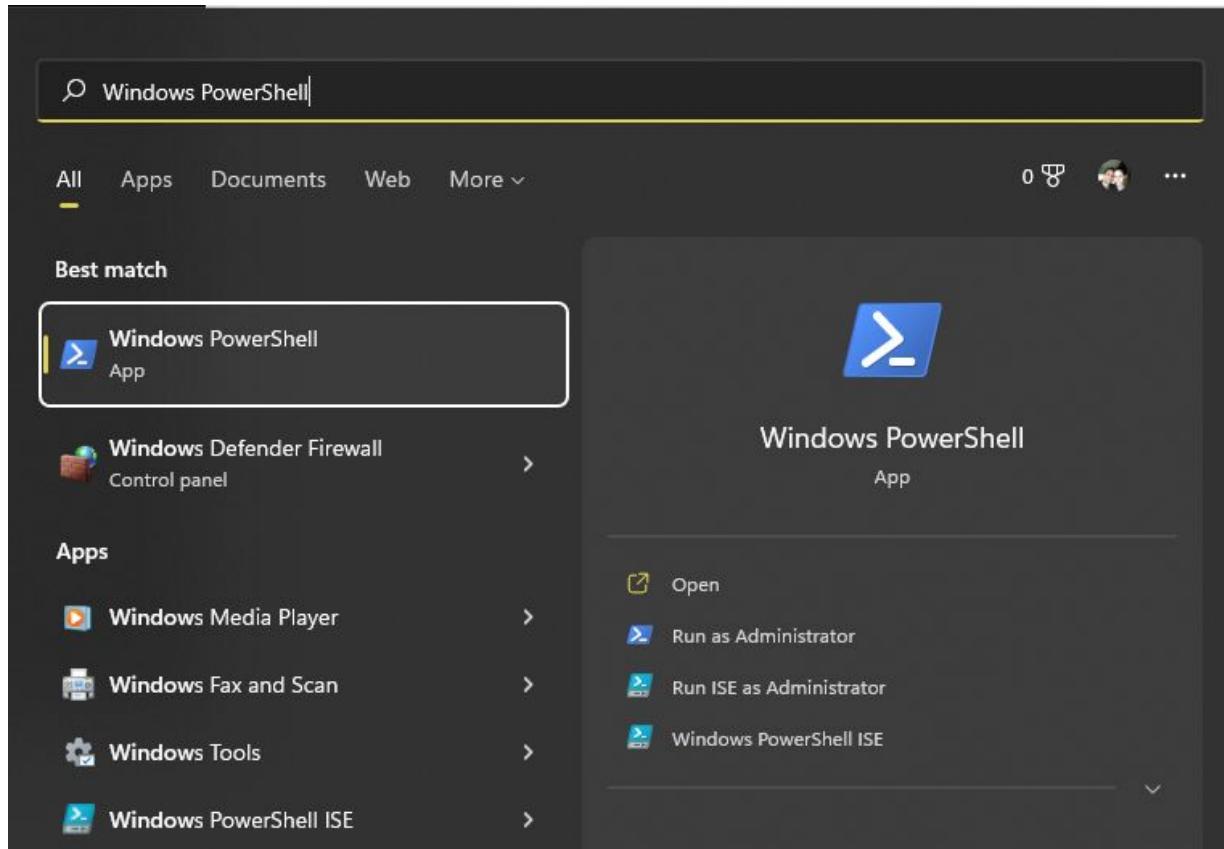


The screenshot shows a Windows Command Prompt window titled "Command Prompt - python". The window displays the following text:

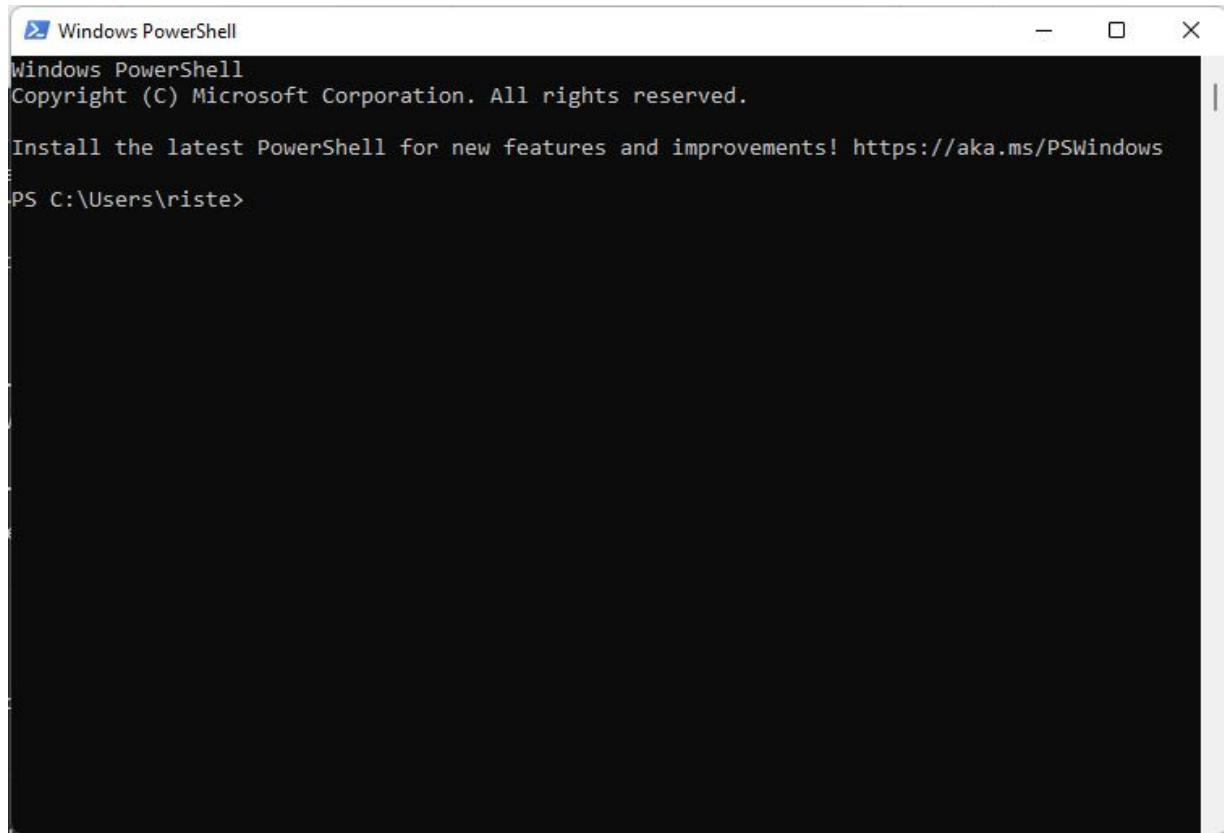
```
Microsoft Windows [Version 10.0.22000.739]
(c) Microsoft Corporation. All rights reserved.

C:\Users\riste>python
Python 3.10.5 (tags/v3.10.5:f377153, Jun  6 2022, 16:14:13) [MSC v.1929 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> -
```

Now we can see that Python version 3.10.5 has been successfully installed and working. In Windows, we can even use PowerShell. PowerShell is even more preferable among developers because the same commands that work on Linux and Mac machines work here as well. So just like we searched for the command prompt, we can search for Windows PowerShell:



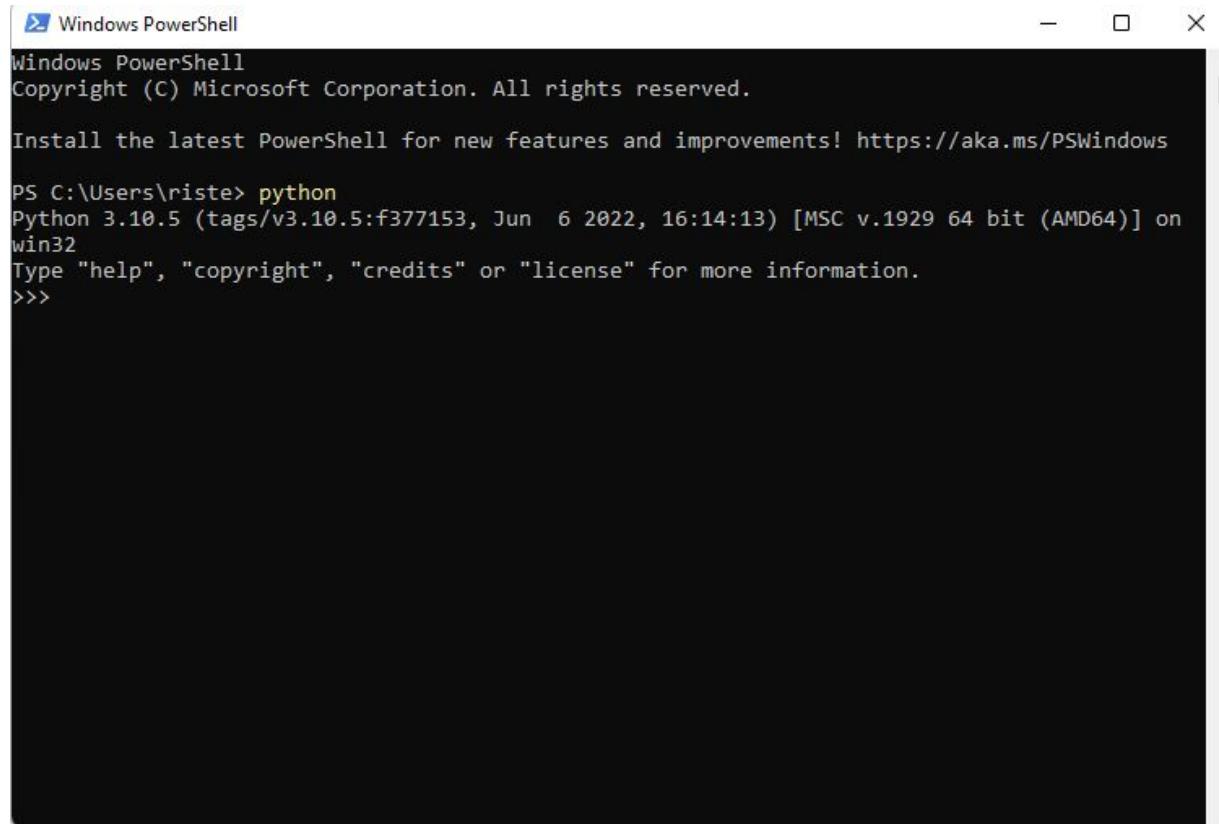
We can open the PowerShell and this is what it will look like:

A screenshot of a Windows PowerShell window titled "Windows PowerShell". The window shows the standard PowerShell startup message: "Windows PowerShell", "Copyright (C) Microsoft Corporation. All rights reserved.", and a link to install the latest version: "Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows". Below this, the command ".PS C:\Users\riste>" is visible, followed by a blank black area where output would normally appear.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows
.PS C:\Users\riste>
```

As you can see, I need to update the PowerShell version as well, but if you haven't got PowerShell on your Windows machine, I suggest you download it because you will be able to use the same commands I use on my Mac. Type Python on the PowerShell and press enter and you should see the same version number of Python that we just installed:



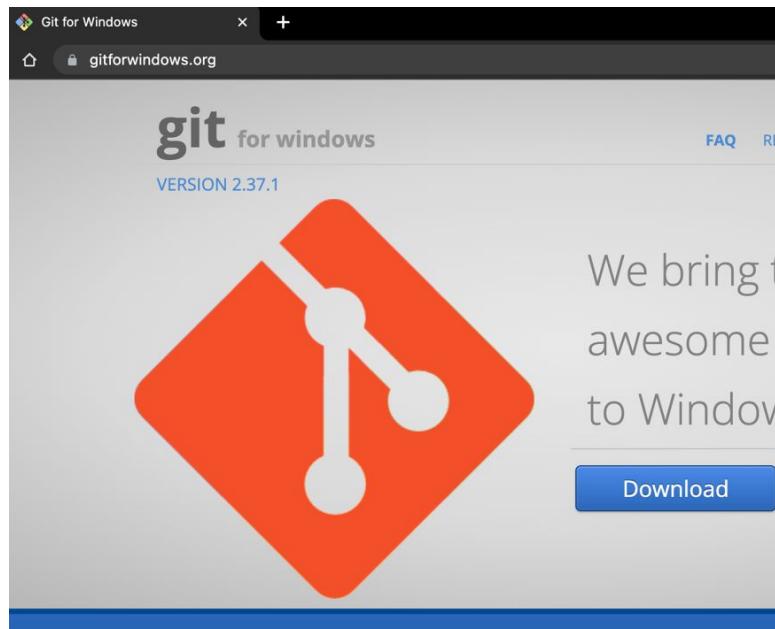
A screenshot of a Windows PowerShell window. The title bar says "Windows PowerShell". The content area shows the following text:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\riste> python
Python 3.10.5 (tags/v3.10.5:f377153, Jun  6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Another way to run Python code on your Windows is to install Git Bash or Git for Windows. This is basically a BASH emulation, which is a shell that will allow you to use UNIX commands:



To summarize, in order to run Python, you can use the Command Prompt, PowerShell, or Git Bash. I recommend sticking to PowerShell if you are a Windows user and updating it if you need to. Now the option we checked to add the PATH allows us to run the Python command in PowerShell. The Python installation was super important and now in the next sections, we will learn how to install different tools so we can write and execute Python code.

*Windows announced you can install Windows terminal if you use Windows 10 or later. This will allow you to use the exact same commands as other Linux and Mac users.*

*Install and get started setting up Windows terminal by clicking this link and following the steps:*

<https://docs.microsoft.com/en-us/windows/terminal/install>

## **LINUX user only - Python Installation**

If you are a Linux user, then you can read the following resources on how to install Python:

Direct link to download the Python:

<https://www.python.org/>

For some installation issues :

<https://realpython.com/installing-python/>

The Installation Guide:

<https://realpython.com/installing-python/>

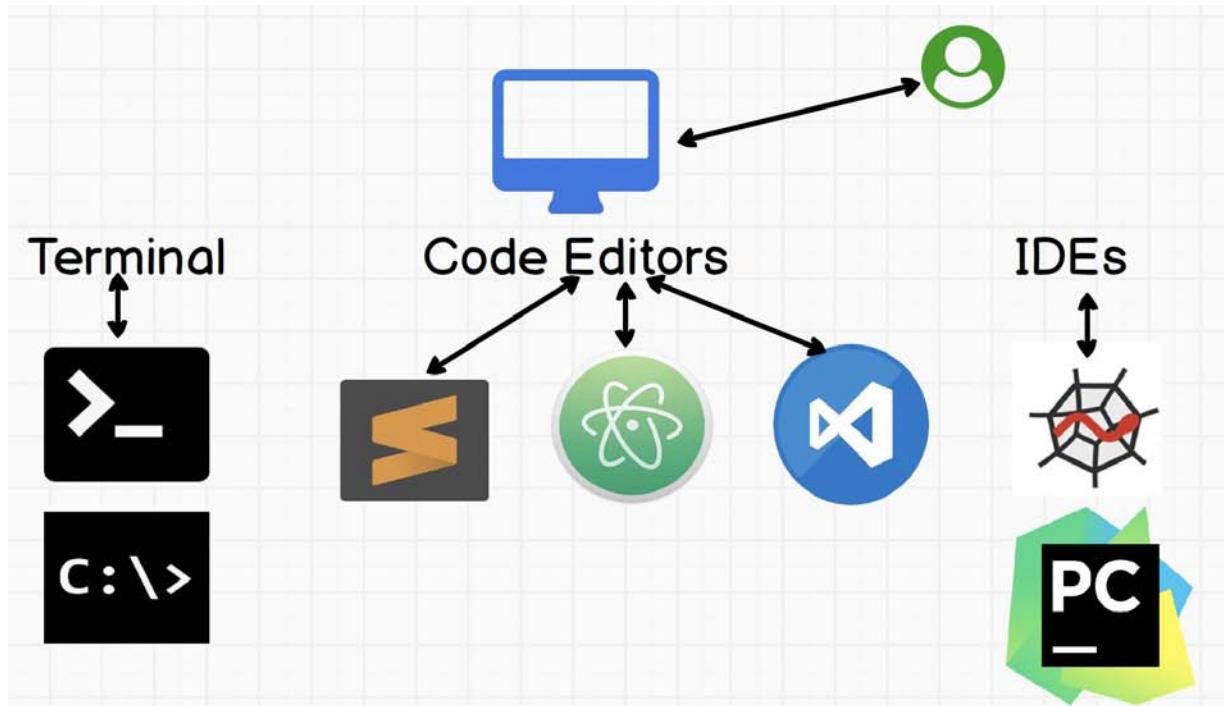
## **Developer Tools**

So far, we executed our code on the website called repl.it.com but we want to be able to use some tools to run and write Python code. In the previous section, we saw that we can use the Command Prompt on Windows and the Terminal on Mac to run Python code. In the Terminal or Command Prompt, we run simple scripts that we want to test quickly but developers will use either Code Editors or IDEs to write and execute Python code. You can write the code in any text/code editor or even in a word document but we don't have the same feeling that we are writing code, it is like we are writing a piece of text and we can easily make mistakes because of the spelling and indentation. We know that proper indentation is very important in Python so no to the word documents. Therefore, a professional developer will need some tools like code editors:

- Sublime
- Atom
- VS Code
- Vim

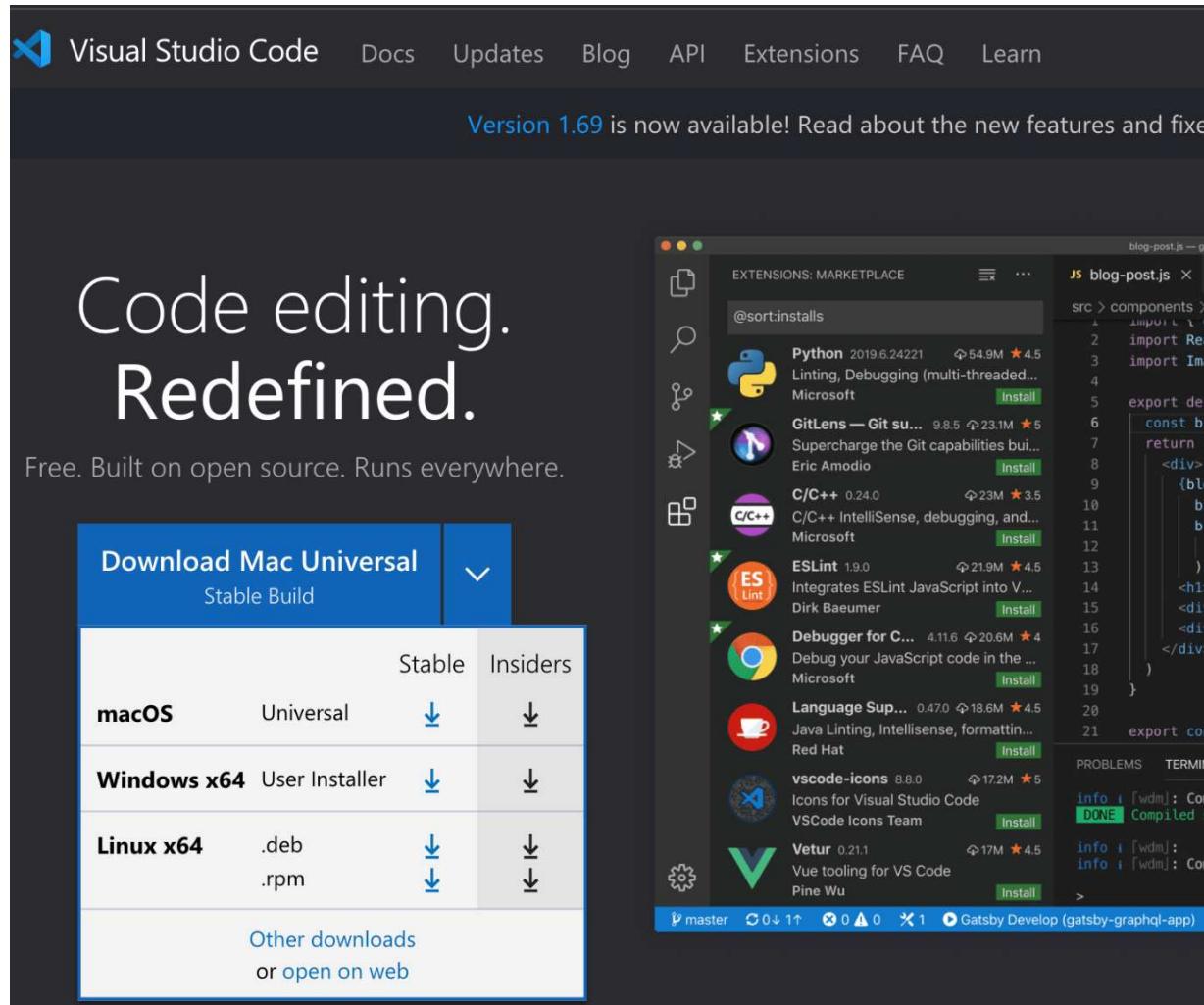
We can even use IDEs like:

- Spyder
- PyCharm
- Thonny



What is the difference between Code Editors and IDEs? Code editors are lightweight and allow us to enhance and ease the process of writing the code. The code editor is used by many programmers because they have built-in features that can speed up the process of writing the code. For example, they use **Emmet** to autocomplete the code and give you suggestions during your code writing and also has **linting** that is used to automatically check the source code for programmatic and stylistic errors. An IDE, on the other hand, is a more complex set that will combine different features like code editors, text editors, debuggers, compilers, code testers, code formatting, and much more. All of these features are under one hood. This is why when we download the IDEs, the file will be much bigger than that of the editors. I will not install all of the tools I have mentioned in the figure above but I will guide you on how to install and use the most important ones. The VS Code (Visual Studio Code) editor can be downloaded from this website:

<https://code.visualstudio.com/>



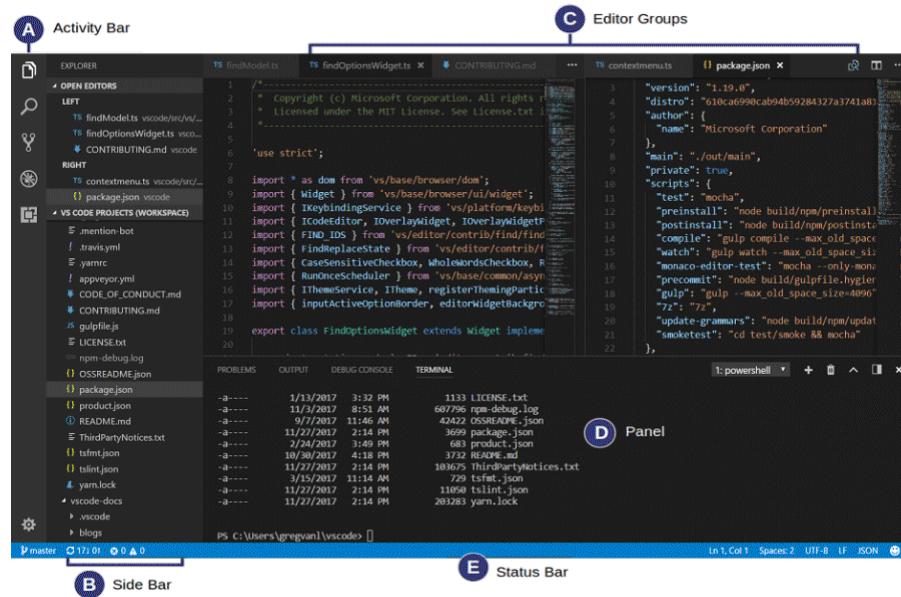
As you can see from the figure above, you need to get the right version for your Operating System. It will be automatically pre-selected for you, and in my case, it is the one that says macOS – Universal. If you are using Windows or Linux, you need to choose the right one for your system. There are also other versions of VS Code if you click on the link saying ‘Other downloads.’ They have an amazing get-started page that will help you if you have any issues. Do not get confused and scared of the screenshots on their website because they have included a few code snippets that you probably don’t understand at this stage.

<https://code.visualstudio.com/docs>

This editor is my favorite and I have used it for all of my projects so far. The code I have written for my HTML and JavaScript books was written in this editor. This means we can use code editors like VS Code not just for one specific language like Python but for all other languages as well. You can click on the download link and install it on your machine because it is very easy and straightforward.

## Explore the Visual Studio Code

I hope you have already installed VS Code (Visual Studio Code Editor). Like many other code editors, the VS Code has a simple but user-friendly interface.

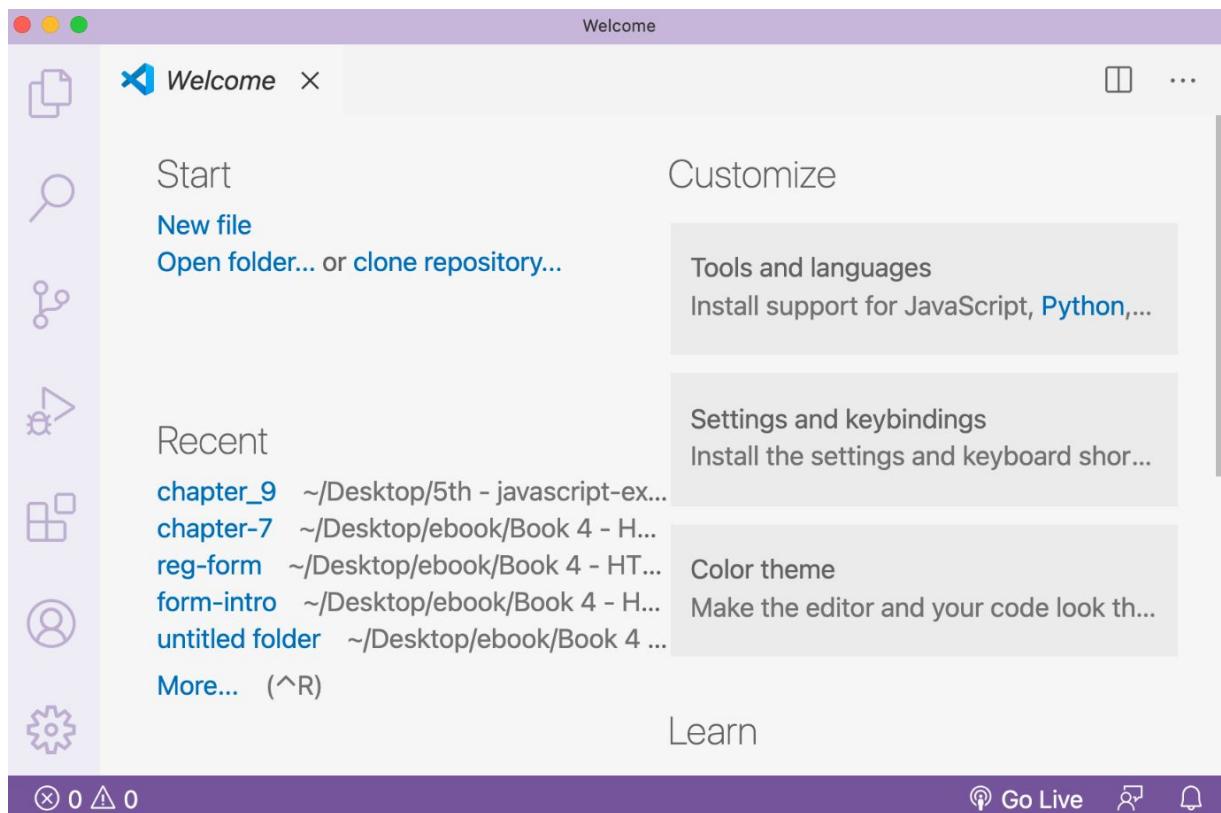


Please don't be afraid of the code, files, and folders in the figure above because it is only for demonstration purposes. As you can see, we have a very basic layout with an explorer on the left side which shows all the files and folders we have opened, and an editor on the right side that shows the content of the files we have opened. The **GUI** or graphical user interface is divided into five different areas (note – all these areas will not be visible in our initial lunch):

- **Editor** – This is the main area where we can edit our files. We can open as many editors as we want and they will be displayed side by side
- **Side Bar** – The side bar contains different views like **Explorer**, which can assist us during the coding
- **Status Bar** – The status bar gives the information on what files are currently being edited
- **Activity Bar** – This area is located on the far left-hand side and it allows us to switch between different views. It also shows the number of changes when **Git** is enabled
- **Panels** – below the editor is where we have the output or debug information – all the errors and warnings. We can even use the integrated terminal if we want.

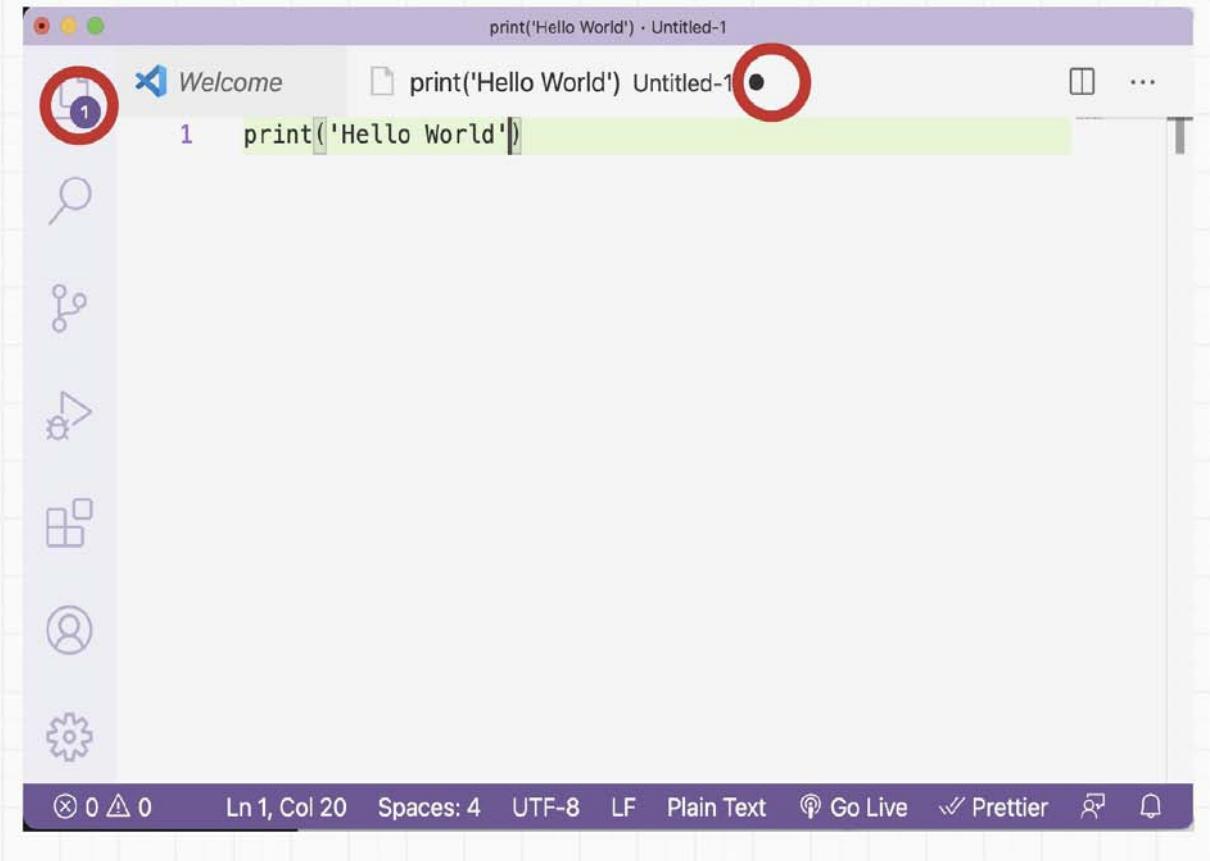
## How To Use The Visual Studio Code

Navigate to your VS Code and open it. The first time you open it, it might look different from mine but it is okay (because I have installed some extensions which I will show you at the end how you can install them too):

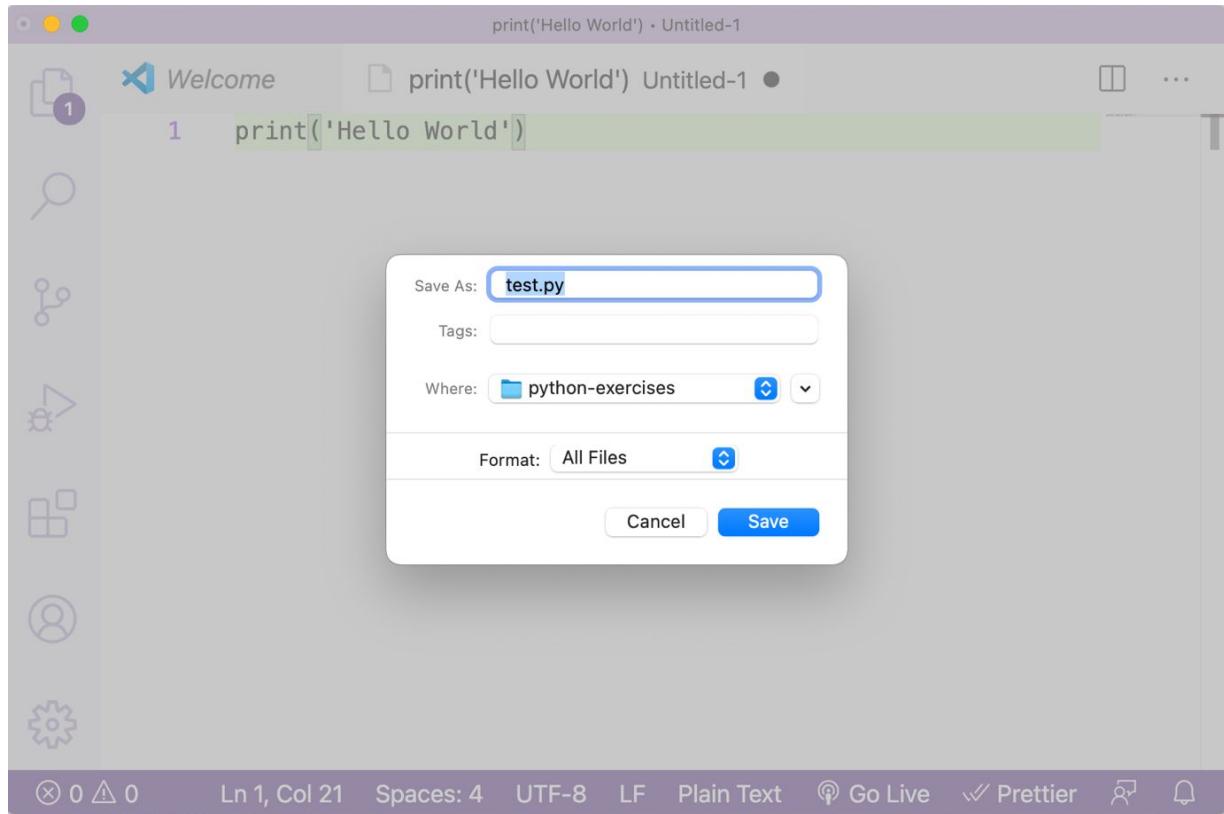


From the figure above, you can see that you can create a new file, open up a folder, etc. Select the new file on the top and it will give you access to the code editor:

The dot's on each side means  
that we have unsaved changed



From the figure above, you can see that I have written the `print ('Hello World')` in the first line but immediately after you start typing the code, you will notice that two dots or circles will appear on each side. This means we have written a code and we haven't saved the changes. We can fix this if we save the file by pressing command + S on Mac or Ctrl +S on Windows. We can even use the menu and go to file > Save as:



As you can see from the figure above, we are saving the file with the .py extension and this means we are working with Python files. Also, it is best practice to create a folder on your desktop or anywhere you think your code files should go and save the files there. You can name your file as you want, but make sure you use the .py file extension and click save and the dots will disappear. Make sure you save the changes after you write your code. After you have saved your file, you will have some suggestions saying that no Python interpreter is selected, and if you want to use Python, you will need to install the recommended extensions:

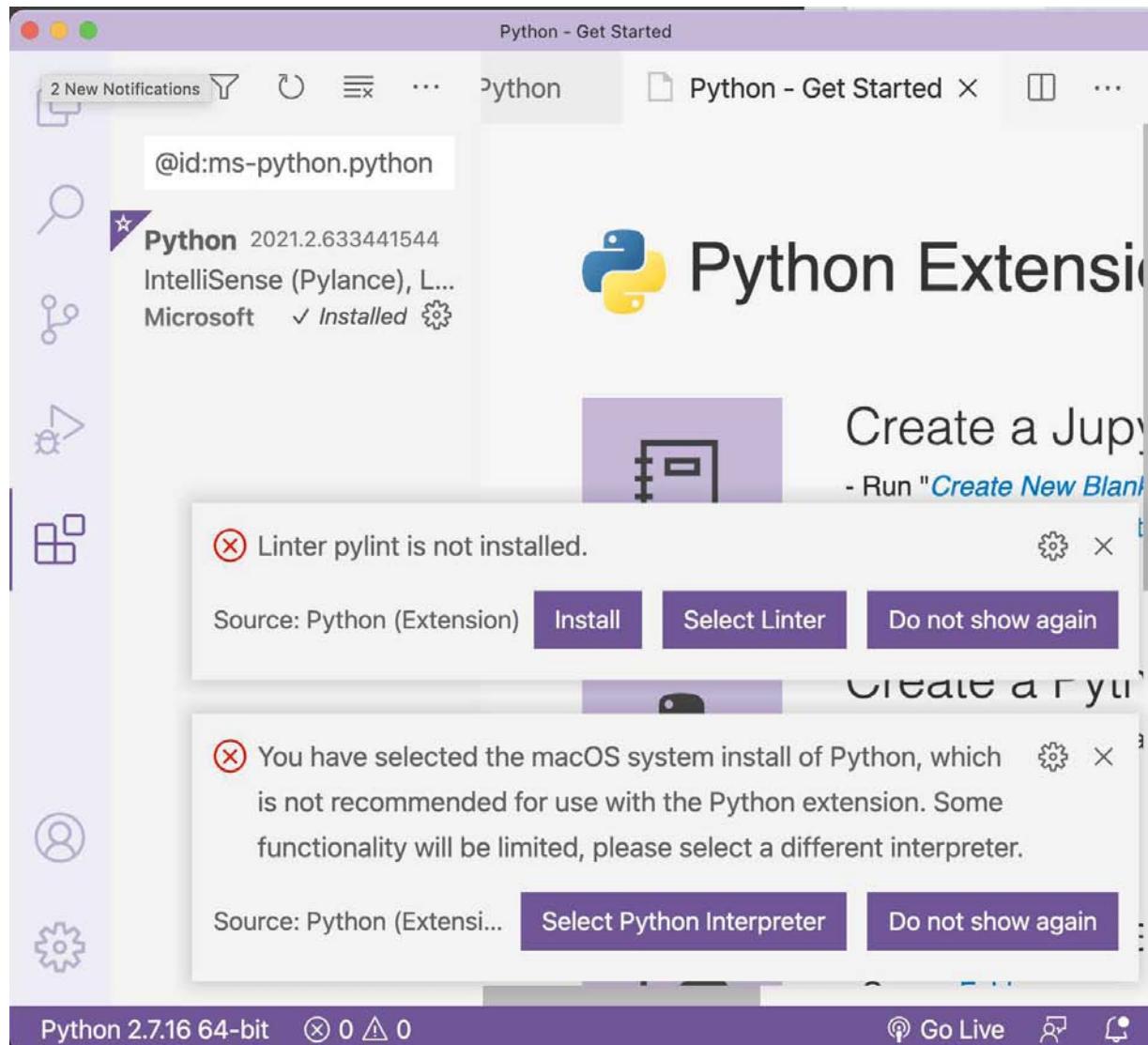
A screenshot of the Visual Studio Code (VS Code) interface. The title bar shows "test.py". The left sidebar has icons for File, Welcome, Search, Share, Open, and Settings. The main editor window displays a single line of code: "print('Hello World')". The status bar at the bottom shows "Ln 1, Col 21" and "Spaces: 4". A modal dialog box is open in the center, asking "Do you want to install the recommended extensions for Python?", with "Install" and "Show Recommendations" buttons. The bottom right corner of the screen has a small icon.

*Note: in the future, the messages might change slightly but please continue reading because the actions you need to perform will be similar if not the same.*

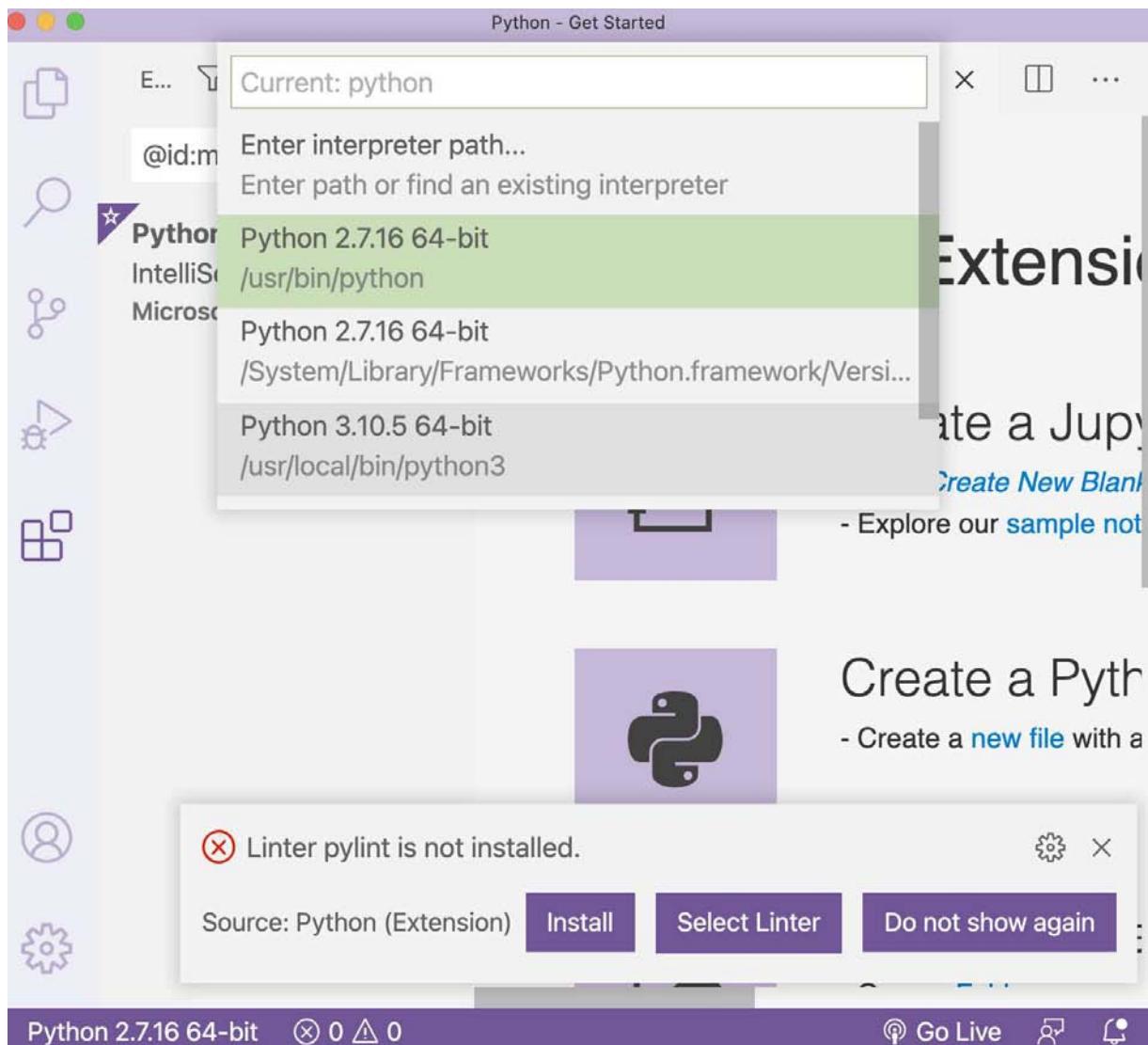
If you see a message like in the figure above, click on the install button and start installing Python.



As you can see in the figure above, Python is installed. After this, I got the following messages telling me about the next steps I should take:



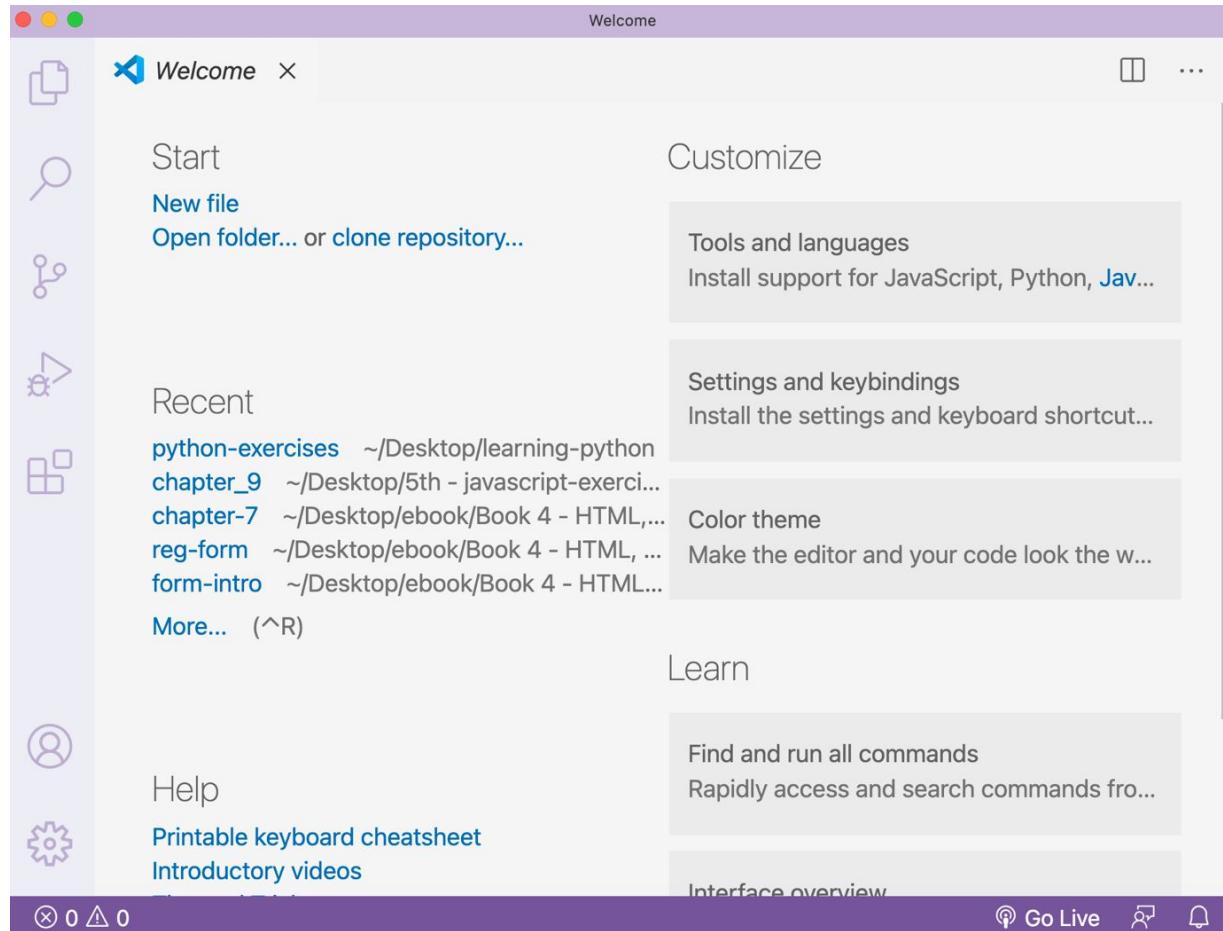
First, we need to click on the 'Select Python Interpreter' button and we need to select the Python version we installed before:



As you can see, my Mac had listed the previously installed versions of Python and this is good because it gives me an option to select the latest one. If you remember, the version I installed was 3.10.5 therefore I need to select that one, but your version might be different so please make sure you select the Python version you downloaded and installed. If you don't remember the version number you can head back to the official website and check the latest version in the downloads section:

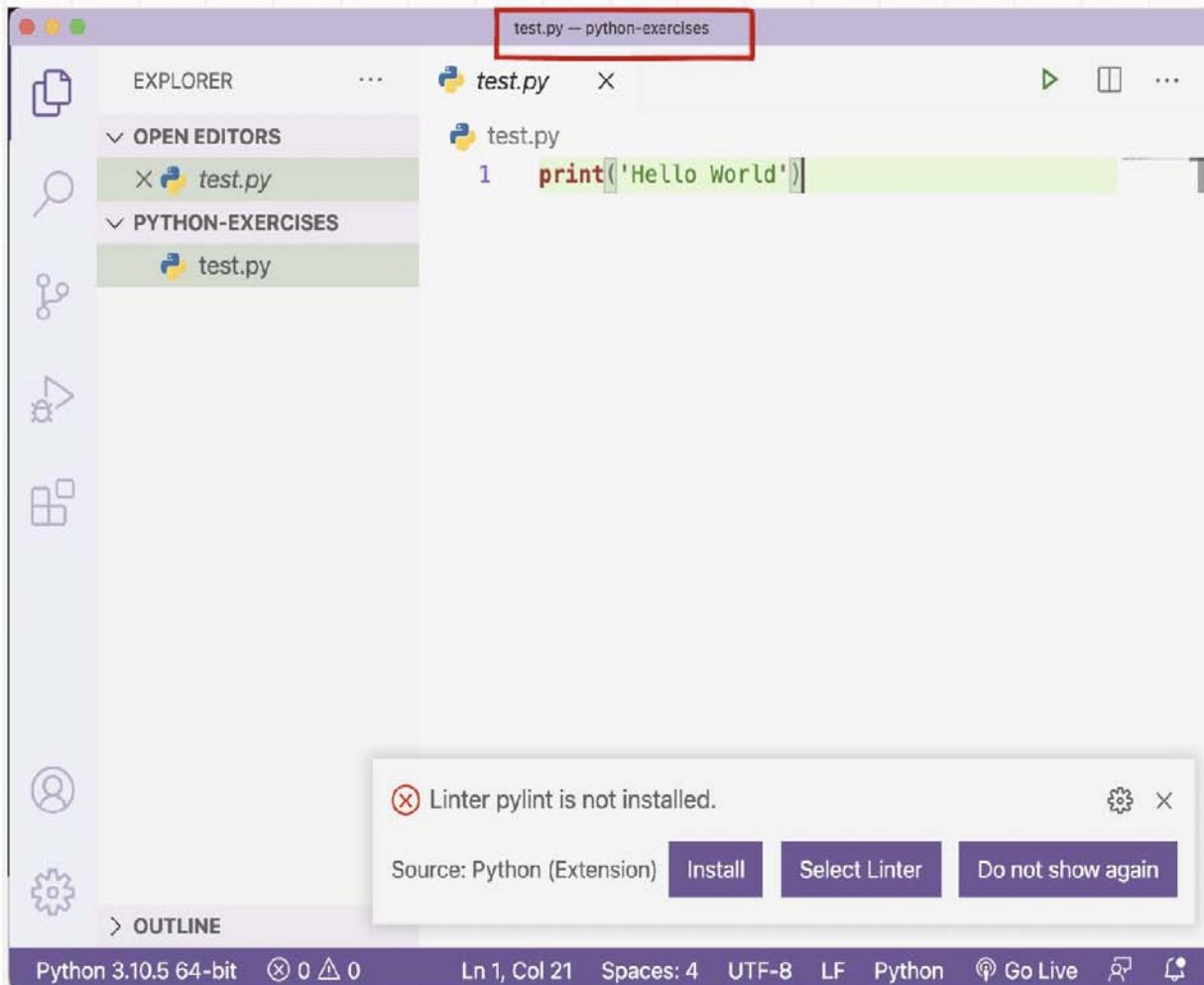
<https://www.python.org/>

Select the right version and that is it for now. Do not worry about installing the Linter pylint for now because you have selected the Python interpreter and we can start writing code, so you can close the message by clicking on the ‘x’. It is best to close the VS Code completely and open it again. Try to open the file you have created by selecting the folder where it was saved. So you need to click on the Open folder blue link:

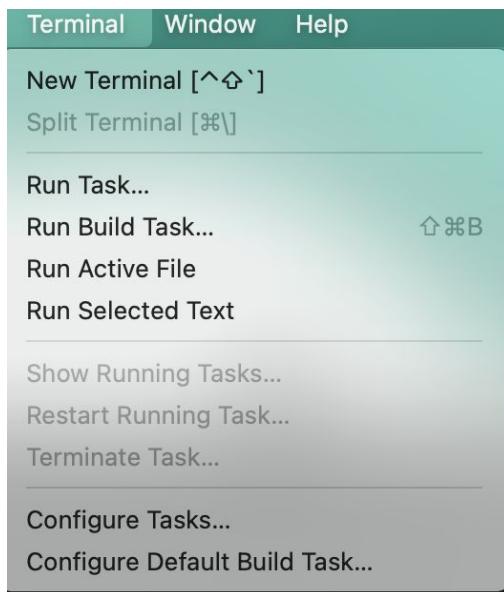


Then you need to select the right folder and this is what you will get:

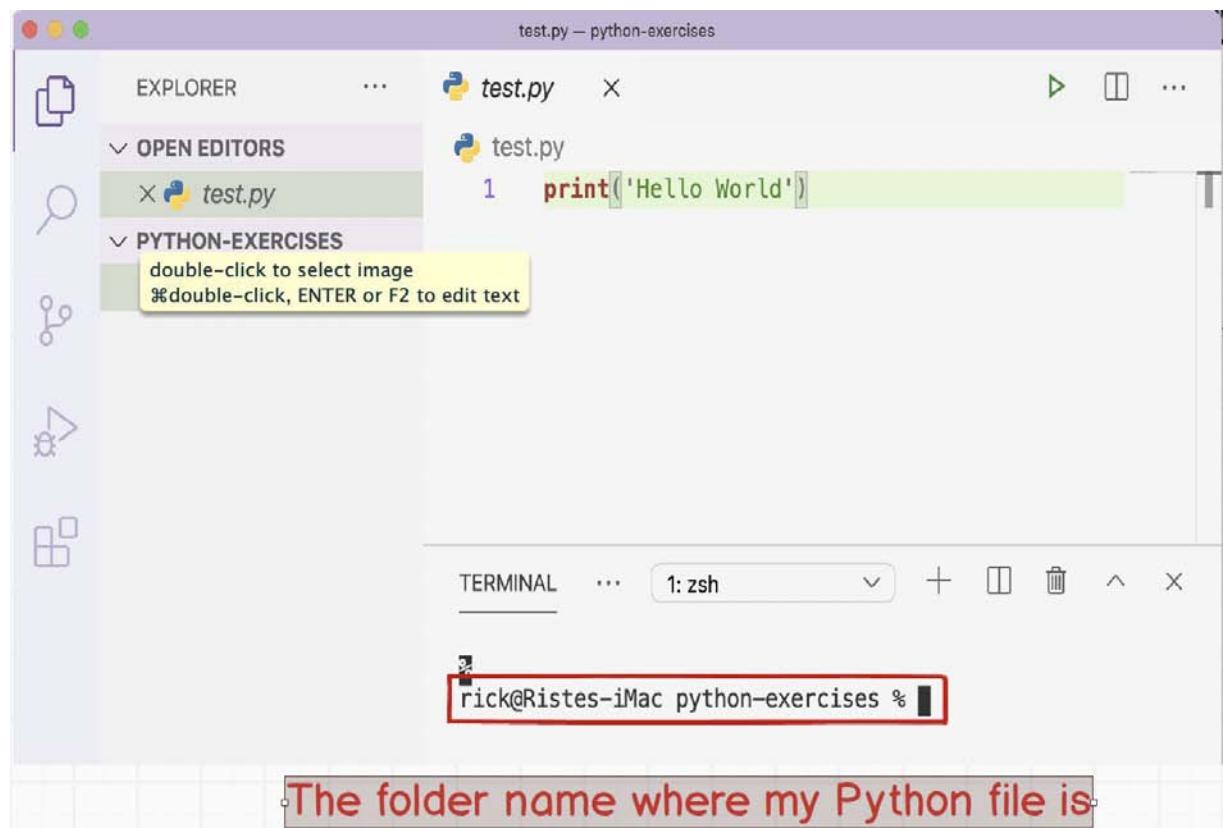
This tells you the name of the file, and location



As you can see, in my case, the file name is called ‘test.py’ and is located in the folder called python-exercises. Your file folder name can be different if you want. Let’s talk about the VS Code editor and what it can do apart from writing code. The VS Code like many other code editors comes with the terminal and you can open the terminal by going into the menu and selecting the terminal:



This will open the terminal:



So what can we do in this terminal? Well, we can run the above file, so make sure you saved everything before you write in the terminal: **python test.py**

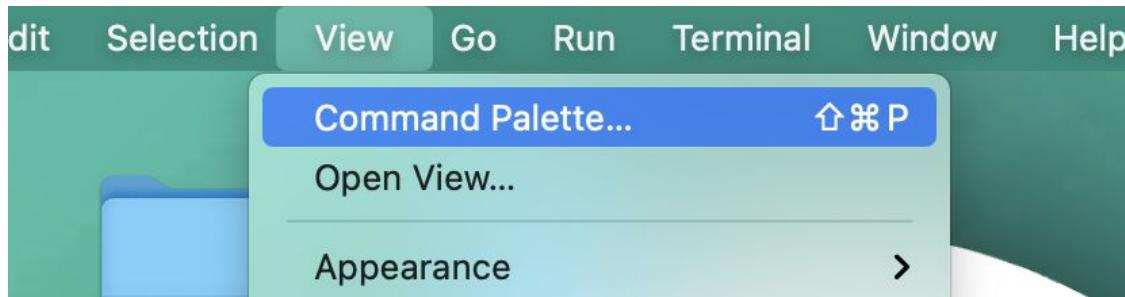
The screenshot shows the Visual Studio Code interface. The title bar says "test.py — python-exercises". The left sidebar has icons for Explorer, Search, Share, and Outline. The Explorer section shows "OPEN EDITORS" with "test.py" selected, and "PYTHON-EXERCISES" with another "test.py". The main area has a Python icon next to "test.py" and the code "print('Hello World')". Below it is a terminal window titled "TERMINAL" showing the command "python test.py" and the output "Hello World". The status bar at the bottom shows "Python 3.10.5 64-bit", "Spaces: 4", "UTF-8", "LF", "Python", "Go Live" button, and a few other icons.

As you can see, I got Hello World as an output. The file name in my case is **test.py** but your file name can be **first.py** for example. In this case, you need to write **python first.py** to get the output. Let's now install Linter, the recommendation we had before so if you don't see it on your screen you can close the VS Code editor and open it again. When you open your folder/file back, a message will pop up. What is linter? Linter will help you with the syntactical and stylistic problems you might have in your Python code. This will help you a lot when you have spelling mistakes and will let you know if the Python syntax is correct. Pylint is one of the more popular versions that do this. There are many other packages available on the market but VS Code is suggesting this one for our Python code. So let's just click on the install button and install it:

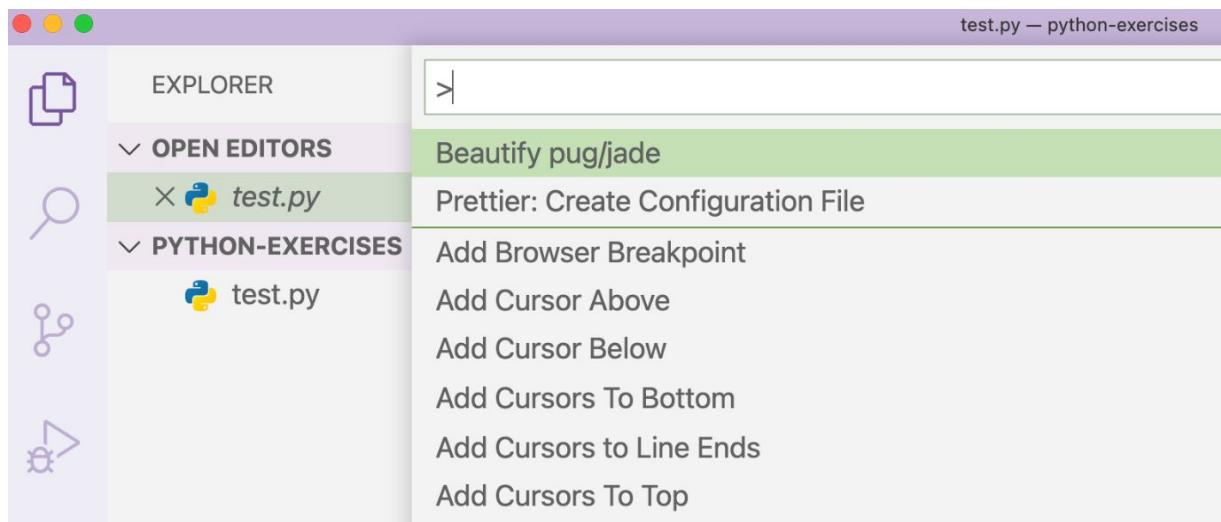
```
TERMINAL ... 2: Python + ⌂ ⌁ ⌂ ⌃ ⌄
and similar are installed in '/Users/rick/Library/Python/3.10/bin'
' which is not on PATH.
Consider adding this directory to PATH or, if you prefer to sup-
press this warning, use --no-warn-script-location.
Successfully installed astroid-2.11.7 dill-0.3.5.1 isort-5.10.1 l-
azy-object-proxy-1.7.1 mccabe-0.7.0 platformdirs-2.5.2 pylint-2.1
4.4 tomli-2.0.1 tomkit-0.11.1 wrapt-1.14.1
WARNING: You are using pip version 22.0.4; however, version 22.1.
2 is available.
You should consider upgrading via the '/usr/local/bin/python3 -m
pip install --upgrade pip' command.
rick@Ristes-iMac python-exercises %
```

Ln 1, Col 21 Spaces: 4 UTF-8 LF Python ⌂ Go Live ⌂ ⌂

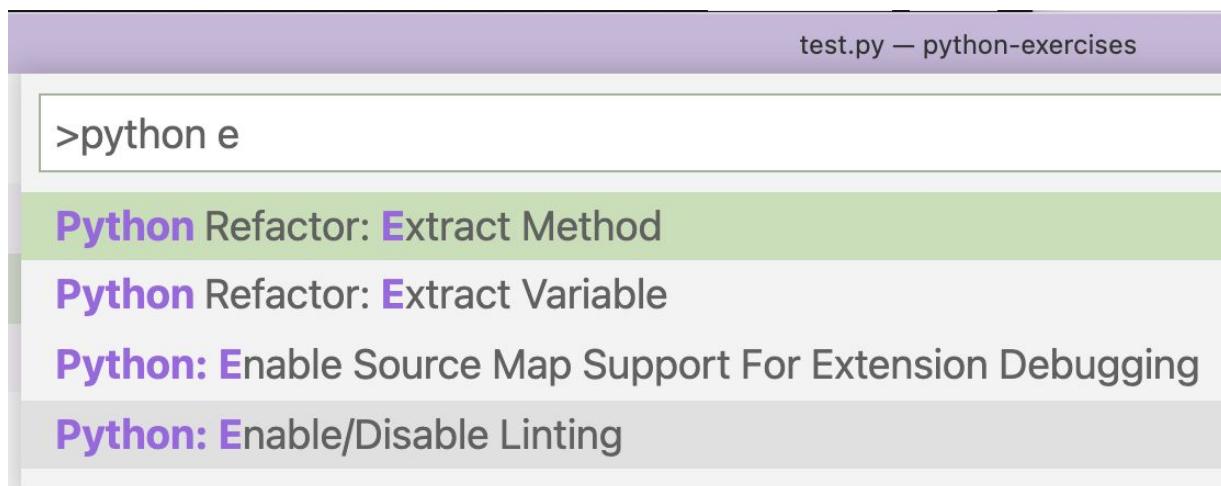
Once the installation is completed, you should have a similar message to the one in the figure above. Important: there shouldn't be any red errors message in your terminal. So let's jump back to the code and try to make a mistake on purpose. Before we make mistakes intentionally, let's turn on linting by going into the menu view>Command Palette:



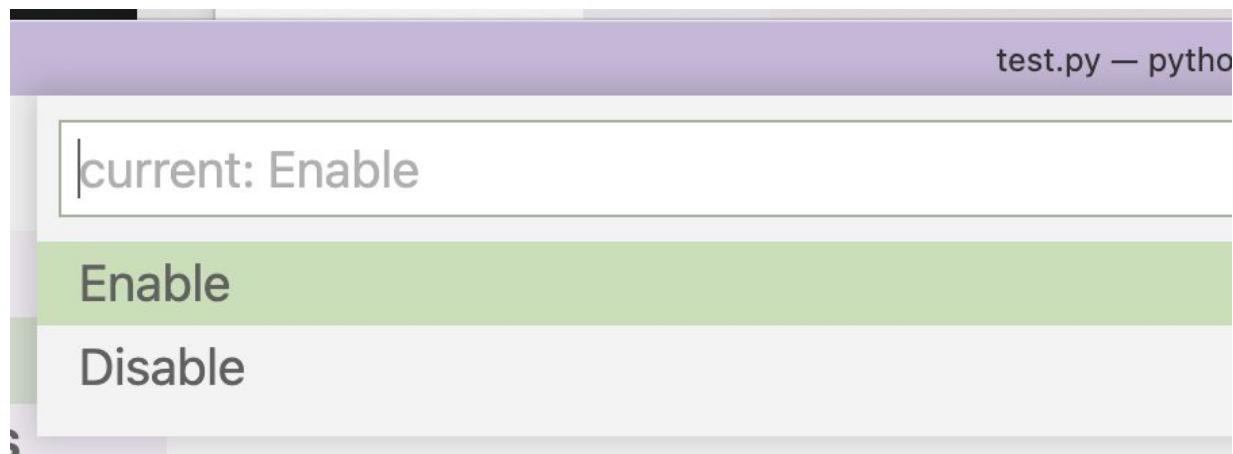
And this should let you type something in your VS Code like in the figure below:



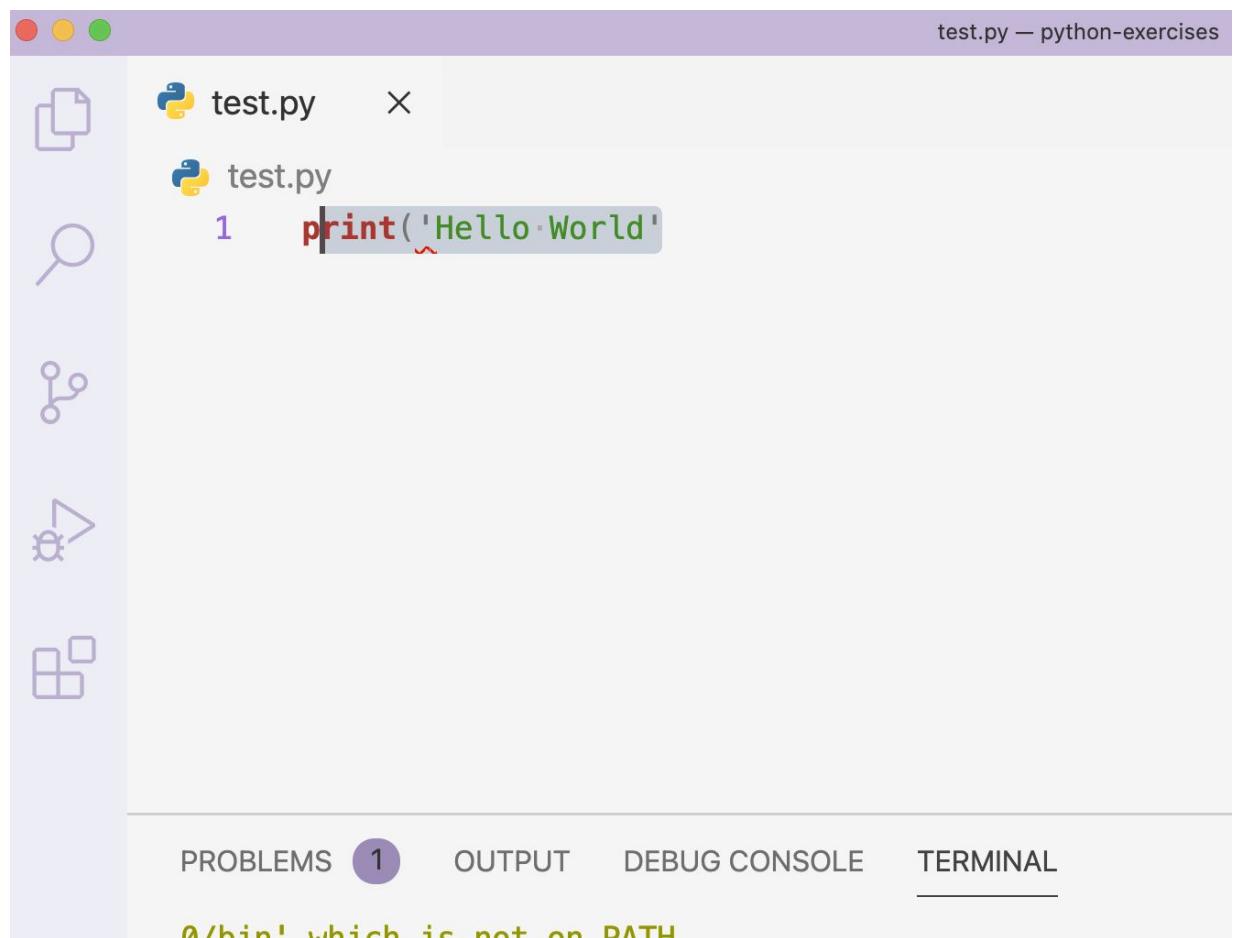
Now when we install the Python Interpreter, we installed a lot of other Python tools in the background that we can now use, so if you start typing **Python: Enable** you will get the following list of options:



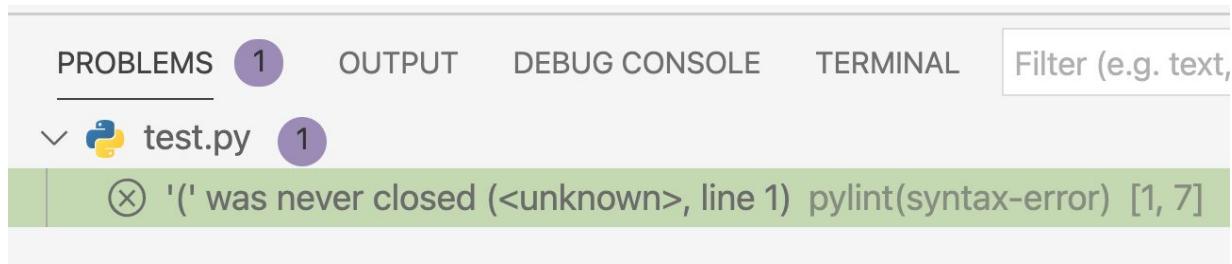
From the figure above, select the **Python: Enable/Disable Linting** option, click on it, and select Enable or On:



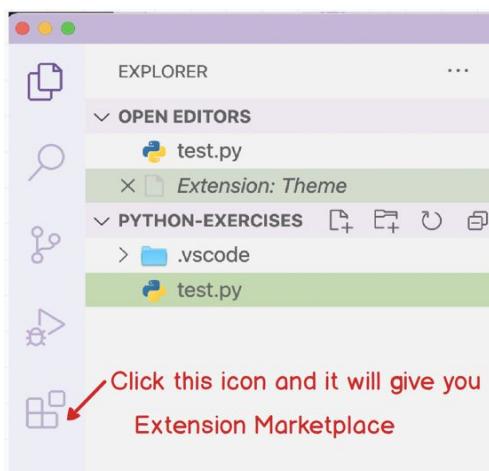
After enabling linting, let's remove the closing bracket of the print function so we are making a mistake intentionally and finally save the test.py file. This will immediately create a problem for us because linting will detect that a bracket is missing and in the PROBLEMS tab/panel we can see a number 1:



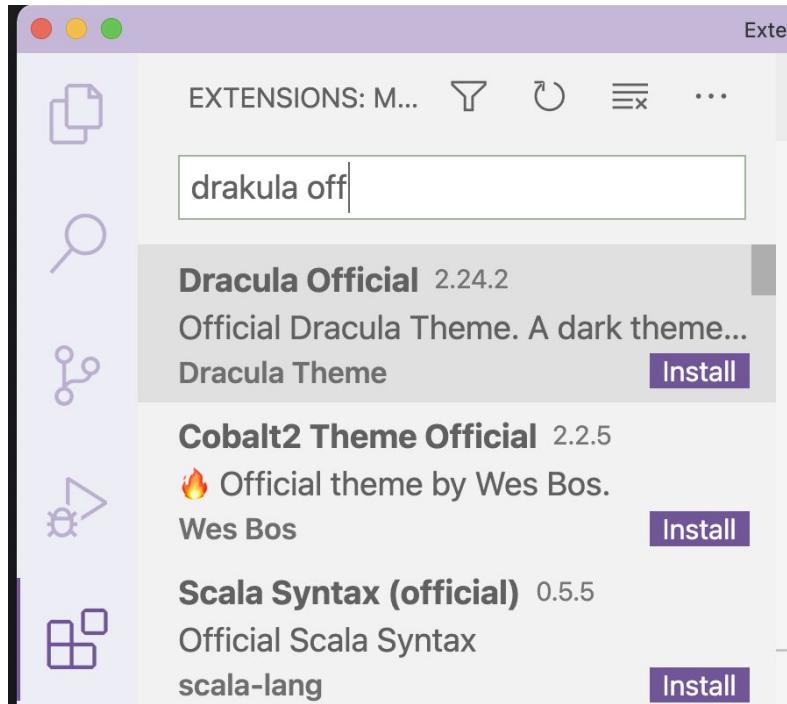
Click on the PROBLEMS tab so you can read the error:



As we can see, this is a syntax error that can be fixed easily because it even shows us the line number where the error occurred. And these are all of the steps that you need to take before you can start coding. Finally, let me show you how you can install extensions in VS Code:



From the figure above, you need to locate the marketplace icon and click on it. From there I would suggest you install some VS Code icon packs that will make your folder and files look better. You can install dark themes like **drakula official** by typing it in the search bar and then clicking on the install button next to it.



Now your VS Code is ready for writing Python code.

## Useful Terminal/Command Prompt Commands

As a future Python developer, you should know how to use the Terminal or Command Prompt. In this section, we will go through some very useful commands. On Mac, you open the terminal by pressing command + space and then you type terminal and press return/enter. On Windows, you need to look for the start Windows icon and search for Command Prompt. The terminals/command prompts are used today but they were used a lot in the past because they run the code. After the GUI (Graphical User Interface), the usage of the terminals decreased. As a developer, you will use the terminal to download and run scripts, to communicate with servers and databases. The commands for the terminal and command prompt are not the same but I will provide screenshots and tips so you can practice the commands on Windows, macOS, and Linux. For example, open your terminal or command prompt and type '**ls**' on Mac and '**dir**' on Windows. The **ls** – (**dir** for Windows) will list everything you will have in the current directory:

```
Last login: Thu Jul 14 16:49:11 on ttys000
rick@Ristes-iMac ~ % ls
Applications          Movies           Users
Desktop              Music            evalonly.txt
Documents             Pictures          pitchshiftLogFile.txt
Downloads             Postman
Library              Public
rick@Ristes-iMac ~ %
```

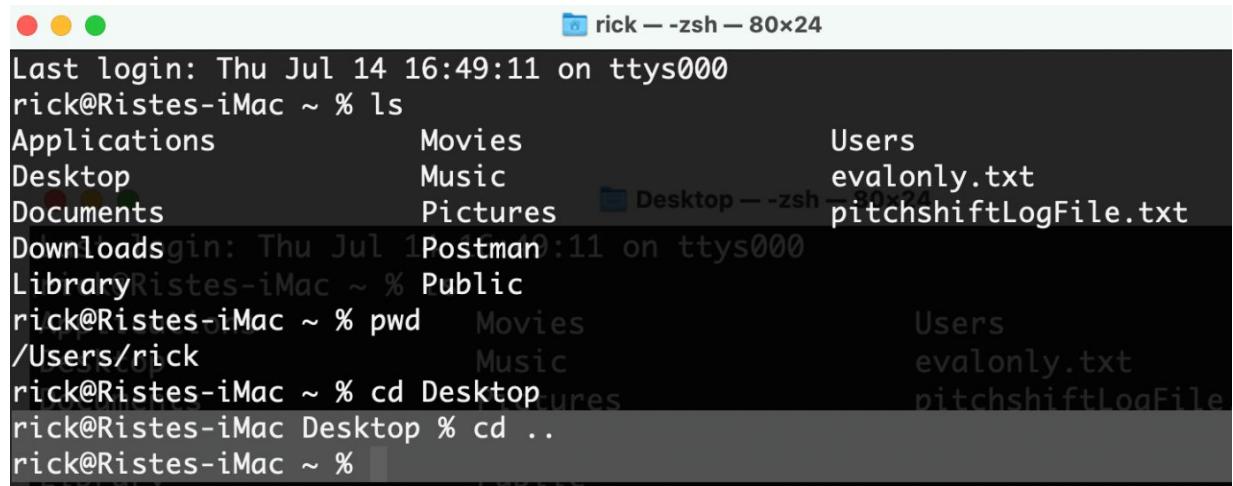
As you can see, everything I have in this directory is listed. How can we check which directory we are in at the moment? For Windows, we need to type ‘cd’ and for macOS, we need to type ‘pwd’:

```
Last login: Thu Jul 14 16:49:11 on ttys000
rick@Ristes-iMac ~ % ls
Applications          Movies           Users
Desktop              Music            evalonly.txt
Documents             Pictures          pitchshiftLogFile.txt
Downloads             Postman
Library              Public
rick@Ristes-iMac ~ % pwd
/Users/rick
rick@Ristes-iMac ~ %
```

As you can see from the figure above, it says I’m in the ‘Users/rick/’ directory. Everything that is listed above is something I have access to. So how can I navigate to my Desktop and see what’s there? Well, for both Mac and Windows, we can write ‘**cd Desktop.**’ The cd means change directory:

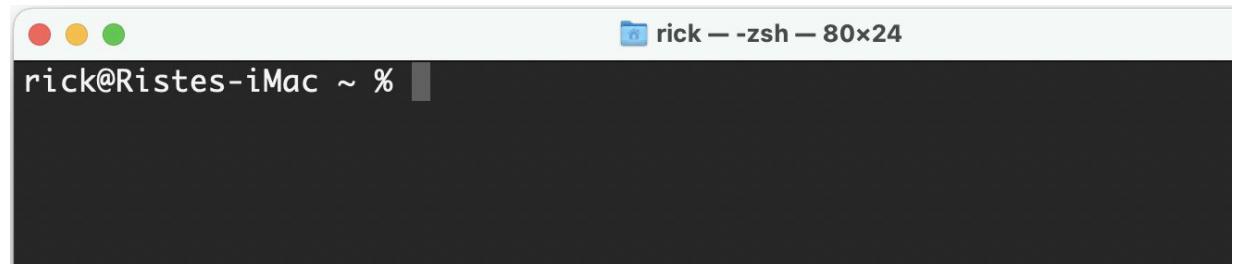
```
Last login: Thu Jul 14 16:49:11 on ttys000
rick@Ristes-iMac ~ % ls
Applications          Movies           Users
Desktop              Music            evalonly.txt
Documents             Pictures          pitchshiftLogFile.txt
Downloads             Postman
Library              Public
rick@Ristes-iMac ~ % pwd
/Users/rick
rick@Ristes-iMac ~ % cd Desktop
rick@Ristes-iMac Desktop %
```

If for some reason you want to get out of the current directory, you can write ‘cd ..’ and this will go from the Desktop back to /Users/rick:



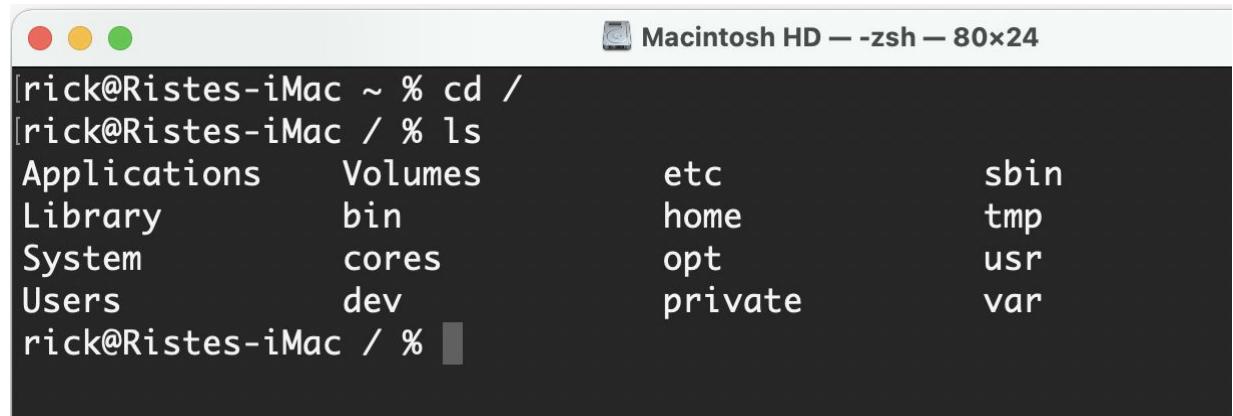
```
Last login: Thu Jul 14 16:49:11 on ttys000
rick@Ristes-iMac ~ % ls
Applications      Movies          Users
Desktop          Music           evalonly.txt
Documents         Pictures        pitchshiftLogFile.txt
Downloads        Postman        rick@Ristes-iMac ~ %
Library          Public          rick@Ristes-iMac ~ %
rick@Ristes-iMac ~ % pwd
/Users/rick
rick@Ristes-iMac ~ % cd Desktop
rick@Ristes-iMac Desktop % cd ..
rick@Ristes-iMac ~ %
```

The terminal now has a lot of text and looks messy. You can use the ‘**clear**’ command on Mac and ‘**cls**’ on Windows to clean the terminal window:



```
rick@Ristes-iMac ~ %
```

If for some reason you want to navigate to the root directory you can use this command ‘cd /’, and don’t forget the forward slash. If you want to list everything, use ‘ls’ or ‘dir’ for windows:



```
[rick@Ristes-iMac ~ % cd /
[rick@Ristes-iMac / % ls
Applications    Volumes        etc          sbin
Library         bin            home         tmp
System          cores          opt           usr
Users           dev            private      var
rick@Ristes-iMac / %
```

If you want to get back to the user directory, you can use the following command: cd ~

```
rick@Ristes-iMac ~ % cd /
rick@Ristes-iMac / % ls
Applications      Volumes          etc          sbin
Library           bin              home         tmp
System            cores            opt          usr
Users             dev              private      var
rick@Ristes-iMac / % cd ~
rick@Ristes-iMac ~ % ls
Applications      Movies           Users
Desktop           Music            evalonly.txt
Documents          Pictures          pitchshiftLogFile.txt
Downloads          Postman
Library            Public
rick@Ristes-iMac ~ %
```

I hope you are enjoying this section so far, but we are not done yet. Let's create a folder with the name 'p1' somewhere on the desktop. We are not going to use the terminal for this so go to your desktop and create the folder p1 where we can create subfolders later on. I need to navigate to the desktop and I'm in the Users directory at this stage 'cd Desktop':

```
rick@Ristes-iMac ~ % ls
Applications      Movies           Users
Desktop           Music            evalonly.txt
Documents          Pictures          pitchshiftLogFile.txt
Downloads          Postman
Library            Public
rick@Ristes-iMac ~ % cd Desktop
rick@Ristes-iMac Desktop %
```

If you type 'ls' or 'dir' you should be able to see what we have on the desktop:

```
[rick@Ristes-iMac ~ % ls
Applications          Movies           Users
Desktop              Music            evalonly.txt
Documents             Pictures          pitchshiftLogFile.txt
Downloads             Postman
Library               Public
[rick@Ristes-iMac ~ % cd Desktop
[rick@Ristes-iMac Desktop % ls
ebook                 p1
everything
rick@Ristes-iMac Desktop %
```

As we can see from the figure above, I have the folder (p1) that I have created. Now if I want to create other folders inside this folder, I need to use the cd command again. So let us type ‘cd p1’ :

```
[rick@Ristes-iMac ~ % ls
Applications          Movies           p1      Users
Desktop              Music            evalonly.txt
Documents             Pictures          pitchshiftLogFile.txt
Downloads             Postman
Library               Public
[rick@Ristes-iMac ~ % cd Desktop
[rick@Ristes-iMac Desktop % ls
ebook                 p1
everything
rick@Ristes-iMac Desktop % cd p1
rick@Ristes-iMac p1 %
```

You can make sure nothing is inside if you use the ls or ‘dir’ command. If you want to create another folder inside the p1, you can use the command called ‘mkdir’ which stands for make directory:

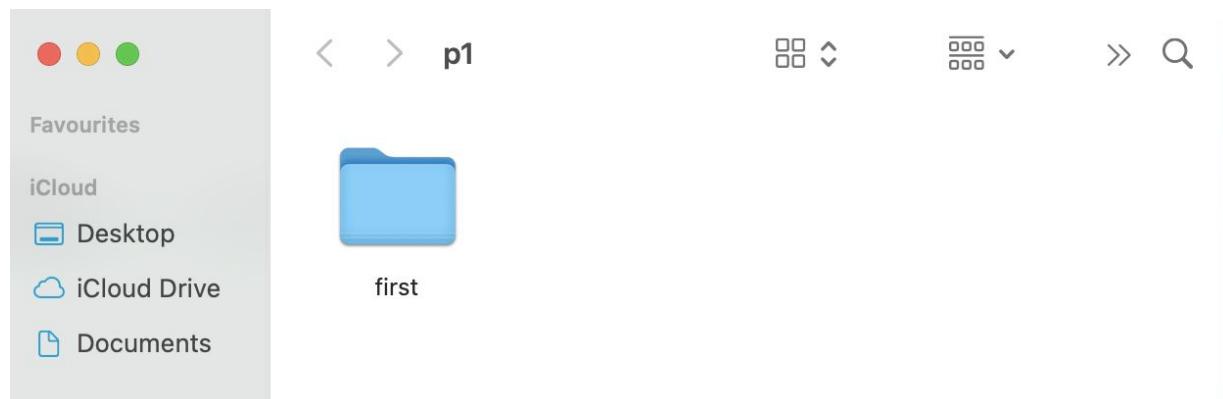
```
rick@Ristes-iMac ~ % ls
Applications           Movies          Users
Desktop               Music           evalo
Documents              Pictures         pitch
Downloads             Postman
Library               Public

rick@Ristes-iMac ~ % cd Desktop
rick@Ristes-iMac Desktop % ls
ebook@Ristes-iMac ~ ls      p1
everything            Movies          ~$rina-Low-Content-KD
rick@Ristes-iMac Desktop % cd p1
rick@Ristes-iMac p1 % ls
rick@Ristes-iMac p1 % mkdir first
rick@Ristes-iMac p1 %
```

Because we are inside the p1 folder, we can write this command ‘open .’ for Mac and ‘start .’ for Windows.

```
rick@Ristes-iMac Desktop % cd p1
rick@Ristes-iMac p1 % ls
rick@Ristes-iMac p1 % mkdir first
rick@Ristes-iMac p1 % open .
rick@Ristes-iMac p1 %
```

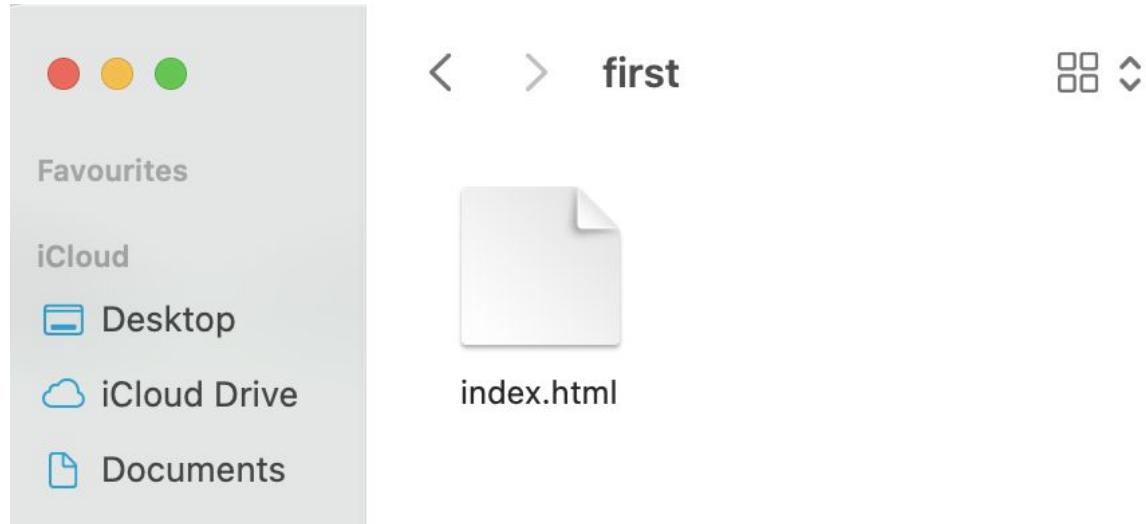
And this is what will happen:



As you can see, inside the p1 folder, we have a new folder named ‘first’ created using the mkdir command. The first folder is empty but let’s create a file inside, for example, an HTML file called **index.html**. First, we need to go inside the folder called ‘first’ so ‘cd first’ and then use the command ‘**touch index.htm**’ for Mac and ‘echo > index.html’ for Windows.

```
[rick@Ristes-iMac Desktop % cd p1
[rick@Ristes-iMac p1 % ls
[rick@Ristes-iMac p1 % mkdir first
[rick@Ristes-iMac p1 % open .
[rick@Ristes-iMac p1 % cd first
[rick@Ristes-iMac first % touch index.html
rick@Ristes-iMac first %
```

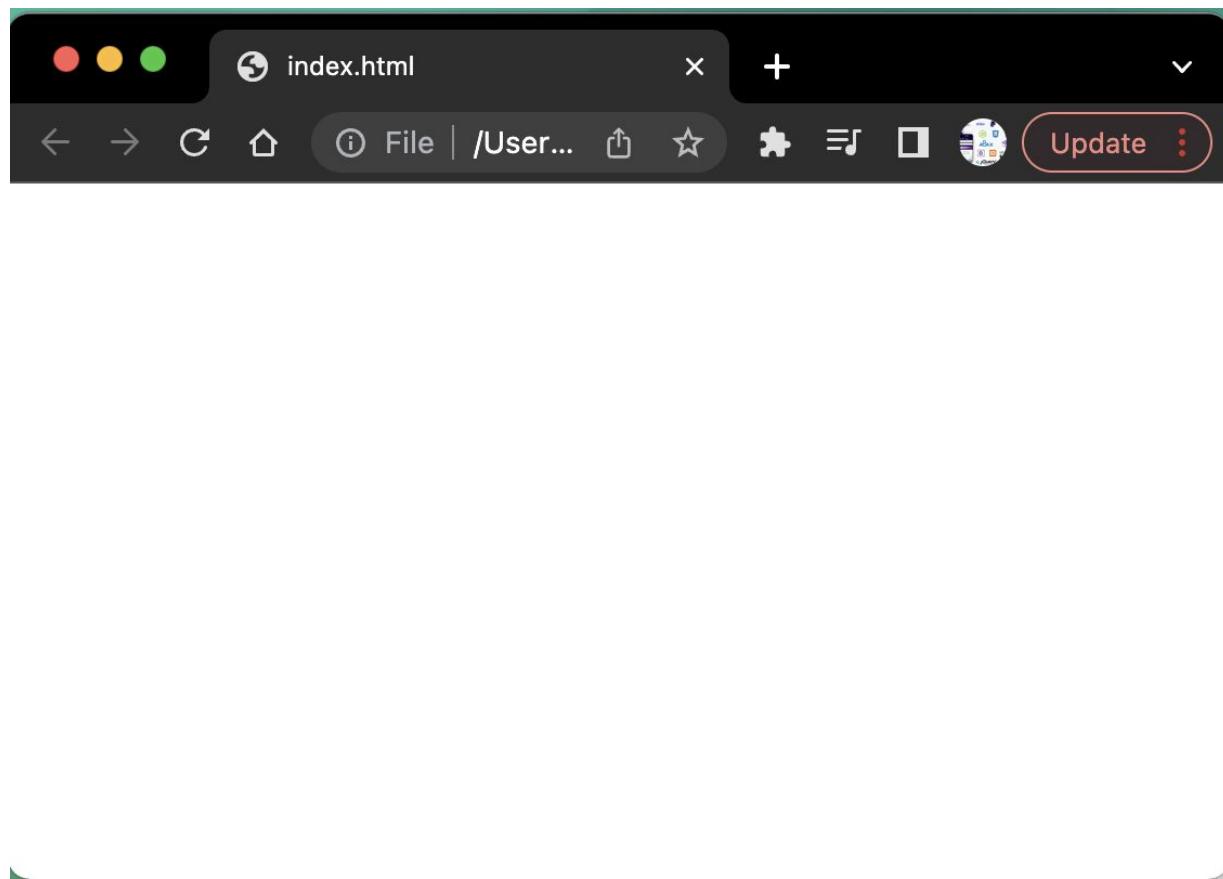
This is what you should have in the p1> first > folder:



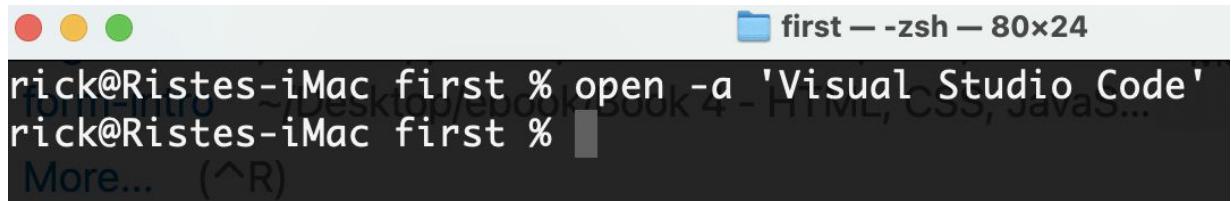
We can open the file (index.html) if we use the ‘open index.html’ or ‘start index.html’ command:

```
[rick@Ristes-iMac Desktop % cd p1
[rick@Ristes-iMac p1 % ls
[rick@Ristes-iMac p1 % mkdir first
[rick@Ristes-iMac p1 % open .
[rick@Ristes-iMac p1 % cd first
[rick@Ristes-iMac first % touch index.html
[rick@Ristes-iMac first % ls
index.html
[rick@Ristes-iMac first % open index.html
rick@Ristes-iMac first %
```

The last command will open the file in our browser:

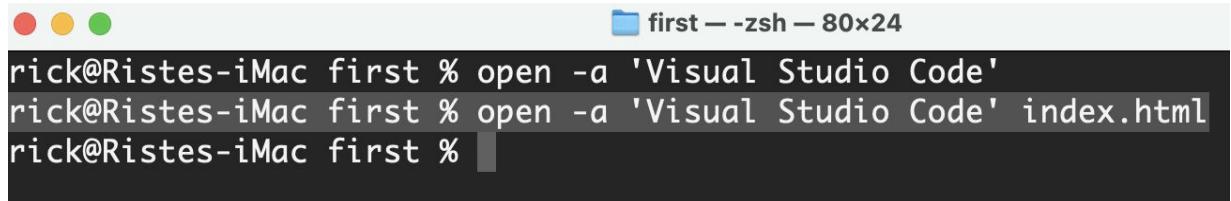


As you can see from the figure above, the HTML file is empty meaning there is no content. We can open the Visual Studio Code directly from the terminal like this:



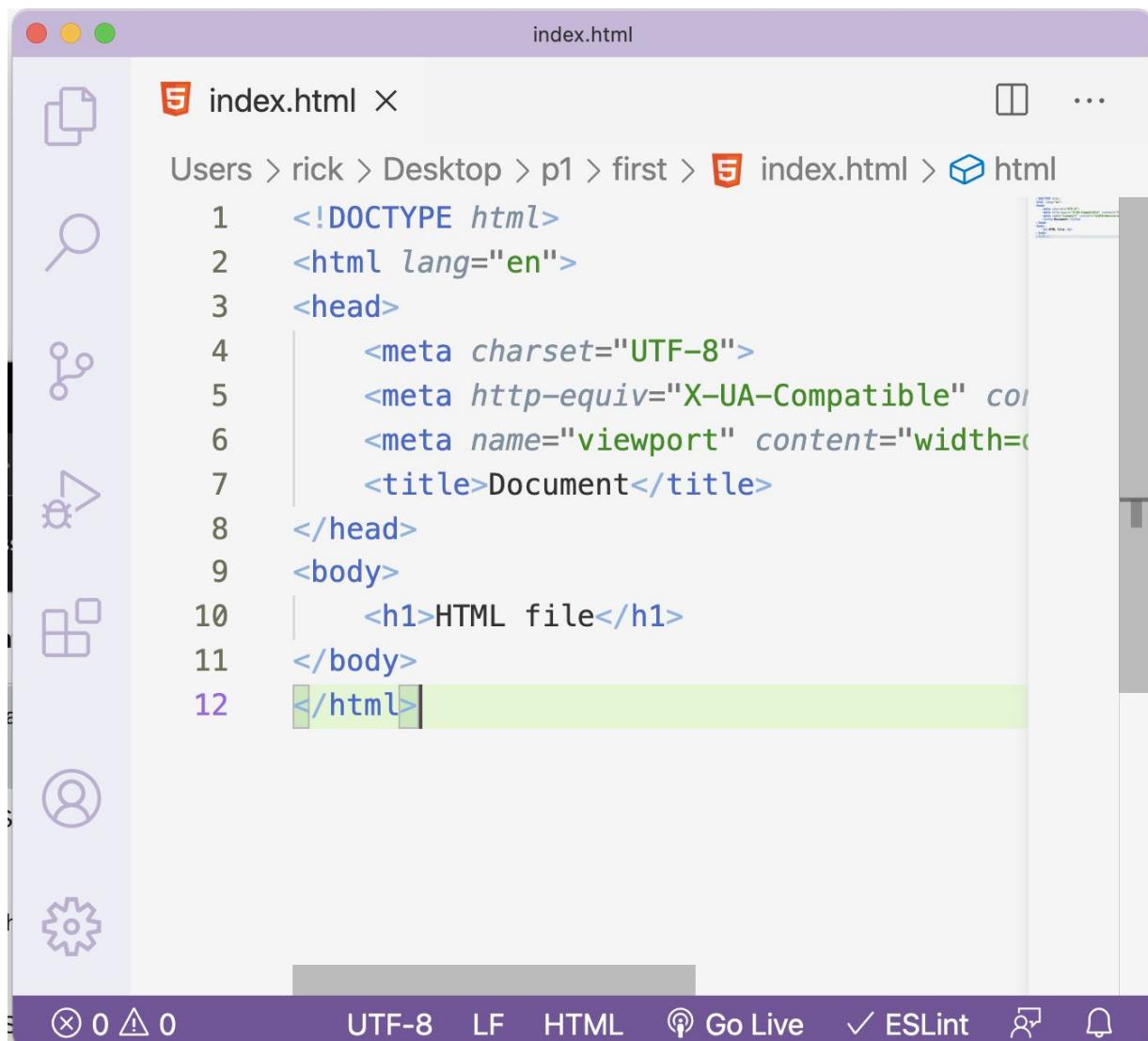
```
first — -zsh — 80x24
rick@Ristes-iMac first % open -a 'Visual Studio Code'
rick@Ristes-iMac first %
More... (^R)
```

This will launch the VS Code for us but if we want to open the index.html file directly, we can type the same command including the file name:



```
first — -zsh — 80x24
rick@Ristes-iMac first % open -a 'Visual Studio Code'
rick@Ristes-iMac first % open -a 'Visual Studio Code' index.html
rick@Ristes-iMac first %
```

This will open the index.html file in the VS Code so you can start writing basic HTML code:

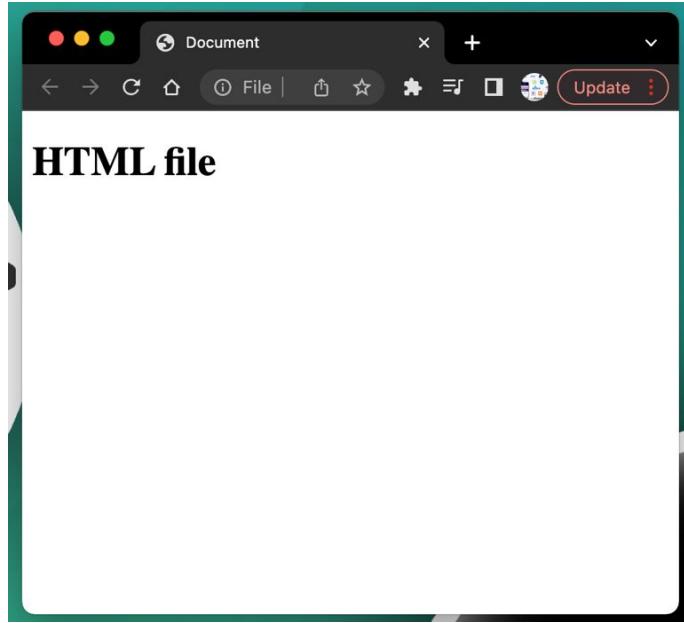


The screenshot shows a code editor window titled "index.html". The sidebar on the left contains icons for file operations like copy, paste, search, and navigation. The main pane displays the following HTML code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8 </head>
9 <body>
10  <h1>HTML file</h1>
11 </body>
12 </html>
```

The status bar at the bottom shows file statistics: 0 changes, 0 errors, and 0 warnings. It also indicates the encoding is UTF-8, line endings are LF, and the file type is HTML. There are also "Go Live" and ESLint status indicators.

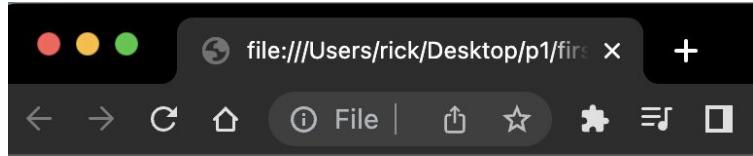
Let's open the index file again by typing 'open index.html' and see if the changes we have made are visible now (save the index.html file before you try to open it):



As you can see through the terminal, you can do the same things you are doing with your mouse and keyboard. If you want to change the file name, for example, from index.html to app.html, you can use the following command ‘mv index.html app.html’ for Mac users and for Windows ‘rename index.html app.html’:

```
rick@Ristes-iMac first % open -a 'Visual Studio Code'  
rick@Ristes-iMac first % open -a 'Visual Studio Code' index.html  
rick@Ristes-iMac first % open index.html  
rick@Ristes-iMac first % mv index.html app.html  
rick@Ristes-iMac first %
```

Now if we go to the web browser this will happen:



## Your file couldn't be accessed

It may have been moved, edited or deleted.

ERR\_FILE\_NOT\_FOUND

This happened because we renamed the index file. If you don't want to type the same commands over and over again, you can press the up arrow on your keyboard and it will start listing the previous commands for you. Let's say you have two files that start with the letter 'a':

app.html

about.html

If you want to access the about.html file and open it from the terminal, as soon as you start typing 'open ab' you can press the tab key and it will autocomplete and give you the correct file name because the terminal knows you are trying to access the about.html file. If you type 'open a' and press tab, it will not autocomplete the file name because it doesn't know which file you referring to because both files start with the same letter 'a'. Let's create another file called app1.html using the touch or 'echo >' command for Windows and ls command to make sure what files we have:

```
rick@Ristes-iMac first % open -a 'Visual Studio Code'
rick@Ristes-iMac first % open -a 'Visual Studio Code' index.html
rick@Ristes-iMac first % open index.html
rick@Ristes-iMac first % mv index.html app.html
rick@Ristes-iMac first % touch app1.html
rick@Ristes-iMac first % ls
app.html      app1.html
rick@Ristes-iMac first %
about.html
```

Let's delete the file we have just created using the '**rm app1.html**' command for Mac users and '**del app1.html**' for Windows:

```
rick@Ristes-iMac first % rm app1.html
rick@Ristes-iMac first % ls
app.html
rick@Ristes-iMac first %
Terminal
```

Let's delete the folder 'first' we created a while ago but to delete the entire folder together with the app.html file inside, we need to go back one level up into the p1 file using the command 'cd ..':

```
[rick@Ristes-iMac first % rm app1.html
[rick@Ristes-iMac first % ls
app.html
[rick@Ristes-iMac first % pwd
/Users/rick/Desktop/p1/first
[rick@Ristes-iMac first % cd ..
[rick@Ristes-iMac p1 % pwd
/Users/rick/Desktop/p1
Let's delete the file we just created using the 'rm' command and name of the file for the users and 'del' command and name of the file for the windows users]
```

Now we can delete the first folder using the ‘rm -r first’ command for Mac and ‘deltree first’ for windows users:

```
/Users/rick/Desktop/p1
rick@Ristes-iMac p1 % rm -r first
rick@Ristes-iMac p1 % ls
rick@Ristes-iMac p1 % pwd
/Users/rick/Desktop/p1
rick@Ristes-iMac p1 %
```

Finally, we can remove the p1 folder because the name is not something I would normally use or want on my desktop ‘rm – r p1’:

```
● ● ● Desktop — -zsh — 80x24
rick@Ristes-iMac p1 % cd ..
rick@Ristes-iMac Desktop % ls
ebook
everything
rick@Ristes-iMac Desktop % rm -r p1
rick@Ristes-iMac Desktop %
```

# **Code Formatting**

When you write code, it is a good idea to know that we can format that code. We have already installed VS Code editor so we can install other extensions that can help automatically format the code for us. In this section, you will learn about PEP-8 and the benefits of using it.

You can read more about PEP:

<https://peps.python.org/>

## **What is PEP?**

The PEP is an abbreviation for Python Enhancement Proposal. PEP-8 is a document that provides guidelines on how to write code that will follow proper logic and coding style. The PEP-8 is the most popular one, but there are many other Enhancement Proposals. The PEP -8 will help us write beautiful code that is clean, readable, and styled. The PEP-8 is a Style Guide for Python code and the documentation is very detailed but long (luckily, you do not need to read the entire documentation):

<https://peps.python.org/pep-0008/>

In the same test.py file, I have added two basic functions:

A screenshot of the Microsoft Visual Studio Code (VS Code) interface. The title bar shows "test.py". The left sidebar has icons for file operations, search, and other tools. The main editor area contains the following Python code:

```
1 def fn_addition(num1,num2):  
2     return num1 + num2  
3 def fn_subtraction(num1,num2):  
4     return num1 - num2  
5 print(fn_addition(2,5))  
6 print(fn_subtraction(6,2))
```

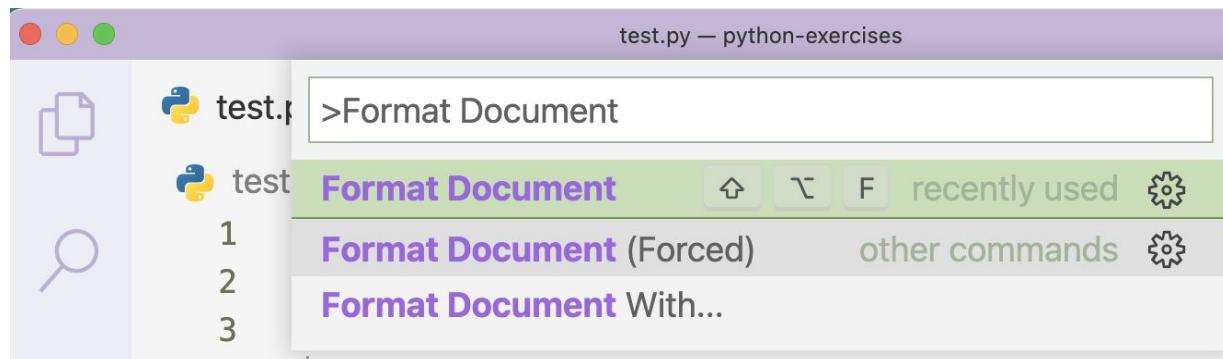
The code is syntax-highlighted, with "def", "print", and variable names in different colors. The last two lines of code, which contain the function calls, are highlighted with a green background.

At the bottom, the status bar shows "Python 3.10.5 64-bit" and other status indicators like "LF" and "Python".

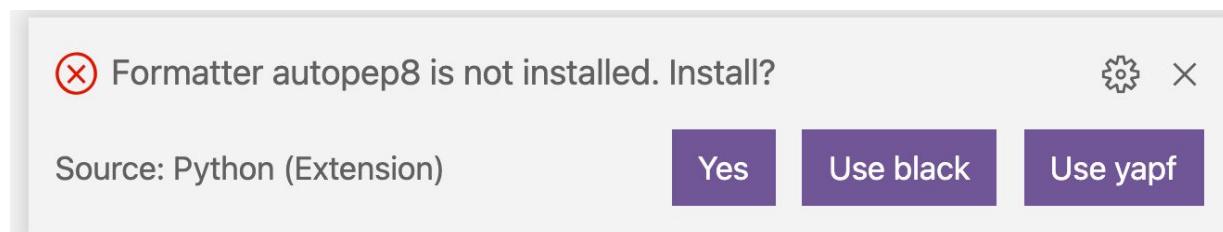
The indentation is visible and if we run the code in the terminal we should have the following output:

```
test.py      ×  
test.py > fn_addition > num1  
1 def fn_addition(num1,num2):  
2     return num1 + num2  
3 def fn_subtraction(num1,num2):  
4     return num1 - num2  
5 print(fn_addition(2,5))  
6 print(fn_subtraction(6,2))  
  
TERMINAL ... 1: zsh  
rick@Ristes-iMac python-exercises % python test.py  
7  
4  
rick@Ristes-iMac python-exercises %
```

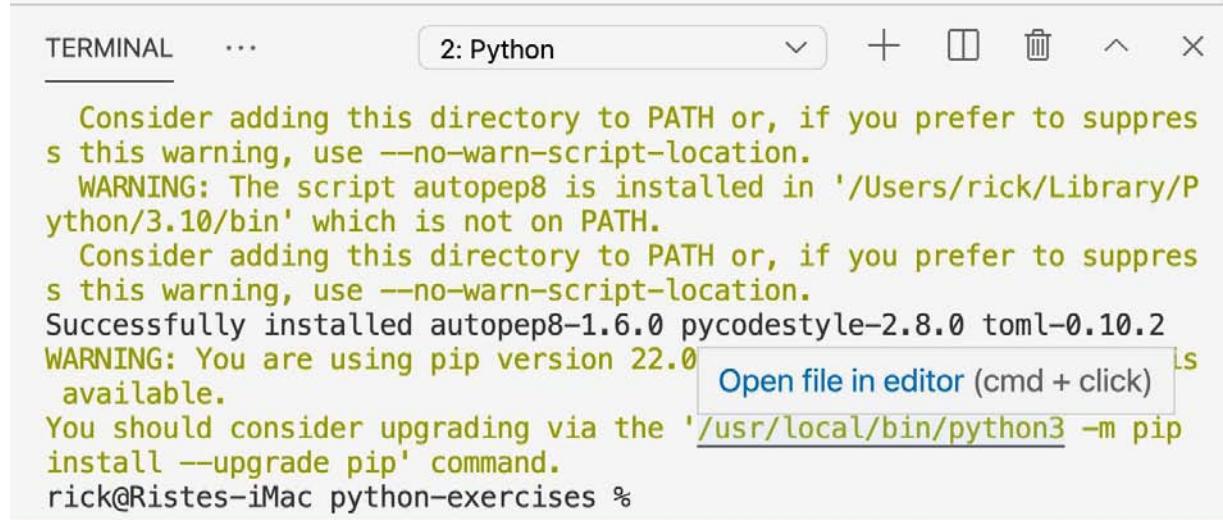
But our code is not formatted according to PEP-8 guidelines. If we go in the menu view > Command Palette and type Format Document and click on it:



Then you will see that the following message says you need to install Autopep8 code formatter:



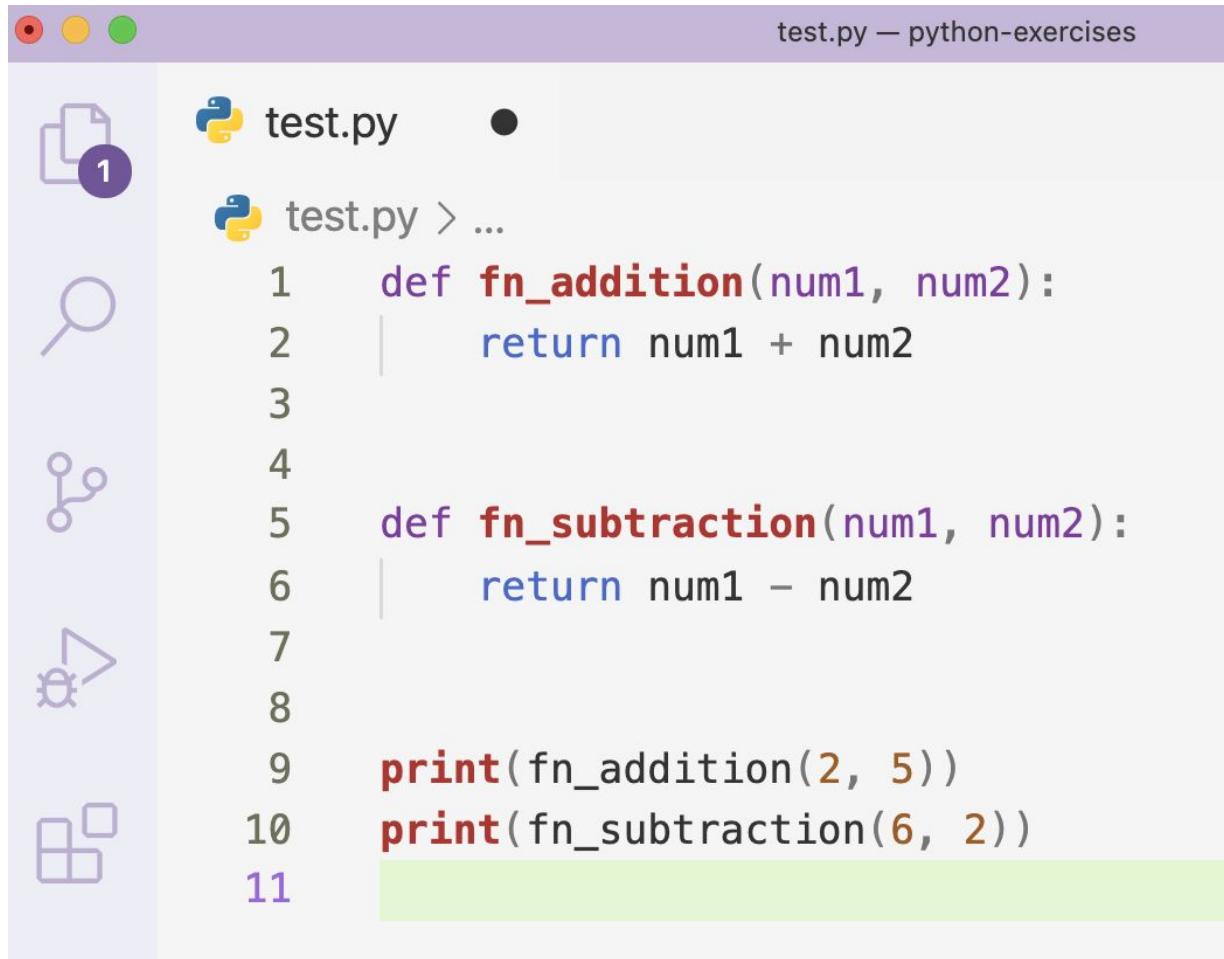
Autopep8 is an extension in VS Code that if enabled will style our code according to PEP-8 guidelines, therefore we should click on the ‘yes’ button and install the extension:



The screenshot shows the VS Code terminal interface with the title bar "TERMINAL" and "2: Python". The terminal window displays the following text:

```
Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.  
WARNING: The script autopep8 is installed in '/Users/rick/Library/Python/3.10/bin' which is not on PATH.  
Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.  
Successfully installed autopep8-1.6.0 pycodestyle-2.8.0 toml-0.10.2  
WARNING: You are using pip version 22.0 [Open file in editor (cmd + click)] available.  
You should consider upgrading via the '/usr/local/bin/python3 -m pip install --upgrade pip' command.  
rick@Ristes-iMac python-exercises %
```

If you have noticed, the code we have in the test.py file remained the same, so again view > Command Palette and type Format Document, and click on it:

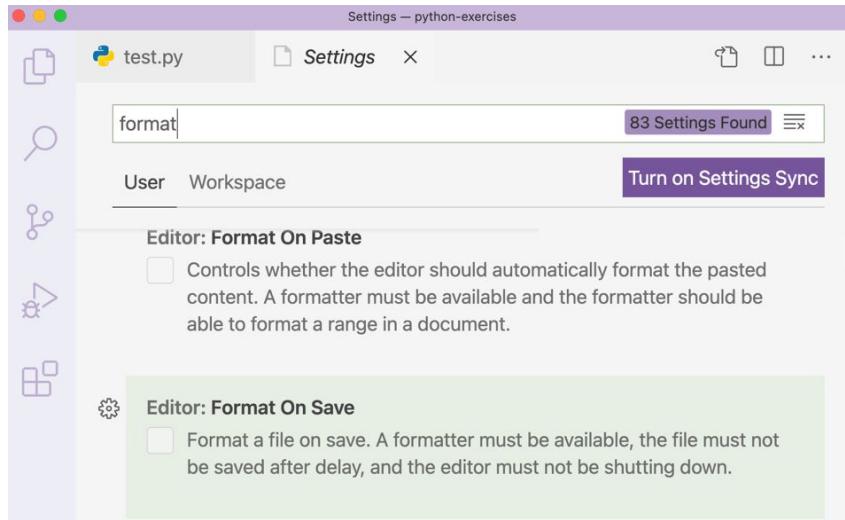


The screenshot shows a code editor window titled "test.py — python-exercises". On the left is a sidebar with icons for file operations (new, open, save, etc.), search, and other tools. The main area displays the following Python code:

```
1  def fn_addition(num1, num2):  
2      return num1 + num2  
3  
4  
5  def fn_subtraction(num1, num2):  
6      return num1 - num2  
7  
8  
9  print(fn_addition(2, 5))  
10 print(fn_subtraction(6, 2))  
11
```

The code is formatted according to PEP-8 conventions, with two blank lines between function definitions.

Our code as you can see looks a bit different, with more spaces in between the function declarations. The code is formatted now according to PEP-8. Why does the code look like this with many spaces between the functions? Well, according to PEP-8 documentation, there should be two empty spaces between the functions. After writing code in your file, you should run the formatter and your code will automatically be formatted for you. Instead of doing this manually every time, you can automate this process. You can do format on save if you go to the menu, code> Preferences> Settings, and type 'format':



There will be a list of options and under the ‘User’ pannel you can scroll down and select the Editor: Format On Save and your formatting will be automatically done. The code formatting will now happen every time you save your code (Command + S) or Control + S for Windows. If interested in learning more about pylint, pep8, or other extensions, please read the VS Code documentation:

<https://code.visualstudio.com/docs/python/linting>

## PyCharm Installation

In this section, you will learn how to install PyCharm on your machines. I have chosen this IDE because it is the best when it comes to writing, executing, and formatting Python code. One downside is that as an IDE, it takes more machine memory because it is a complete package. One of the many benefits compared to other Code editors is that you don’t have to install extensions and plugins in order to run/execute Python. Some will argue that code editors are extremely good because they can be used for multiple languages. Both IDEs and Code editors have pros and cons but I would like to give you an option to use whatever you feel is best for you. PyCharm was developed by the company Jet Brains and they have a free version that you can use to run your Python code and there is also a full-fledged professional version that you need to pay for. The paid version will include extensions and 17 different tools which you don’t need at this stage .

<https://www.jetbrains.com/pycharm/>



The Python IDE  
for Professional  
Developers

[DOWNLOAD](#)

Full-fledged Professional or Free Community

When you click on the download button, it will take you to the download page where you can choose the right version. It will automatically detect your operating system but if it doesn't, please select the operating system and download the free community version.

# Download PyCharm

Windows

macOS

Linux

## Professional

For both Scientific and Web Python development. With HTML, JS, and SQL support.

[Download](#)

.dmg (Intel) ▾

Free 30-day trial available

## Community

For pure Python development

[Download](#)

.dmg (Intel) ▾

Free, open-source

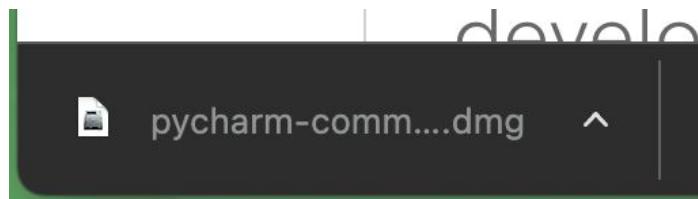
My operating system is macOS and the version I should download is called Community and it is free, open-source, and pure for Python development. Click the download button and follow the installation wizard. If you are a Windows user, the installation will be slightly different compared to mine but in the end, you will have the same IDE. You will also need to use your email address before you can start downloading the PyCharm:

## Getting Started

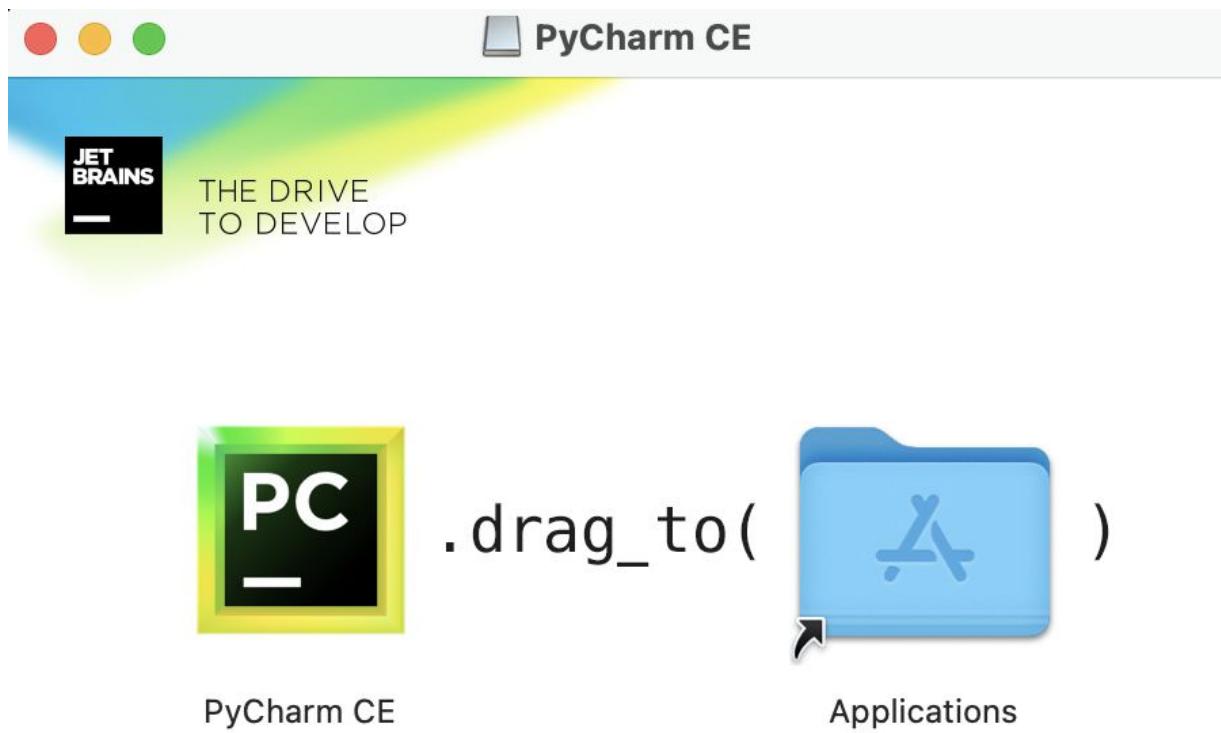
Send me helpful educational materials  
during my evaluation period

your-email@gmail.com|

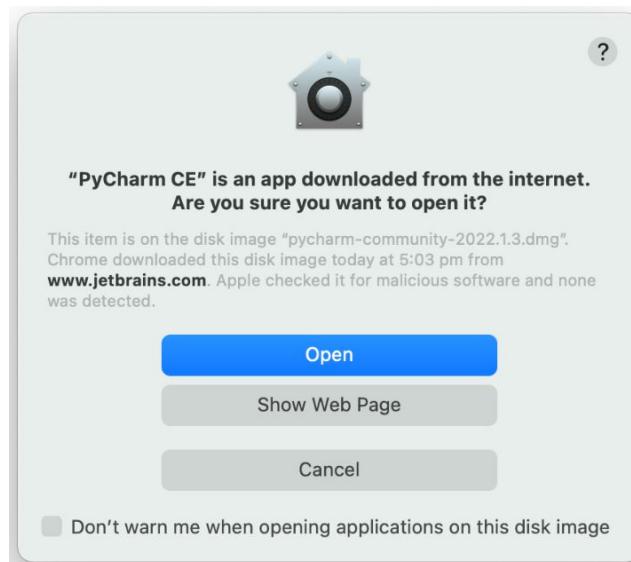
After you download it, you can start the installation process:



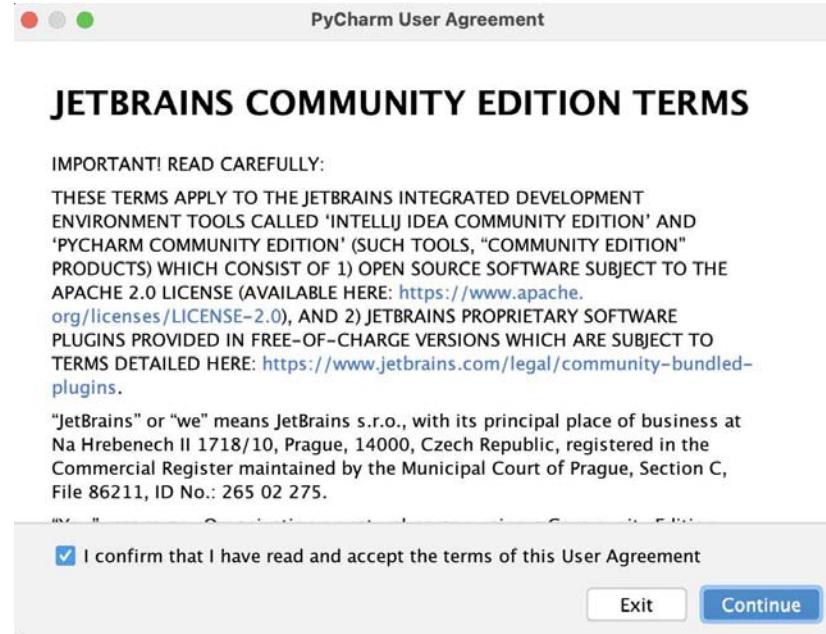
The macOS users will see this screen where you need to drag the PyCharm CE to the applications, and in Windows, there will be an installer wizard:



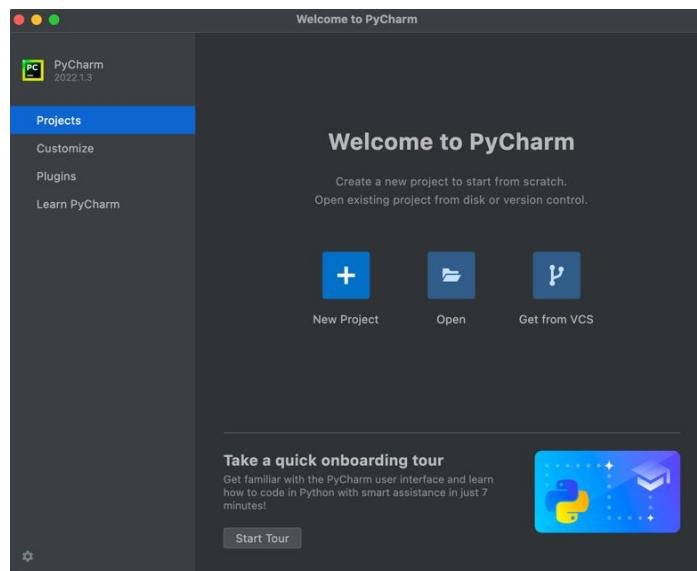
After we drag the icon, you can double-click on the PyCharm CE icon so it can open the IDE. If you have a previous version of PyCharm, you can do a clean install and you don't have to import the settings. After double-clicking on the icon, it will give you the following window:



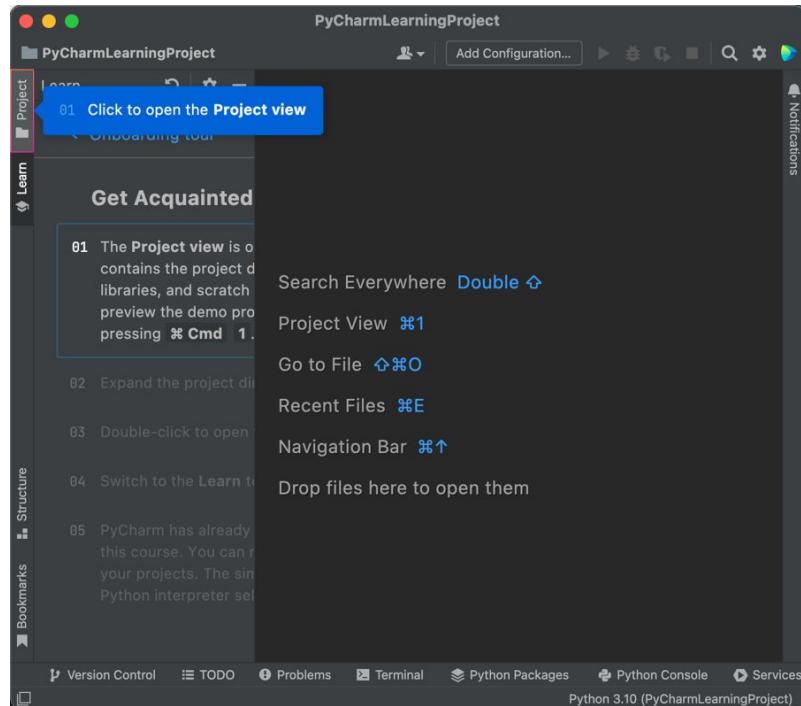
Read the JETBRAINS terms of use, tick the box on the left bottom corner and press the Continue button:



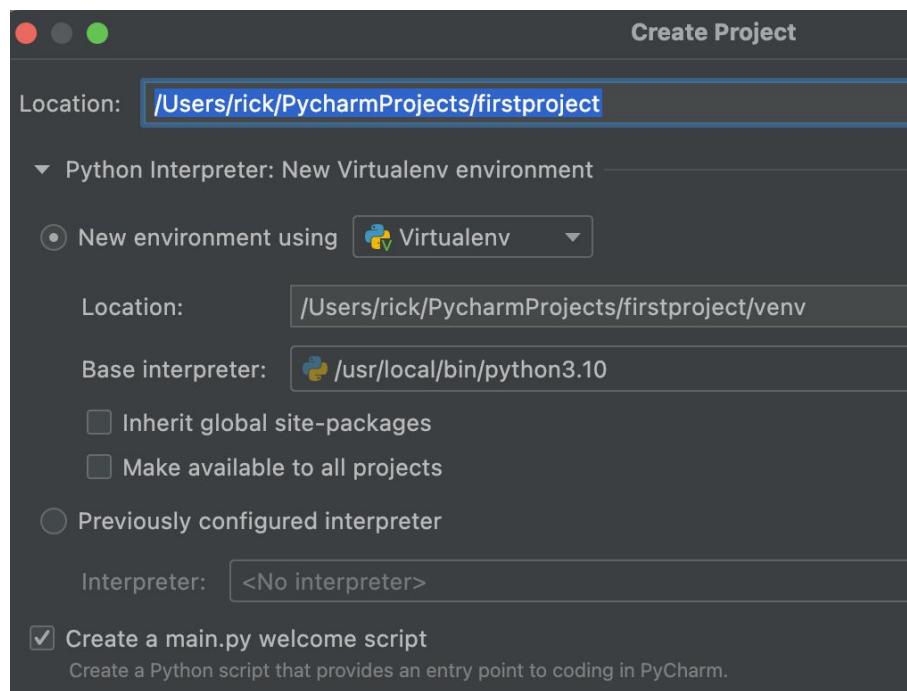
After this, you should see the welcome page:



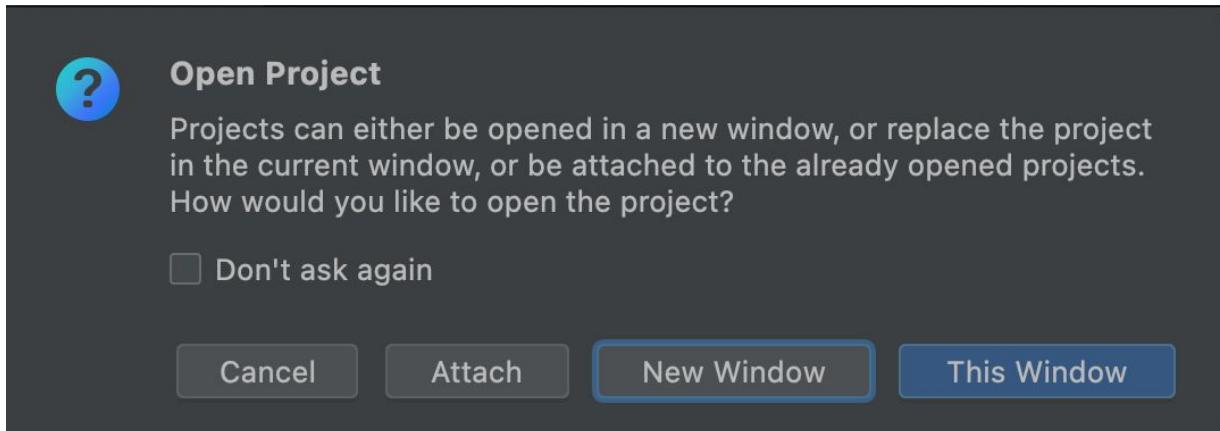
You can click on the Start Tour and you can read all of the tips and tricks so you can get familiar with PyCharm:



If you want to create a new project, you can select file> New Project:



Make sure you set where you want your PyCharm files to be saved and you can name your project whatever you like, in my case, I will name it ‘firstproject.’ The Base Interpreter should be the latest Python version you have installed. For me, this was version 3.10 and it was pre-selected. Finally, you can create the project by clicking on the ‘Create’ button. If you get this message after creating the project, you can select ‘This Window’



The project will be created for you and you will have a file usually called main.py:

A screenshot of the PyCharm IDE interface. On the left, the 'Project' tool window shows a single project named 'firstproject' containing a 'venv' folder and a 'main.py' file. The 'main.py' file is open in the center editor pane. The code in 'main.py' is as follows:

```
firstproject - main.py
firstproject ~/PycharmProject
Project  venv
          main.py
External Libraries
Scratches and Consoles

firstproject - main.py
main.py x

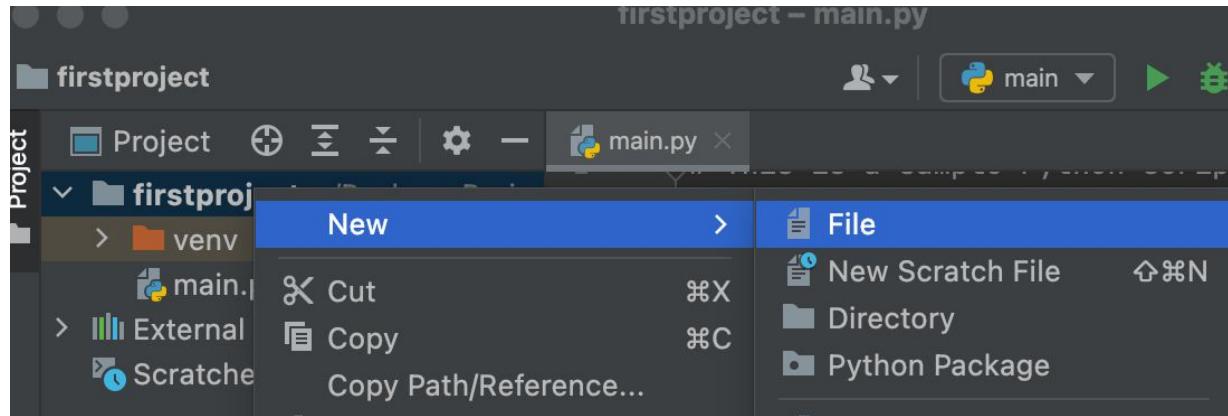
# Press ^R to execute it or replace it with your code
# Press Double ⌘ to search everywhere for class

def print_hi(name):
    # Use a breakpoint in the code line below to debug your script.
    print(f'Hi, {name}')  # Press ⌘F8 to toggle the breakpoint.

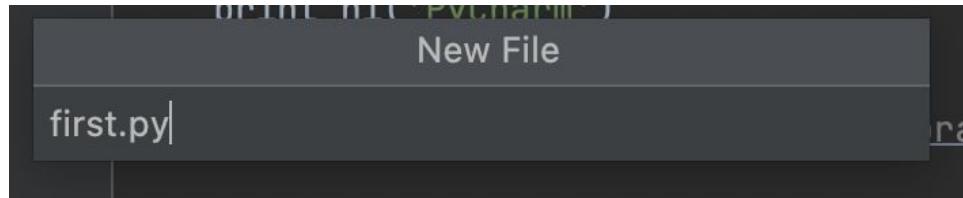
# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    print_hi('PyCharm')

# See PyCharm help at https://www.jetbrains.com/help/pycharm/
```

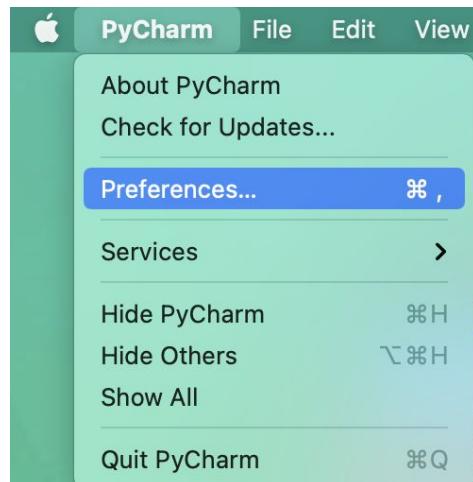
As you can see, the **firstproject** is created and inside we have a file called main.py with some starting code. If you want to create a new file, you can select the firstproject folder, right click on it, and then select new file:



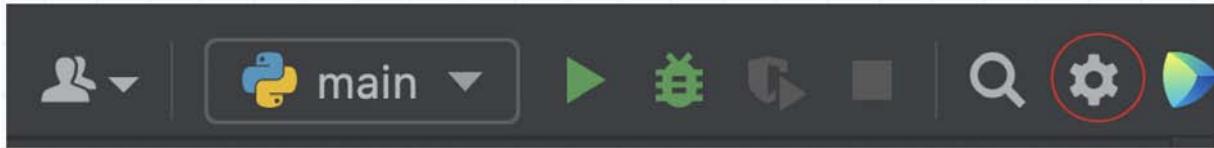
Name your file whatever you like, mine will be first.py (make sure you type the right extension):



Click return/enter and the file will be created inside the firstproject. You can do further customization to the PyCharm if you go to the menu, PyCharm>preferences :



Or you can click on the IDE and Project Setting icon and it will lead you straight to the Preferences section:

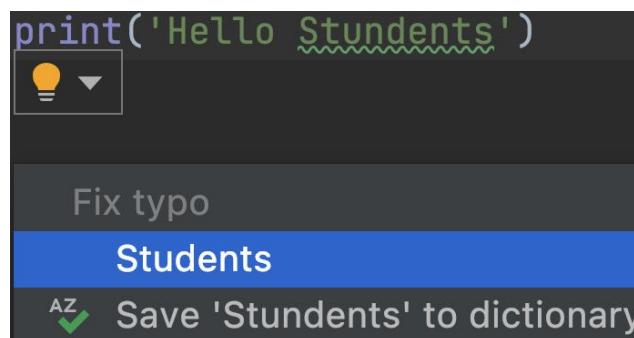


I have just set the font size to be bigger so I don't have to strain my eyes too much during the coding. Let's start by writing a piece of code like a simple `print()` function with bad spelling:

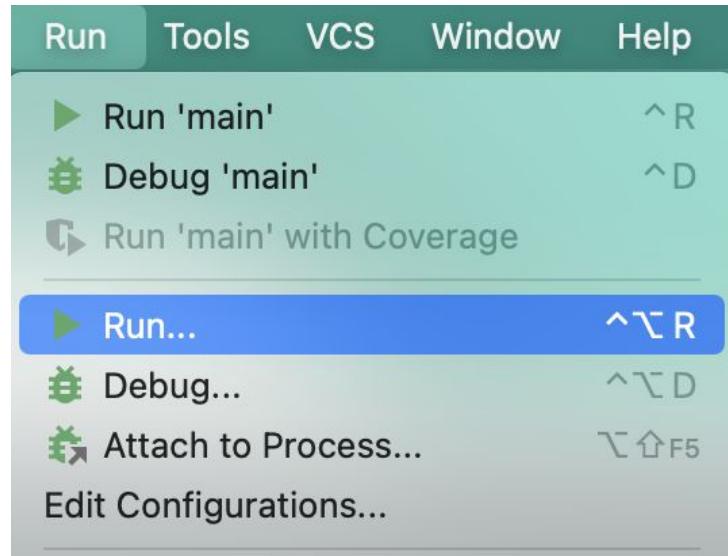
```
first.py X
1 print('Hello Stundents')
```

A screenshot of the PyCharm code editor. The window title is "first.py X". In the code editor, line 1 contains the Python command `print('Hello Stundents')`. The word "Stundents" is underlined with a yellow wavy line, indicating a spelling error. A small orange lightbulb icon is positioned below the line number 1.

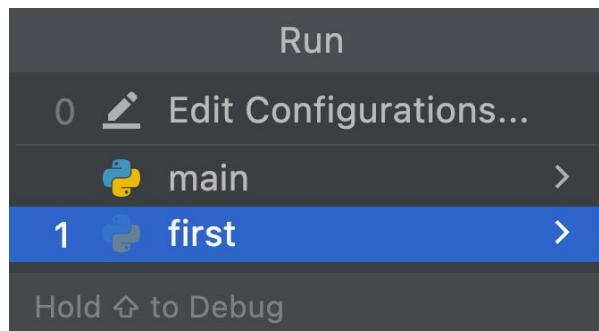
As you can see, the word 'Stundents' should be 'Students' so we get an automatic syntax error detection plus the closing ')' bracket is underlined in yellow. The yellow underline means we need to add a new line and the warning will disappear. You can click on the light bulb icon and it will help you fix the problem:



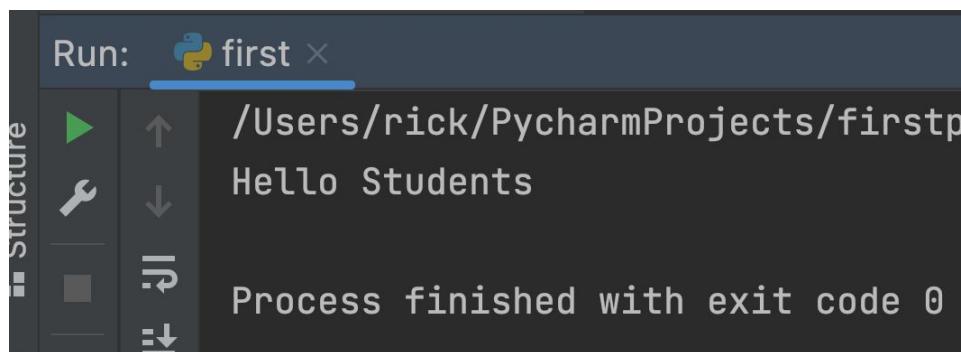
Finally, how we can run the code we have in the `first.py` file? Well, you need to go to the menu and select Run as in the figure below:



And because you have two files, main.py and first.py, it will ask us which file you want to run:



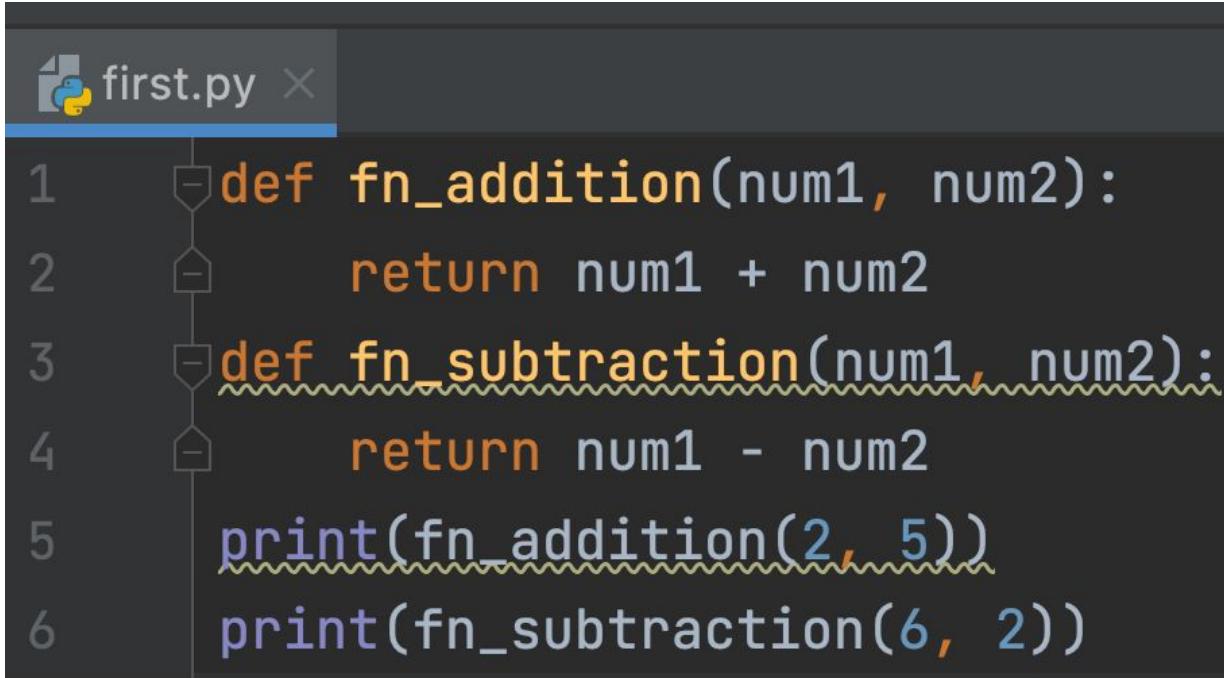
After you select the correct file, the terminal will be opened at the bottom with a message that comes from the print function:



PyCharm is a very good IDE because you don't have to install packages, extensions, etc. to write and execute your Python code. Please read the PyCharm documentation and learn some of its features.

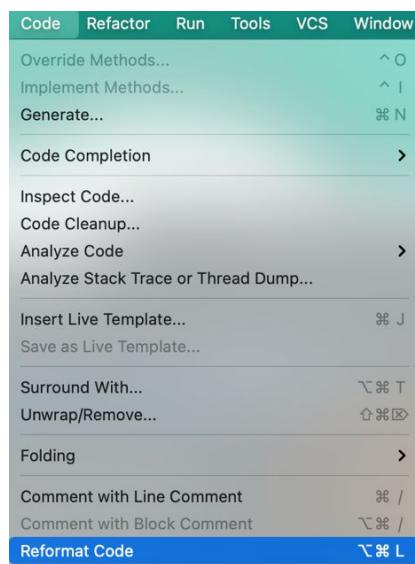
## What about code formatting in PyCharm?

If you have longer code that is more than one line we can go to Code>Reformat Code and the code will be reformatted automatically for you. Here is an example copied from the file we created in the VS Code section:

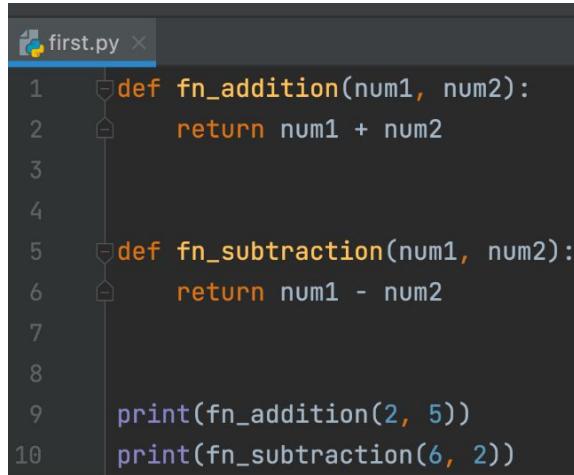


```
1 def fn_addition(num1, num2):
2     return num1 + num2
3 def fn_subtraction(num1, num2):
4     return num1 - num2
5 print(fn_addition(2, 5))
6 print(fn_subtraction(6, 2))
```

All we need is to go to the menu section and select code>Reformat code:



Let's look at our first.py file and check if the code is now formatted correctly:



A screenshot of the PyCharm IDE interface. The title bar says "first.py". The code editor contains the following Python code:

```
1 def fn_addition(num1, num2):  
2     return num1 + num2  
3  
4  
5 def fn_subtraction(num1, num2):  
6     return num1 - num2  
7  
8  
9 print(fn_addition(2, 5))  
10 print(fn_subtraction(6, 2))
```

As you can see from the figure above, the Reformat code works the same as in VS Code. These are some links I want you to have in case you want to read more about PyCharm:

Installation Guide:

<https://www.jetbrains.com/help/pycharm/installation-guide.html>

Learn PyCharm:

<https://www.jetbrains.com/pycharm/learn/>

PyCharm Guide:

<https://www.jetbrains.com/pycharm/guide/>

## **Summary**

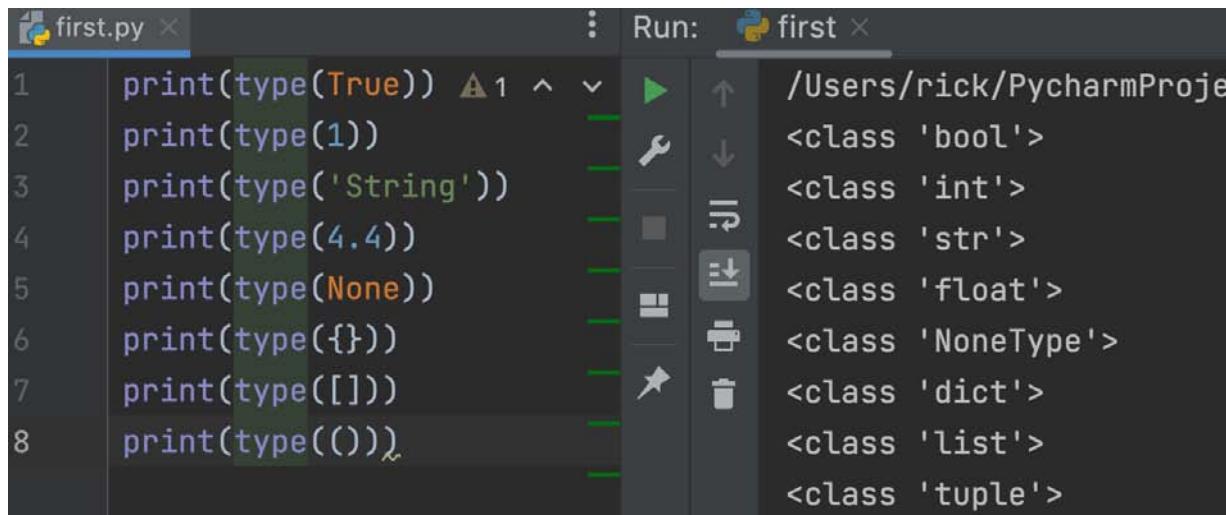
In this chapter, you have learned a lot. We started by installing Python on different operating systems and then we installed different programming tools. We have installed the VS Code editor which is one of the best code editors in the market, but if you don't like editors then I have provided the best IDE for Python called PyCharm. You don't have to use them both, it is up to you to decide which one is the best for you. Everything you have learned in this chapter will make you a better Python developer. We shouldn't forget to mention that you have learned the importance of having clean and formatted code and together with the terminal commands, you are now becoming a professional developer.

# Chapter 4 – Object Oriented Programming – Advanced Python

Python is an Object-Oriented Programming language. This term is used by many other languages and it's fundamental. It's so important to understand the concept of having Objects because in Python everything can be considered as an Object. OOP languages are organized around Objects and these Objects can be created and modified during runtime. We will learn where these Objects come from in the next sections but each object is known as an instance of a class. The Classes in OOP are the templates that allow us to create objects. In this chapter, we will learn what OOP is and what its most important features are.

## Class keyword

In the previous chapter, we have seen the keyword ‘class’ when we tried to confirm the data type using the function type():



The screenshot shows a PyCharm IDE interface. On the left is a code editor with a file named 'first.py' containing the following code:

```
1 print(type(True))  
2 print(type(1))  
3 print(type('String'))  
4 print(type(4.4))  
5 print(type(None))  
6 print(type({}))  
7 print(type([]))  
8 print(type(()))
```

On the right is a 'Run' tool window showing the output of the code:

Output
<class 'bool'>
<class 'int'>
<class 'str'>
<class 'float'>
<class 'NoneType'>
<class 'dict'>
<class 'list'>
<class 'tuple'>

The class keyword is everywhere in Python. I mentioned that everything in Python can be considered an Object. Each object has access to properties,

methods, and attributes. The Classes in Python will allow us to create objects, so they are a blueprint for creating objects. This means that the classes can help us create new Objects and expand the initial list of data types. This also means that we can get creative and create our own data types that are different from the built-in ones. This chapter will help you to gain an in-depth understanding of these concepts. Therefore, Object-Oriented Programming is the most fundamental paradigm. Before Object-Oriented Programming, our code was a list of procedures where each procedure was executed in a specific order but this was changed when OOP was introduced. Please note that you can continue using repl.it.com to run the code that we are going to cover in this chapter or you can use some of the tools we have installed in the previous chapter. To put your curiosity at ease, let's start learning OOP.

# **Why Object-Oriented Programming?**

Object-Oriented Programming is very useful because we can write a piece of code and reuse that code by creating different Classes and instances of these classes. Each of these Classes can add new features without having to write/change or modify the existing code. The complex programs require us to use Classes and Objects because we can create multiple manageable chunks of code.

## **The Principles of Object-Oriented programming**

The following principles are generally applicable to all Object-Oriented Programming languages.

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

## **What other languages support Object Oriented Programming?**

The oldest and most well-known object-oriented language is Java. Then there is C# which is another OOP language developed by Microsoft. Some of the more famous OOP languages are PHP, Ruby, TypeScript, and Python.

## **How can we create Classes in Python?**

In order to create a class, we need to use the keyword `class`, and then we should provide the class name.

Example:

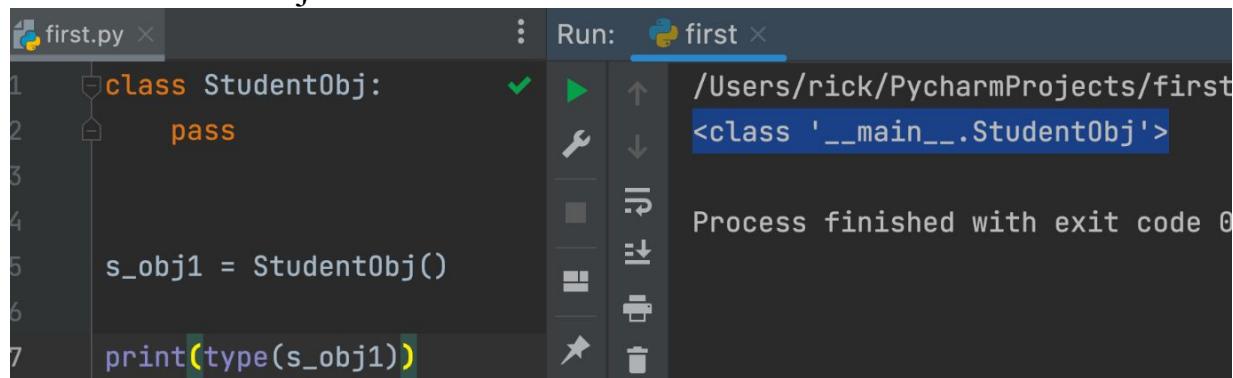
```
class Student:
```

As you can see, the Class name starts with a capital letter and this is a naming convention that you should get used to it. Another rule is that the

class name should be singular, for example, you should not name your classes like this ‘class Students.’ If the class name needs to consist of two words then you should use the following convention:

```
class StudentObj:
```

As we can see, each of the words starts with a capital letter, which is different compared to the snake case we used so far. The snake case is used to join multiple words with underscore like student\_obj. After the Class name, we have a colon ( : ) and here we will need to write the class body. But before we do this, let’s create an Object for this class. If we leave the code like this, it will throw an error so let’s add the ‘pass’ keyword and create our first Object.



```
first.py
1  class StudentObj:
2      pass
3
4
5  s_obj1 = StudentObj()
6
7  print(type(s_obj1))
```

Run: first  
/Users/rick/PycharmProjects/first  
<class '\_\_main\_\_.StudentObj'>  
Process finished with exit code 0

On the right side of the figure, we have the output where we have the type class and ignore the ‘main’ keyword for now because we will explain what it means in future sections.

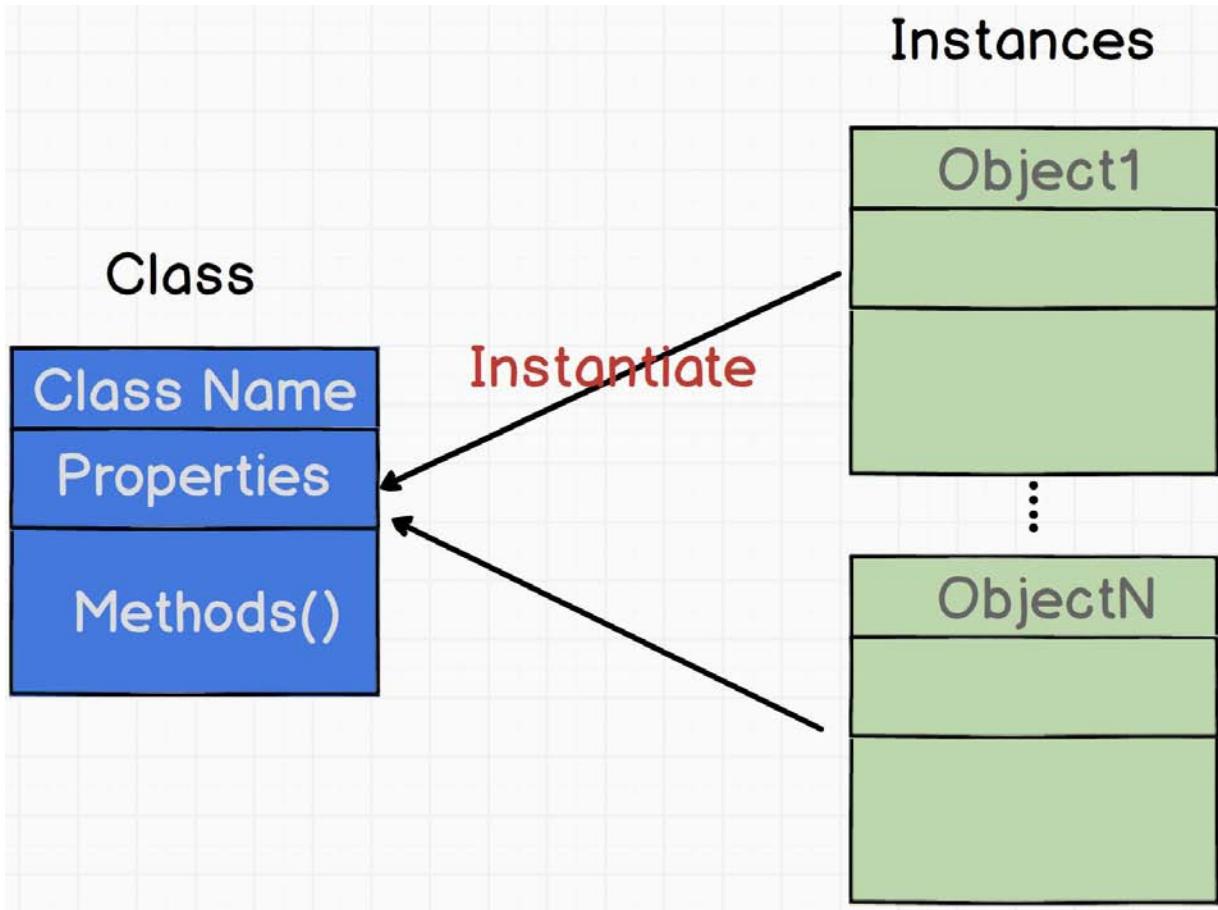
## What are Classes and Objects in Python?

You need to understand these two keywords Classes and Objects/instances. The Classes are the blueprint or templates for creating Objects. When we create Classes, we define a set of properties and methods that will represent the base of that class. The methods of the Class are like the functions and the properties are like variables. The Objects will use the class template so it can access a copy of the properties and methods from the Class.

Keywords so far:

- **Classes**
- **Objects**

- **Properties**
- **Methods.**



From the figure above, we have a few new keywords that I need to explain. On the left side, we have a Class and this represents what we want to create, and what kind of properties and methods we want that Class to have. So, it is a blueprint or template. On the right side with green background color, we have the Objects. When we want to create an Object from a particular Class, we need to instantiate that class. The Objects we are creating are called Class Instances and we can create as many instances as we want. If you have learned other OOP languages, then you are familiar with this syntax because they all share the same or similar syntax:

```
# class template
class StudentObj:
    pass
```

```
# Create new object by instantiating the StudentObj class
s_obj1 = StudentObj()

print(type(s_obj1))
```

The **s\_obj1** is an Object or an Instance of the class `StudentObj`.

## Class Constructor, attributes, methods

In this section, we will create a Class called `Person` and we are going to learn a few more new features. Do not worry if you don't understand all of them at first, it takes time but trust me, in the future, this will be like very easy for you. So let's create our first real class called `Person`:

```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f'{self.name}')
```

In Python, the Class methods start with a double underscore `__` and they end with a double underscore as well. Generally, in Python, the methods are special types of functions. We have covered functions in the previous chapter so we are using the exact same syntax with the keyword '`def`' in front. So, the difference between functions and methods is that methods are specific to a Class, so they belong to that Class and functions belong to the global scope. That is why the authors in most literature you will read use functions instead of methods when they describe Classes, they can be easily mixed up. In our case, the methods with a double underscore are special and they are called **dunder** methods or magic methods. The method at the top of the Class declaration is known as the **constructor** method or `init()`. This method will automatically be called when we instantiate Objects from the class `Person`. We don't have any objects at this stage but this method will be

called immediately. Let us create a **person\_obj1** which will be an instance from the Person class and try to run the following code:

```
chapter4 > person.py
Project main.py x classes.py x person.py x Run: person x
1 class Person:      ▲ 1 ^ v
2     def __init__(self, name):
3         self.name = name
4
5     def greet(self):
6         print(f'{self.name}')
7
8
9 person_obj1 = Person()
10 print(person_obj1)
11 |
```

```
/Users/rick/PycharmProjects
/chapter4/venv/bin/python
/Users/rick/PycharmProjects
/chapter4/person.py
Traceback (most recent call last):
File "/Users/rick
/PycharmProjects/chapter4
/person.py", line 9, in
<module>
    person_obj1 = Person()
TypeError: Person.__init__() missing 1 required positional
argument: 'name'
```

From the picture above, on the right side we have a **TypeError** saying that the Person method **\_\_init\_\_()** is missing the **name** argument, but how does it know? Well, we are creating an instance of the Person Class, therefore when an Object of the Person Class is created, the constructor of that Class will be called:

```
person_obj1 = Person()
print(person_obj1)
```

This will run the constructor function **init()** and it will try to do the following assignment:

**self.name = name**

But the problem is that I never give any arguments during the constructor call because we called the Class like this **Person()** without passing an argument. In the constructor method, after the **self**-keyword, we are expected to provide a parameter name:

```
def __init__(self, name):
```

This means that after the self-keyword if we have parameter/parameters, we must provide them in the Class call, for example, Person('Jason'). We need to link them like in the figure below:

The screenshot shows a code editor with a dark theme. A Python class named 'Person' is defined. The code is as follows:

```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f'{self.name}')

person_obj1 = Person()
print(person_obj1)
```

Annotations are present in the code:

- A red arrow points from the word "name" in the `__init__` method signature to the variable `name` in the assignment statement `self.name = name`.
- A yellow circle highlights the opening parenthesis of the `Person()` call in the line `person_obj1 = Person()`.
- A yellow arrow points from the highlighted parenthesis to the `name` parameter in the `__init__` method signature.

Therefore, when we instantiate an Object of the Class Person, we need to provide the argument or arguments that are listed in the init constructor. In our case, the init constructor requires only one parameter and that is the 'name'.

The screenshot shows the PyCharm IDE interface. On the left, the code editor displays `first.py` with the following content:

```
1 class Person:
2     def __init__(self, name):
3         self.name = name
4
5     def greet(self):
6         print(f'{self.name}')
7
8
9 person_obj1 = Person('Jason')
10 print(person_obj1)
```

On the right, the 'Run' tool window shows the output of the run command:

```
Run: first
Process finished with exit code 0
```

The output window also shows the memory address of the `Person` object: `<__main__.Person object at 0x103c161d0>`.

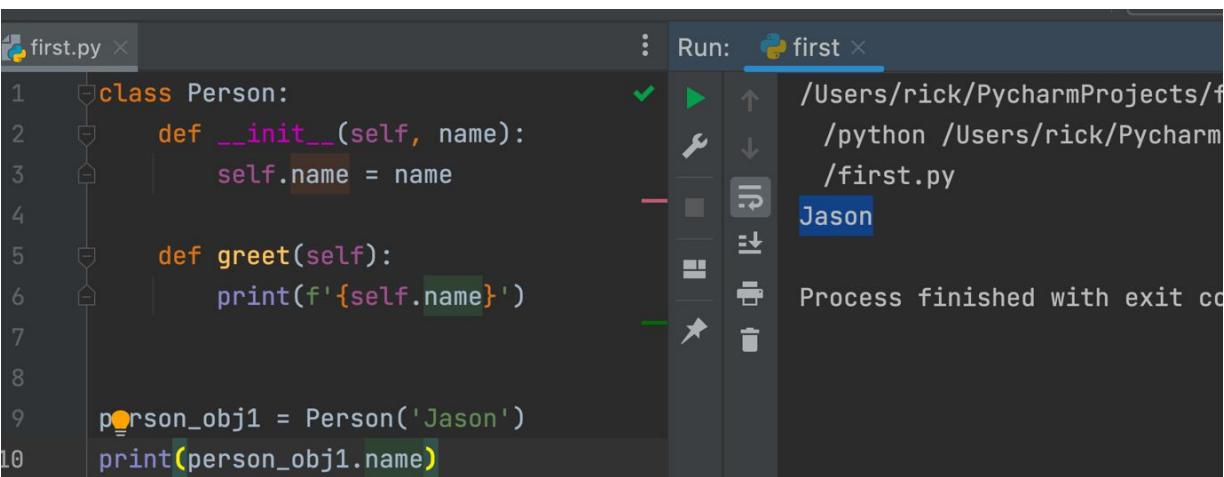
As you can see, we add one argument in the class call:

```
person_obj1 = Person('Jason')
```

Now if we run the code again the error will be gone. The ‘self’ keyword we have in the `init()` method refers to the current instance of the class (`person_obj1`). In our case the `self`-parameter is a reference to the `person_obj1` which is the only instance of the class, therefore `person_obj1` will have access to the Class methods and properties. The ‘self’ keyword can be named whatever we like but it has to be the first parameter in the constructor because it’s considered a default parameter of any method in the class. This is what is happening behind the scenes:

```
class Person:  
    def __init__(self, name):  
        self.name = name # Jason  
  
    def greet(self):  
        print(f'{self.name}')  
  
#self refers to person_obj1  
  
person_obj1 = Person('Jason')  
print(person_obj1.name)
```

Let's access that name property from the Object because the Objects are Instances of a Class and they have access to the Class methods and properties:



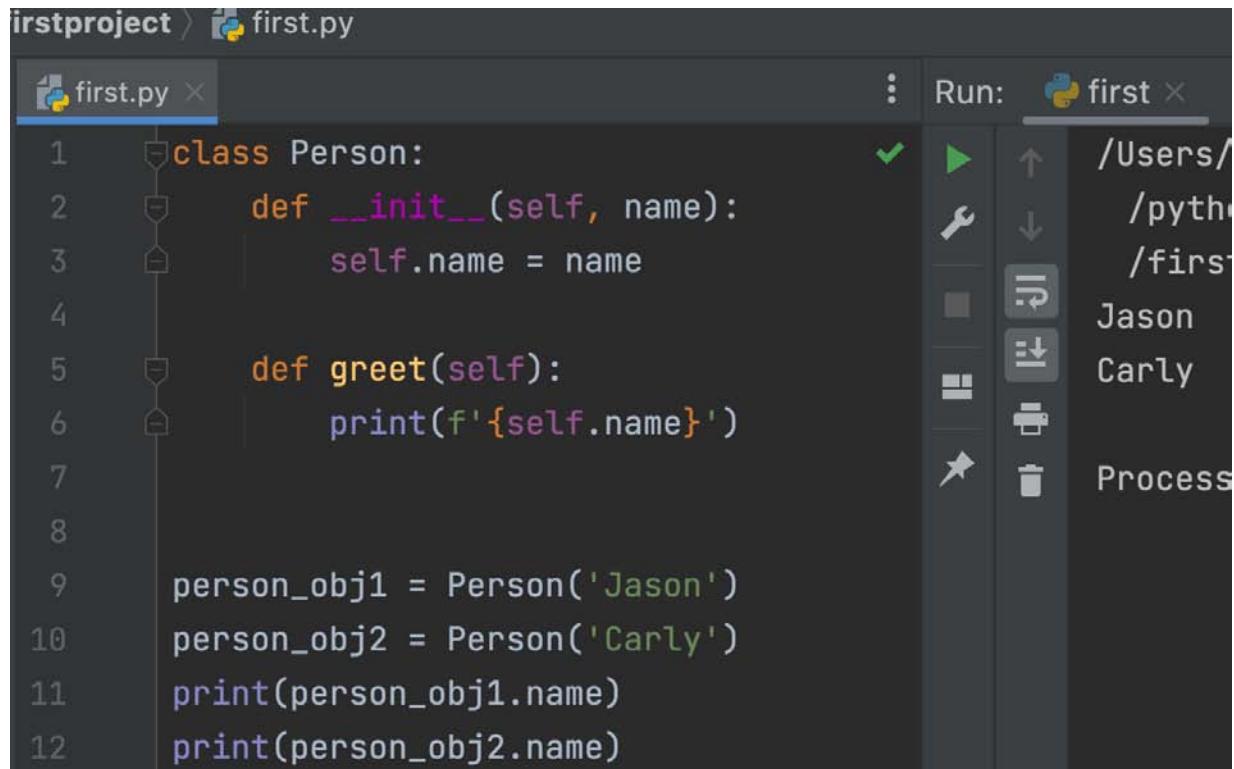
The screenshot shows a PyCharm interface with two panes. The left pane displays the code for 'first.py':

```
1 class Person:  
2     def __init__(self, name):  
3         self.name = name  
4  
5     def greet(self):  
6         print(f'{self.name}')  
7  
8  
9     person_obj1 = Person('Jason')  
10    print(person_obj1.name)
```

The right pane shows the 'Run' tool window with the configuration set to 'first' and the output window displaying the result of running the script:

```
Run: first  
/Users/rick/PycharmProjects/f  
/python /Users/rick/Pycharm  
/first.py  
Jason  
Process finished with exit co
```

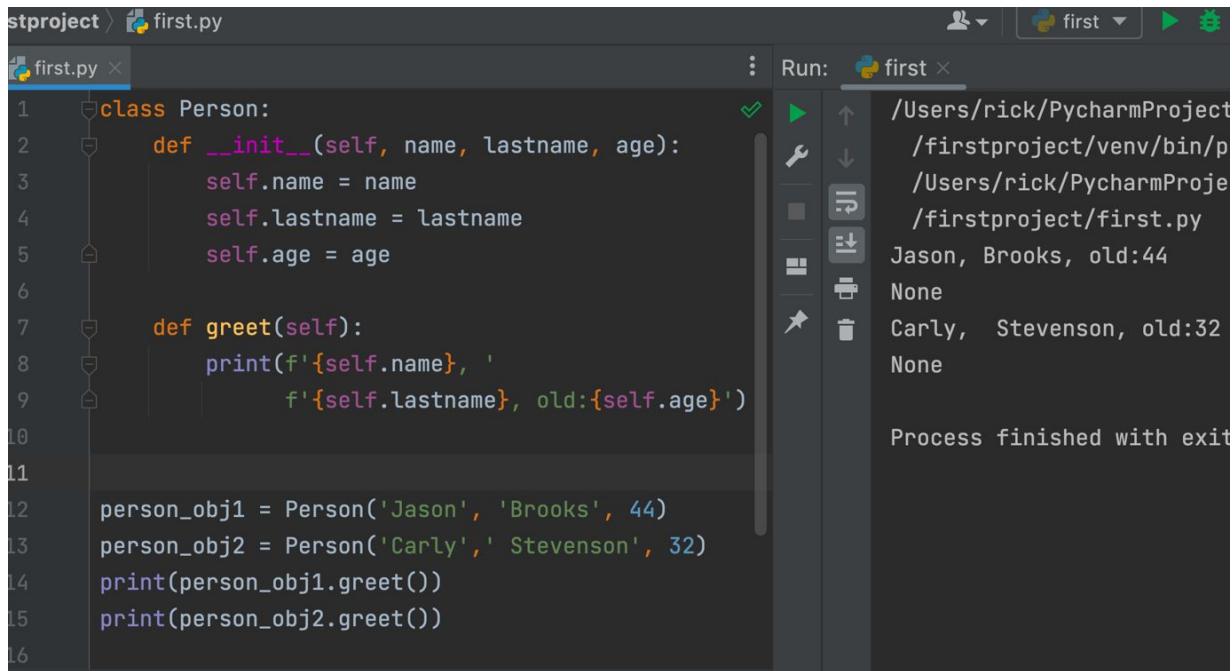
Let's create another instance/object from the Person class this time with a different name:



The screenshot shows the PyCharm IDE interface. The left pane displays the code for `first.py`. The code defines a `Person` class with an `__init__` method that initializes the `name` attribute. It also contains a `greet` method that prints the object's name. Below the class definition, two instances are created: `person_obj1` with name 'Jason' and `person_obj2` with name 'Carly'. The right pane shows the run configuration for the `first` script, with the working directory set to `/Users/Python/first`. The run button is highlighted.

```
1  class Person:
2      def __init__(self, name):
3          self.name = name
4
5      def greet(self):
6          print(f'{self.name}')
7
8
9  person_obj1 = Person('Jason')
10 person_obj2 = Person('Carly')
11 print(person_obj1.name)
12 print(person_obj2.name)
```

As you can see, we have created another instance of the Person Class and this instance now has a different name attribute (Carly) and when the `__init__` method is called, the name parameter will assign 'Carly' because the of 'self' keyword which will tell the constructor that the parameter belongs to the `person_obj2` Object. So, we have followed the good old DRY (DO Not Repeat Yourself) principle because we were able to create two separate instances/objects that called the same class with different name attributes. The objects accessed the same class and its methods and properties. If this was not OOP, then we would have to write the same code for both of the objects. This is making the code to be more dynamic. The self-keyword is bound to the object that is instantiated. In the Person Class, the `init` constructor is very simple, accepting only one parameter but it can be much more complex, for example, let's say we want the constructor to be called with name, last name, and age parameters and we can also change the `greet()` function to print all of the details:



The screenshot shows the PyCharm IDE interface. On the left, the code editor displays `first.py` with the following content:

```
1 class Person:
2     def __init__(self, name, lastname, age):
3         self.name = name
4         self.lastname = lastname
5         self.age = age
6
7     def greet(self):
8         print(f'{self.name}, '
9             f'{self.lastname}, old:{self.age}')
10
11 person_obj1 = Person('Jason', 'Brooks', 44)
12 person_obj2 = Person('Carly', 'Stevenson', 32)
13 print(person_obj1.greet())
14 print(person_obj2.greet())
```

On the right, the 'Run' tool window shows the output of the script:

```
/Users/rick/PycharmProject
/firstproject/venv/bin/python
/firstproject/first.py
Jason, Brooks, old:44
None
Carly, Stevenson, old:32
None

Process finished with exit
```

Now when we instantiate an Object from the `Person` class, we need to pass two additional arguments like last name and age. The `name`, `lastname`, and `age` are class attributes and they can be accessed by the Object-instance. The class attributes are properties that the Object has access to using the dot notation and the property:

```
print(person_obj1.name)
print(person_obj2.age)
```

As you can see, the `person_obj1` and `person_obj2` have access to class attributes without the brackets because they are not methods. If we want to access a class method directly, we must use the brackets () like this:

```
print(person_obj1.greet())
print(person_obj2.greet())
```

Finally, the `person_obj1` and `person_obj2` are instances of the class `Person` but they are unique and different from each other, please do not confuse them. We have used the class `Person` as a template to create/instantiate objects that are stored in different places in computer memory. We can find this very easily if we print only:

```
print(person_obj1)
print(person_obj2)
```

The output will be:

```
<__main__.Person object at 0x10cddfdf0>
<__main__.Person object at 0x10cddebff0>
```

As you can see, both Person objects are stored in different locations in the computer memory. This is perfect because now each of these objects can have its own methods and attributes that will be hidden from the other object.

In summary, the Class can have attributes and methods. The attributes are initialized by the constructor method that will have a ‘self’ keyword as a default parameter. When we create a new object or instance of the class Person, we immediately call the constructor and initialize all of the parameters to the values that are unique to that object. Each instance of the Person class can access the class methods and attributes and they are independent of each other, meaning they are stored in different memory locations.

## Class Object Attribute

A Class Object attribute is different compared to the rest of the attributes in the Class. Let’s first create one:

```
class Person:
    # Class Object Attribute
    is_person = True

    # Constructor
    def __init__(self, name, lastname, age):
        self.name = name # name attribute
        self.lastname = lastname # lastname attribute
        self.age = age # age attribute
    # Method
```

```
def greet(self):
    print(f'{self.name}, '
          f'{self.lastname}, old:{self.age}')

person_obj1 = Person('Jason', 'Brooks', 44)
person_obj2 = Person('Carly', 'Stevenson', 32)
print(person_obj1.greet())
print(person_obj2.greet())
```

From the figure above, we have the **is\_person** Class Object attribute that is set to be Boolean True. The difference between this attribute and the rest of the attributes is that the Class Object attributes are **static**. In the method `greet(self)`, we pass the `self`-keyword so we can get access to the name, lastname, and age from the object that calls the method because the `self` is bound to the object. But this static attribute can also be called by the class itself:

```
class Person:
    # Class Object Attributes
    is_person = True

    # Constructor
    def __init__(self, name, lastname, age):
        self.name = name # name attribute
        self.lastname = lastname # lastname attribute
        self.age = age # age attribute

    # Method
    def greet(self):
        print(f'{self.name}, '
              f'{self.lastname}, old: {self.age}. '
              f'Called by the class itself: {Person.is_person}. '
              f'Called by the self keyword {self.is_person}')
```

```
person_obj1 = Person('Jason', 'Brooks', 44)
person_obj2 = Person('Carly', 'Stevenson', 32)
print(person_obj1.greet())
print(person_obj2.greet())
```

Class object attributes are the ones that will not change, they are static and can be accessed directly by the class itself or the instances/objects. The class object attributes belong to the class and they are initialized to some value in the Class itself, and the attributes defined in the constructor or other methods like name, age, and lastname will have values that are dynamically assigned by the calling objects. If we run the above code the output is:

```
Jason, Brooks, old: 44. Called by the class itself: True. ↴
↳ Called by the self keyword True
None
Carly, Stevenson, old: 32. Called by the class itself:
    True. Called by the self keyword True
None
```

## \_\_init\_\_ Constructor

I have already explained what the \_\_init\_\_ constructor is in the sections above, but I would like to add a few more things that I think are important when it comes to class instructors. Let's have a look at the constructor again:

```
def __init__(self, name, lastname, age):
    self.name = name # name attribute
    self.lastname = lastname # lastname attribute
    self.age = age # age attribute
```

And the object-instance that will call the constructor immediately:

```
person_obj1 = Person('Jason', 'Brooks', 44)
```

As soon as the person\_obj1 is instantiated from the class Person, the constructor function will be called. Why is the constructor function so important? Well, the constructor will do its job and it will construct a new Object or instance of the Person Class so that the new object can use the Class attributes and methods. The constructor's first parameter is the self-keyword. This self-keyword as you can see have a different color if we compare it to the rest of the constructor parameters. When I first started to learn Python, I understood that the self is just a reference to the calling object, and each object will have its own self. Each object we create is unique and therefore the self-keyword must belong to only one object.

Please check the following figure:



I hope you now understand why the constructor is called when we instantiate the object. Please don't get scared by my Picasso drawing. The constructor can be much more complex, for example, let's allow new objects to be instantiated from the Person Class if they meet a certain condition like an age limit. For example, let's add a condition in the constructor so it will check the age before we create the actual object. Have a look at the complete code:

```
class Person:
    # Class Object Attributes
    is_person = True

    # Constructor
    def __init__(self, name, lastname, age):
        if age >= 18:
            self.name = name # name attribute
            self.lastname = lastname # lastname attribute
            self.age = age # age attribute

    # Method
```

```
def greet(self):
    print(f'{self.name}, '
          f'{self.lastname}, old: {self.age}.'
          f'Called by the class itself: {Person.is_person}.'
          f'Called by the self keyword {self.is_person}')
```

```
person_obj1 = Person('Jason', 'Brooks', 17)
print(person_obj1.greet())
```

Because of the if-condition inside the **init** constructor, we will never instantiate the person\_obj1 as it is because the current age argument that we pass is 17. This will make the condition be evaluated to Boolean False. If we try to run this code, we will get **AttributeError**:

```
<__main__.Person object at 0x10b6cbdc0>
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects
  /firstproject/first.py", line 22, in
    <module>
      print(person_obj1.name)
AttributeError: 'Person' object has no
attribute 'name'
```

Very good, now if we change the age to 18 or bigger, the if condition will be evaluated to true and the object will be successfully created:

```
person_obj2 = Person('Andy', 'Garcia', 20)
print(person_obj2.greet())
```

The above code will print this out:

```
Andy, Garcia, old: 20. Called by the class itself: True. Called by the self keyword True
```

## Constructor with default values

We know the functions can have default values for their parameters, in case the calling object misses out on any of the values. The constructor is a function, a special function, you probably know this because of the ‘def’ keyword in front, so it can have default values as well:

```
# Constructor
def __init__(self, name='John', lastname='Doe', age=18):
    if age >= 18:
        self.name = name # name attribute
        self.lastname = lastname # lastname attribute
        self.age = age # age attribute
```

Now, this allows us to create objects or instances from the Person Class calling the constructor like this:

```
person_obj3 = Person()
print(person_obj3.name)
```

The output from the code above will be John even though we didn’t pass any arguments to the Person() constructor. The default values we have in our code are called safeguards so we can have proper control of how we instantiate Objects from a Class.

## Create Class Methods using **@classmethod**

In this section, we will create a Class method that belongs to the Class. To achieve this, we need to use the **@classmethod** keyword which is known as a **decorator**. We need to use the decorator before the function declaration:

```
# Class Method
@classmethod
def date_created(today_date,year):
    print(today_date,year)
```

As we can see, before the method declaration, we used the `@classmethod` decorator and then the rest of the method syntax is the same as before. Can this method be accessed by the Objects of the Person Class? Let's try to run the following code:

```
person_obj3.date_created('14/05',2023)
```

This will cause the following error:

```
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects
  /firstproject/first.py", line 32, in
    <module>
      person_obj3.date_created('14/05',
                                2023)
TypeError: Person.date_created() takes 2
           positional arguments but 3 were given
```

The error says we supplied 2 parameters but 3 were given. If you remember, each method inside the Class should have the `self`-keyword as a first parameter, but here we are missing this keyword. The Class method does not belong to the Object therefore the '`self`' keyword is no longer needed but we need to use a keyword that will tie this method to the Class. Therefore, we use the '`cls`' which stands for class and is standard nowadays. The Class method must receive the `class` keyword as a first argument:

```
# Class Method  
@classmethod  
def date_created(cls,today_date,year):  
    print(today_date,year)
```

Now the object can call this method:

```
person_obj3.date_created('11/11', 2023)
```

So why do we need this kind of method when the object can access it just like the rest of the methods? Well, this kind of method belongs to the class itself so you are not required to instantiate the class in order to use it:

```
# the class can call this method without objects  
Person.date_created('12/12', 2023)
```

This will still print the output because the method is a class method and doesn't require an Object to access it. These types of methods are rare in Python but it is good to know them.

To summarize what we have learned about the Class method:

- the class method is bound to the class and not to the object
- the class method has access to the class and its state like all of the attributes of that class
- It can access and modify the class states and attributes

## Static method `@staticmethod`

The static method is very similar to the class method but the difference is that we don't have to use the class implicit keyword 'cls'. The static method does not have access to the Class state and therefore cannot modify the Class. So why do we need these static class methods then? Well, these methods can be used as utility-type methods that can take parameters and perform some action. To create a static method, we need to use the decorator `@staticmethod`:

```
# Static Method  
@staticmethod  
def is_adult(age):  
    return age >= 18
```

```
print(Person.is_adult(19))
```

This will return true because the age (19 >=18) which is evaluated to True:

## 1<sup>st</sup> Pillar of OOP - Encapsulation

Encapsulation is one of the four most important principles of the OOP. We have already covered this concept but now hopefully the definition will make sense. Encapsulation refers to binding the data with the functions and methods that operate or manipulate that data. The data will be the attributes and the functions will be the methods. In our example, we have a Class called Person that encapsulates the data which will be the attributes like name and age to the functions or methods that manipulate that data. You can understand the encapsulation as a package because it holds the data and the functions enclosed in one single box:

```
#encapsulation
class Person:

    # Constructor
    def __init__(self, name='John', lastname='Doe', age=18):
        if age >= 18:
            self.name = name # name attribute
            self.lastname = lastname # lastname attribute
            self.age = age # age attribute

    # Method
    def greet(self):
        print(f'The data and functions are encapsulated, \n'
              f'My name is: {self.name}, \n'
              f'My last name: {self.lastname}, \n'
              f'and I\' old:{self.age} ')

person_obj1 = Person('Jason', 'Mamoa', 53)
person_obj1.greet()
```

## 2<sup>st</sup> Pillar of OOP – Abstraction

Abstraction means hiding information and only giving access to what is necessary. Abstraction is the second most important principle that we will cover in OOP. In other words, we are hiding the internal implementation of the methods from outside users by only providing access to limited data. Take a look at the following Object that accesses the greet() method:

```
# Method
def greet(self):
    print(f'The data and functions are encapsulated, \n'
          f'My name is: {self.name}, \n'
          f'My last name: {self.lastname}, \n'
          f'and I\' old:{self.age} ')
```

💡

```
person_obj1 = Person('Jason', 'Mamoa', 53)
person_obj1.greet()
```

```
The data and functions are encapsulated,
My name is: Jason,
My last name: Mamoa,
and I' old:53
```

The Object **person\_obj1** does not know how the `greet()` method is implemented in the Class `Person`, so that information is hidden but the Object gets an output which means it gets only the essential data the method `greet()` provides. The `person_obj1` knows what the `greet()` function does and it can get the output but it doesn't know the internal implementation.

## Python Private vs Public Variables

In this section, we will discuss what private and public variables are, what they mean, and how can we use them in our code. Please check out the following code:

```
# private and public variables
class Person:
```

```
# Constructor
def __init__(self, name='John', lastname='Doe', age=18):
    if age >= 18:
        self.name = name
        self.lastname = lastname
        self.age = age

# Class method
def greet(self):
    print(f'The data and functions are encapsulated, \n'
          f'My name is: {self.name}, \n'
          f'My last name is: {self.lastname}, \n'
          f'and I\'m {self.age} old.')

person_obj1 = Person('Jason', 'Mamoa', 53)
person_obj1.greet()
person_obj1.lastname = 'Bourne'
```

```
print('-----')
person_obj1.greet()
```

This is the output of the last greet method():

```
The data and functions are encapsulated,  
My name is: Jason,  
My last name is: Mamoia,  
and I'm 53 old.
```

---

```
The data and functions are encapsulated,  
My name is: Jason,  
My last name is: Bourne,  
and I'm 53 old.
```

I know you didn't expect this outcome because I have used an instance of the class Person and changed the value of the lastname attribute. This is totally opposite to the second pillar of OOP called abstraction because this principle holds that Object should not be able to access and modify data but to get data/information back. In Java, we can create private variables and they will remain private for the instances. This is a nice concept in Java and all we need to do is to write the private keyword in front:

```
# Private  
private String name;  
private String lastname;  
private int age;
```

But we cannot do the same thing in Python. In Python, when we want to say that this attribute or variable is private, we need to write an underscore in front:

```
self._name = name # name attribute  
self._lastname = lastname # lastname attribute  
self._age = age # age attribute
```

This is not the exact same thing that we have in Java. It will not throw an error if we try to run the above code again because the value of lastname will be changed, so what is the point of writing underscore then? Well, the point is that when other programmers see this naming convention, they will understand that we are trying to say this attribute is private and we should not try to change its value. All of the other variables are considered public but the ones with an underscore are considered private. This is how we can mimic the privacy in Python - by adding an underscore in front, but if someone tries to modify it, it will still go through. In Python, we cannot achieve true privacy like in other OOP languages but at least we have a naming convention that uses a single underscore before the attribute name. When we use `_lastname` we indicate that this is a private variable/attribute in Python. For example, the method `greet()` now must also include the underscore in front of the variables so it can work without any errors:

```
# Method
def greet(self):
    print(fThe data and functions are encapsulated, \n'
          fMy name is: {self._name},\n'
          fMy last name: {self._lastname},\n'
          fand I\' old:{self._age} ')
```

## 3<sup>rd</sup> Pillar of OOP – Inheritance

Inheritance is one of the core principles of OOP. The inheritance allows us to create a class that will inherit methods and properties from another class. The class that is being inherited from is called a Parent class or base class and the class that inherits the methods and properties from the parent class is called a child class or derived class. So far, we have used the Person class as an example to explain some of the principles of OOP. The Person class has name, lastname, and age as properties, and one method called `greet` to print those properties. In order to show you inheritance, I will need to create another class called Student that will inherit the functionality from the parent class Person. Let's see how we can achieve this in Python:

```

# inheritance
class Person:
    # Constructor
    def __init__(self, name='John', lastname='Doe', age=18):
        if age >= 18:
            self._name = name # name attribute
            self._lastname = lastname # lastname attribute
            self._age = age # age attribute

    # Method
    def greet(self):
        print(f'The data and functions are encapsulated, \n'
              f'My name is: {self._name},\n'
              f'My last name: {self._lastname},\n'
              f'and I\'m old:{self._age} ')
print('----- Person Class -----')

person_obj1 = Person('Jason', 'Mamoa', 53)
person_obj1.greet()

class Student(Person):
    pass

print('----- Student Class -----')

student_obj1 = Student('Rick', 'Jameson', 33)
print(student_obj1.greet())

```

Output:

----- Person Class -----

The data and functions are encapsulated,  
My name is: Jason,  
My last name: Mamoia,  
and I' old:53

----- Student Class -----

The data and functions are encapsulated,  
My name is: Rick,  
My last name: Jameson,  
and I' old:33

This is the syntax for inheritance in Python:

```
class Student(Person):
```

The Parent class should be included in the brackets of the child class and that is all. In this case, the Student class doesn't have its own properties and methods but it can have new methods and they will not be shared with the parent class. This means that the child class can inherit everything from the parent class and can have its own attributes and methods. We use the pass keyword whenever we don't want to include any code. Let's discuss the following code:

```
student_obj1 = Student('Rick', 'Jameson', 33)  
print(student_obj1.greet())
```

As you can see, the student class was able to use the parent class Person \_\_init\_\_ constructor method to initialize the name, lastname, and age and after that to use the print method so it can print the details of the student\_obj1. So far, our child class inherits all of the properties/attributes

and methods from the parent class but we can add a new constructor for the child class as well that will have its own attributes. The common attributes between a Person and a Student are:

- name
- lastname
- age

We can add the following new attributes for the Student class:

```
class Student(Person):  
    def __init__(self, name, lastname, age, dob, address):  
        Person.__init__(self, name, lastname, age)  
        self.dob = dob  
        self.address = address
```

From the code above, the Student constructor will have all of the attributes from the parent class Person plus two new, the dob (date of birth) and address. Instead of writing the self.name = name for all of the parent attributes, we can call the Person init constructor and include the parent attributes directly in the Student class:

```
Person.__init__(self, name, lastname, age)
```

For the Student class, we add self.dob = dob and self.address = address because they are the new attributes that belong to the student class and they are not inherited from any other class. The student object now will need to include the dob and address attributes:

```
student_obj1 = Student('Rick', 'Jameson', 33,  
'29/09/1987', 'Melbourne/Australia')
```

If there was no inheritance, the Student class and Person class would have looked like this:

```
class Person:  
    # Constructor  
    def __init__(self, name='John', lastname='Doe', age=18):  
        if age >= 18:
```

```

        self._name = name # name attribute
        self._lastname = lastname # lastname attribute
        self._age = age # age attribute

class Student:
    def __init__(self, name, lastname, age, dob, address):
        if age >= 18:
            self._name = name # name attribute
            self._lastname = lastname # lastname attribute
            self._age = age # age attribute
        self.dob = dob
        self.address = address

```

As you can see, we are not following the DRY (Do Not Repeat Yourself) principle because we are writing the same code for the same attributes twice. Imagine we have a few more classes like Undergraduate and Postgraduate, then we would need to repeat the same code four times. I hope you understand why inheritance is so important in OOP languages.

## Method overriding

Method overriding happens when we want to use the same method name in the child class but we want the output to be different from the parent class method. Both methods need to have the same name. For example, in our Parent class, we have the greet() method, which is suitable and works perfectly for the Person class but although the child class Student can access it, it cannot use it to print its own two new attributes. Here is the greet() method from the Parent class:

```

# Method
def greet(self):
    print(f'The data and functions are encapsulated, \n'
          f'My name is: {self._name},\n'
          f'My last name: {self._lastname},\n'
          f'and I\' old:{self._age} ')

```

The instantiated student\_obj1 from the Student class can access this method, thanks to the inheritance like this:

```
student_obj1 = Student('Rick', 'Jameson', 33,  
'29/09/1987', 'Melbourne/Australia')  
  
print(student_obj1.greet())
```

The output will be:

```
----- Student Class -----  
The data and functions are  
encapsulated,  
My name is: Rick,  
My last name: Jameson,  
and I' old:33
```

Great! It works but what about the dob and address attributes that we have only in the Student, we want them to be included in the output. This is where method overriding comes into action. We can use the same method name greet() and create it for the Student class (we should write this method in the Student class):

```
# Method overriding  
def greet(self):  
    print(f'My name is: {self._name},\n'  
        f'My last name: {self._lastname},\n'  
        f'and I'm {self._age}. I'm student: \n'  
        f'born at: {self.dob},\n'  
        f'that lives in : {self.address}')
```

The output will be:

```
----- Student Class -----  
My name is: Rick,  
My last name: Jameson,  
and I' old:33. I'm student:  
born at: 29/09/1987,  
that lives in : Melbourne/Australia
```

Perfect! Now we have two methods with the same name in the two classes. When an instance of person class is created and called the greet() method, the method created in the Person class will be called but when an instance of the child class is created and called the greet() method then the Interpreter will get the method from the student class.

Here is the entire code:

```
# inheritance  
class Person:  
    # Constructor  
    def __init__(self, name='John', lastname='Doe', age=18):  
        if age >= 18:  
            self._name = name # name attribute  
            self._lastname = lastname # lastname attribute  
            self._age = age # age attribute  
  
    # Method  
    def greet(self):  
        print(f'The data and functions are encapsulated, \n'  
              f'My name is: {self._name},\n'  
              f'My last name: {self._lastname},\n'
```

```

fand I\n old:{self._age} ')

print('----- Person Class -----')

person_obj1 = Person('Jason', 'Mamoa', 53)
person_obj1.greet()

class Student(Person):
    def __init__(self, name, lastname, age, dob, address):
        Person.__init__(self, name, lastname, age)
        self.dob = dob
        self.address = address

    # Method overriding
    def greet(self):
        print(f'My name is: {self._name},\n'
              f'My last name: {self._lastname},\n'
              f'and I\n old:{self._age}. I\n'm student: \n'
              f'born at: {self.dob},\n'
              f'that lives in : {self.address}')

```

```

print('----- Student Class -----')

student_obj1 = Student('Rick', 'Jameson', 33, '29/09/1987',
'Melbourne/Australia')

print(student_obj1.greet())

```

## **isinstance() function – Python**

Before we start learning about the new `isinstance()` function in Python, I would like to give you another inheritance example and repeat some of the important features we have learned so far about classes in Python. Classes, not just in Python but in any programming language, can be a difficult

concept to understand at first, and this is one of the reasons why I wanted to summarize. Let's start from the basic building blocks of a class. Each class can have attributes and methods. The number of attributes and methods depend on what the class is about, this means it can have many attributes or no attributes at all. The class important method is the constructor method `__init__`. In this method, the first parameter is called 'self' and the rest of the parameters are the class attributes. The class can be instantiated, meaning we can create object/objects from that class. The process of creating a new object is called instantiation. Therefore, the objects are instances of a class. When we create an instance the constructor method is called immediately. But how does the class know which object was instantiated and how to initialize the attributes to values? The first parameter of the constructor is going to tell the class which object is currently in use. The methods inside the classes are nothing but functions, but we do not call them functions because these methods can be used internally for the class itself or by its objects/instances. The instance/object of the class can access all of the attributes and methods. The class can have its own class attributes and methods and they are different from the normal attributes and methods because we don't need to instantiate the class in order to use them. The class itself can call them directly. Class inheritance is a very important pillar of OOP because it allows us to use a class and create a child class that can inherit all of the attributes and methods from that parent class. This saves time because we are not creating code duplicates. In some literature, you will find that the child class is also called:

- Subclass
- Derived class
- Concrete class

The parent class or the class that other classes inherits can be known as:

- Parent
- Super
- Abstract class

I hope now everything is starting to make sense and here is another example of inheritance:

```

# inheritance

class Mammal:
    def __init__(self, name):
        self.name = name

    # eat method
    def eat(self):
        print(f'{self.name} eats different types of foods!')

    # walk method
    def walk(self):
        if self.name != 'bat':
            print(f'{self.name} can walk!')
        else:
            print(f'The {self.name} is the only mammal that can fly!')

print('***** Mammal class instances output! *****')
mammal_obj1 = Mammal('Elephant')
mammal_obj1.eat()
mammal_obj1.walk()
mammal_obj2 = Mammal('bat')
mammal_obj2.eat()
mammal_obj2.walk()

class Dog(Mammal):
    def __init__(self, name, breed, legs):
        Mammal.__init__(self, name)
        self.breed = breed
        self.legs = legs

    # eat method overriding
    def eat(self):
        print(f'{self.name} eats only dog food!')

```

```

# details method - unique for the class Dog
def details(self):
    print(f'The {self.name} is a {self.breed} \n'
        f'and like all dogs have {self.legs}-legs!')

print('***** Dog class instances output! *****')
dog_obj1 = Dog('Benn', 'labrador', 4)
dog_obj1.eat()
dog_obj1.walk()

```

The output:

```

***** Mammal class instances output! *****
Elephant eats different types of foods!
Elephant can walk!
bat eats different types of foods!
The bat is the only mammal that can fly!
***** Dog class instances output! *****
Benn eats only dog food!
Benn can walk!

```

As you can see in the above inheritance example, we have the Mammal class which is the parent and the Dog class is the one that inherits. We have used the inheritance so the Dog class can inherit all of the attributes and methods like eat and walk from its parent Mammal class. In the Dog class, we also have method overriding because the ‘eat’ method from the parent class wasn’t suitable for use on Dog instances. The Dog class can also have its own method/methods like ‘details’. This method cannot be accessed by the Mammal class instances.

## **isinstance()**

This is a built-in function in Python that allows us to check if something is an instance of a class.

Here is the syntax:

```
isinstance(instance, Class)
```

From the previous example, we can check if the `dog_obj1` is an instance of the `Dog` class:

```
print(isinstance(dog_obj1, Dog))
```

If we run this code, it will give us `True` because the `dog` object is an instance of the class `Dog`. But if we try to see if the `dog_obj1` is an instance of the `Mammal` class, what do you think will happen?

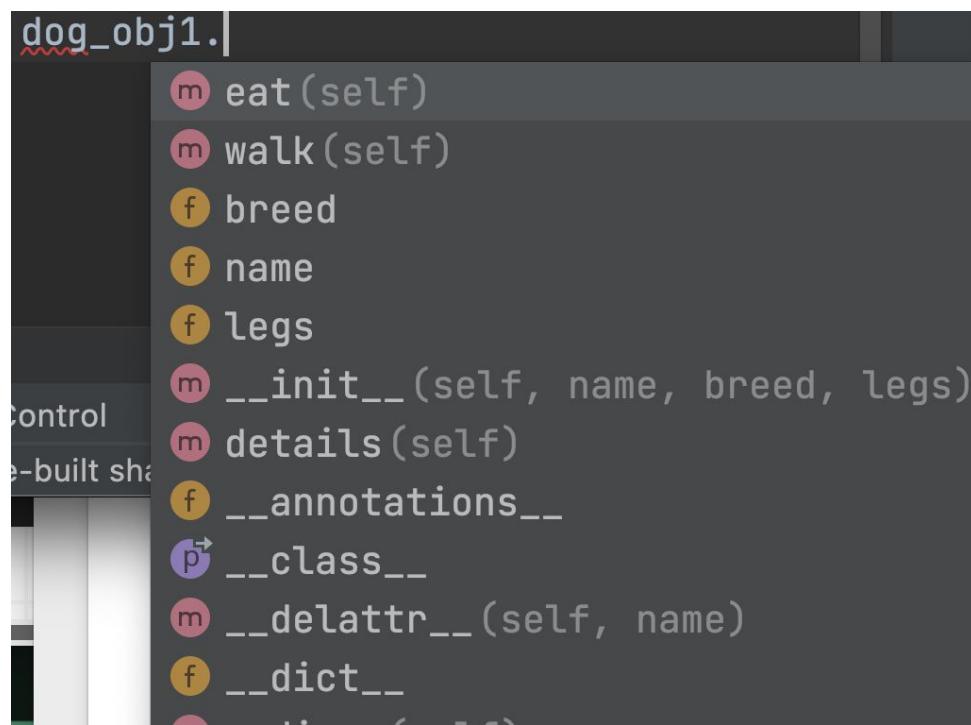
```
print(isinstance(dog_obj1, Mammal))
```

The output:

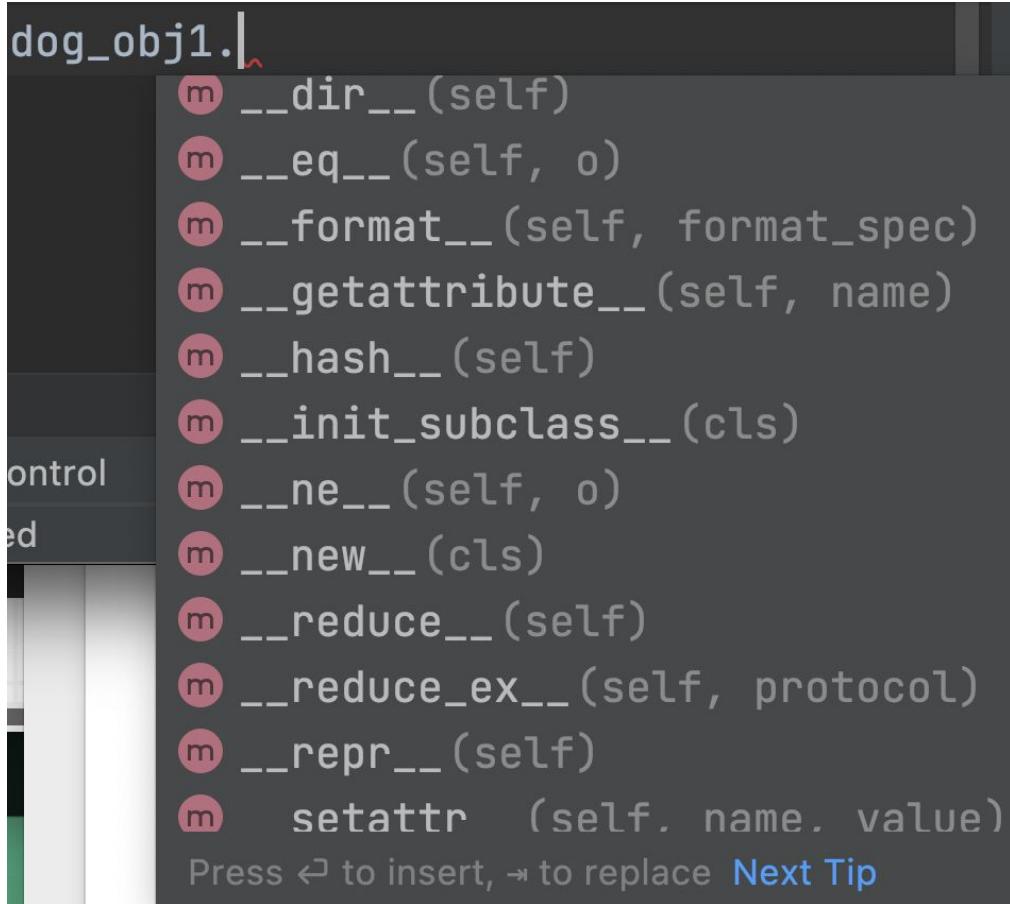
```
print(isinstance(dog_obj1, Dog))
print(isinstance(dog_obj1, Mammal))
```

True	True
------	------

This will print `True` because the `dog_obj1` is technically an instance of the `Mammal`. After all, the `Dog` class is the child class or subclass of the `Mammal` class. This is confusing but it is very logical because the `Dog` class is a subclass of `Mammal` and therefore every instance of the `Dog` class will be an instance of the `Mammal` class as well. If you remember, I mentioned that in Python, everything is an object. I can prove this if I type the name of the instance and dot like this:



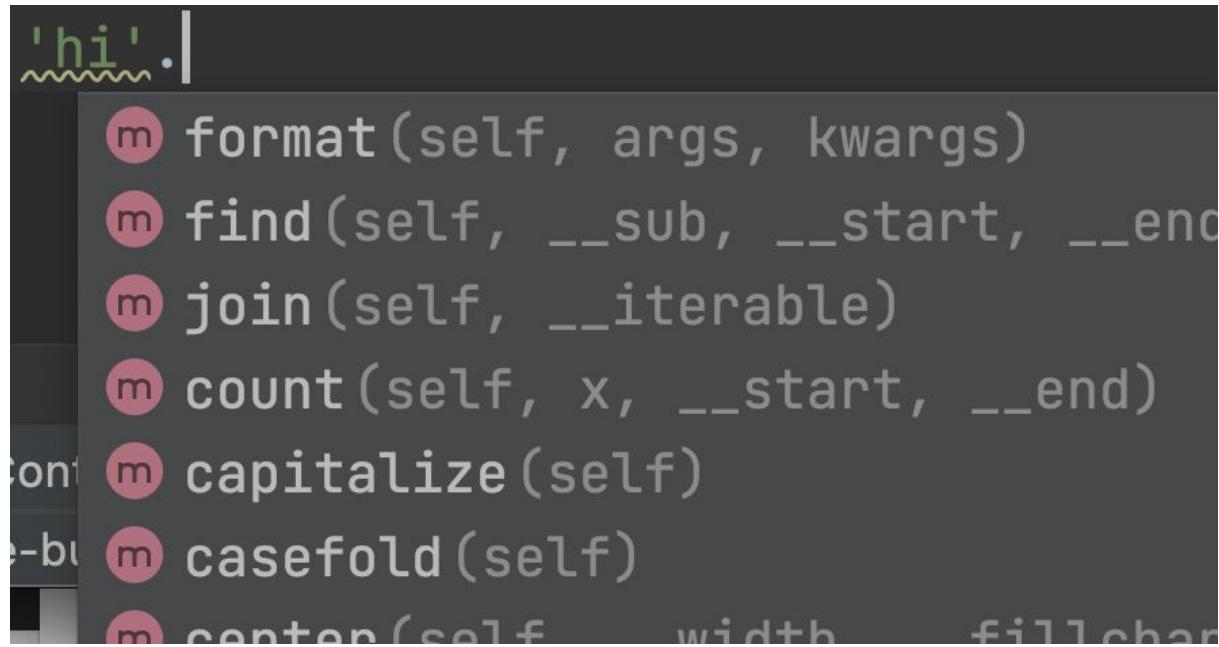
As you can see, the `dog_obj1` has access to all of the attributes and methods from the `Dog` and `Mammal` class because of the inheritance but if I scroll further, there will be more functions available to this `dog_obj1` instance:



Where are these functions coming from? Python, as we know, comes with pre-built methods and functions and in this case, the `dog_obj1` or any instance will inherit from the base `Object` class that Python comes with. This means that `dog_obj1` is an instance of `Dog` class, `Mammal` class, and `Object` class as well. If I try to print this code, the result must be True:

```
print(isinstance(dog_obj1, Dog))      True
print(isinstance(dog_obj1, Mammal))    True
print(isinstance(dog_obj1, object))    True
```

That is why the `dog_obj1` has access to all of the methods from the `Dog`, `Mammal`, and `Object` base classes. The methods are also accessible for Python Strings, Lists, Dictionaries, etc:



A screenshot of a Python code editor showing the string class methods. The code editor has a dark theme with syntax highlighting. The string 'hi' is typed, followed by a dot and a list of methods. The methods listed are: format(self, args, kwargs), find(self, \_\_sub, \_\_start, \_\_end), join(self, \_\_iterable), count(self, x, \_\_start, \_\_end), capitalize(self), casefold(self), center(self, width, fillchar). The method names are preceded by a magenta circle icon.

```
'hi'.|
    m format(self, args, kwargs)
    m find(self, __sub, __start, __end)
    m join(self, __iterable)
    m count(self, x, __start, __end)
cont m capitalize(self)
e-bu m casefold(self)
    m center(self, width, fillchar)
```

Now we know that although the Mammal class is the first/parent class, it also inherits from the base Object class as well, therefore we can even write it like this:

```
class Mammal(object):
```

## 4<sup>th</sup> Pillar of OOP – Polymorphism

This pillar has a very long and unusual name because it comes from the Greek word that means many forms (poly – many, morphism – forms). Polymorphism refers to a function or a class method that has the same name but is being used in different ways and in different scenarios. Polymorphism is one of the most fundamental pillars of OOP. In the previous parent-child class example, we saw that different classes can share their methods, therefore the method 'eat' is shared between the two classes but each of them does something different with this method. Same method name, but different functionality and output:

```
***** Mammal class instances output! *****
Elephant eats different types of foods!
***** Dog class instances output! *****
Benn eats only dog food!
```

Another simple example of polymorphism is when we use the built-in `len()` function in Python. These functions can be called/used by many different Python data types:

```
print('***** len() function polymorphism example! *****')
str_obj = "Polymorphism"
list_obj = ["Java", "Python", "Ruby", "C#", "PHP", "C++"]
dict_obj = {
    "brand": "Tesla",
    "model": "Funny names :)",
    "year": 2023
}
print(len(str_obj))
print(len(list_obj))
print(len(dict_obj))
```

Output:

```
, ,
***** len() function polymorphism example! *****
12
6
3
```

This proves that the `len()` function works on different data types like strings, lists, and dictionaries and it gives us result, so the `len()` function actually takes many shapes. Another way to demonstrate polymorphism is to use the for loop and call the same method but using a different class instance:

```
animals = [mammal_obj1, dog_obj1]
```

```
for animal in animals:  
    animal.eat()
```

Both objects in the lists belong to the Mammal-Dog inheritance example and the result will be:

```
Elephant eats different types of  
foods!  
Benn eats only dog food!
```

From the code above, we can see that we have two different outputs because in the for loop, this is what is happening:

```
mammal_obj1.eat()  
dog_obj1.eat()
```

## super() function

In order to explain the **super()** function, I would like to give you an example where we have two classes - Person and Employee. If you want you can try and write the code yourself based on the description below, but please don't get too hard on yourself if you can't do it.

### Description:

The program should have two classes. The first class will be called Person and the second class will be called Employee. You will need to guess their relationship so you can write the inheritance correctly. The person class will have the following attributes:

name

`dob` (date of birth)  
`id_number`

In the person class, you should have the `info()` method that will print all of the Person details in a new line. The class Employee should be able to use all of the attributes and methods from the parents class but it should also have new attributes:

`salary`  
`position`

The employee class will have its own `info()` method with the same name as from the Person class so it can print the parent class details plus the new Employee details as well (method overriding). In the end, you need to create an instance of the class Person and call the info method then create another instance of the Employee class and call the `info()` method.

Here is the final code:

```
# Parent class
class Person(object):

    def __init__(self, name, dob, id_number):
        self.name = name
        self.dob = dob
        self.id_number = id_number

    def info(self):
        print(f'name: {self.name}\n'
              f'dob: {self.dob}\n'
              f'id: {self.id_number}')


# child class
class Employee(Person):
    def __init__(self, name, dob, id_number, salary, position):
        # invoking the __init__ of the parent class
        Person.__init__(self, name, dob, id_number)
```

```
self.salary = salary  
self.position = position  
  
def info(self):  
    print(f'Name: {self.name}\n'  
        f'DOB: {self.dob}\n'  
        f'ID: {self.id_number}\n'  
        f'Salary: {self.salary}\n'  
        f'Position: {self.position}')
```

```
# Instance of Person class  
print('# Instance of Person class')  
person_obj = Person('Andy', '29/09/1999', 1334524)  
person_obj.info()
```

```
# Instance of Employee class  
print('# Instance of Employee class')  
employee_obj = Employee('James', '29/09/1999', 1334524, 7000,  
'accountant')  
employee_obj.info()
```

As you can see in the Employee class, we call the `__init__` constructor directly from the Person class so we don't need to write all of the attributes Person has in the Employee constructor again. This will be wasting time and unnecessary code repetition. But this is one way of doing it, and there is another way that require us to use the `super()` function. The `super()` function needs to be called in the child class and in our case the Employee class. The `super()` function doesn't even require the 'self' keyword:

```
class Employee(Person):  
    def __init__(self, name, dob, id_number, salary, position):  
        super().__init__(name, dob, id_number)
```

The `super()` is called super because it refers to the parent class (Person) and the parent classes are also known as super or abstract classes. This is how we can use the `super()` to link up the parent and child classes

# Code introspection in Python

Introspection allows us to determine the type of object during the runtime. As we know, everything in Python is considered an Object and every object has attributes and methods. We also learned that the parent class inherits from the base class known as Object. In Python, we have a built-in function that we can use for code introspection. The first function is a well-known function that we have used so many times so far:

## Type()

```
# 1 type()
str_obj = 'Hi'
list_obj = [1, 5, 7]
int_obj = 10
float_obj = 10.3
dict_obj = {
    "type": "1"
}

print(type(str_obj))
print(type(list_obj))
print(type(int_obj))
print(type(float_obj))
print(type(dict_obj))
```

Output:

```
<class 'str'>
<class 'list'>
<class 'int'>
<class 'float'>
<class 'dict'>
```

## dir()

this function will return the list of methods and attributes associated with the object

```
print(dir(dict_obj))
```

The output:

```
['__class__', '__class_getitem__', '__contains__',
 '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__ior__', '__iter__', '__le__', '__len__', '__lt__',
 '__ne__', '__new__', '__or__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__ror__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
 'popitem', 'setdefault', 'update', 'values']
```

## str()

This function will convert everything to a string

```
# 3 str()
list_obj1 = [1, 2, 4]
print(type(list_obj1))
# convert to str
print(type(str(list_obj1)))
```

Output:

```
<class 'list'>
<class 'str'>
```

## id()

This function will return an id of an object

```
# 4 id()
m = [1, 2, 3, 4, 5]
```

```
# print id of m
print(id(m))
```

Output:

```
4478974848
```

Other useful code introspection methods are:

```
help()
hasattr()
getattr()
repr()
callable()
isinstance
issubclass
__doc__
__name__
```

You can read more about them if you open the following link from the Python documentation:

<https://docs.python.org/3/library/functions.html>

<b>Built-in Functions</b>			
<b>A</b> abs() aiter() all() any() anext() ascii()	<b>E</b> enumerate() eval() exec()	<b>L</b> len() list() locals()	<b>R</b> range() repr() reversed() round()
<b>B</b> bin() bool() breakpoint() bytearray() bytes()	<b>F</b> filter() float() format() frozenset()	<b>M</b> map() max() memoryview() min()	<b>S</b> set() setattr() <u>slice()</u> sorted() staticmethod() str() sum() super()
<b>C</b> callable() chr() classmethod() compile() complex()	<b>H</b> hasattr() hash() help() hex()	<b>N</b> next()	<b>T</b> tuple() type()
<b>D</b> delattr() dict() dir() divmod()	<b>I</b> id() input() int() isinstance() issubclass() iter()	<b>P</b> pow() print() property()	<b>V</b> vars()
			<b>Z</b> zip()
			<b>-</b> <u>__import__()</u>

## Dunder/Magic methods

Dunder or magic methods are very important in Python. We have already briefly mentioned them but in this section, we will talk about them in more detail. One example of the dunder method we have used is the `__init__` method or known as constructor. The `__init__` is a magic method because it will be called automatically when an instance(object) from that class is created. Luckily for us, Python provides many dunder/magic methods like the `len()`, `print()`, and `[]`. They are all dunder methods. We have seen from the previous section that when we use the `dir()` function on a class instance, the output will be a list of methods inherited from the base Object class:

```
print(dir(person_obj))
```

Output:

```
['__class__', '__class_getitem__', '__contains__',
 '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__ior__', '__iter__', '__le__', '__len__', '__lt__',
 '__ne__', '__new__', '__or__', '__reduce__', '__reduce_ex__',
 '__repr__', '__reversed__', '__ror__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
 'popitem', 'setdefault', 'update', 'values']
```

We said that double underscore is what makes these methods be considered as dunder or special but the `len()` function does not have underscores and I stated it is a dunder method. Well, the `len()` is actually implemented with the help of the dunder methods. The `len()` function will give us the length, for example, if we have a list like this:

```
list_obj = [1, 3, 5, 7, 9]
```

We can use the `len()` function to get the list length:

```
print(len(list_obj))
```

The result will be 5 because we have five items in the list above. In the background, the len() is implemented as a dunder method and Python allows us to use this kind of method directly on the object or instances of our classes. I will create a class Car and one instance just to show you how we can call some of the dunder methods from the figure above:

```
# car class
```

```
class Car(object):
    def __init__(self, color, make, year, size):
        self.color = color
        self.make = make
        self.year = year
        self.size = size
```

```
bmw_car = Car('red', 'bmw', 2022, 'sedan')
```

Now we can call some of the dunder/magic methods on this class like the \_\_str\_\_() directly on the bmw\_car instance like this:

```
print(bmw_car.__str__())
```

The output will be:

```
<__main__.Car object at 0x10976bd30>
```

The \_\_str\_\_() dunder method will return a str version of the object. The str is the built-in string class. But what do you think will happen if we use the built-in str() Python function instead of the dunder method?

```
print(str(bmw_car))
```

The output will be exactly the same, and why am I showing you this? Well, the dunder method \_\_str\_\_() allows us to use the Python built-in function

`str()` and avoid complications. If you want to read about the dunder methods, you can click on the following link but sometimes Python documentation can be overwhelming:

<https://docs.python.org/3/reference/datamodel.html#specialnames>

If you read only a few paragraphs from the link above, you will find out that we can perform basic customization on these methods and that is what I want to teach. In short, we can modify the existing dunder methods and create our own methods. The magic/dunder methods can help us override the functionality for the built-in functions and make them custom for specific classes. For example, let's modify the `__str__()` dunder method to return something instead of a memory location:

```
# car class

class Car(object):
    def __init__(self, color, make, year, size):
        self.color = color
        self.make = make
        self.year = year
        self.size = size

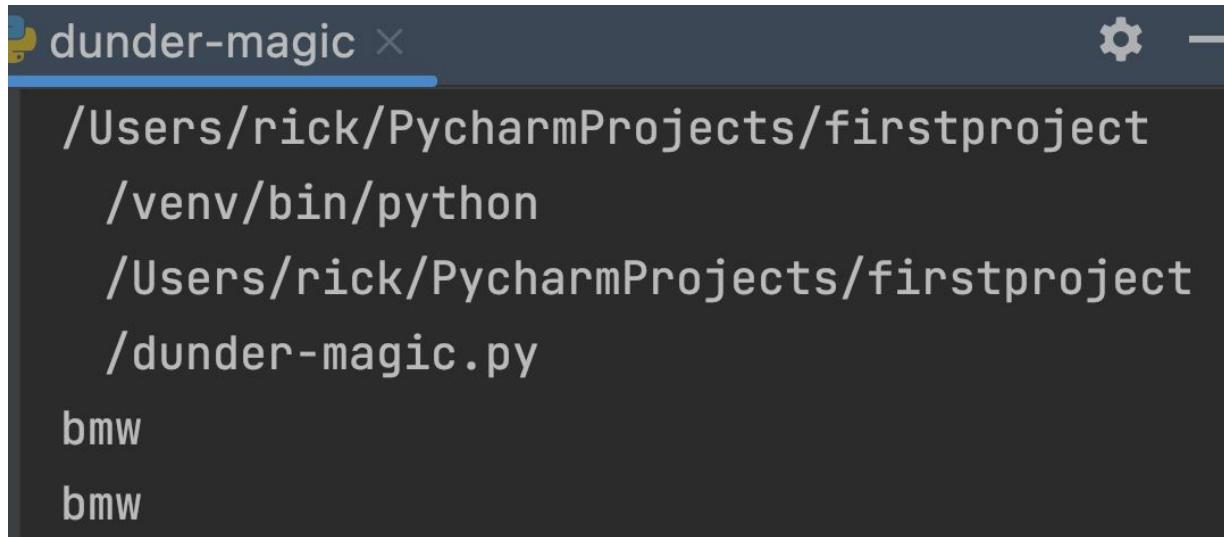
    def __str__(self):
        return f'{self.make}'
```

```
bmw_car = Car('red', 'bmw', 2022, 'sedan')
```

```
# call the dunder __str__ method:
print(bmw_car.__str__())

# the exact same as above is the built-in str() function
print(str(bmw_car))
```

If we run the same code now, the output should change:



```
dunder-magic ×
/Users/rick/PycharmProjects/firstproject
/venv/bin/python
/Users/rick/PycharmProjects/firstproject
/dunder-magic.py
bmw
bmw
```

As you can see, we used the dunder methods to change what the method returns and we can override the functionality completely. You should not modify the dunder methods but you should note that these modifications are not going to work on every class because they are not permanent and they are not global. In our case, we haven't changed the Python method `__str__()` completely. It will be changed just for the 'Car' class and the instances of that class. This means that if I have another class and create an instance from that class then I try to use the dunder string method `__str__()`, the output will be the same as from the original Python implementation. Let us see another dunder method `__del__`. This method is very similar to the destructors in C++ and these destructors are used to destroy the object state. So, instead of destroying the variable or the object, let us just modify this `__del__` to print 'Deleted':

```
def __del__(self):
    print('Deleted')
```

And we can call it like this:

```
# call the del method
del bmw_car
```

```
# if we try to access the bmw_car object an error
print(bmw_car)
```

Output:

```
Deleted  
Traceback (most recent call last):  
  File "/Users/rick/PycharmProjects/chapter4/  
    /dunder_magic_methods.py", line 41, in  
      <module>  
        print(bmw_car)  
NameError: name 'bmw_car' is not defined
```

It will still run the destructor at the end of the program and it will delete all of the references to the object. The `del` is usually not that popular in Python and you can run into some problems if you are using it because it will not delete just a single object or variable but all of the references connected to that object or variable. But it is always nice to know it exists. There is another dunder method called `delete`. The `__del__` and `__delete__` are both dunder methods in Python. The `__delete__` will delete the attribute which is a descriptor and we can execute it when needed. It will not run at the end of the program like `__del__`.

```
# Calling __delete__  
def __delete__(self, instance):  
    print("Deleted")
```

Let us talk about another dunder method `__call__`. The call will give our class unique features:

```
def __call__(self):  
    return 'Called!'
```

Now our actual object `bmw_car` can be called using these brackets '()' , or the same way we call the function:

```
print(bmw_car())
```

The result will be:

```
5
bmw
bmw
Called!
Deleted
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/chapter4
    /dunder_magic_methods.py", line 48, in
      <module>
        print(bmw_car)
    NameError: name 'bmw_car' is not defined
```

Another dunder method that I would like to mention is `__getitem__()`. This method if used in a class will allow the instance of that class to use the square brackets [] and pass the index to get an item or a value from a list, dictionary, or tuple. So let us define another attribute inside our class that will be from the type dictionary:

```
class Car(object):
    def __init__(self, color, make, year, size):
        self.color = color
        self.make = make
        self.year = year
        self.size = size
        self.engine_dict = {
            'automatic': 'Yes',
            'manual': 'No'
        }
```

Now we can create the dunder `__getitem__()` method:

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

Finally, we can call the instance of the class in our case `bmw_car` like this:

```
# call the bmw_car instance using the square brackets
print(bmw_car['manual'])
```

This will give us the value of the key ‘manual’ which is ‘No’. I think you now understand that we can modify the dunder methods and customize what they return. It’s not the best idea to modify the dunder methods but sometimes our code logic requires us to do this. The dunder methods are indeed magic because they can help us change the way methods return something for specific classes. This is all about dunder methods but you can read the documentation if you want to learn more about them. In the next section, we will focus on another OOP feature called multiple inheritance.

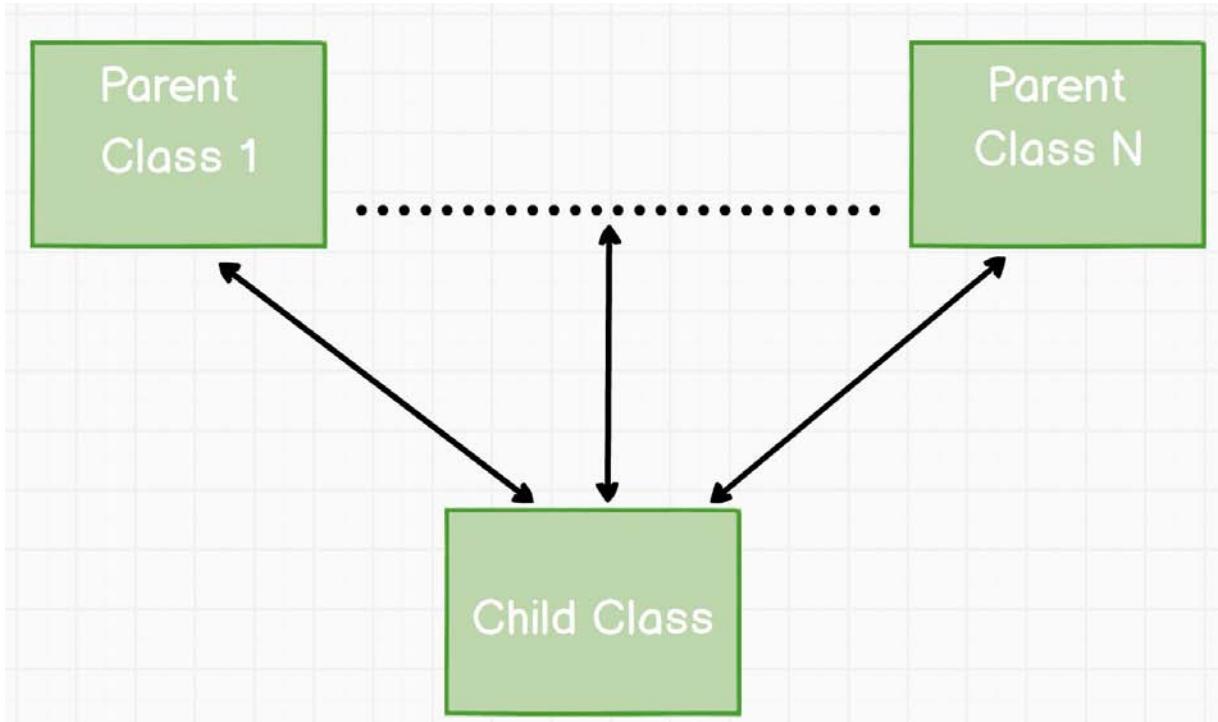
## Multiple Inheritance

The inheritance allows us to have a class known as child class that can inherit the properties and methods from another class. But there can be different types of inheritances:

- 1) Single inheritance
- 2) Multi-level inheritance
- 3) Multiple inheritance
- 4) Multipath inheritance
- 5) Hybrid Inheritance
- 6) Hierarchical Inheritance

In our case, we want to explore multiple inheritance, what it does, and what it means in Python. So far, we have done a single inheritance where the child class had only one parent/base class but the child class can have multiple

parents. The derived (child) class can inherit all of the properties and methods from multiple classes. Please take a look at the figure below:



We can have multiple parent classes and the child class can inherit all of the methods and properties defined in the parent classes. This is very powerful and not all of the programming OOP languages allow it. In the following example, we will have father and mother class that will be inherited by the child class:

```
# Multiple Inheritance
```

```
# father class
class Father(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def get_name(self):
        print(f'Name: {self.name}')
```

```

def get_age(self):
    print(f'Age: {self.age}')

class Mother:
    def __init__(self, name, eyes):
        self.name = name
        self.eyes = eyes

    def get_name(self):
        print(f'Name: {self.name}')

    def get_eyes(self):
        print(f'Color of eyes: {self.eyes}')

class Child(Father, Mother):
    def __init__(self, name, personality, gender):
        self.name = name
        self.personality = personality
        self.gender = gender

    def child_info(self):
        print(f'Name: {self.name}\n'
              f'Last name: {self.personality}\n'
              f'gender: {self.gender}\n')

# child object

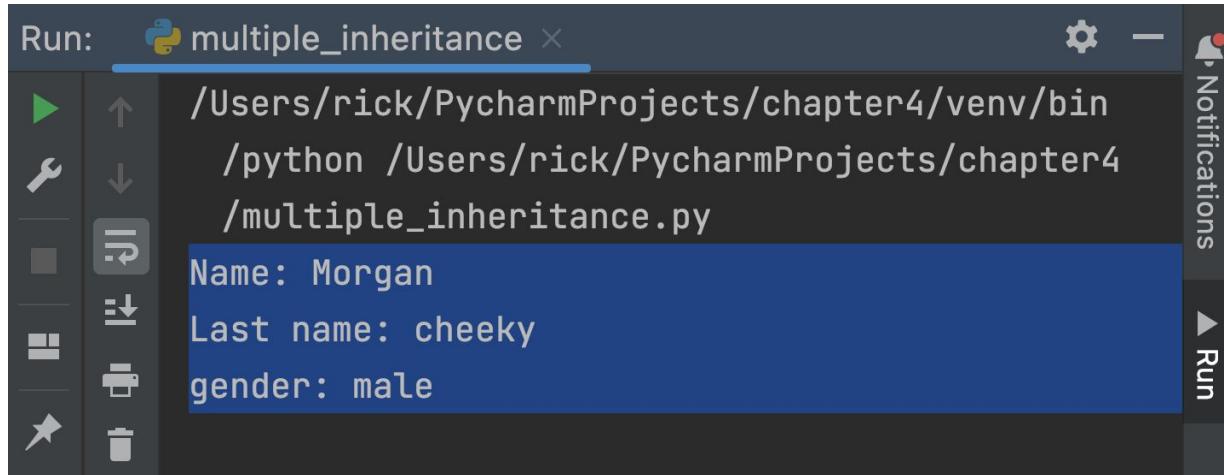
morgan = Child('Morgan', 'cheeky', 'male')
morgan.child_info()

```

As you can see, we can do multiple inheritance if we pass the names of the classes one after the other, separated with commas:

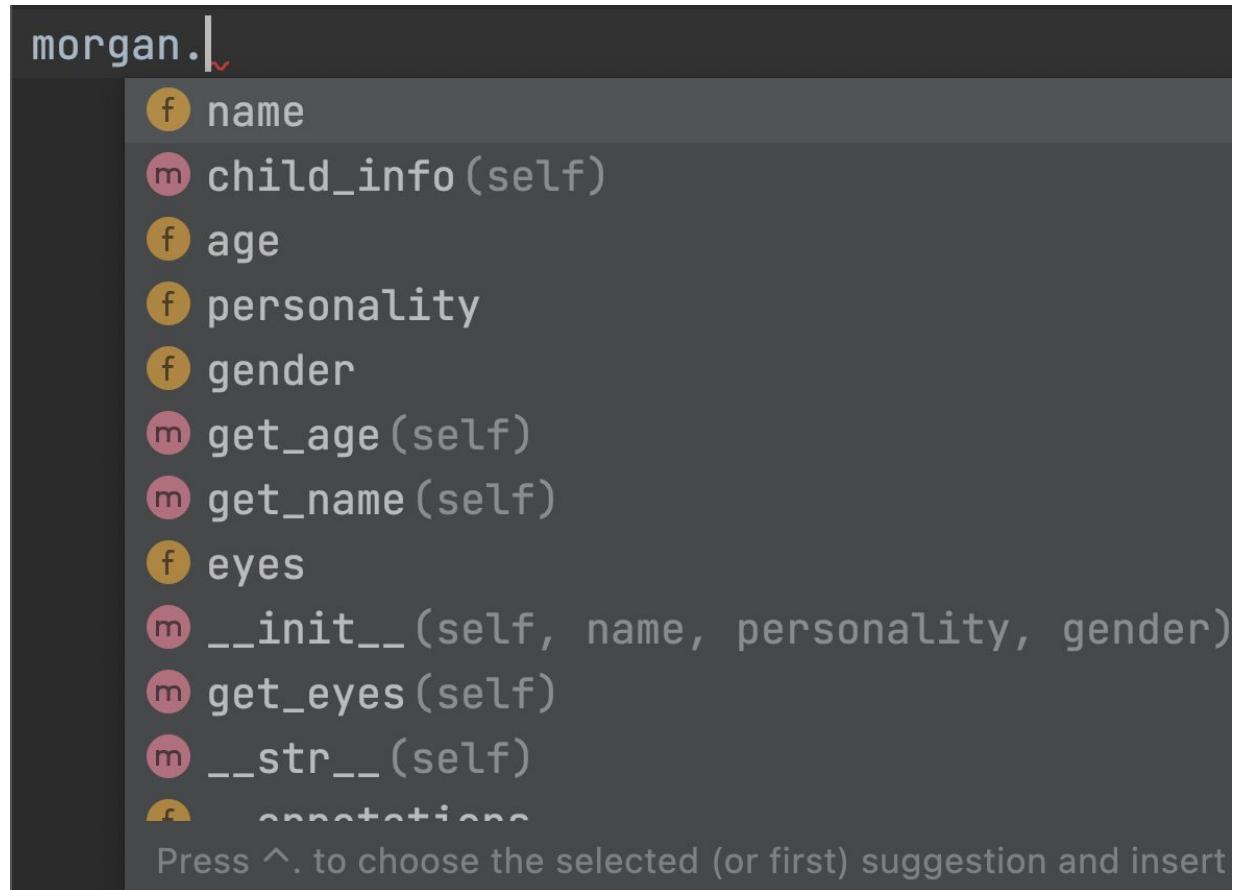
```
class Child(Father, Mother):
```

If we run the following code, the ‘morgan’ instance will call its own method ‘child\_info’:



```
Run: multiple_inheritance ×
/Users/rick/PycharmProjects/chapter4/venv/bin
python /Users/rick/PycharmProjects/chapter4
/multiple_inheritance.py
Name: Morgan
Last name: cheeky
gender: male
```

But because we have multiple inheritance, this automatically means we can access all of the father and mother functionalities (attributes and methods). If we type the instance name and dot, we will have a list of every property and method the ‘morgan’ object have available:



From the figure above, we can see that we can access the name property and get\_name() method defined in the father class because of the inheritance. So let's call the get\_name() method and print its result:

```
# child object
morgan = Child('Morgan', 'cheeky', 'male')
morgan.child_info()
morgan.get_name()
```

Name: Morgan  
Last name: cheeky  
gender: male  
Name: Morgan

This works just fine, let's test the get\_age() method now:

```
# child object
morgan = Child('Morgan', 'cheeky', 'male')
morgan.child_info()
morgan.get_name()
morgan.get_age()
```

```
Name: Morgan
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/firstproject/multiple-inheritance.py", line 46, in <module>
    morgan.get_age()
File "/Users/rick/PycharmProjects/firstproject/multiple-inheritance.py", line 13, in get_age
    print(f'Age: {self.age}')
AttributeError: 'Child' object has no attribute 'age'
```

From the figure above, we can see that we have an attribute error saying that the ‘Child’ object does not have an attribute for age, and this is true because this is our Morgan object:

```
morgan = Child('Morgan', 'cheeky', 'male')
```

We can solve this by adding age at the end. For example, let’s 2 (for two years):

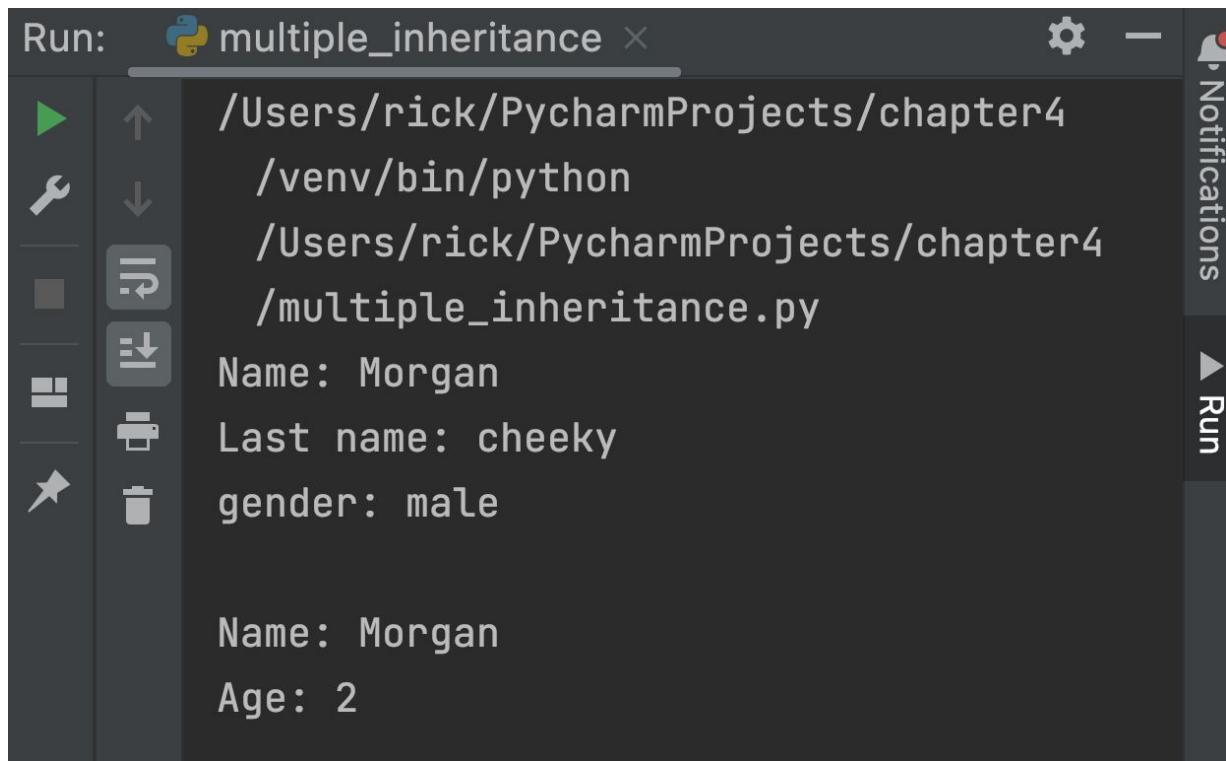
```
# child object
morgan = Child('Morgan', 'cheeky', 'male', 2)
morgan.child_info()
morgan.get_name()
morgan.get_age()
```

```
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/firstproject/multiple-inheritance.py", line 43, in <module>
    morgan = Child('Morgan', 'cheeky', 'male', 2)
TypeError: Child.__init__() takes 4 positional arguments but 5 were given
```

Now we have a `TypeError` saying the child constructor `__init__()` method does not take 5 arguments, so what can we do? Well, one thing we haven’t done is that we never called the father constructor in the child constructor so let’s change that and a few lines of code in the `Child` constructor function:

```
class Child(Father, Mother):
    def __init__(self, name, personality, gender, age):
        # call the father constructor
        Father.__init__(self, name, age)
        self.name = name
        self.personality = personality
        self.gender = gender
```

As you can see, we have called the Father constructor and we also added the age attribute to the child `__init__()` constructor as well. Finally, we can run this code again:



The screenshot shows the PyCharm interface with the 'Run' tool window open. The title bar says 'Run: multiple\_inheritance'. The tool window lists the Python environment path: '/Users/rick/PycharmProjects/chapter4 /venv/bin/python'. Below that, it shows the script path: '/Users/rick/PycharmProjects/chapter4 /multiple\_inheritance.py'. The output pane displays the following text:  
Name: Morgan  
Last name: cheeky  
gender: male  
  
Name: Morgan  
Age: 2

As you can see, the child now is 2 years old, and everything works. Let's call some of the mother methods and see what will happen:

```
# child object
morgan = Child('Morgan', 'ch
morgan.child_info()
morgan.get_name()
morgan.get_age()
morgan.get_eyes()

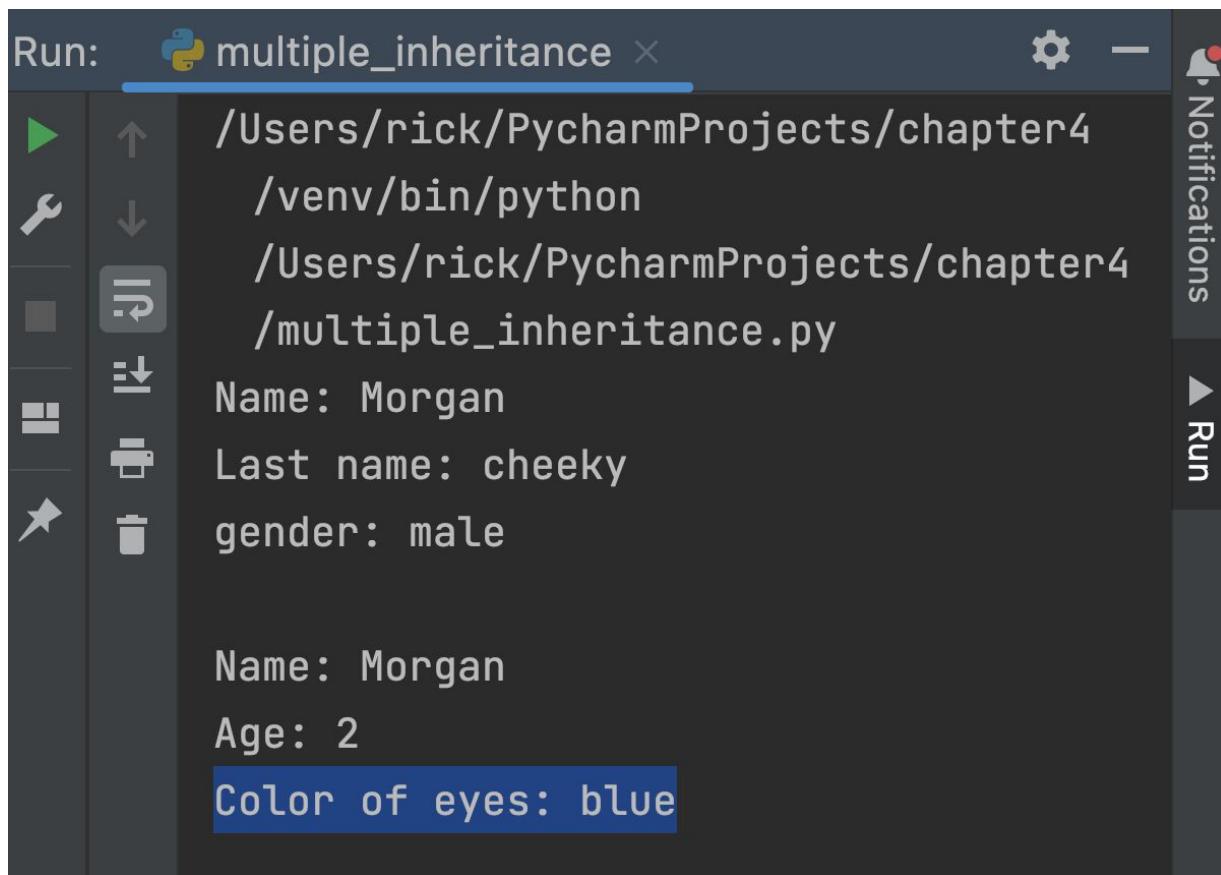
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects
    /firstproject/multiple-inheritance.py",
    line 46, in <module>
      morgan.get_eyes()
  File "/Users/rick/PycharmProjects
    /firstproject/multiple-inheritance.py",
    line 25, in get_eyes
      print(f'Color of eyes: {self.eyes}')
AttributeError: 'Child' object has no
attribute 'eyes'
```

Same error, the child object doesn't have the 'eyes' attribute and if we add that attribute the problem will not go away unless we include the Mother constructor in the child class:

```
class Child(Father, Mother):
    def __init__(self, name, personality, gender, age, eyes):
        # call the father constructor
        Father.__init__(self, name, age)
        Mother.__init__(self, name, eyes)
        self.name = name
        self.personality = personality
        self.gender = gender
```

Let's test this by adding 'blue' eyes:

```
morgan = Child('Morgan', 'cheeky', 'male', 2, 'blue')
```



The screenshot shows the PyCharm Run window with the title "multiple\_inheritance". The run configuration path is set to "/Users/rick/PycharmProjects/chapter4 /venv/bin/python /Users/rick/PycharmProjects/chapter4 /multiple\_inheritance.py". The output pane displays the following text:

```
Name: Morgan
Last name: cheeky
gender: male

Name: Morgan
Age: 2
Color of eyes: blue
```

Perfect! Now everything works. Let us summarize the pros and cons of multiple inheritance:

#### Pros:

- a) The child class can inherit functionalities from multiple classes
- b) This allows us to create complex relationships between the classes
- c) Multiple inheritance syntax resembles how classes are in the real world, therefore we can better describe the classes and establish relationships between them
- d) Having the ability to use the parents' methods and attributes is a huge bonus because any OOP language strives for code reusability, and here we reuse the code from other classes
- e) Building applications using multiple inheritance will take less time (code reusability) and less memory

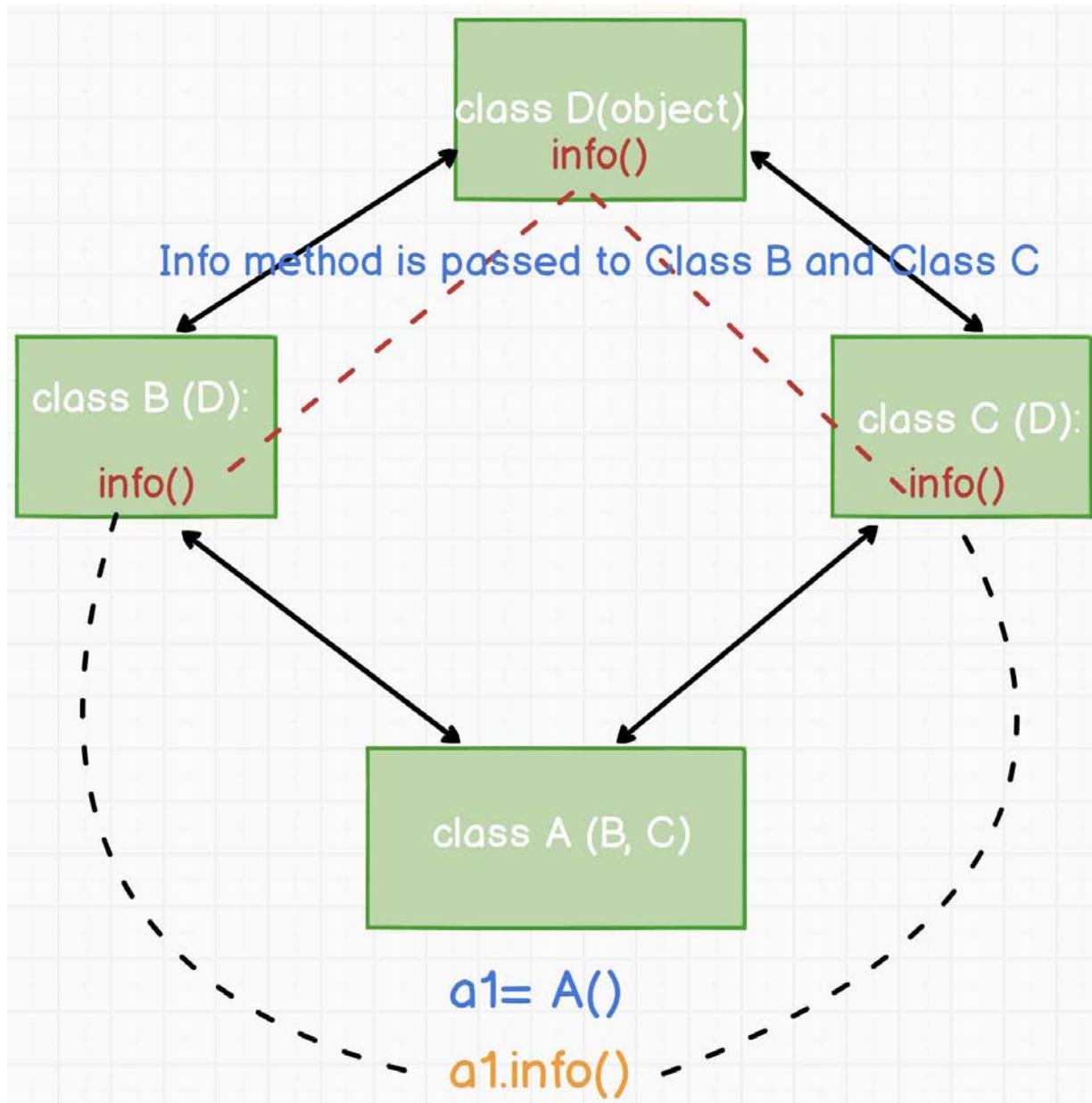
#### Cons:

- a) I mentioned that Python is one of the few languages that allow us to implement multiple inheritance, which can cause a lot of confusion

- b) It is really hard to have multiple inheritances where multiple classes have the same method name
- c) The maintenance of the code is much more tasking compared to applications with single inheritance

## MRO – Method Resolution Order

When we are dealing with multiple inheritance, we can run into problems. That is why I mentioned that multiple inheritance is not a feature that all OOP programming languages support because they want to avoid complexity. For example, we have class A that inherits from two other classes, B and C, and this is the exact same scenario we had with the ‘Child, Father, and Mother’ classes in the previous section. But imagine that both of these classes (B and C) inherit from a fourth-class D. Let us have a look at the following figure:



I know this figure is confusing so let me explain what is happening. The two classes, `B` and `C`, inherit from the Parent class, `D`, where we have a method called `info()`. Because of the inheritance, this method is now available in classes `B` and `C`. Class `A` on the other hand inherits all of the functionalities from classes `B` and `C`. When we instantiate the class `A` and call the `info()` method, where will this method come from? Will it come from class `B` or class `C`? This is decided by the linearization algorithm. The MRO (Method Resolution Order) is an algorithm that will decide how the classes are ordered and from which class to use the method. So, the linearization will create a list of all ancestor classes including the class itself and it will order

them by putting the first as closest and the last as furthest. In simple terms, the MRO is a rule that Python will follow to determine which object to call first. Okay let's create a new example where we can see how the MRO algorithm works:

```
# MRO - Method Resolution Order
```

```
# class A
```

```
class A:  
    def __init__(self, length=5):  
        self.length = length  
  
    def info(self):  
        return self.length
```

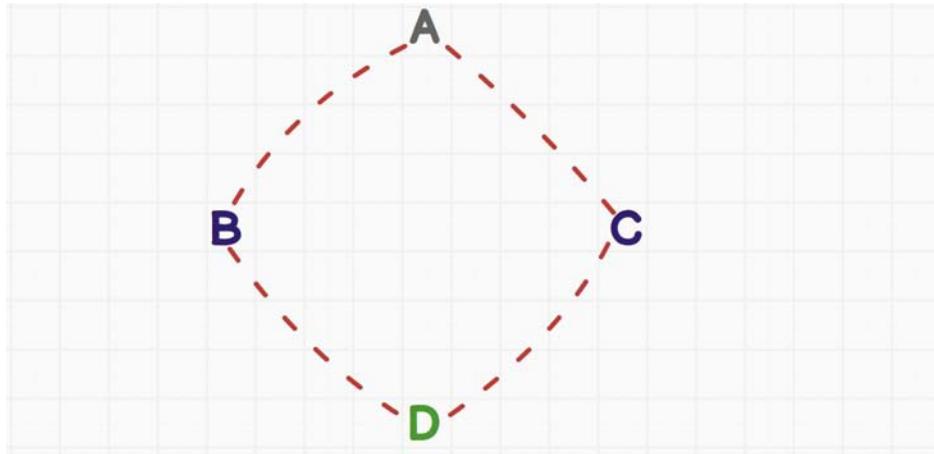
```
class B(A):  
    pass
```

```
class C(A):  
    def __init__(self, length=2):  
        self.length = length  
  
    def info(self):  
        return self.length
```

```
class D(B, C):  
    pass
```

```
# instance of class D  
d_obj = D()  
print(d_obj.info())
```

If we sketch this entire relationship, we will end up with this drawing:



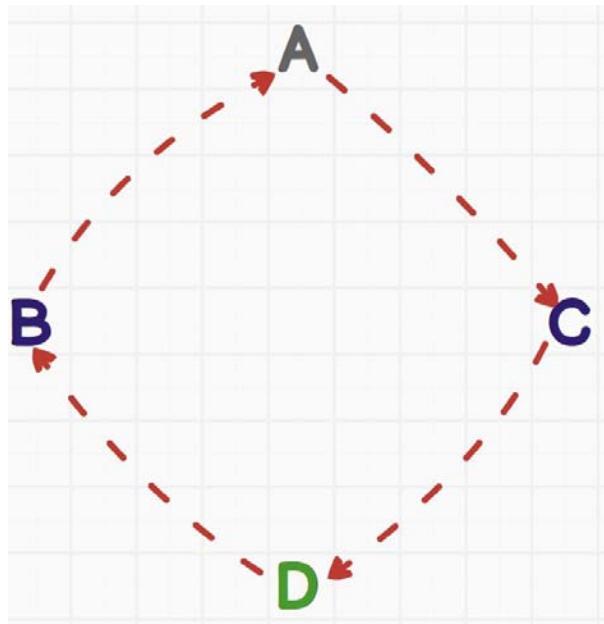
**Class D have multiple inheritance (B, C)**

**B inherits from A, C inherits from A**

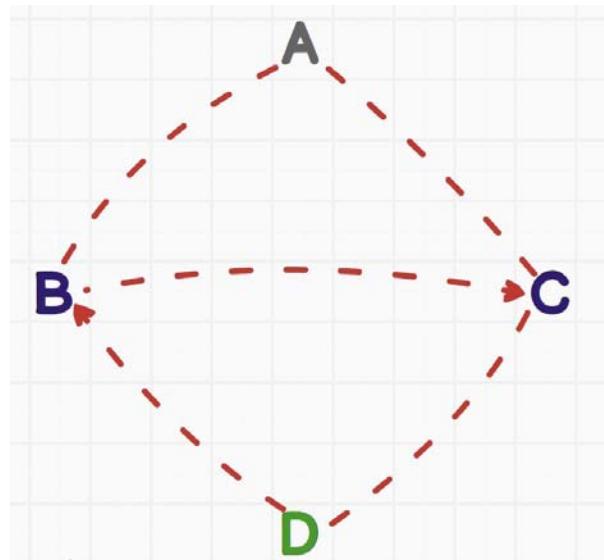
If we create an instance of the D class and call the info method, what will be the output?

```
# instance of class D
d_obj = D()
print(d_obj.info())
```

The output will be 2 and not 5, why is this? Well, because the job of the MRO algorithm is not going to follow our human logic and it will not go clockwise and get the method from class A directly:



So MRO will consider what is first in line when we have methods or attributes that have the same name. It will consider the classes from which the inheritance should happen first:



In Python, there is A method called `mro()` that will tell us the exact order that the algorithm will use so all we need to do is to type:

```
print(D.mro())
```

Output:

1

2

3

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
 <class '__main__.A'>, <class 'object'>]
```

4

From the figure above, we can see the algorithm order. First, it will check if the info method is in class D, then it will go to the B class and check, then C, then A, and finally the base Object class that each of the classes inherits from. The Method Resolution Order is not easy to understand in more complex scenarios. Can you guess the following output:

```
class First:
    pass
```

```
class Second:
    pass
```

```
class Third:
    pass
```

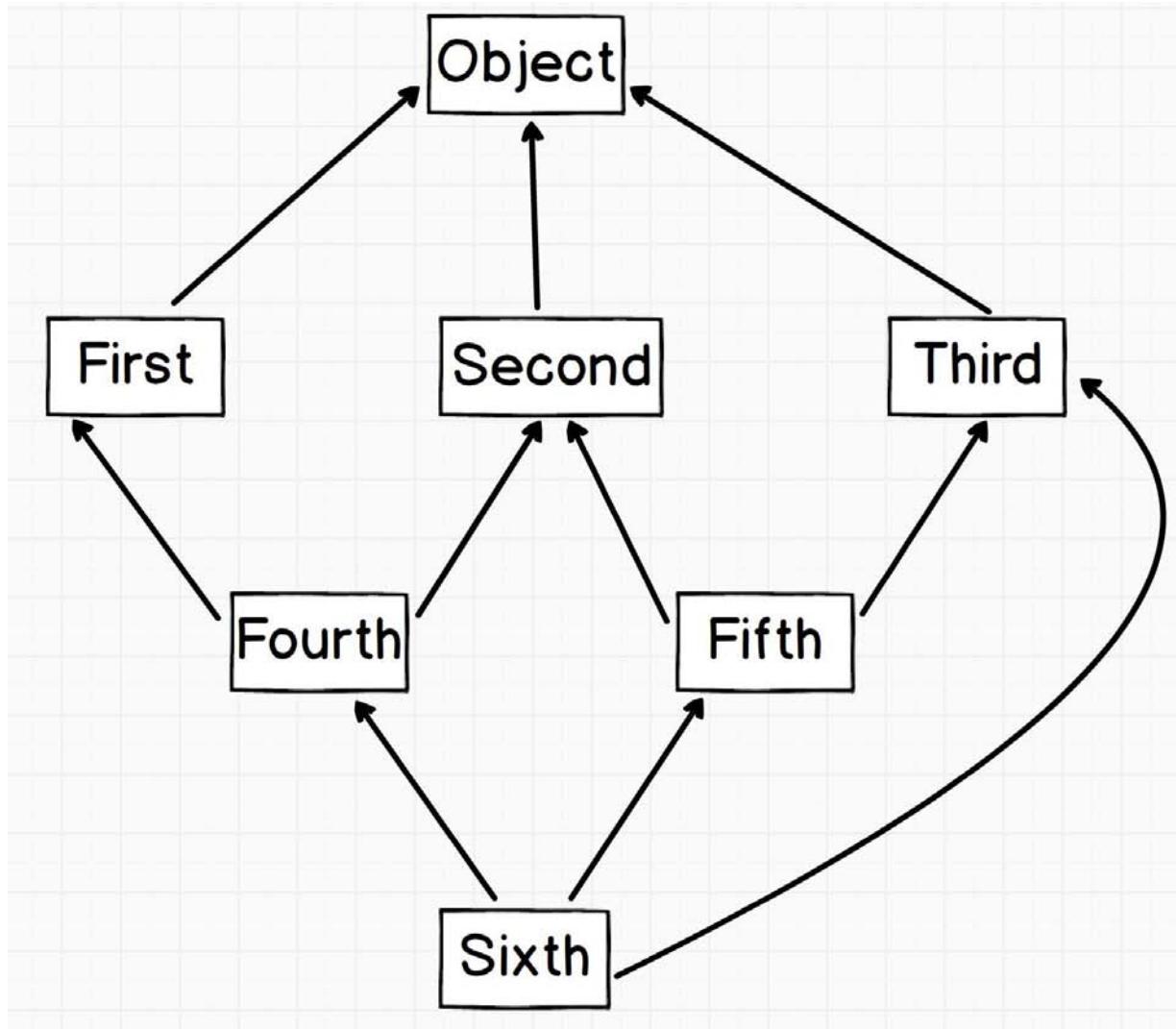
```
class Fourth (First, Second):
    pass
```

```
class Fifth (Second, Third):
    pass
```

```
class Sixth (Fifth, Fourth, Third):
    pass
```

```
print(Sixth.mro())
```

The diagram of this relationship:



Here is the output (please try and guess the output):

```
[<class '__main__.Sixth'>, <class '__main__.Fifth'>,
 <class '__main__.Fourth'>, <class '__main__.First'>,
 <class '__main__.Second'>, <class '__main___.Third'>, <class 'object'>]
```

Why did the MRO visit the Sixth then the Fifth and not the Fourth class in the figure above? Well, because for MRO, the order we are passing the classes is important. Look at the last line of code:

```
class Sixth(Fifth, Fourth, Third):
```

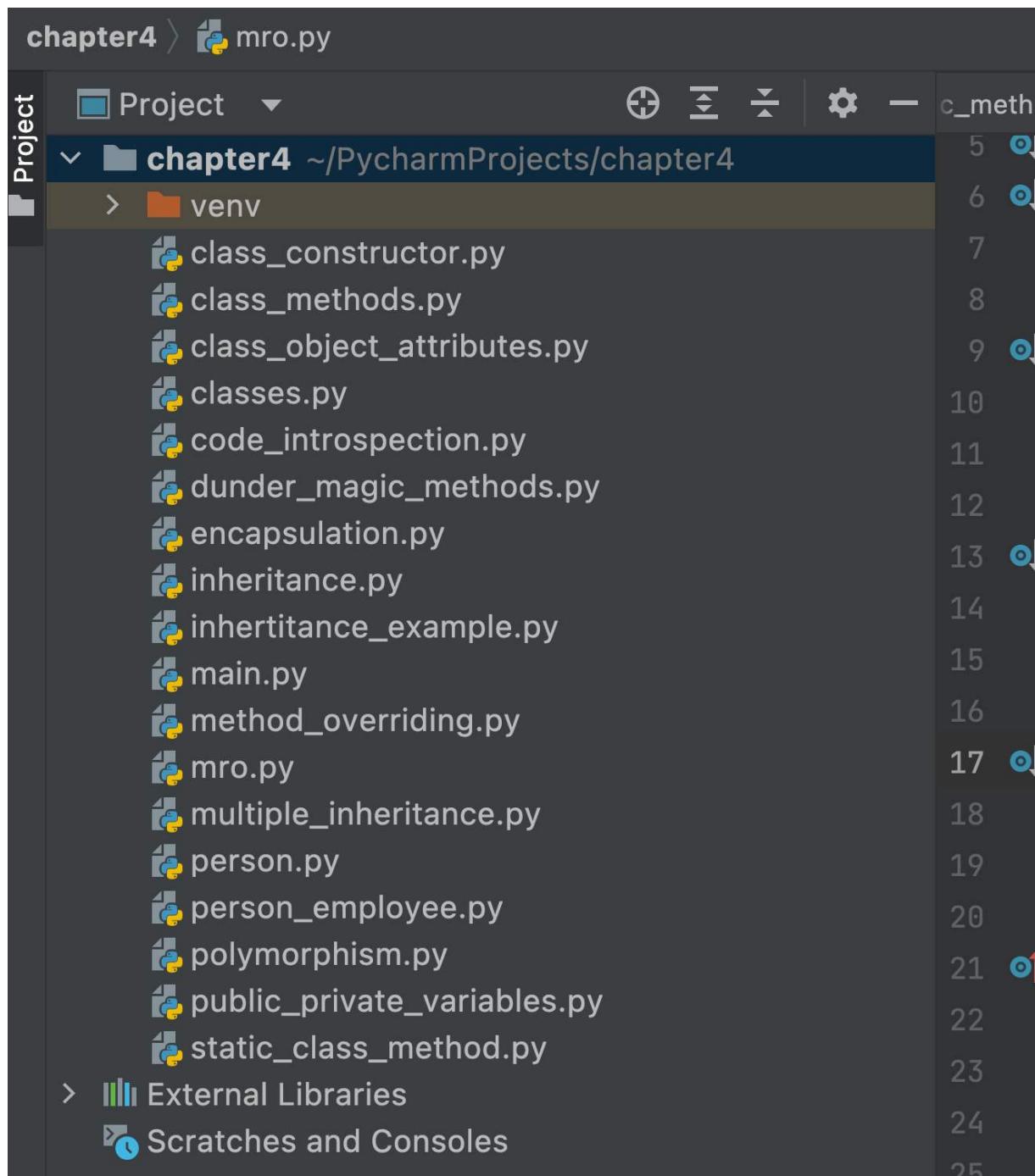
From the code above, the order starts with Fifth, then Fourth, and then Third. But from the figure above, we can see that it starts with Sixth, then it goes to Fifth and Fourth, but instead of going to Third, it goes directly to First, Second, and finally Third. This is happening because of one algorithm the MRO uses called **Depth First Search** but this is a more advanced concept and it's not in the scope of this book. The algorithm for MRO is not the same one that was used in the previous versions of Python.

## **Summary**

This chapter was packed with information and new concepts. I know this was the hardest chapter so far for some of you. My recommendation for you is to not get discouraged because I can assure you if you keep practicing and reading, you will learn all of the OOP features. Take your time, read some of the sections again and never give up. Let's get ready for our new chapter called modules and I promise, the hardest features are already covered.

# **Chapter 5- Phyton – Modules and Packages**

In this chapter, we will learn all about Python modules and packages. A Python module is simply a file that contains Python code. Inside a module, we can define classes, functions, and variables. In a module, we organize related code. So far, we have organized and grouped the code in a few different ways. The first way is to use files so here are a few of the files I have created in the previous chapter:

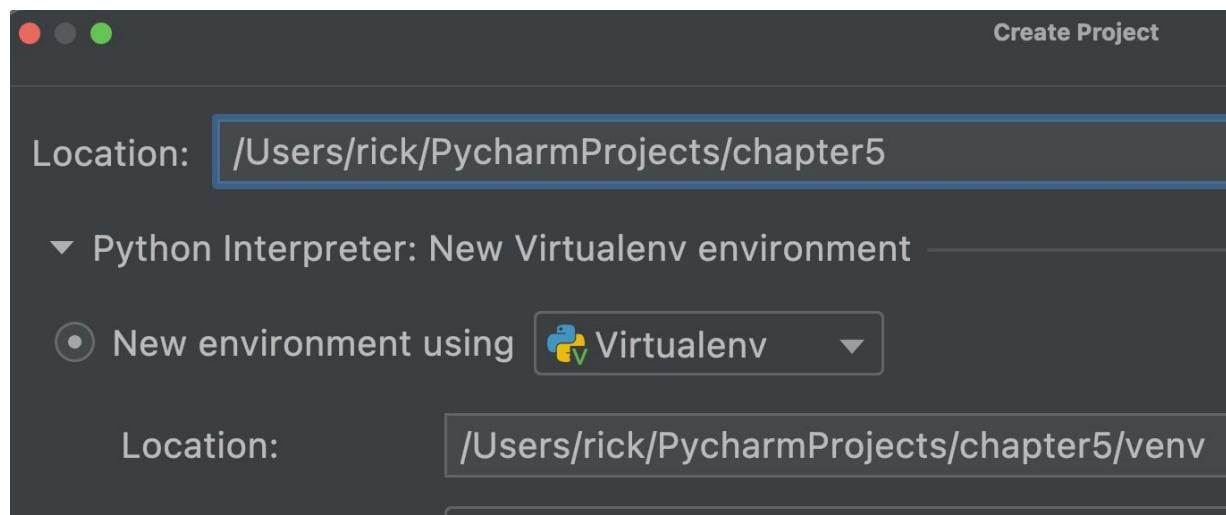


Organizing the code into different files is one way but we can organize the code into functions as well. As you know, the functions in Python are treated as single units or blocks so we can call and run them when we want. The third way to organize the code is to use classes. The classes are blueprints or templates from which we can instantiate or create objects. Python has provided us with different ways to organize and group related

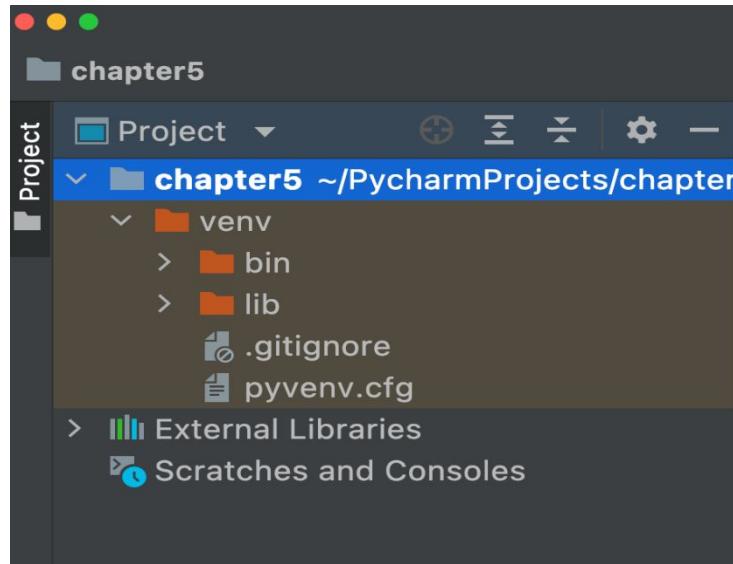
code. In the next section, we will talk about how we can use modules to organize and group the code.

## Why We Need Python Modules?

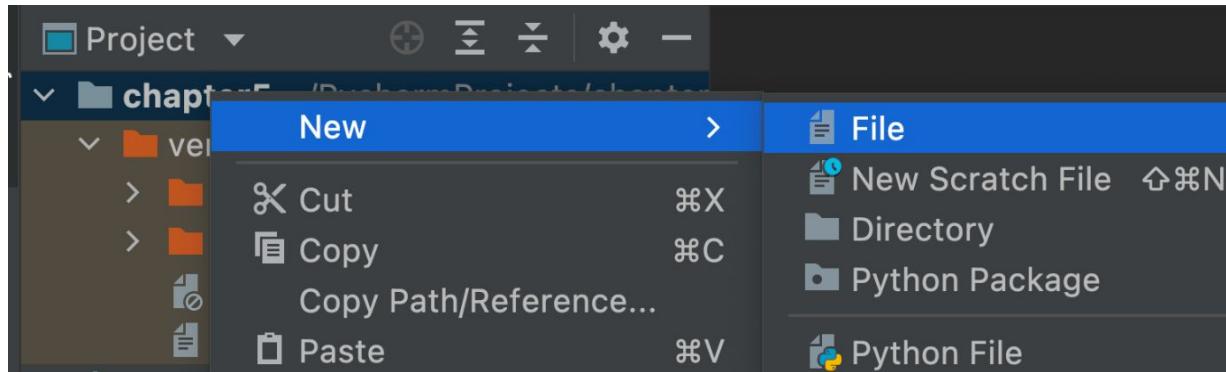
We need modules because we want to split large code into meaningful smaller sections. The classes, functions, and files are not a solution if we want to write a complex program. That is why we need modules. So, what is a module? A module is a single Python file ‘.py’ where we have code. We use modules to divide the code into multiple related files. In my **PyCharm** IDE, I will create a couple of files that will help me to explain how the modules work. First things first, let’s create a new project called chapter5 (if you are trying this on your own, you can name your folder whatever you like). So how can we create a new project in PyCharm? Easy, go to the menu and select **file > New Project**:



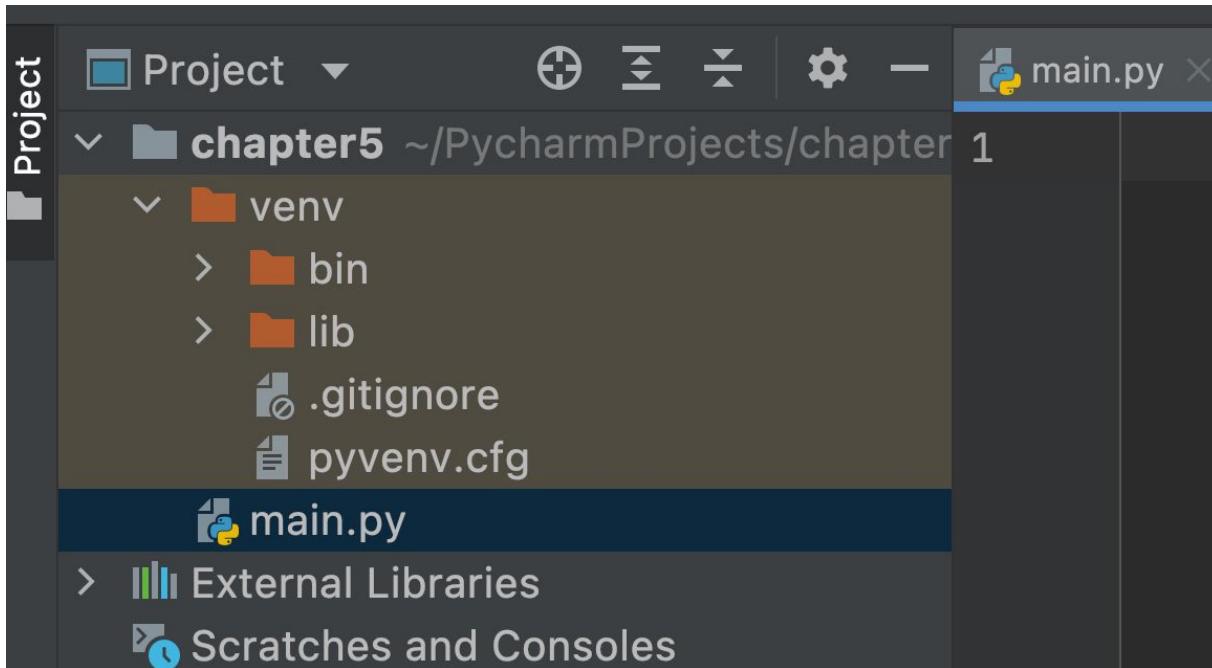
After you have created the folder, you will have a directory called **venv**, you don’t have to worry about what it does and what it means at this stage:



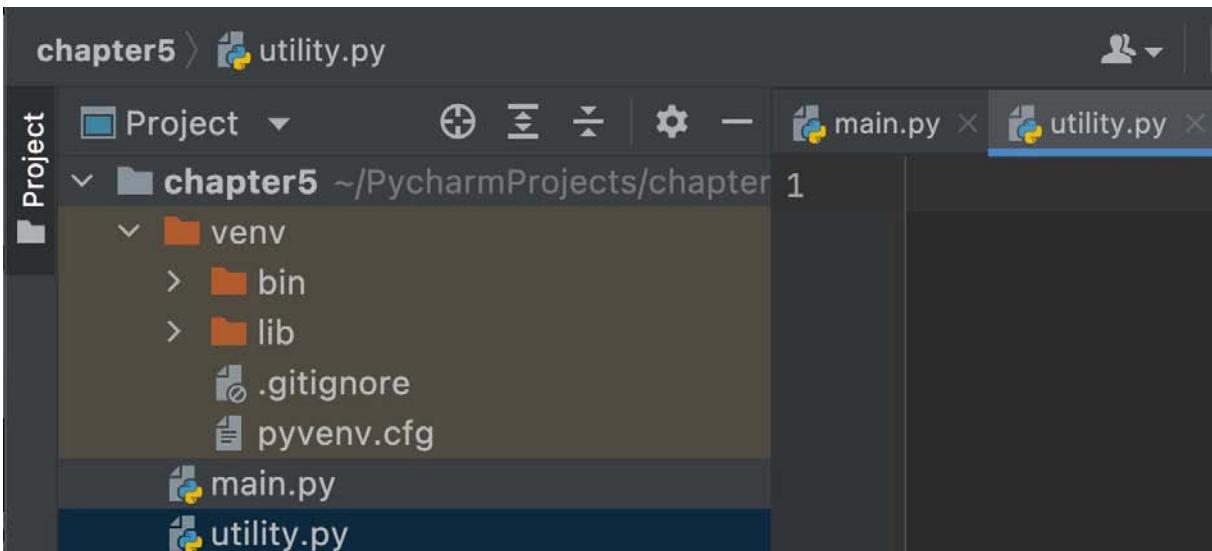
When you create new files or modules as we now call them, you should not create them inside the **venv** folder but outside in the **chapter5** directory. Let's create a file called **main.py**:



Here is the new file now:



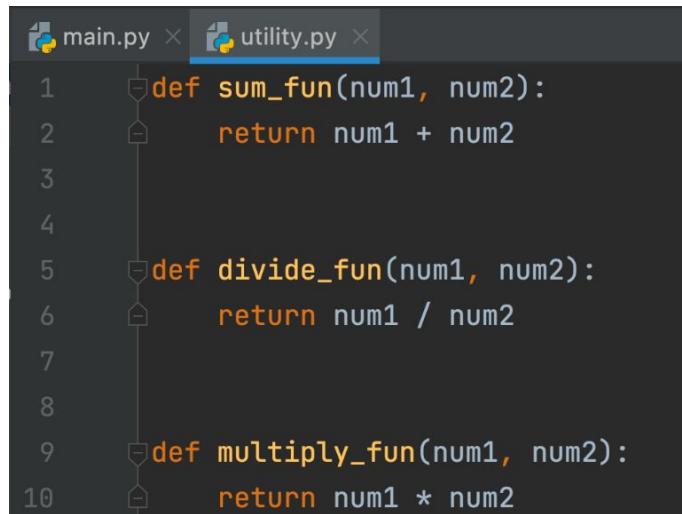
We can repeat the same step and create another Python file called **utility.py**. Now our chapter5 folder looks like this:



The two files that we created are modules, **main.py** is the first module and **utility.py** is the second module. When we are creating the module names, we need to use the **snake\_case** syntax just as we did for the variables. This means that if the module name consists of two words, we need to use an underscore between them. All of the letters should be lowercase:

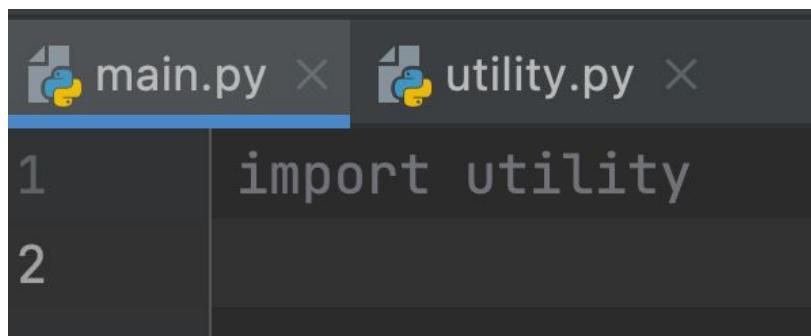
## my\_file.py

In the **utility.py** module, we can write some basic math functions and the **main.py** module is the one that will call and use these functions. Let's write a few simple math functions in the **utility.py** module first:



```
main.py × utility.py ×
1 def sum_fun(num1, num2):
2     return num1 + num2
3
4
5 def divide_fun(num1, num2):
6     return num1 / num2
7
8
9 def multiply_fun(num1, num2):
10    return num1 * num2
```

What is the purpose of my **main.py** module? Well, I want to use the functions defined in the utility in the **main.py** module. The question is how can I connect both of them? In Python, we can use the keyword ‘import’ when we want to import a file. In the **main.py** file is where we import syntax:



```
main.py × utility.py ×
1 import utility
2
```

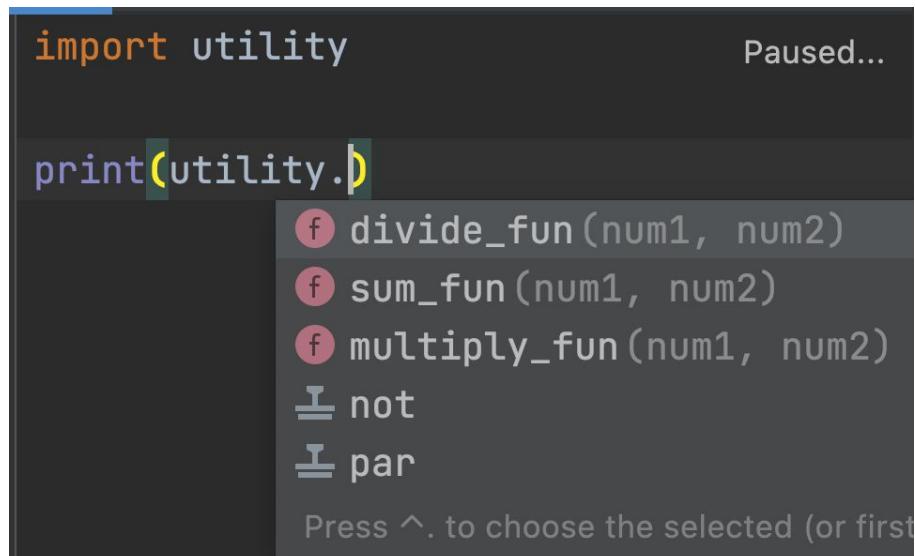
As you can see from the figure above, I didn't include the extension ‘import utility.py’ because it is assumed that the module is a Python file. If we print the utility, what do you think will be the output:

```
import utility  
print(utility)
```

Output:

```
<module 'utility' from  
'/Users/rick/PycharmProjects/chapter5/utility  
.py'>
```

As you can see, it will print the actual file path of the **utility.py** file. It would be a different path if you are trying this on your machine, or if you are still using '**replit**'. It will provide you with a different path because that website will generate a path for you. Now our files/modules are linked and we only need to use the dot syntax to start using the functions:



The screenshot shows a code editor window in PyCharm. The code being typed is:

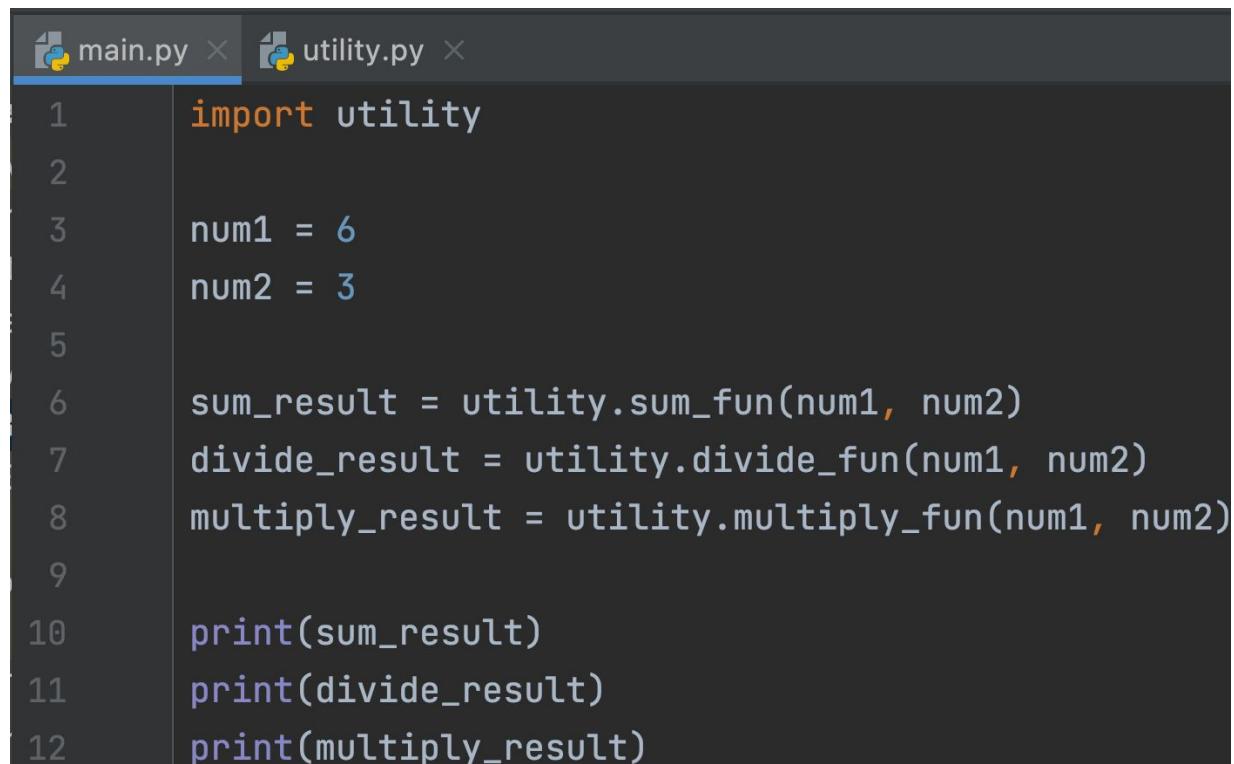
```
import utility  
print(utility.
```

A code completion dropdown menu is open at the end of the line, listing the following suggestions:

- f divide\_fun(num1, num2)
- f sum\_fun(num1, num2)
- f multiply\_fun(num1, num2)
- l not
- l par

At the bottom of the dropdown, there is a message: "Press ^ to choose the selected (or first)".

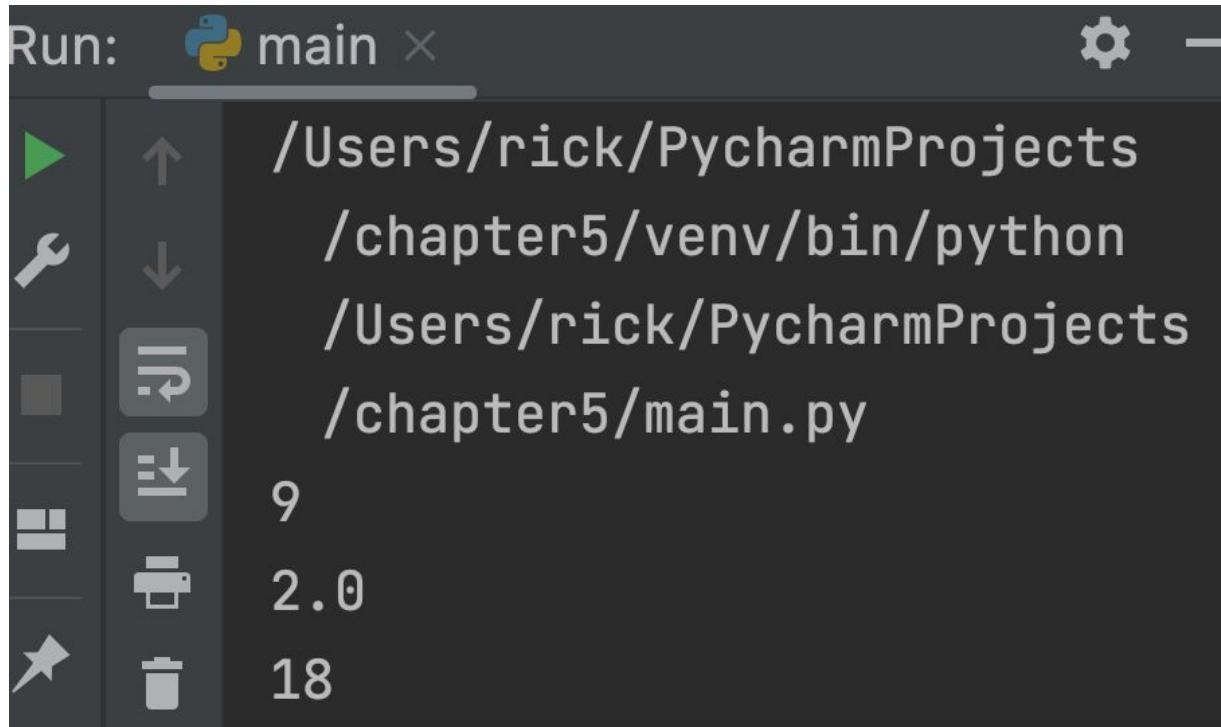
As you can see, whatever we have declared in the utility module is now accessible in the main.py module. Let's do some basic maths calculations using the functions from the utility file:



The screenshot shows a code editor interface with two tabs at the top: "main.py" and "utility.py". The "main.py" tab is active, displaying the following Python code:

```
1 import utility
2
3 num1 = 6
4 num2 = 3
5
6 sum_result = utility.sum_fun(num1, num2)
7 divide_result = utility.divide_fun(num1, num2)
8 multiply_result = utility.multiply_fun(num1, num2)
9
10 print(sum_result)
11 print(divide_result)
12 print(multiply_result)
```

The output is:

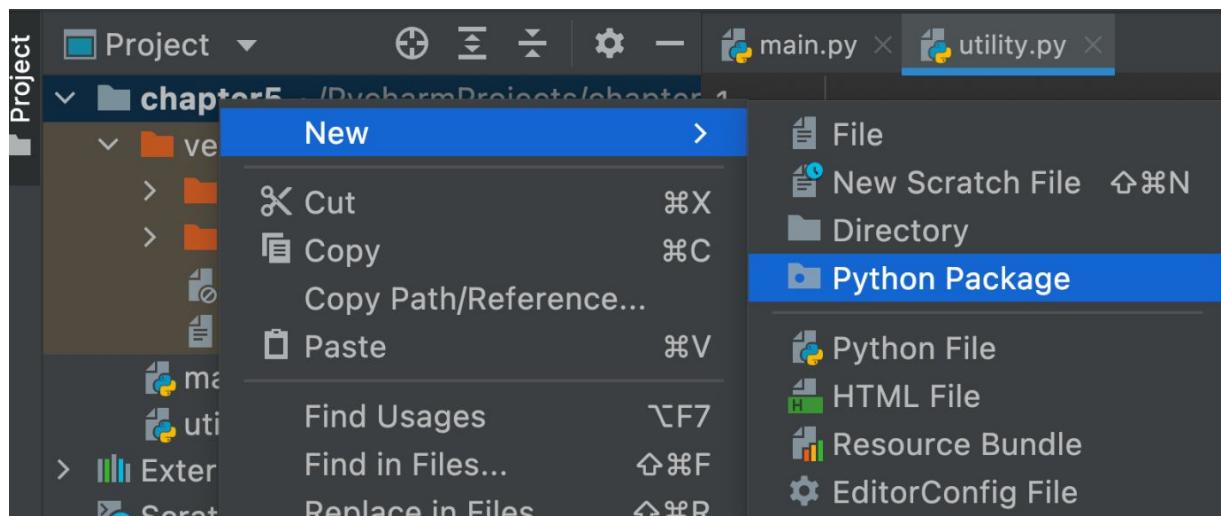


What if I have a very big project with a lot of files that I need to import?  
Well, you can do multiple imports in the main file like this:

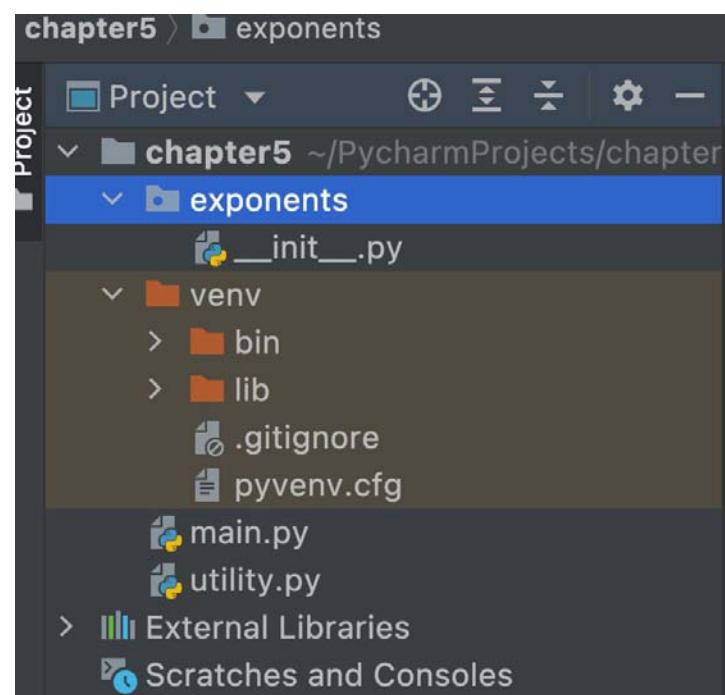
```
import utility
import utility1
import utility2
import utility3
```

## Python Packages

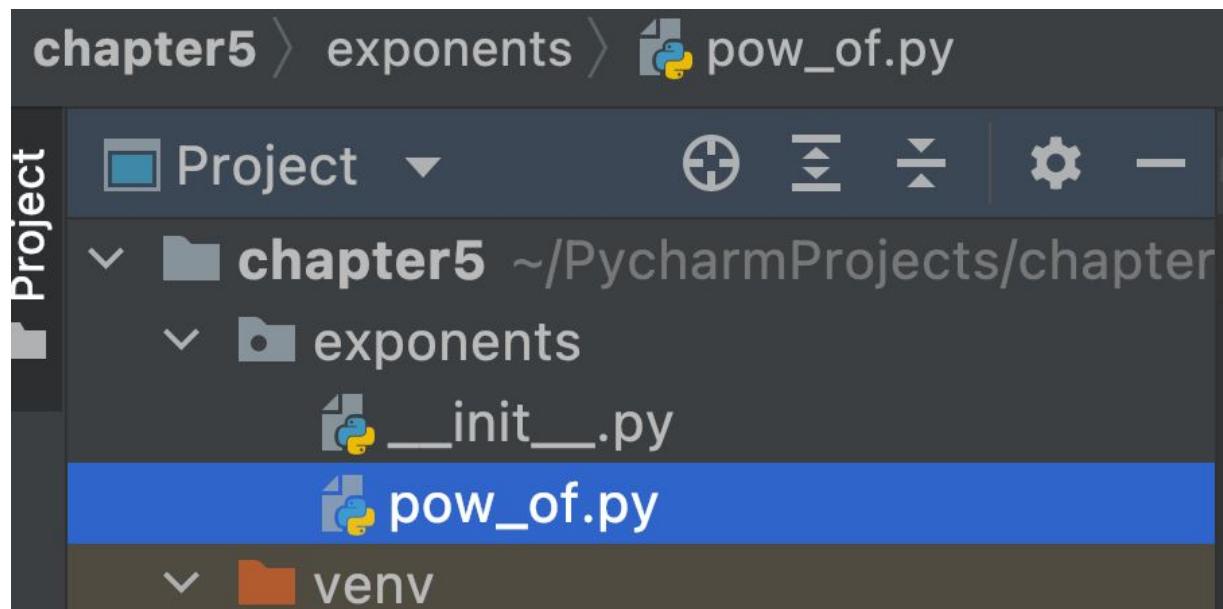
Imagine that our project is getting bigger and every day we add new functionality and we want that functionality to be included in a separate folder or package. In Python, we can create new packages if we do right click on chapter5 and then select New > Python Package and name the package 'exponents' (you can use a different name if you want to):



As you can see, we can even create a package and even a new directory, but let's select the package:



Inside the chapter5, we have created a package called **exponents**, and inside, there is a file called `__init__.py`. This file will be empty if we open it and it came with the package we created. In this **exponents** folder, we need to create a new file called **pow\_of.py**.



In this file, I will add one function that will calculate the power of two:

```
def pow_fun(num):
    return num ** 2
```

Let's try to import pow\_of inside the main.py file with the import syntax:

```
1 import utility
2 import pow_of
```

Now let's print the following code:

```
pow_result = pow_of.pow_fun(num1)
print(pow_result)
```

The output will be:

```
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/chapter5
    /main.py", line 2, in <module>
      import pow_of
ModuleNotFoundError: No module named 'pow_of'
```

As you can see, the module cannot be found. Why we are getting this error? The package we created called exponents maybe look like a normal folder or directory but it's considered a package. The package is on a different level from the rest of the modules like main and utility. Therefore, the files from this package can be imported into Python like this:

```
import exponents.pow_of
```

Before we do anything, let's print this and check out the output:

```
print(exponents.pow_of)
```

The output will be a path to the file pow\_of located in the package exponents (if you try to run the same example on your computer, the path will be different):

```
<module 'exponents.pow_of' from
  '/Users/rick/PycharmProjects/chapter5/exponents/pow_of.py'>
```

Finally, let's check if the function we have declared in the pow\_of module can now be accessed in the main file:

```
print(exponents.pow_of.pow_fun(num1))
```

Output:

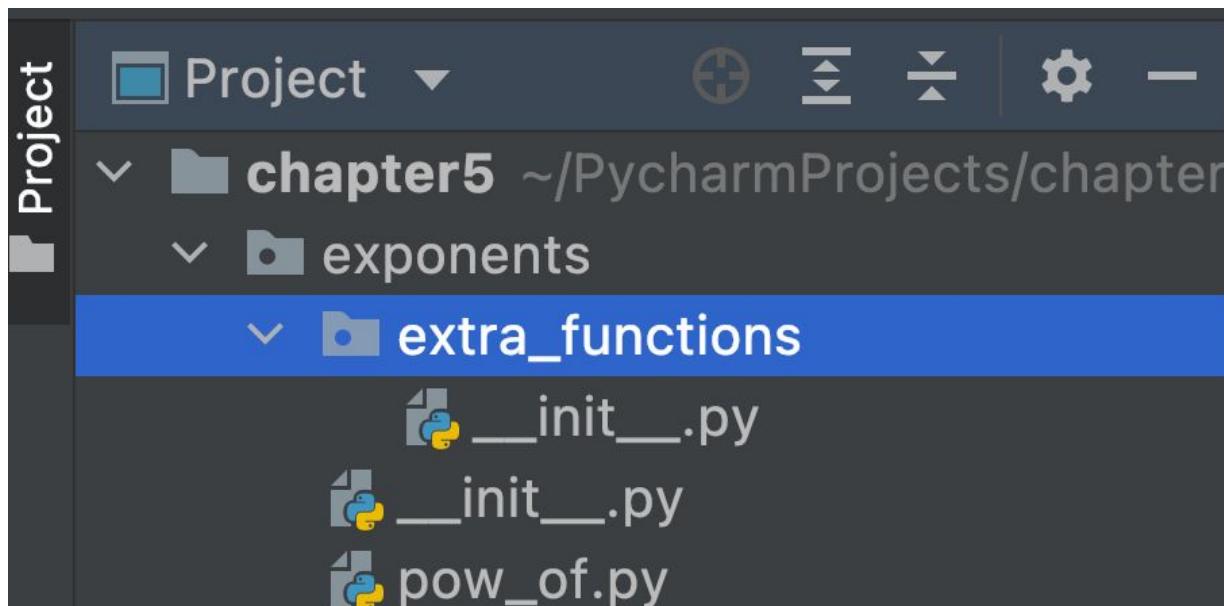
```
main.py × utility.py × pow_of.py × Run: main × /Users/PycharmProjects/cha.../pyt.../User/PycharmProjects/cha.../cha.../9 2.0 18 36 Processes exit
```

```
1 import utility
2 import exponents.pow_of
3
4
5 num1 = 6
6 num2 = 3
7
8 sum_result = utility.sum_fun(num1, num2)
9 divide_result = utility.divide_fun(num1, num2)
10 multiply_result = utility.multiply_fun(num1, n
11
12 print(sum_result)
13 print(divide_result)
14 print(multiply_result)
15 print(exponents.pow_of.pow_fun(num1))
```

It looks like everything we have done is working because we got the result 36 and 6 raised to the power of 2 is exactly 36. Okay, now, let's talk about the `__init__.py` file that was created for us when we created the package. If you are using [replit.com](#), you will not have this file because the website usually hides these kinds of files. Why does Python create this empty file? This file is created because it will tell Python that this is a package from where you need to import files.

## Import modules using `from-import` syntax

In this section, we will learn different ways we can import modules. Let's create another package inside the existing 'exponents' package called 'extra\_functions':



In the extra\_functions, we can create a new file called modulo\_fun.py and inside this file, we can add this function:

```
def modulo_fn(num1, num2):
    return num1 % num2
```

And let's call this module in our main.py module using the import statement that we learned before:

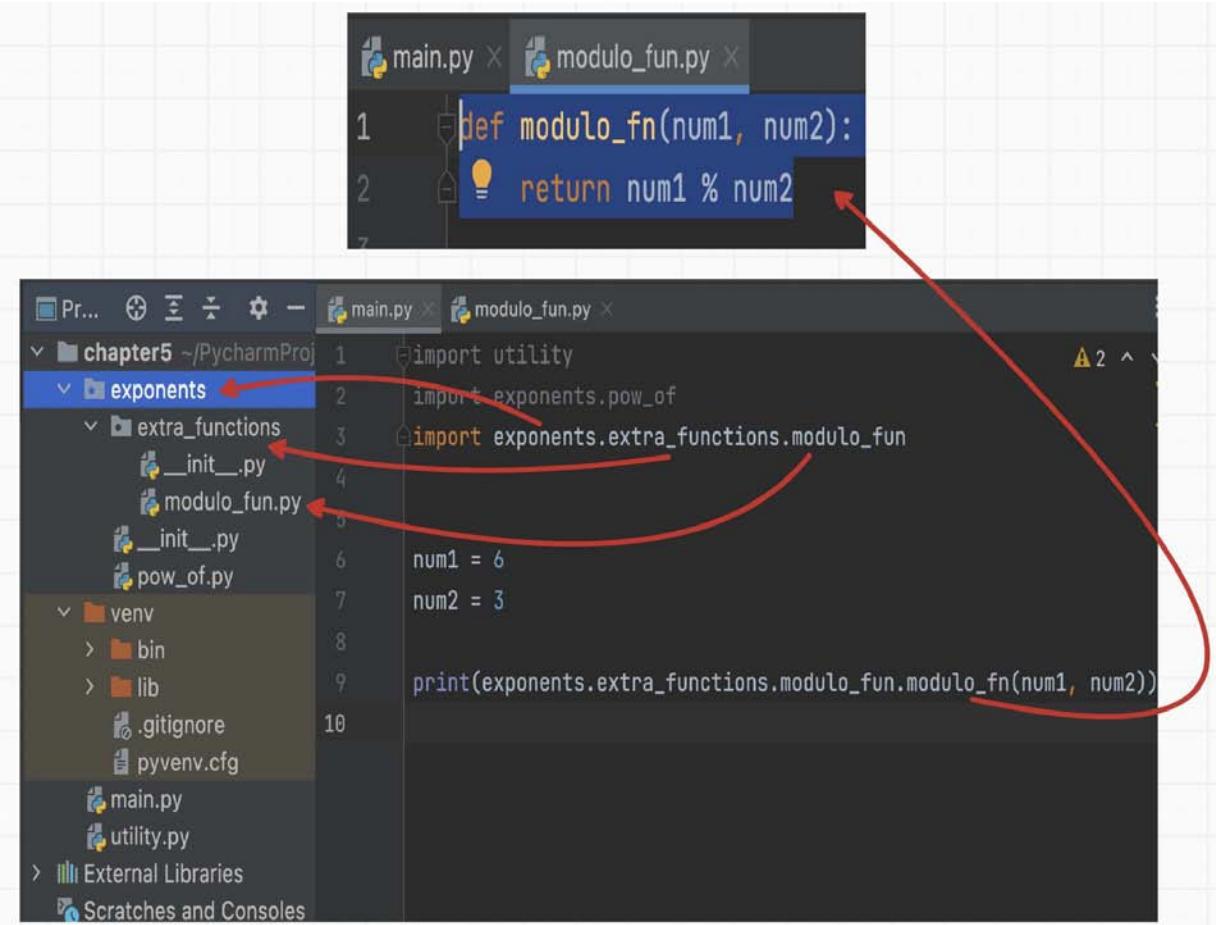
```
import exponents.extra_functions.modulo_fun
```

As you can see, the import line is getting longer and it doesn't look nice. Let's try to use the modulo\_fn in our main.py file:

```
import utility
import exponents.pow_of
import exponents.extra_functions.modulo_fun

num1 = 6
num2 = 3
print(exponents.extra_functions.modulo_fun.modulo_fn(num1, num2))
```

This is how all of the files are connected:



So instead of calling (package\_name.package\_name.module.function), we can do something like this:

```
from exponents.extra_functions.modulo_fun import modulo_fn
```

And now the modulo\_fn can be used directly in the main.py as it was declared there:

```
print(modulo_fn(num1, num2))
```

What will happen if there are multiple functions in the module that we want to import? Let's rewrite the import utility statement to this:

```
from utility import sum_fun, divide_fun, multiply_fun
```

As you can see, we listed all of the functions, separating them with commas and we can call these functions from the main.py module like this:

```
print(sum_fun(num1, num2))
print(divide_fun(num1, num2))
print(multiply_fun(num1, num2))
```

The syntax is very simple as long as we follow the from-import rule:

```
from module_name import things
```

But if we want to import the entire module and use the module like before to access its functions, we can simply do this:

```
from exponents.extra_functions import modulo_fun
```

This approach is better when we want to avoid name collisions. What does name collision mean? This means that we can have a function with the same name in different modules and when we import them using the last approach, it will not create a name collision.

## Import modules using from-import \* syntax

There is another way we can import everything that we have in a module and that is if we use the star at the end. Let's create a new file which will be a copy of the utility.py file and call it utility1.py:

The screenshot shows the PyCharm IDE interface. On the left, the project structure is displayed under 'chapter5 ~/PycharmProj'. It includes a 'venv' folder containing 'bin' and 'lib', and a 'main.py' file. Under 'exponents', there is a 'extra\_functions' directory containing four files: '\_\_init\_\_.py', 'modulo\_fun.py', '\_\_init\_\_.py', and 'pow\_of.py'. On the right, the code editor shows 'utility1.py' with the following content:

```
def sum_fun1(num1, num2):
    return num1 + num2

def divide_fun1(num1, num2):
    return num1 / num2

def multiply_fun1(num1, num2):
    return num1 * num2
```

As you can see, I have changed the function names as well to avoid any naming collision, and let's test this syntax now:

```
from utility1 import *
```

The star at the end means import everything from the utility1 module and now you can call the functions from this module like this:

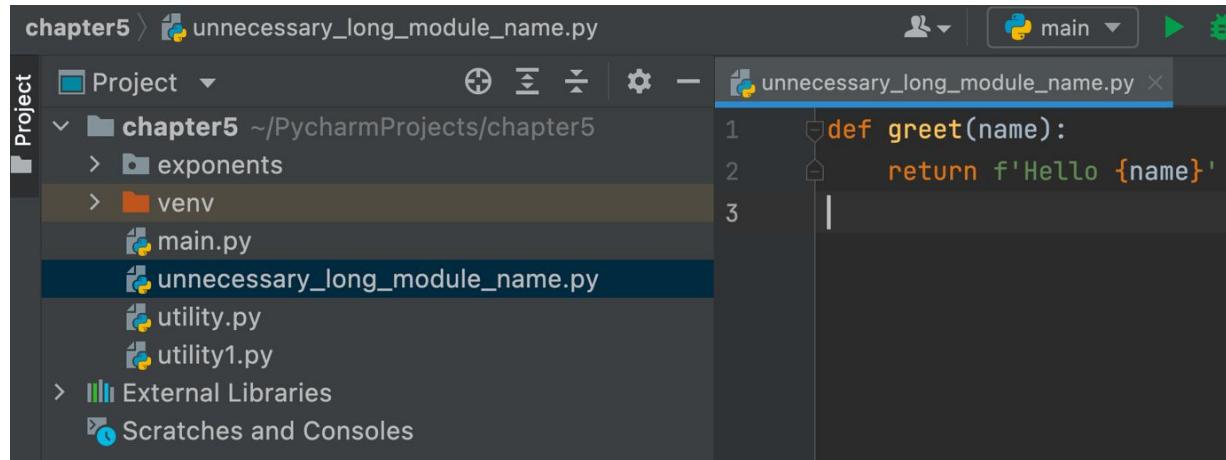
```
print(utility1.sum_fun1(num1, num2))
print(utility1.divide_fun1(num1, num2))
print(utility1.multiply_fun1(num1, num2))
```

There is no one rule of import syntaxes you must use, just go with the one you are most comfortable with.

## Python Module alias

Let's look at how we can use aliases in Python to import a specific module. In order to show you this, I will create another module with a long and

unusual name ‘unnecessary\_long\_module\_name.py’:



As you can see, I have only one function there to greet the user. Now we can import this function using the well-known from-import syntax:

```
from unnecessary_long_module_name import greet
```

Or we can make it shorter and more meaningful if we create an alias like this:

```
import unnecessary_long_module_name as long_module
```

Instead of the name `long_module`, we can even use one letter like ‘`u`’ and make it even shorter. Now let’s see if we can use the `greet()` function:

```
print(long_module.greet('Kevin Hart'))
```

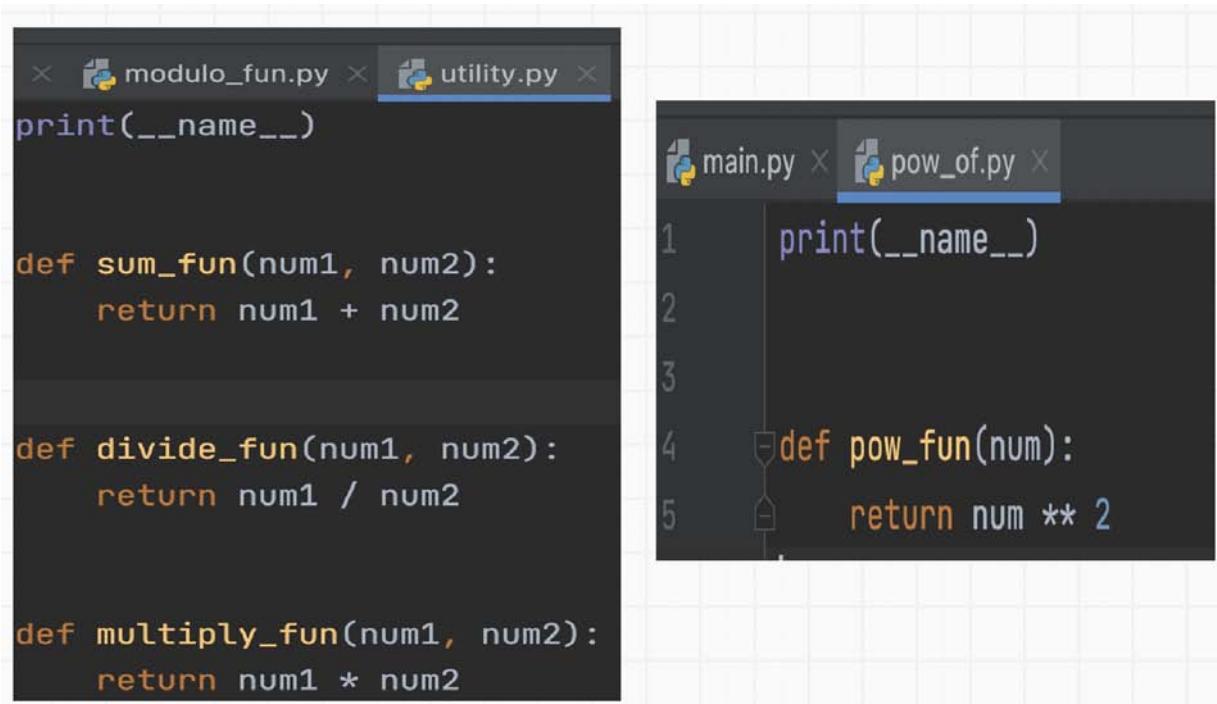
If we run the main module, we will get:

```
Hello Kevin Hart
```

By doing this, we can definitely create custom names that will avoid naming collisions.

## Python `__name__`

I think now it's the right time to explain what double underscore `__name__` means. If you haven't seen this code so far, double underscore name (`__name__`), I can assure you that one day you will find this piece of code and use it. So, what is Python variable `__name__`? This variable with two underscores before and after the variable name has a special meaning for Python when it comes to modules. To show what it means, I have made a copy of the previous modules and created a new folder called **chapter5.1**. Inside this folder, there are the utility.py and pow\_of.py where I have added at the top the following code: `print(__name__)`:



```
utility.py:
print(__name__)

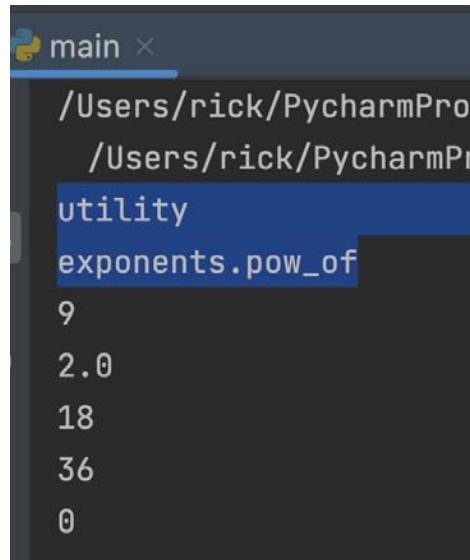
def sum_fun(num1, num2):
    return num1 + num2

def divide_fun(num1, num2):
    return num1 / num2

def multiply_fun(num1, num2):
    return num1 * num2

pow_of.py:
1 print(__name__)
2
3
4 def pow_fun(num):
5     return num ** 2
```

Now when I run the main.py, I get the following output:



```
main ×
/Users/rick/PycharmPro
/Users/rick/PycharmPro
utility
exponents.pow_of
9
2.0
18
36
0
```

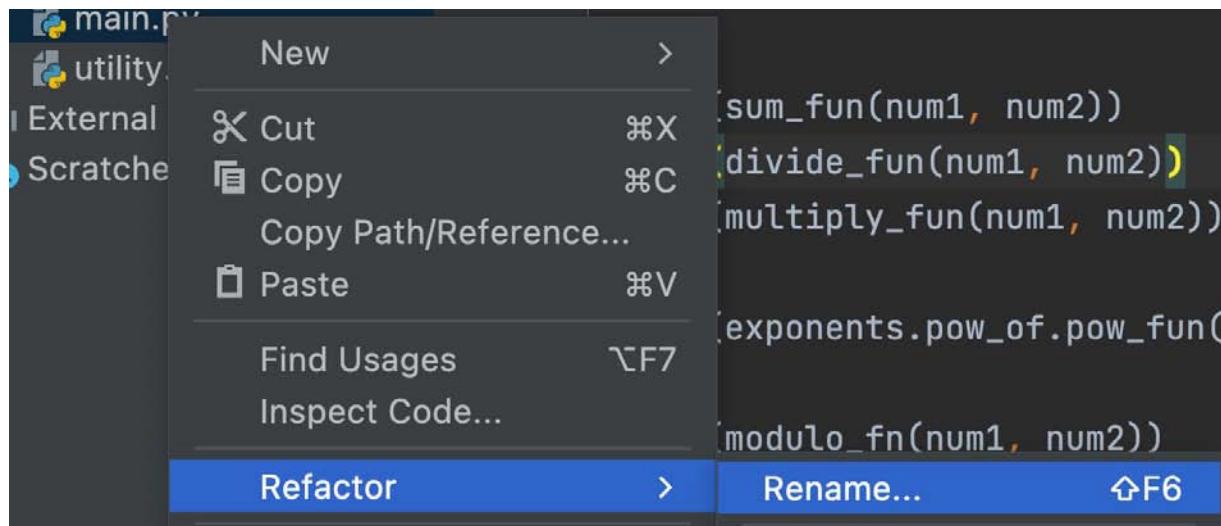
This is printing back the names of the modules because the Python Interpreter starts working from our main.py file and from there it calls the rest of the modules. Each time the Interpreter visits a file outside the main file, it will run that file and it will be stored in the memory, including the name of the module. This is efficient because in the main.py file we can do multiple calls to the functions that belong to external files and the interpreter will not run the utility or pow\_of file over and over again, instead, it will take them from the memory because it has direct access. What do you think will happen if we try to print the name of the module directly in the main.py file?

```
print(__name__)
```

The output will be:

```
__main__
```

Let's change the **main.py** file name to **main\_file.py**. All we need to do is right-click on the main.py file and then select Refactor > Rename:



After you rename this file let's run the `main_file.py`:

The screenshot shows the PyCharm interface with the code editor containing `main_file.py` and the Run tool window. The code imports functions from `utility`, `exponents`, and `extra_functions`. It defines variables `num1` and `num2`, and prints the results of various operations. The Run tool window shows the output of the code execution.

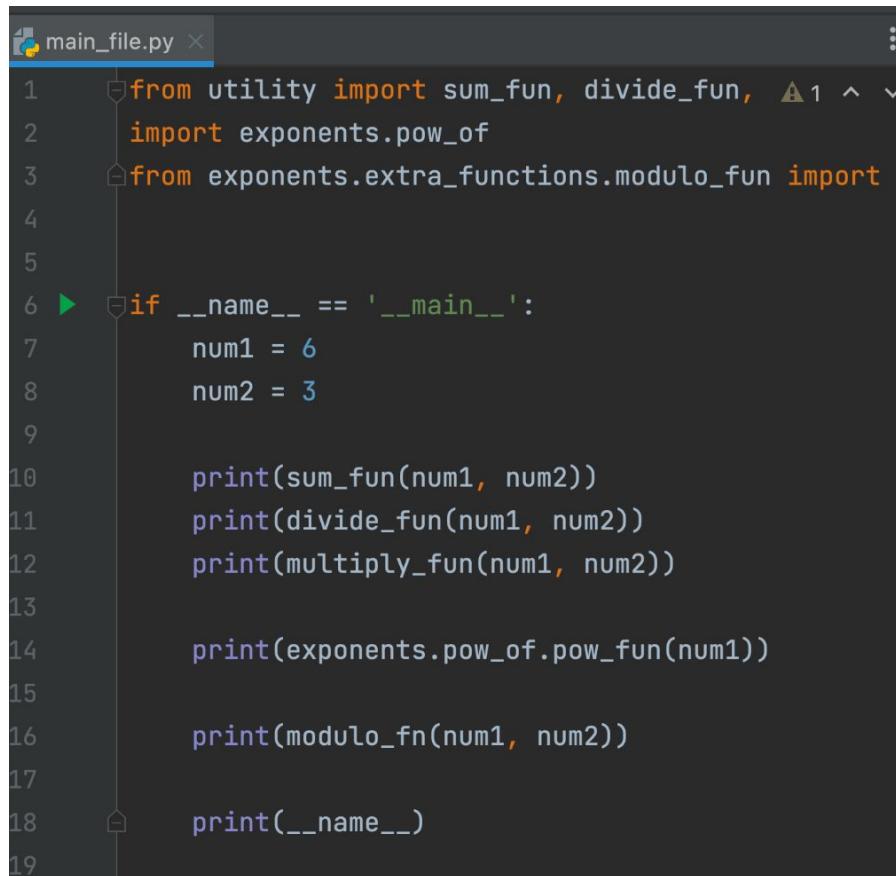
Output
/Users/ri.../chapter.../Users/ri.../chapter.../utility
9
2.0
18
36
0
__main__
Process f...
code 0

As we can see from the figure above, the name of the file is still `__main__` not `main_file`, but why is this? In PyCharm, when we want to run the code, we usually click the RUN button and it will ask which file we want to execute. The file we are selecting will be the main Python, and that is why

the `__main__` was printed because the current file we are running is the main file regardless of what name it has. Now let's discuss this famous line that you will see:

```
if __name__ == '__main__':
```

This if-statement will be true only if this line is located in the main file, it will not work in the utility or any other files that are not main. We can now wrap the rest of the code like this:

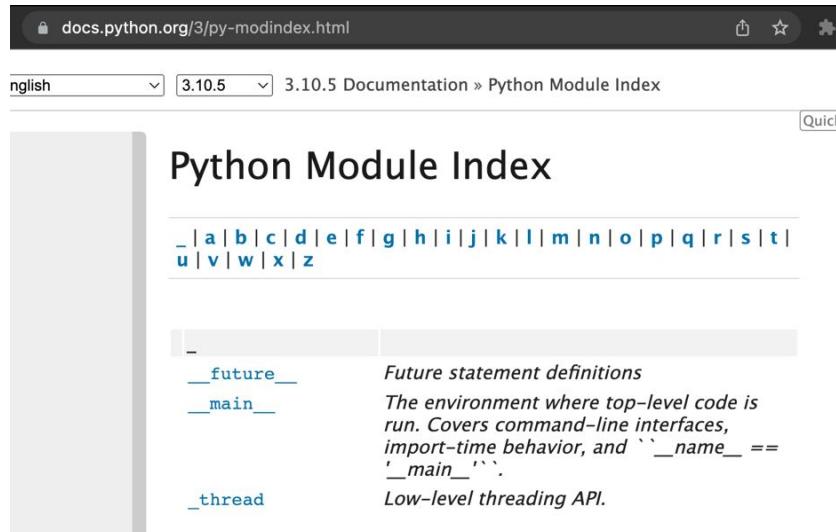


```
main_file.py
1  from utility import sum_fun, divide_fun, ^
2      import exponents.pow_of
3  from exponents.extra_functions.modulo_fn import ^
4
5
6  ▶ if __name__ == '__main__':
7      num1 = 6
8      num2 = 3
9
10     print(sum_fun(num1, num2))
11     print(divide_fun(num1, num2))
12     print(multiply_fun(num1, num2))
13
14     print(exponents.pow_of.pow_fun(num1))
15
16     print(modulo_fn(num1, num2))
17
18     print(__name__)
19
```

This means if the file we are running is the main file in Python, the indented code will be executed.

## Python Built-in Modules

It is time to learn about Python built-in modules. There is a huge list of these built-in modules if we visit the official Python documentation:



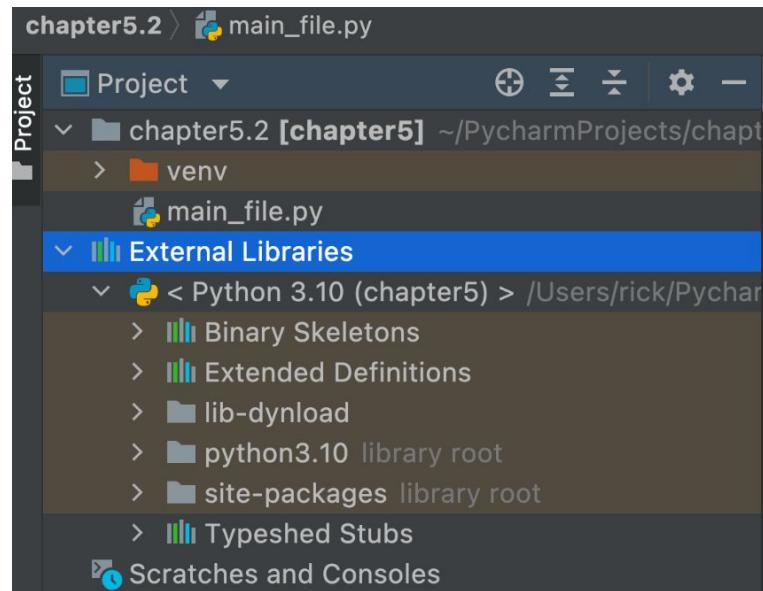
The figure above is just a tiny screenshot of the documentation that lists the modules, and we don't have to remember all of them, but we need to get familiar with what Python offers so we can use these built-in features when we need them.

The link to the documentation:

<https://docs.python.org/3/py-modindex.html>

Python built-in modules are loaded automatically as the shell starts. We have seen the Python built-in functions but we have such functions in the modules as well, we just need to know how to import them. Why do you need Python built-in modules? Imagine you want to create an application to send emails to a list of customers, you can check if this type of module exists so you can use it and finish your project very fast instead of trying to code the entire application from scratch. These modules come with the installation of Python but we need to import them if we want to use them in our code. For example, so far, we have created simple math functions but there is a module that deals with complex mathematical calculations so we can use these functions without creating them. The module is called 'math'. In other programming languages, these built-in modules are known as libraries and they do the same thing. We can actually see these modules

when we create a new project in PyCharm. In your project, click on the External Libraries list:



You can also find them if you can navigate to ‘python 3.10 library root.’ Yours might be a bit different, but click on it and it will expand and give us a list of installed packages and modules. As an exercise, let’s import some of the existing modules like the math module, and try to print where it comes from:

A screenshot of the PyCharm interface. The top bar shows "main\_file.py" and "Run: main\_file". The code editor on the left contains the following Python code:

```
1 import math
2
3 print(math)
```

The "Run" tool window on the right shows the current working directory and the path to the "math" module:

```
/Users/rick/PycharmProjects
chapter5/venv/bin/python
/Users/rick/PycharmProjects
chapter5.2/main_file.py
<module 'math' from '/Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/math.cpython-310-darwin.so'>
```

From the figure above, we can see the actual path where this module is coming from. Another way to display the list of the available modules is to use the **help** command:

```
help('modules')
```

Here is part of the modules list:

```
Please wait a moment while I gather a list of all available modules...

/Users/rick/PycharmProjects/chapter5/venv/lib/python3.10/site-packages/_dist
.py:36: UserWarning: Setuptools is replacing distutils.
  warnings.warn("Setuptools is replacing distutils.")

__future__          _testmultiphase      getpass           sched
abc                 _thread             gettext           secrets
_aix_support        _threading_local   glob              select
_ast                _tkinter            graphlib         selectors
 asyncio            _tracemalloc       grp               setuptools
_bisect              _uuid              gzip              shelve
_blake2              _virtualenv       hashlib          shlex
_bootsubprocess     _warnings          heapq             shutil
_bz2                 _weakref           hmac              signal
_codecs              _weakrefset        html              site
_codecs_cn           _xxsubinterpreters http              smtpd
_codecs_hk           _xxtestfuzz       idlelib          smtplib
_codecs_iso2022      _zoneinfo         imaplib          sndhdr
_codecs_jp            abc              imgihdr         socket
_codecs_kr            aifc              imp              socketserver
_codecs_tw           antigavity       importlib       sqlite3
_collections         argparse          inspect         sre_compile
_collections_abc      array             io               sre_constants
```

Back to the ‘math’ module, if you are interested to know what functionality this module offers, we can use the **help** keyword again and provide the name of the module as an argument like this:

```
help(math)
```

The help(math) will give us a complete picture of what is in this module. If we are interested in knowing only the methods in this module, we can use the dir() function:

```
print(dir(math))
```

Here is the entire list:

```
/Users/rick/PycharmProjects/chapter5/venv/bin/python
/Users/rick/PycharmProjects/chapter5.2/main_file.py
['__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'acos', 'acosh', 'asin',
 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb',
 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf',
 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm',
 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan',
 'tanh', 'tau', 'trunc', 'ulp']
```

Let's use some of these methods, for example, the log() that will return the natural logarithm of different numbers:

```
print(math.log(2.35657))
print(math.log(3))
print(math.log(1))
```

The output will be:

```
0.8572071720115416
1.0986122886681098
0.0
```

We can even use the `math.pow()` method so it will return the value of x raised to the power of n.

```
print(math.pow(3, 3))
```

Output:

```
27.0
```

And finally, let's find the square root of a number:

```
print (math.sqrt(9))
print (math.sqrt(25))
print (math.sqrt(16))
```

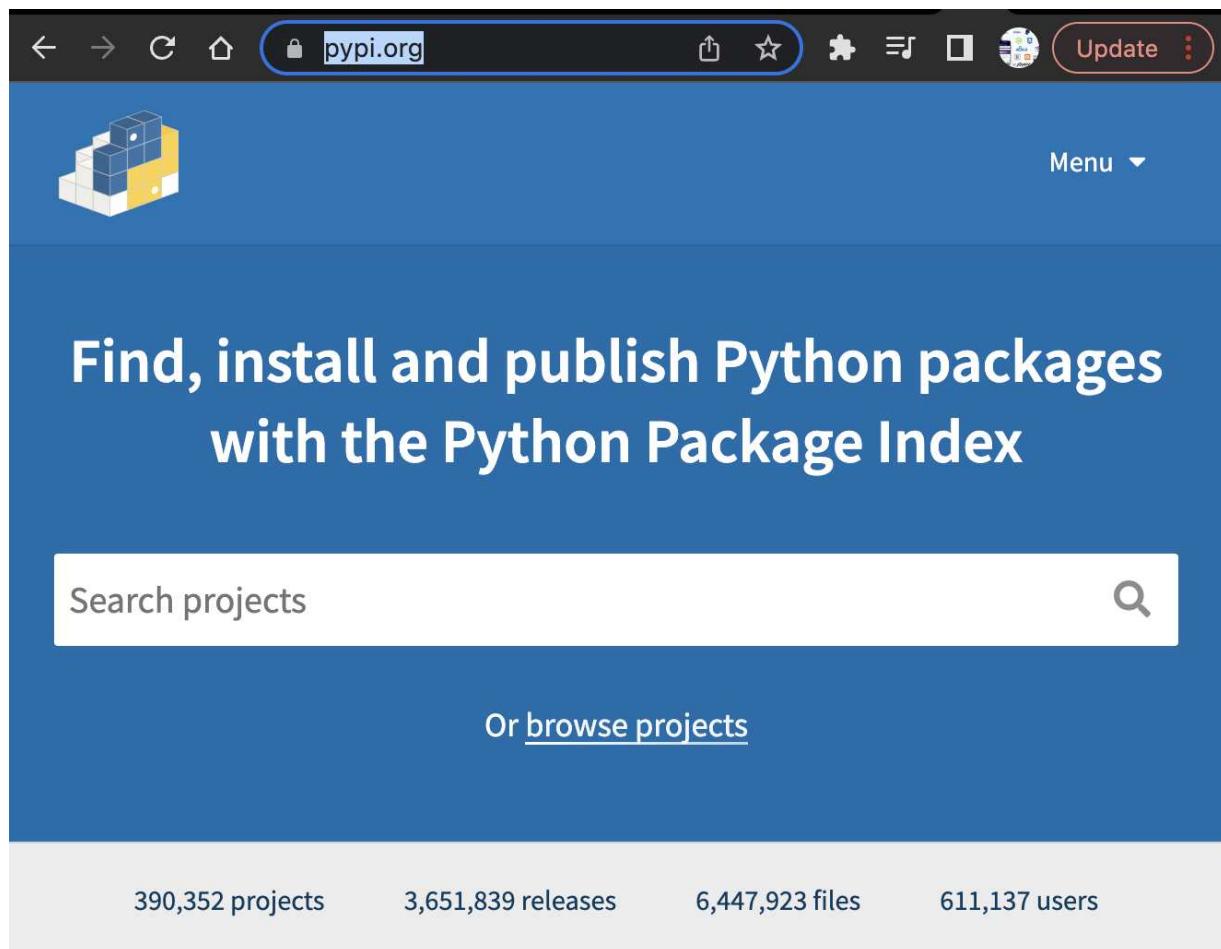
Output:

```
3.0
5.0
4.0
```

## The biggest reason why Python is a great programming language

So far, we have learned how we can create modules and packages, and how to import these modules from the packages as well. In the previous sections, we learned about the built-in library of modules that comes installed with Python. But there is one more thing I want to mention that makes Python one of the greatest languages ever. These are the millions of python modules that are created by various Python developers and shared so we don't need to waste time creating them. These modules can be downloaded and imported into our projects because they are developed by the Python

community. Python developers from all around the world created different modules that we can use. How can we use other developers' code? You can use the 'pip install' command that will enable you to use these modules and pieces of code in your projects in seconds. But where I can find and read more about what other developers created and shared? There is a specific website called 'pypi.org' that is Python Package Index. **PyPi** is a repository of software for Python programming languages that helps us to find and install software that has been developed by the Python community. Here is the link and screenshot to the actual Python Package Index (PyPi):



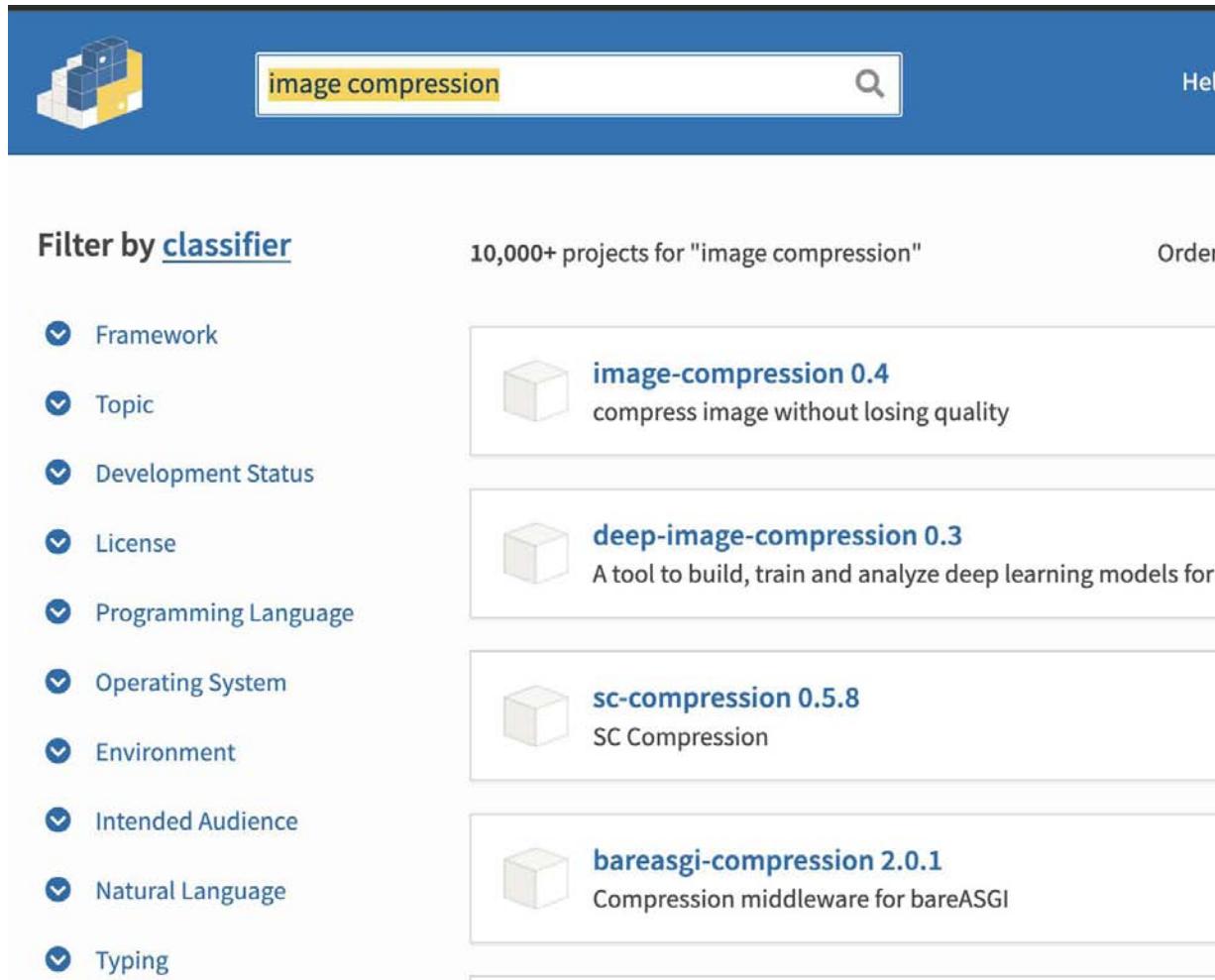
The Python Package Index (PyPI) is a repository of software for the Python programming language.

PyPI helps you find and install software developed and shared by the Python community. [Learn about installing packages](#).

Package authors use PyPI to distribute their software. [Learn how to package your Python code for PyPI](#).

<https://pypi.org/>

At the moment, we have 611,137 users and over 390,352 projects, and these numbers will only grow in the future. This shows you the real power of Python. For example, if you want your website to be faster and all the images to be optimized, you can search if something like that exists in pypi:



The screenshot shows the Python Package Index (PyPi) search results for the query "image compression". The search bar at the top contains the text "image compression". Below the search bar, there is a "Filter by classifier" section with several checkboxes:

- Framework
- Topic
- Development Status
- License
- Programming Language
- Operating System
- Environment
- Intended Audience
- Natural Language
- Typing

On the right side, the search results are displayed in a list:

- image-compression 0.4**  
compress image without losing quality
- deep-image-compression 0.3**  
A tool to build, train and analyze deep learning models for
- sc-compression 0.5.8**  
SC Compression
- bareasgi-compression 2.0.1**  
Compression middleware for bareASGI

You will probably have a list of different projects and all you need to do is find a package that suits your project and install it. Not all of the packages will be suitable and maintained regularly so you need to decide on which of the packages is the best fit for your project. I have shown you these two websites (Python Module Index Documentation and PyPi) so you can find modules and packages that will help you build your program faster.

## Install Python Packages using PyPi repository

We know that **PyPi** is a repository website where we can find, install, and even publish Python packages. Just to give you a demonstration of how to find and install a package using PyCharm, I have done a very simple Google search and I have found something very interesting. The package that we will install is funny but useless, meaning it will only tell us Python jokes and nothing else. There are multiple packages like this on Google. When you type Fun/Weird Python Packages, you will find very interesting packages and experiments. The package that we are going to install is called **pyjokes**. Let us open the website [pypi.org](https://pypi.org) and search for the project called **pyjokes** on the search bar like this:



This will list all of the projects but make sure you click on the one that says 'One-line Jokes for programmers (jokes as service)':

A screenshot of the PyPi search results for "pyjokes". The search bar at the top shows "pyjokes". On the left, there is a sidebar with filter options: "Filter by classifier" (with checkboxes for Framework, Topic, Development Status, License, and Programming Language), "12 projects for 'pyjokes'", and "Order by Relevance". The main area displays two project cards. The first card, "pyjokes 0.6.0", has a description "One line jokes for programmers (jokes as a service)" and a date "Aug 15, 2019". The second card, "pyjokes-hebrew 0.0.5", has a description "with this module you can get a random joke in hebrew!!!!!!!" and a date "Sep 30, 2020". A red double-headed arrow highlights the "pyjokes 0.6.0" card.

As you can see, the pyjokes had a couple of versions and at the moment the latest version is 0.6. If you open this package, you can find additional information:

pyjokes 0.6.0

pip install pyjokes

Latest version

Released: Aug 15, 2019

One line jokes for programmers (jokes as a service)

**Project description**

Navigation

- Project description
- Release history
- Download files

pyjokes

One line jokes for programmers (jokes as a service)

---

**Project links**

Homepage

---

**Statistics**

View statistics for this project via [Libraries.io](#), or by using [our public dataset on Google BigQuery](#)

---

**Installation**

Install the *pyjokes* module with pip.

See the [documentation](#) for installation instructions.

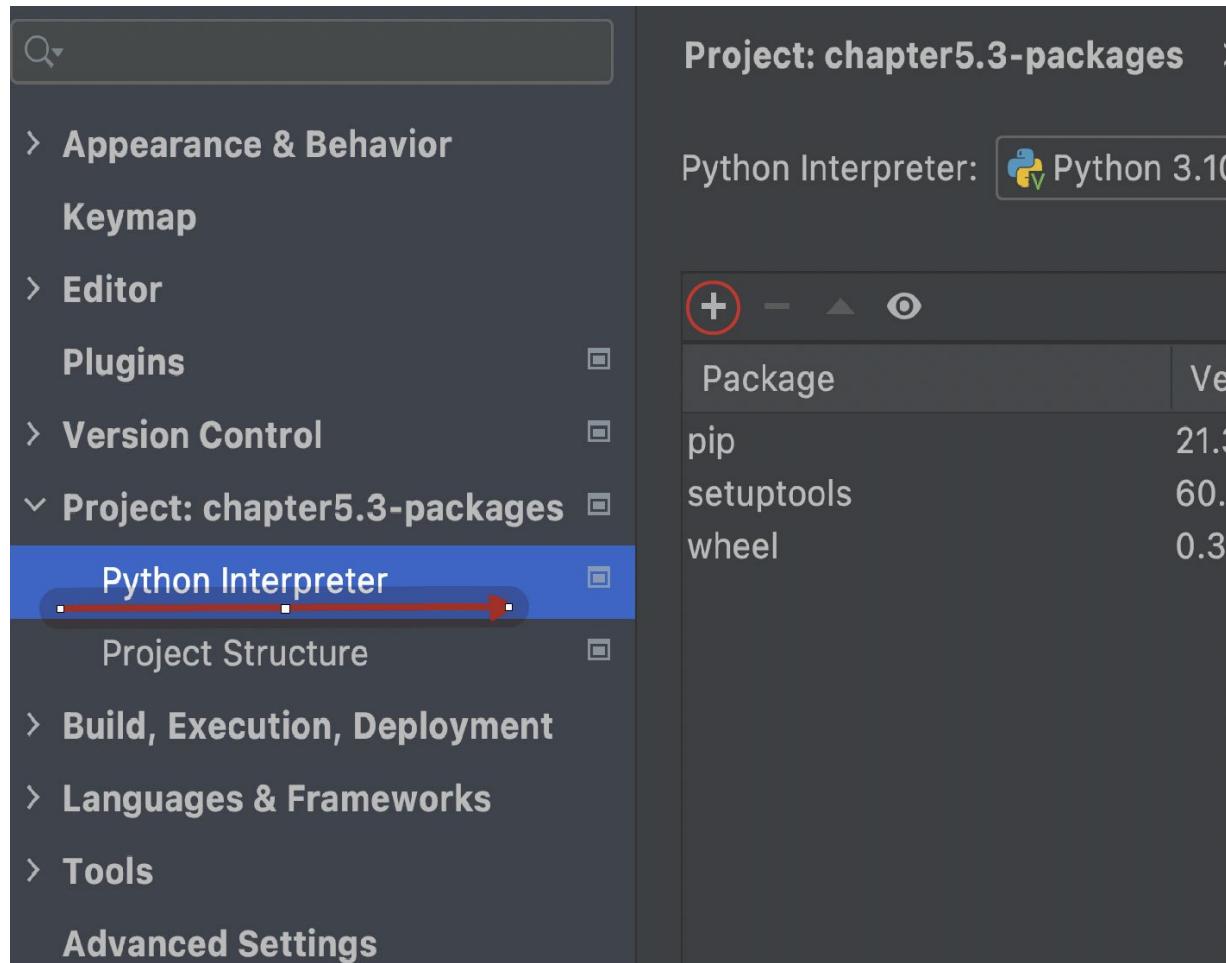
**Usage**

Once installed, simply call *pyjoke* from the command line or add it to your .bashrc file to see a joke every time you open a terminal.

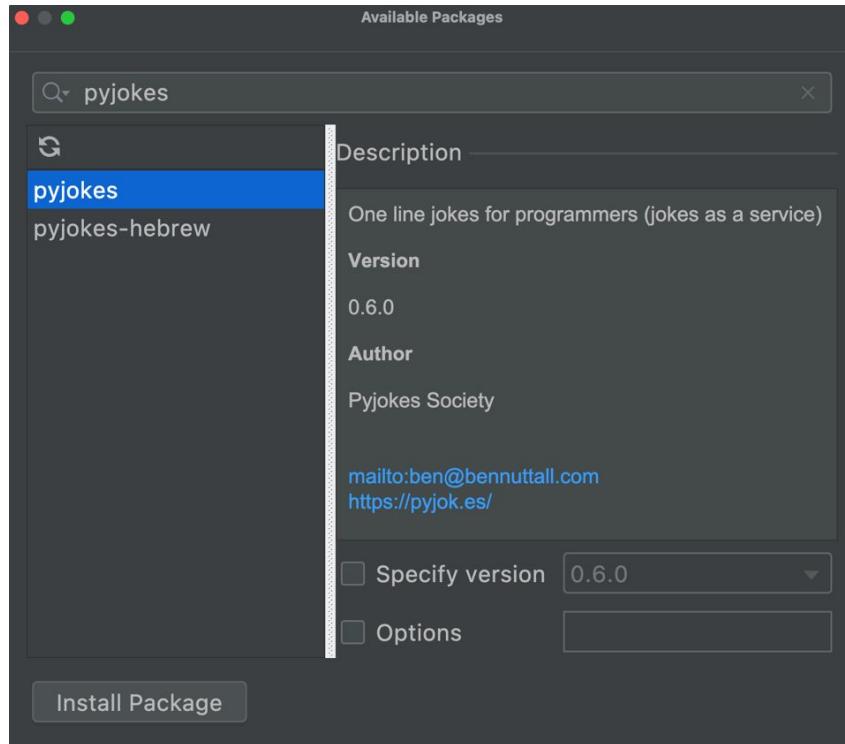
Use the -c flag to get jokes from a specific category. Options:

From this page, you can read more about this project, how to install the pyjokes module with pip, and how to use it. You can download the files if you want, or visit the GitHub page. GitHub is like a Facebook (meta) or Twitter for programmers. It's a social network where you can upload your work so the other developers can clone and download it. You can also see some statistics, for example, how many people liked your project (there will be stars) or how many people forked and used your code in their own projects. There are different ways we can install this package/module, and first I will try to show you how you can do it with PyCharm. I have created

a copy of my previous projects and I called it ‘chapter5.3-packages.’ If you are trying this on your own, you can name your project whatever you want. Now let’s go to the menu and select **PyCharm>Preferences** and **Select Project:** and then find and select the **Python Interpreter:**



As we can see on the right side, we have a few packages already pre-installed like **pip** and **setuptools** so all you need to do is locate the plus sign where you can add or search for a new package, in our case, the package we need to install is **pyjokes**:



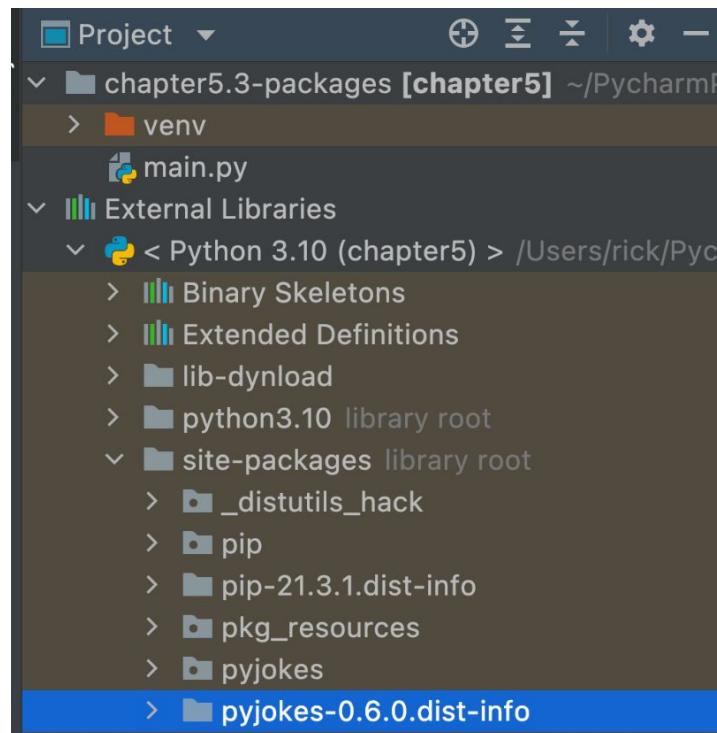
Make sure the right package is selected and click on the button that says **Install Package**. After the installation, you will get a message saying you have successfully installed the **pyjokes** package. We can confirm this if we go back to the Python Interpreter where you could see if the package is listed:

## Project: chapter5.3-packages > Python Interpreter

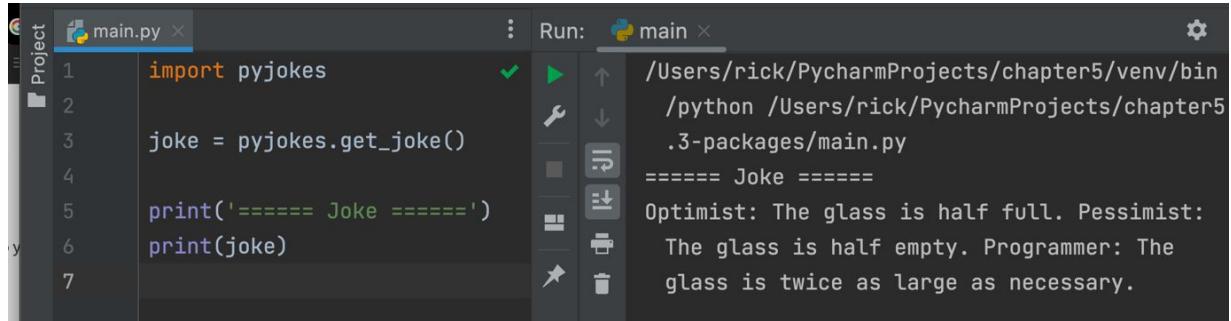
Python Interpreter:  Python 3.10 (chapter5) ~/PycharmPr

Package	Version
pip	21.3.1
pyjokes	0.6.0
setuptools	60.2.0
wheel	0.37.1

The package will also be in the External Libraries > Site Packages > **pyjokes**:



Let's now read the quick description on the [PyPi](#) website and there it says we can access the jokes if we first import the **pyjokes** and then use some of the two functions (**get\_joke** or **get\_joke**):



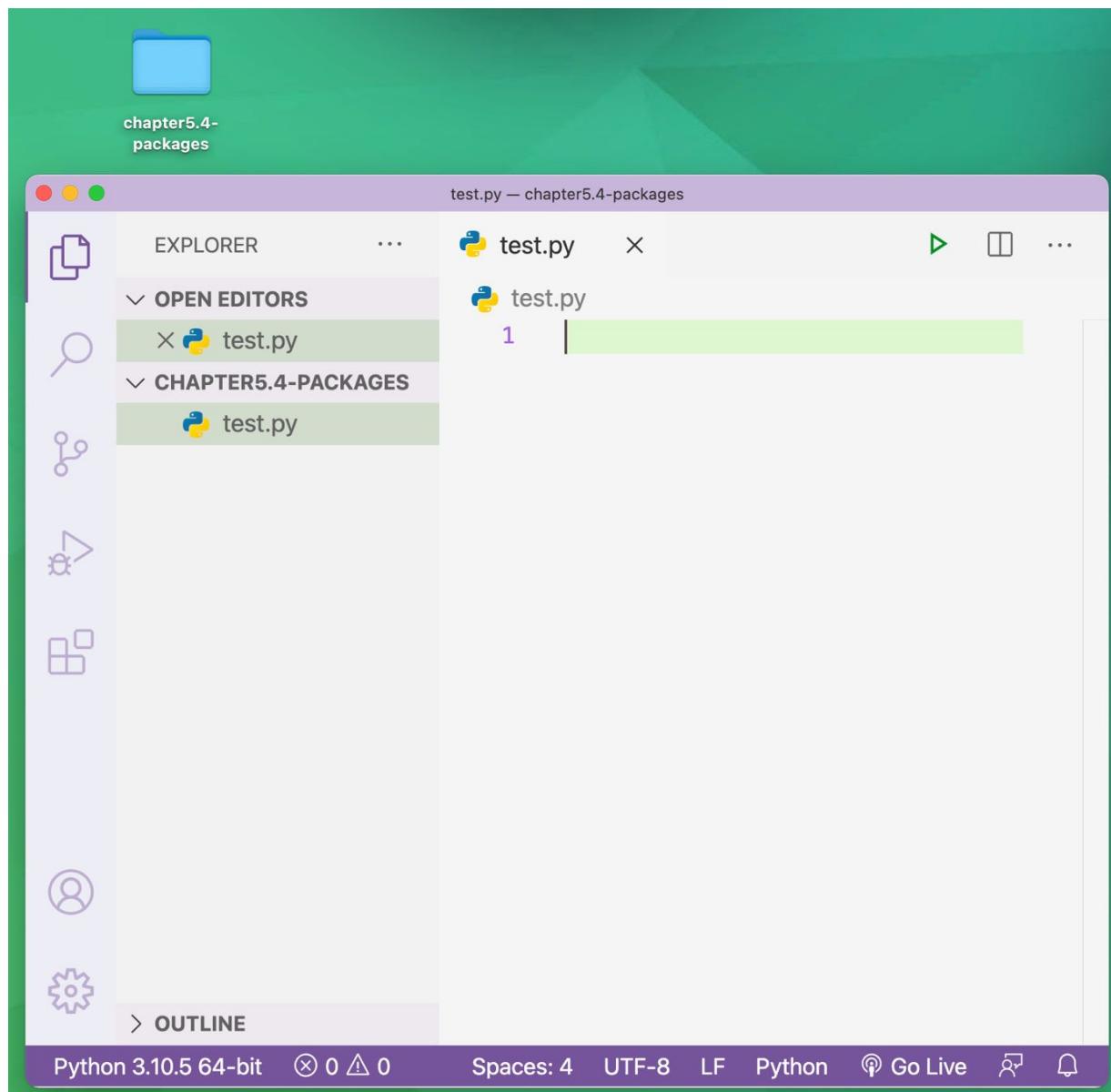
The screenshot shows the PyCharm IDE interface. On the left is the Project tool window, and the main area displays the code in `main.py`:

```
1 import pyjokes
2
3 joke = pyjokes.get_joke()
4
5 print('===== Joke =====')
6 print(joke)
7
```

On the right, the Run tool window shows the output of the run command:

```
/Users/rick/PycharmProjects/chapter5/venv/bin
python /Users/rick/PycharmProjects/chapter5
.3-packages/main.py
===== Joke =====
Optimist: The glass is half full. Pessimist:
The glass is half empty. Programmer: The
glass is twice as large as necessary.
```

As you can see, we can install packages easily with PyCharm but this is not the way most developers install packages. Another, more professional way to install packages is to use the terminal or Command Prompt. On my Desktop I have created a folder called `chapter5.4-packages` and I have opened it with VS Code editor and created a file inside this folder called `test.py`:



The test.py file is empty for now but let's add the pyjokes code from PyCharm:

The screenshot shows the PyCharm IDE interface. The top bar indicates the file is 'test.py — chapter5.4-packages'. The left sidebar has icons for Explorer, Search, and Packages. The 'EXPLORER' section shows 'OPEN EDITORS' with 'test.py' and 'CHAPTER5.4-PACKAGES' with 'test.py'. The main editor window displays the following Python code:

```
1 import pyjokes
2
3 joke = pyjokes.get_joke()
4
5 print('===== Joke =====')
6 print(joke)
7
```

The terminal below shows the output of running the script:

```
rick@Ristes-iMac chapter5.4-packages % python3 test.py
Traceback (most recent call last):
  File "/Users/rick/Desktop/chapter5.4-packages/test.py", line 1, in <module>
    import pyjokes
ModuleNotFoundError: No module named 'pyjokes'
rick@Ristes-iMac chapter5.4-packages %
```

The status bar at the bottom shows 'Python 3.10.5 64-bit' and 'Spaces: 4 UTF-8 LF Python Go Live'.

ModuleNotFoundError because the pyjokes package was installed for the PyCharm IDE only and we can't use it everywhere. So how can we fix this? Well, if we go back to pypi.org we can see there is a pip command to install packages using our terminals or command prompt:

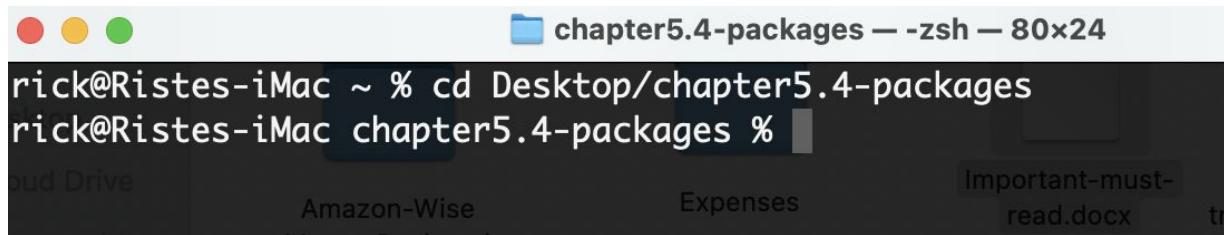


Copy the command ‘pip install pyjokes’ and open the terminal or command prompt. If you forget some of the terminal commands, I suggest you go back and read that section again. Instead of using the built-in terminal in VS Code editor, I will use the machine terminal and check the current directory using the ‘pwd’ command:



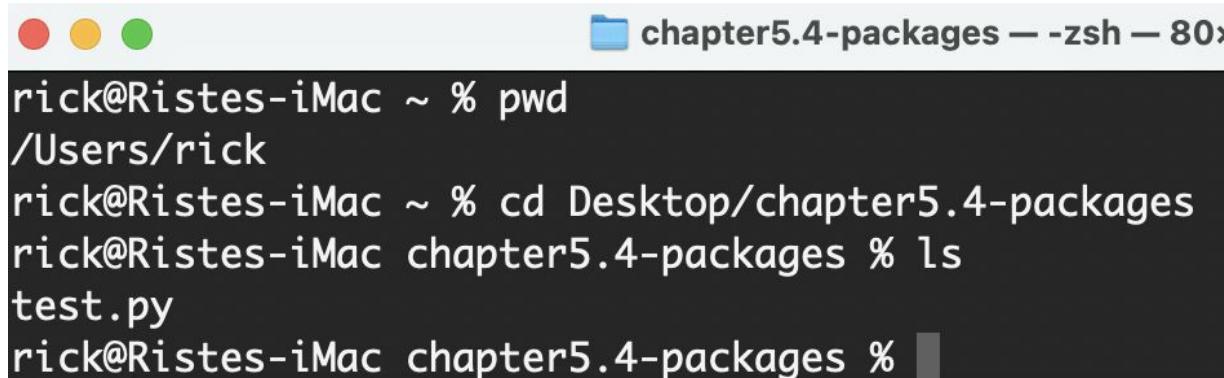
```
rick@Ristes-iMac ~ % pwd  
/Users/rick  
rick@Ristes-iMac ~ %
```

Now I need to go to the Desktop and in the ‘chapter5.4-packages’ folder where my test.py file is:



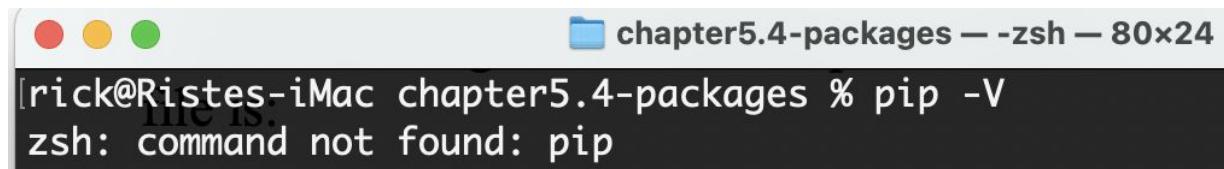
```
rick@Ristes-iMac ~ % cd Desktop/chapter5.4-packages  
rick@Ristes-iMac chapter5.4-packages %
```

As you can see, we are in chapter5.4 folder. Let’s list all available files there:



```
rick@Ristes-iMac ~ % pwd  
/Users/rick  
rick@Ristes-iMac ~ % cd Desktop/chapter5.4-packages  
rick@Ristes-iMac chapter5.4-packages % ls  
test.py  
rick@Ristes-iMac chapter5.4-packages %
```

The test.py file is the only file I have in the folder chapter5.4-pakcages. Finally, we can check the pip version. When we installed Python on our machines, it also installed the **pip**. Let’s check the pip version by typing **pip -V**, and based on your machine this might work or might give you an error like this one:



```
[rick@Ristes-iMac chapter5.4-packages % pip -V  
zsh: command not found: pip
```

If this is the case, we can type pip3 -V and now we should get the right version:

```
rick@Ristes-iMac chapter5.4-packages % pip3 -V  
pip 22.0.4 from /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.1  
0/site-packages/pip (python 3.10)  
rick@Ristes-iMac chapter5.4-packages %
```

For me, this means I can install pyjokes using the pip3 instead of the pip command (pip3 install pyjokes):

```
rick@Ristes-iMac chapter5.4-packages % pip3 install pyjokes  
Collecting pyjokes  
  Using cached pyjokes-0.6.0-py2.py3-none-any.whl (26 kB)  
Installing collected packages: pyjokes  
Successfully installed pyjokes-0.6.0  
WARNING: You are using pip version 22.0.4; however, version 22.2.1 is available.  
You should consider upgrading via the '/Library/Frameworks/Python.framework/Vers  
ions/3.10/bin/python3.10 -m pip install --upgrade pip' command.  
rick@Ristes-iMac chapter5.4-packages %
```

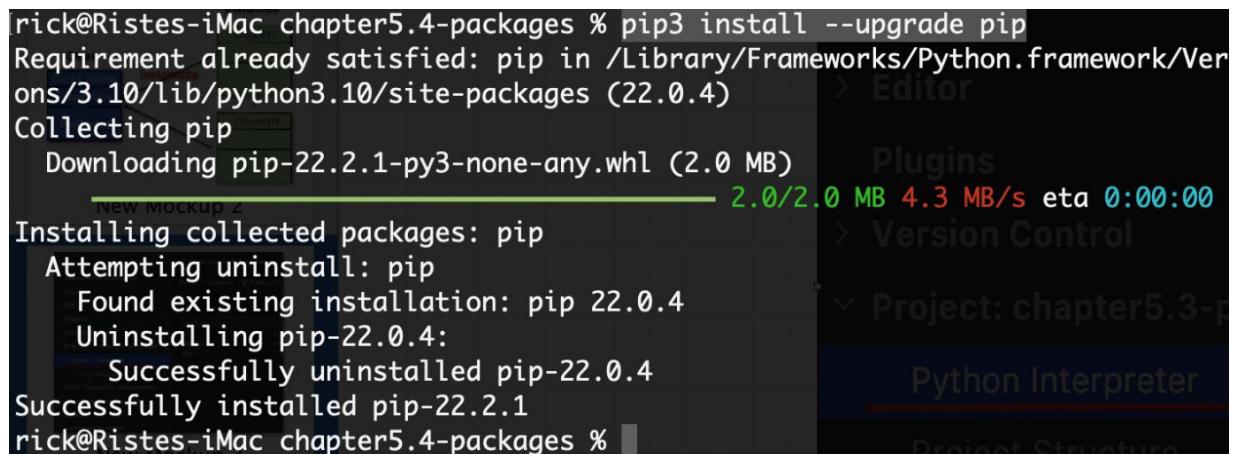
After you run the ‘pip3 install pyjokes,’ you should get the following message ‘Successfully installed pyjokes.’ We can also upgrade package versions as well. As you can see, I have some warnings in green color alerting me that there is a new version of pip available:

**pip install --upgrade pip**

In my case, I need to use the following command:

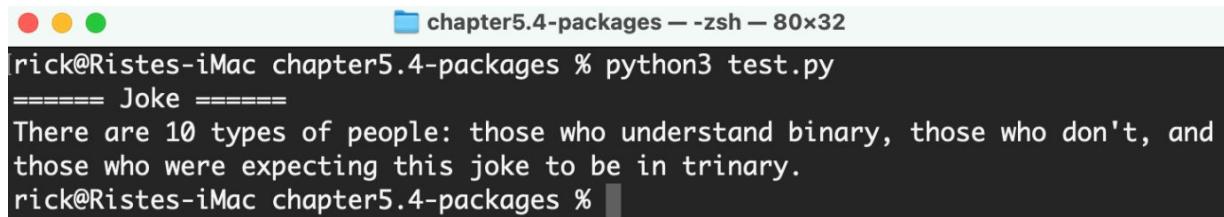
**pip3 install --upgrade pip**

As you can see, after I run the pip3 install command, I have the following window:



```
rick@Ristes-iMac chapter5.4-packages % pip3 install --upgrade pip
Requirement already satisfied: pip in /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages (22.0.4)
Collecting pip
  Downloading pip-22.2.1-py3-none-any.whl (2.0 MB)
    2.0/2.0 MB 4.3 MB/s eta 0:00:00
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 22.0.4
    Uninstalling pip-22.0.4:
      Successfully uninstalled pip-22.0.4
Successfully installed pip-22.2.1
rick@Ristes-iMac chapter5.4-packages %
```

Let's now test if the package pyjokes is actually working and for this, we can use the built-in terminal in VS Code or the machine terminal. I will show you both ways, here is the terminal output:



```
chapter5.4-packages — zsh — 80x32
rick@Ristes-iMac chapter5.4-packages % python3 test.py
===== Joke =====
There are 10 types of people: those who understand binary, those who don't, and those who were expecting this joke to be in trinary.
rick@Ristes-iMac chapter5.4-packages %
```

If we go back to the VS Code editor and use the built-in terminal, you should have this output:

The screenshot shows the VS Code interface. In the Explorer sidebar, there are two entries under 'OPEN EDITORS': 'test.py' and 'CHAPTER5.4-PACKAGES'. The 'CHAPTER5.4-PACKAGES' entry contains a file named 'test.py'. The main editor tab shows the following Python code:

```
1 import pyjokes
2
3 joke = pyjokes.get_joke()
4
5 print('===== Joke =====')
6 print(joke)
7
```

The terminal at the bottom shows the output of running the script:

```
rick@Ristes-iMac chapter5.4-packages % python3 test.py
===== Joke =====
A programmer crashes a car at the bottom of a hill, a b
ystander asks what happened, he says "No idea. Let's pu
sh it back up and try again".
rick@Ristes-iMac chapter5.4-packages %
```

VS Code terminal is working as well and that is basically what I wanted to show you about how to find, install, and run packages from the Python Package Index website. Finally, if you want to uninstall some of the packages you have already installed, you can use the command pip3 uninstall and the name of the package. For example, if you want to uninstall the previous package completely you can use one of the following commands:

**pip uninstall pyjokes**

Or if you are using the pip command only:

**pip3 uninstall pyjokes**

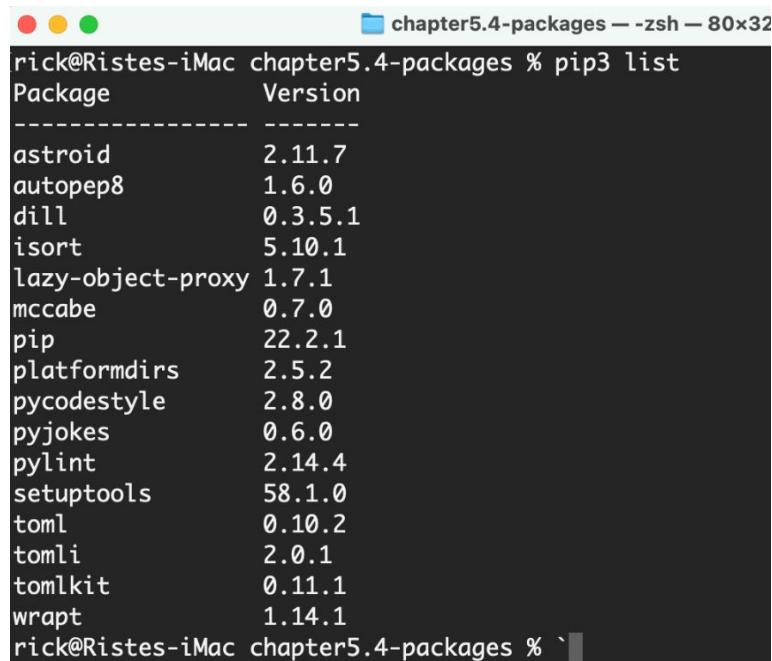
If for some reason you want to install a specific version of a package, you can also do this from your terminal as long as you know the version number. Most of the packages will have a different number like the pyjokes have 0.6.0 and this is called versioning because each new version will bring

something new, maybe some bug was discovered in version 0.5.0 and they rectified and updated it in the new version 0.6.0. Therefore, the version numbers mean something, they are not just random numbers. If you need to install a different version of the package, you can do it like this:

**pip3 install pyjokes==0.5.0**

This will install the previous version of pyjokes. You can also list all of the packages installed using the following command:

**pip3 list**



A screenshot of a terminal window titled "chapter5.4-packages -- zsh -- 80x32". The command "pip3 list" has been run. The output shows a table with two columns: "Package" and "Version". The packages listed are: astroid (2.11.7), autopep8 (1.6.0), dill (0.3.5.1), isort (5.10.1), lazy-object-proxy (1.7.1), mccabe (0.7.0), pip (22.2.1), platformdirs (2.5.2), pycodestyle (2.8.0), pyjokes (0.6.0), pylint (2.14.4), setuptools (58.1.0), toml (0.10.2), tomli (2.0.1), tomlkit (0.11.1), and wrapt (1.14.1).

Package	Version
astroid	2.11.7
autopep8	1.6.0
dill	0.3.5.1
isort	5.10.1
lazy-object-proxy	1.7.1
mccabe	0.7.0
pip	22.2.1
platformdirs	2.5.2
pycodestyle	2.8.0
pyjokes	0.6.0
pylint	2.14.4
setuptools	58.1.0
toml	0.10.2
tomli	2.0.1
tomlkit	0.11.1
wrapt	1.14.1

## **Summary**

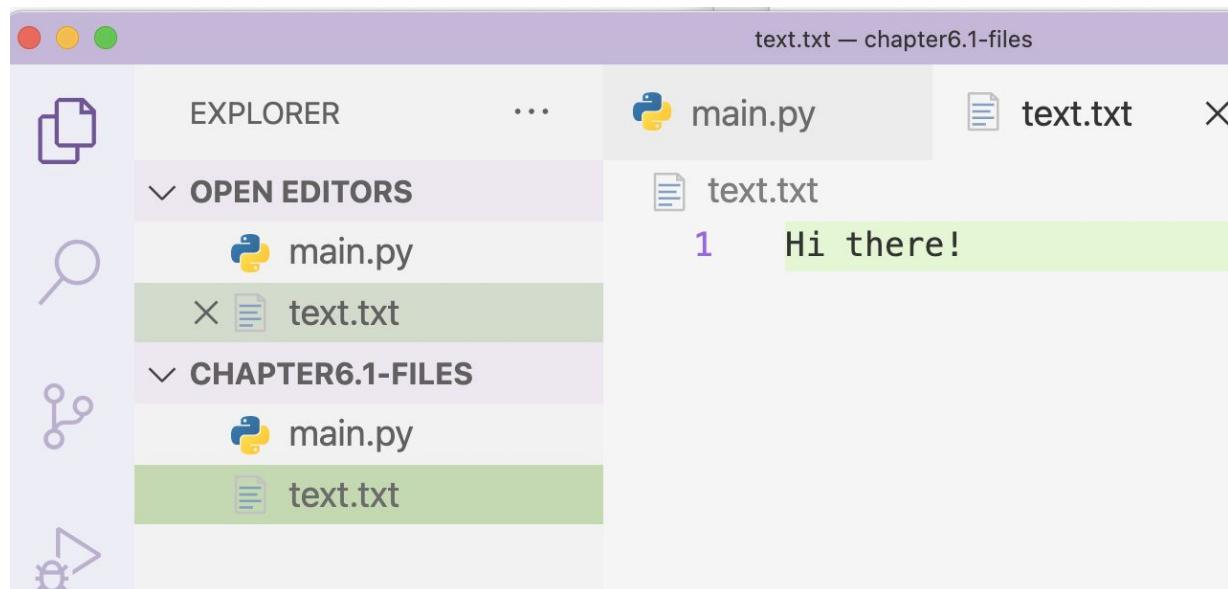
This chapter was one of my favorites because we learned a lot about modules and packages. I hope by now you see the real power of Python because it doesn't matter how great a developer or programmer you are, it will be super hard to shine if you don't have a backup from your community. In the next chapter, we will focus on reading and writing to files.

# Chapter 6 – Working with Files in Python

In most programming languages, you will find the phrase **FILE I/O**, but what does it mean? The letter ‘I’ stands for **input** and the ‘O’ stands for **output**. In this chapter, we will cover how we can work with files in Python. Python has several functions for handling files like open, write, and read. The key function for working with files in Python is the open() function so let’s start with it.

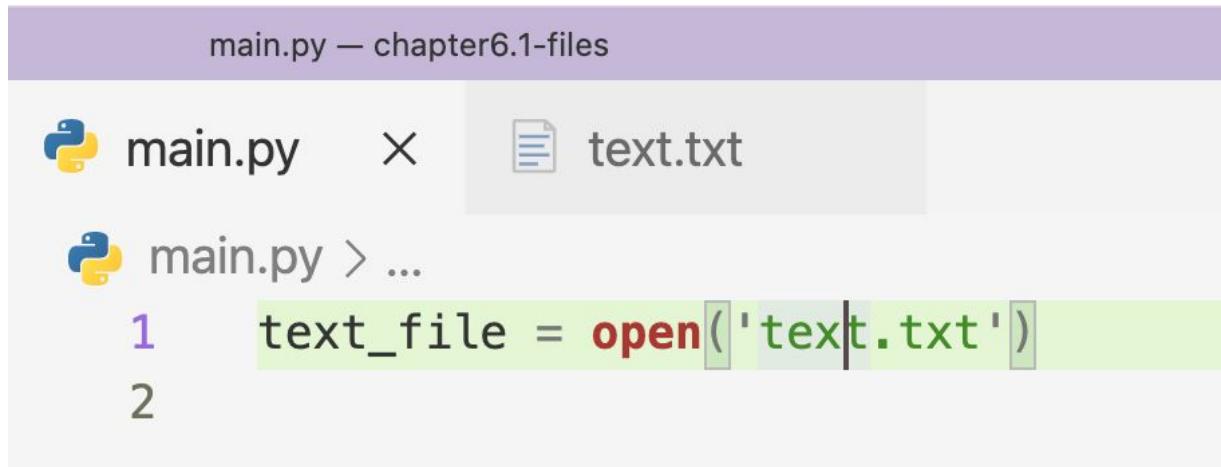
## Open

In this chapter, I will use the VS Code editor instead of the PyCharm because I want you to have a book that uses both development tools. I will create a folder on my desktop called ‘**chapter6.1-files**’ and inside I will add two files, the first one will be a Python file called main.py and the second will be a text file called text.txt file:



As you can see from the figure above, the text.txt file contains the simple text ‘Hi there!’ and the main.py file is empty for now. The idea is to use the

text.txt file and open it from our main.py file and if you want to do this, you can use the open () function that is built-in Python:



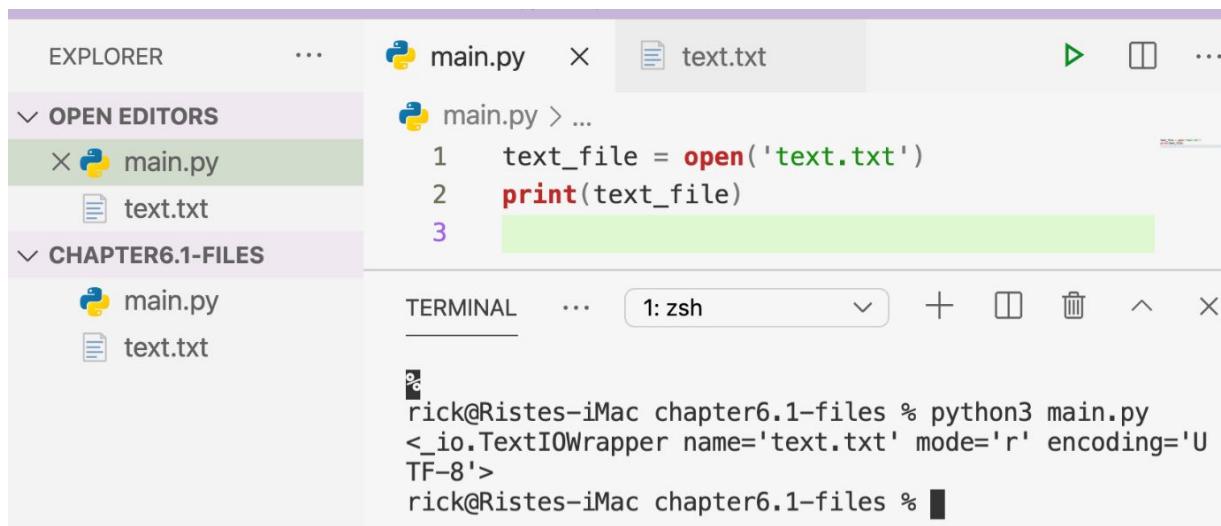
```
main.py — chapter6.1-files
```

main.py    X    text.txt

main.py > ...

```
1  text_file = open('text.txt')
2
```

Here we will use the function open that takes one argument. The argument is the file name and we need to pass it as a string or in quotes (single or double). Whatever the function open() returns, the result will be stored in a variable called text\_file. Let's run the main file and print what we have in the **text\_file** (I will use the VS Code terminal):



EXPLORER    ...    main.py    X    text.txt    ▶    ⚡    ...

OPEN EDITORS

- main.py
- text.txt

CHAPTER6.1-FILES

- main.py
- text.txt

TERMINAL    ...    1: zsh    +    ⚡    ⚡    ⚡    ⚡

```
% rick@Ristes-iMac chapter6.1-files % python3 main.py
<_io.TextIOWrapper name='text.txt' mode='r' encoding='UTF-8'
rick@Ristes-iMac chapter6.1-files %
```

The output is the following:

```
<_io.TextIOWrapper name='text.txt' mode='r' encoding='UTF-8'>
```

We can see that we have an **IO** object called **TextIOWrapper**, and then we have the file name, the mode ‘r’ (we will learn what this means), and finally, the encoding which is **UTF-8**. The encoding tells us how the file is encoded. The goal is to read the content of this file, and luckily for us, Python gives us the function to read called `read()`:

You can run the file if you click on this triangle

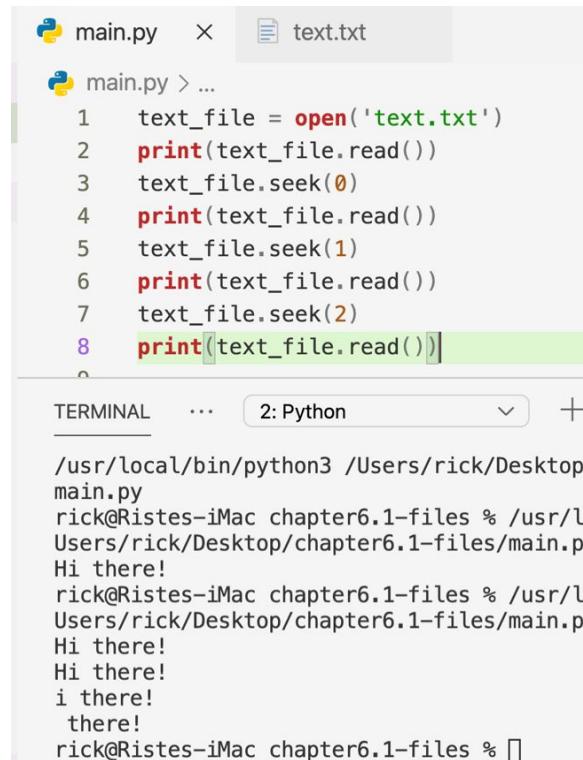
```
main.py -- chapter6.1-files
main.py  x  text.txt
main.py > ...
1  text_file = open('text.txt')
2  print(text_file.read())
3

TERMINAL  ...  2: Python  +  ×
/usr/local/bin/python3 /Users/rick/Desktop/chapter6.1-files/
main.py
rick@Ristes-iMac chapter6.1-files % /usr/local/bin/python3 /
Users/rick/Desktop/chapter6.1-files/main.py
Hi there!
rick@Ristes-iMac chapter6.1-files %
```

As you can see from the figure above, we can click on the triangle to run the code instead of typing in the terminal `python3 main.py`. Regardless of how you execute the `main.py` script, you will have the same output, but be careful, we can read the file only once because of the `open()` function. Python treats the `open` function as a cursor, therefore if you want to read the same file multiple times, you will run into some problems. Okay let’s test this and call the `read()` function multiple times:

```
print(text_file.read())
print(text_file.read())
print(text_file.read())
print(text_file.read())
```

If you run the above code, only the first read function will work and will produce an output, the rest of the read functions will not produce results. This means that the contents of the file are read as a cursor, one character at a time, and at the end of the first read(), the cursor will reach the end of the file. We can fix this if we use the seek() method where we can pass the index which will determine the position where the reading should start. If we want to start from the beginning, the index should be zero:



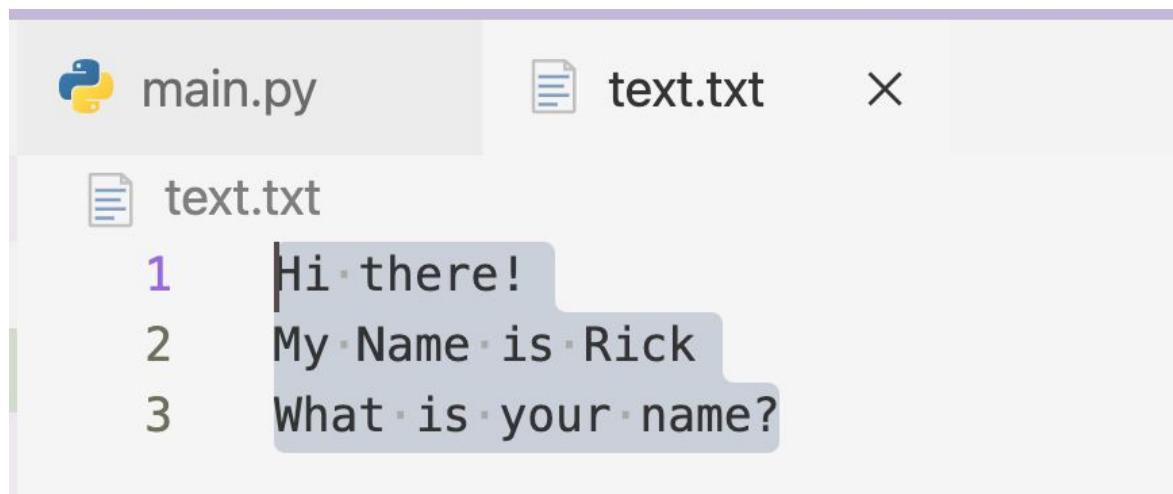
The screenshot shows a Python development environment with two panes. The top pane is a code editor for 'main.py' containing the following code:

```
main.py > ...
1  text_file = open('text.txt')
2  print(text_file.read())
3  text_file.seek(0)
4  print(text_file.read())
5  text_file.seek(1)
6  print(text_file.read())
7  text_file.seek(2)
8  print(text_file.read())
```

The bottom pane is a terminal window titled '2: Python' showing the execution of the script and its output:

```
/usr/local/bin/python3 /Users/rick/Desktop/main.py
rick@Ristes-iMac chapter6.1-files % /usr/l
Users/rick/Desktop/chapter6.1-files/main.p
Hi there!
rick@Ristes-iMac chapter6.1-files % /usr/l
Users/rick/Desktop/chapter6.1-files/main.p
Hi there!
Hi there!
i there!
there!
rick@Ristes-iMac chapter6.1-files %
```

If your text.txt file has multiple lines of content and you need to read the first line only, you can use the method called readline(). Let's add a few more lines in the text.txt file and test this readline() method:



main.py

text.txt

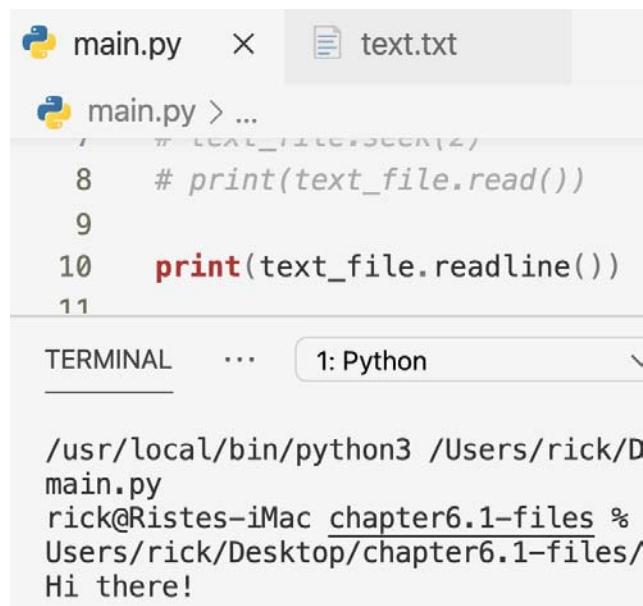
text.txt

1 Hi there!

2 My Name is Rick

3 What is your name?

Let's call the readline() method:



main.py

text.txt

main.py > ...

8 # print(text\_file.read())

9

10 print(text\_file.readline())

11

TERMINAL 1: Python

```
/usr/local/bin/python3 /Users/rick/D  
main.py  
rick@Ristes-iMac chapter6.1-files %  
Users/rick/Desktop/chapter6.1-files/  
Hi there!
```

If you need to read all of the lines, then you can use the method called readlines() but this method will return a List data type with all of the lines from the file. The list will also include the newline character '\n' that we use to create a new line. Let's use this method and check the output:

The screenshot shows a Jupyter Notebook interface. At the top, there are two tabs: 'main.py' and 'text.txt'. Below the tabs, the code in 'main.py' is displayed:

```
8     # print(text_file.read())
9
10    print(text_file.readline())
11    print(text_file.readlines())
12
```

Below the code, there is a terminal window showing the output of running the script:

```
/usr/local/bin/python3 /Users/rick/Desktop/
main.py
rick@Ristes-iMac chapter6.1-files % /usr/local/bin/python3 /Users/rick/Desktop/chapter6.1-files/main.py
Hi there!

rick@Ristes-iMac chapter6.1-files % /usr/local/bin/python3 /Users/rick/Desktop/chapter6.1-files/main.py
Hi there!

['My Name is Rick\n', 'What is your name?']
```

As a good practice, we need to close the file at the end because you might want to use it somewhere else so it's a common practice to close the file/files that are opened. To do this, we can call the `close()` method:

```
text_file.close()
```

In the next section, I will teach you another way to open and close a file using the ‘with’ statement.

## Write and Append to Files

In the previous section, we saw how can we open files, read from them using some methods, and finally we had to close them so they can be used again. Today there is a much nicer way of doing I/O in order to avoid closing the file manually. This new way of opening and closing the files can be achieved using the ‘with’ statement. I will duplicate the `main.py` file and rename it to `main1` so I can show you the new features from this section. Let's open the `text.txt` file using the `with` statement:

The screenshot shows a Python code editor with three tabs: 'main1.py', 'main.py', and 'text.txt'. The 'main1.py' tab is active, displaying the following code:

```
1 with open('text.txt') as new_text_file:  
2     print(new_text_file.read())  
3
```

The output window below shows the terminal session:

```
Hi there!  
My Name is Rick  
What is your name?  
rick@Ristes-iMac chapter6.1-files %
```

As you can see, I don't need to close the file at the end using the 'with' statement. This is what developers are using today.

## Write to a file

Remember at the beginning of this chapter when we printed the file, we had access to the mode='r':

```
<_io.TextIOWrapper name='text.txt' mode='r' encoding='UTF-8'>
```

The mode ='r' stands for reading and we can even specify the mode directly into the 'with' statement like this:

```
with open('text.txt', mode='r') as new_text_file:  
    print(new_text_file.read())
```

The 'r' is the default parameter, we don't need to specify it, but if you are interested in writing to a file, we need to use the mode='w', where 'w' stands for writing into a file. If we change the mode to write and run the

same file, we will end up with an error because we are trying to read from a file but the mode is set to write:

The screenshot shows a code editor with two tabs: 'main1.py' and 'main.py'. The 'main1.py' tab contains the following code:

```
1  with open('text.txt', mode='w') as new_text_file
2  |     print(new_text_file.read())
3
```

The 'text.txt' file is shown in the background. Below the code editor is a terminal window titled '1: Python' with the following output:

```
File "/Users/rick/Desktop/chapter6.1-files/main1.py",
2, in <module>
    print(new_text_file.read())
io.UnsupportedOperation: not readable
rick@Ristes-iMac chapter6.1-files %
```

Before we fix this problem, please check if your text.txt file content is there. Well, the text file would be empty because we tried to write to a file without specifying what needs to be written and that will delete the existing content, therefore you must be careful when you are using the write mode. What if you want to read and write in the same file? We can use the mode='r+' if we want to read and write. Okay let's write some content back to a text.txt file first:

The screenshot shows a code editor interface. At the top, there are two tabs: 'main1.py' and 'text.txt'. The 'main1.py' tab is active, displaying the following Python code:

```
1  with open('text.txt', mode='r+') as new_text_file:  
2      content = new_text_file.write('Hi there!\n'  
3                                         'my name is Rick\n'  
4                                         'What is your name?')  
5
```

The code uses the `open` function with mode `'r+'` to open the file `'text.txt'` for reading and writing. It then writes three lines of text to the file: "Hi there!", "my name is Rick", and "What is your name?". Lines 1, 2, and 3 are highlighted in green, while line 4 is highlighted in yellow.

Below the code editor is a terminal window titled '1: Python'. It shows the command `/usr/local/bin/python3 /Users/rick/Desktop/chapter6.1-files/main1` being run, followed by the output: `rick@Ristes-iMac chapter6.1-files % /usr/local/bin/python3 /Users,rp/chapter6.1-files/main1.py` and `rick@Ristes-iMac chapter6.1-files %`.

If we open the `text.txt` file, we can see that we have successfully written the new content:

The screenshot shows a code editor interface with two tabs: 'main1.py' and 'text.txt'. The 'text.txt' tab is active, displaying the following text:

```
1  Hi there!  
2  my name is Rick  
3  What is your name?
```

The first line, "1 Hi there!", is highlighted in green, indicating it is the most recent content written to the file.

This works perfectly but let's do some further tests, for example, I don't think the content is good enough and I want to write a new text and overwrite the previous one. I can just simply do this:

The screenshot shows a code editor with two tabs: 'main1.py' and 'text.txt'. The 'main1.py' tab contains the following code:

```
1 with open('text.txt', mode='r+') as new_text_file:  
2     content = new_text_file.write('Hi readers!')  
3
```

The 'text.txt' tab shows the file content after running the script:

```
/usr/local/bin/python3 /Users/rick/Desktop/chapter%  
rick@Ristes-iMac chapter6.1-files % /usr/local/bin/python3 /p/chapter6.1-files/main1.py  
rick@Ristes-iMac chapter6.1-files % /usr/local/bin/python3 /p/chapter6.1-files/main1.py
```

Let's open the text file and check if the new content 'Hi readers!' has been added:

The screenshot shows a code editor with two tabs: 'main1.py' and 'text.txt'. The 'text.txt' tab displays the following content:

```
1 Hi readers!  
2 What is your name?
```

As you can see from the figure above, we have the new content plus some bits of the old content as well. Why is this happening? The reason this is happening is because the 'with' statement will reset the open() method or will close it for us so the cursor will be back at zero index and we actually add the content at the beginning of the old existing one.

## Append

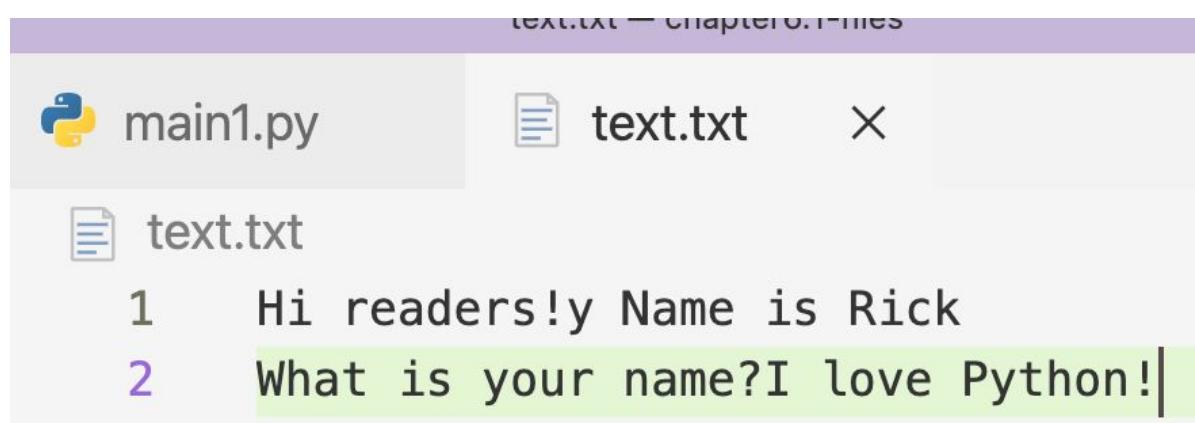
When we want to add text or content at the end of the existing content, we can use the append mode. All we need to add is the letter ‘a’ in the mode like this:

```
mode='a'
```

For example, let’s append the text ‘I love Python’ to the existing file:

```
with open('text.txt', mode='a') as new_text_file:  
    content = new_text_file.write('I love Python!')
```

And if we open the text.txt file, the text should be added at the end:



The screenshot shows a code editor with two tabs: 'main1.py' and 'text.txt'. The 'main1.py' tab contains the following code:

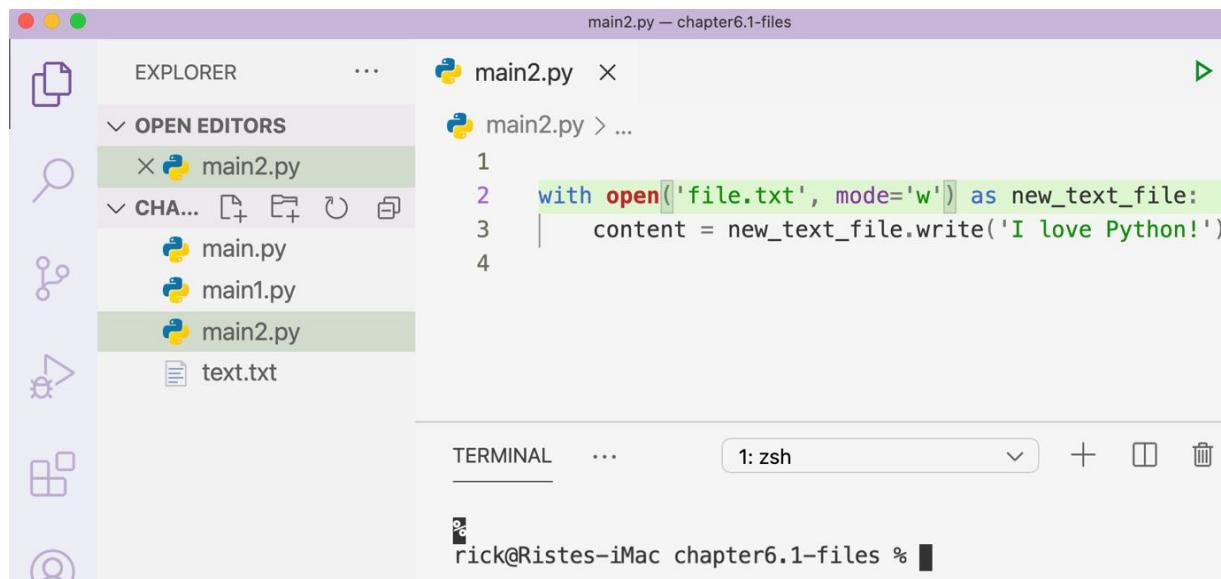
```
with open('text.txt', mode='a') as new_text_file:  
    content = new_text_file.write('I love Python!')
```

The 'text.txt' tab shows the contents of the file:

```
1 Hi readers!y Name is Rick  
2 What is your name?I love Python!
```

The text starting with '2' is highlighted in green.

As an exercise, copy the main1.py file and create a new file called main2.py. I would like you to experiment a little bit. In the open() method, instead of loading the file **text.txt**, try using a file that does not exist in your directory/folder, for example, use a file called ‘file.txt’. After you have done everything, run the file and observe what will happen. Here is how the main2.py file looks before I click on the run button :



```
main2.py — chapter6.1-files
```

EXPLORER ...

OPEN EDITORS

- main2.py
- CHAPTER6.1-FILES

  - main.py
  - main1.py
  - main2.py
  - text.txt

main2.py X

main2.py > ...

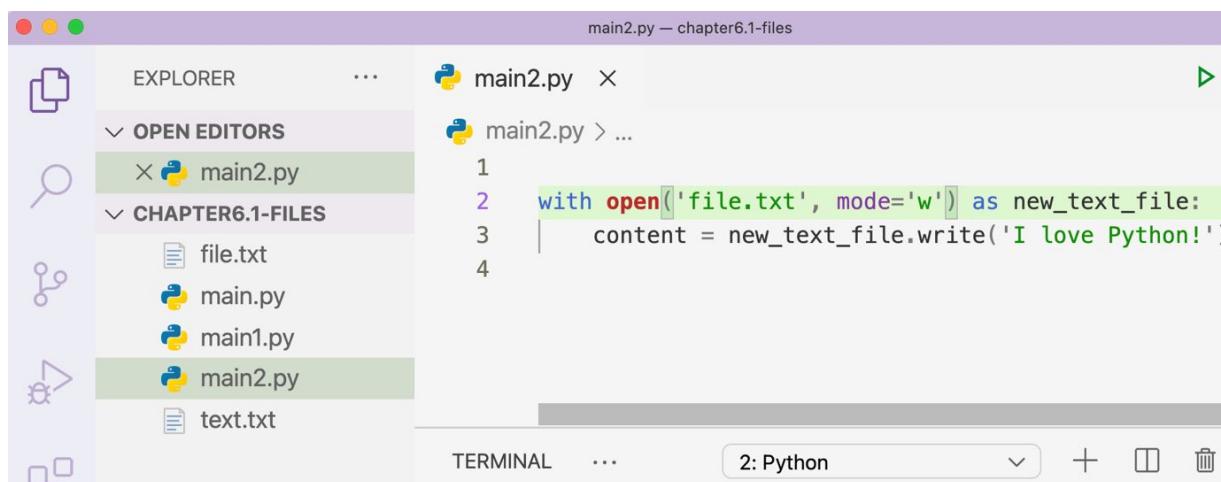
```
1
2 with open('file.txt', mode='w') as new_text_file:
3     content = new_text_file.write('I love Python!')
4
```

TERMINAL ...

1: zsh + □ ─

rick@Ristes-iMac chapter6.1-files %

If I run the main2.py file, this will be the output:



```
main2.py — chapter6.1-files
```

EXPLORER ...

OPEN EDITORS

- main2.py
- CHAPTER6.1-FILES

  - file.txt
  - main.py
  - main1.py
  - main2.py
  - text.txt

main2.py X

main2.py > ...

```
1
2 with open('file.txt', mode='w') as new_text_file:
3     content = new_text_file.write('I love Python!')
4
```

TERMINAL ...

2: Python + □ ─

As you can see from the figure above, a new file called **file.txt** was created with ‘I love Python!’ as content. The big question is why this happened. We tried to write to a file that doesn’t exist, so the open method will try to find this file but because it can’t it will create a new file for us and add ‘I love Python!’. Let’s delete the ‘file.txt’ and change the mode to append and run the file again. The result will be identical for both modes. Finally, let’s delete the file.txt and use the read and write mode=’r+’ and run the file, what do you think will happen? Here is the output:

The screenshot shows the VS Code interface with the following details:

- EXPLORER** sidebar: Shows files in the current workspace, including `main2.py`, `main2.py > ...`, `main.py`, `main1.py`, `main2.py` (selected), and `text.txt`.
- EDITOR**: A code editor window titled "main2.py — chapter6.1-files" containing the following Python code:

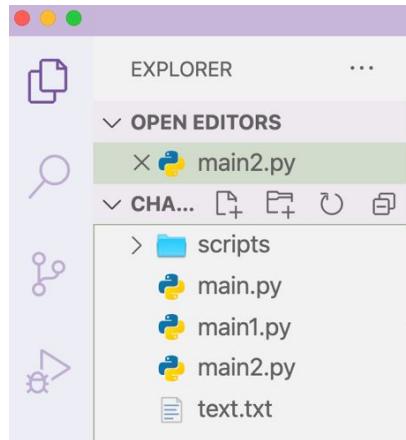
```
1
2 with open('file.txt', mode='r+') as new_text_file:
3     content = new_text_file.write('I love Python!')
```
- TERMINAL**: A terminal window titled "2: Python" showing the command-line output of running the script:

```
rick@Ristes-iMac chapter6.1-files % /usr/local/bin/python3 /Users/rick/Desktop/chapter6.1-files/main2.py
rick@Ristes-iMac chapter6.1-files % /usr/local/bin/python3 /Users/rick/Desktop/chapter6.1-files/main2.py
Traceback (most recent call last):
  File "/Users/rick/Desktop/chapter6.1-files/main2.py", line 2, in <module>
    with open('file.txt', mode='r+') as new_text_file:
FileNotFoundError: [Errno 2] No such file or directory: 'file.txt'
rick@Ristes-iMac chapter6.1-files %
```

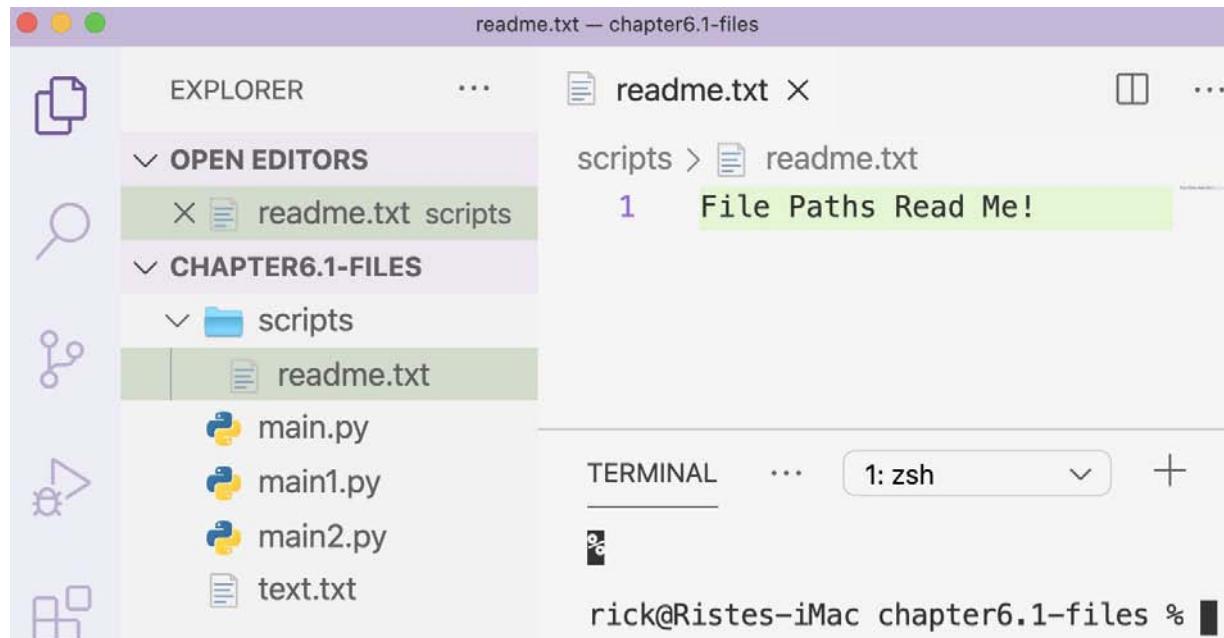
From the figure above, you can see that no file was created and we got the `FileNotFoundException`. This is happening because we are trying to read from a file that doesn't exist.

## File Paths

In this section, we will discuss the importance of file paths. The paths are very important and they can get complex sometimes because the files we are trying to open/load/read are not always located in the same level/directory as the main file. So far, our files (`main.py`, `main1`, `main2`, `text.txt`, `file.txt`) were located in the same folder but we cannot expect to have all of the files like this in the same directory or folder. Let's create a folder inside the `chapter6.1-files` and call it 'scripts':



Inside this folder, let's create a file called `readme.txt` where we add the following simple text:



Great! Now let's create a new `main3.py` file and try to open the `readme.txt` file like we have done it so far:

The screenshot shows a Python code editor interface. On the left, there's a sidebar with icons for file operations like copy, paste, search, and share. The main area has tabs for 'readme.txt' and 'main3.py'. The 'main3.py' tab is active, displaying the following code:

```
1
2 with open('readme.txt', mode='r') as new_text_file:
3     print(new_text_file.read())
4
```

Below the code editor is a terminal window titled '2: Python'. It shows the command being run and the resulting error message:

```
/usr/local/bin/python3 /Users/rick/Desktop/chapter6.1-files/main3.py
rick@Ristes-iMac chapter6.1-files % /usr/local/bin/python3 /Users/rick/Desktop/chapter6.1-files/main3.py
Traceback (most recent call last):
  File "/Users/rick/Desktop/chapter6.1-files/main3.py", line 2, in <module>
    with open('readme.txt', mode='r') as new_text_file:
FileNotFoundError: [Errno 2] No such file or directory: 'readme.txt'
rick@Ristes-iMac chapter6.1-files %
```

As you can see, we are trying to open a file that does not exist at the same level as the main3.py file. That is why we are getting the FileNotFoundError error. The easy solution is to list the directory name in our case ‘scripts’ in the actual path and add a forward slash before the file name. I know this sounds confusing and difficult but here is what you need to write:

```
with open('scripts/readme.txt', mode='r') as new_text_file:
    print(new_text_file.read())
```

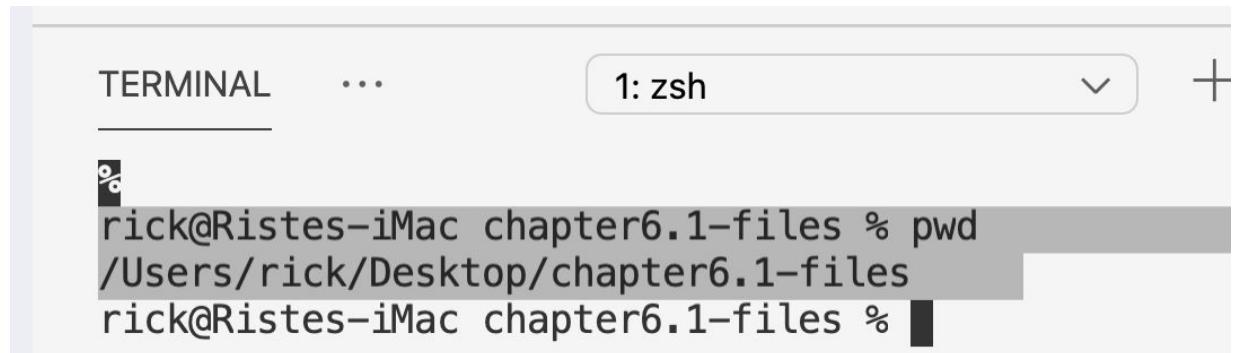
If you run the main3.py file after we specified the correct path to the file, you will get the output we wanted:

## File Paths Read Me!

Sometimes for Windows users instead of a forward slash, we need to use a backslash like this to load the correct file:

```
with open('scripts\readme.txt', mode='r') as new_text_file:
    print(new_text_file.read())
```

This is known as the **relative path**. There is another path known as the **absolute path** and for us to find out the absolute path where the files are located, we can open the terminal or use the built-in VS Code terminal and type the **pwd** command or **cd** for Windows:



The screenshot shows the VS Code interface with a terminal window open. The tab bar at the top says "TERMINAL" and "1: zsh". The terminal window displays the following text:  
%  
rick@Ristes-iMac chapter6.1-files % pwd  
/Users/rick/Desktop/chapter6.1-files  
rick@Ristes-iMac chapter6.1-files %

As you can see, my absolute path will be '/Users/rick/Desktop/chapter6.1-files' and instead of using the relative path, I can use the absolute path (your path will be different from mine):

```
main3.py > ...  
1  
2   with open('/Users/rick/Desktop/chapter6.1-files/scripts/readme.txt', mode='r') as new_text_file:  
3     print(new_text_file.read())
```

If you run the file with the absolute path, you should have the same output. The absolute path is not that common compared to relative paths because they are nicer and shorter to write. As a developer, you will see this relative path:

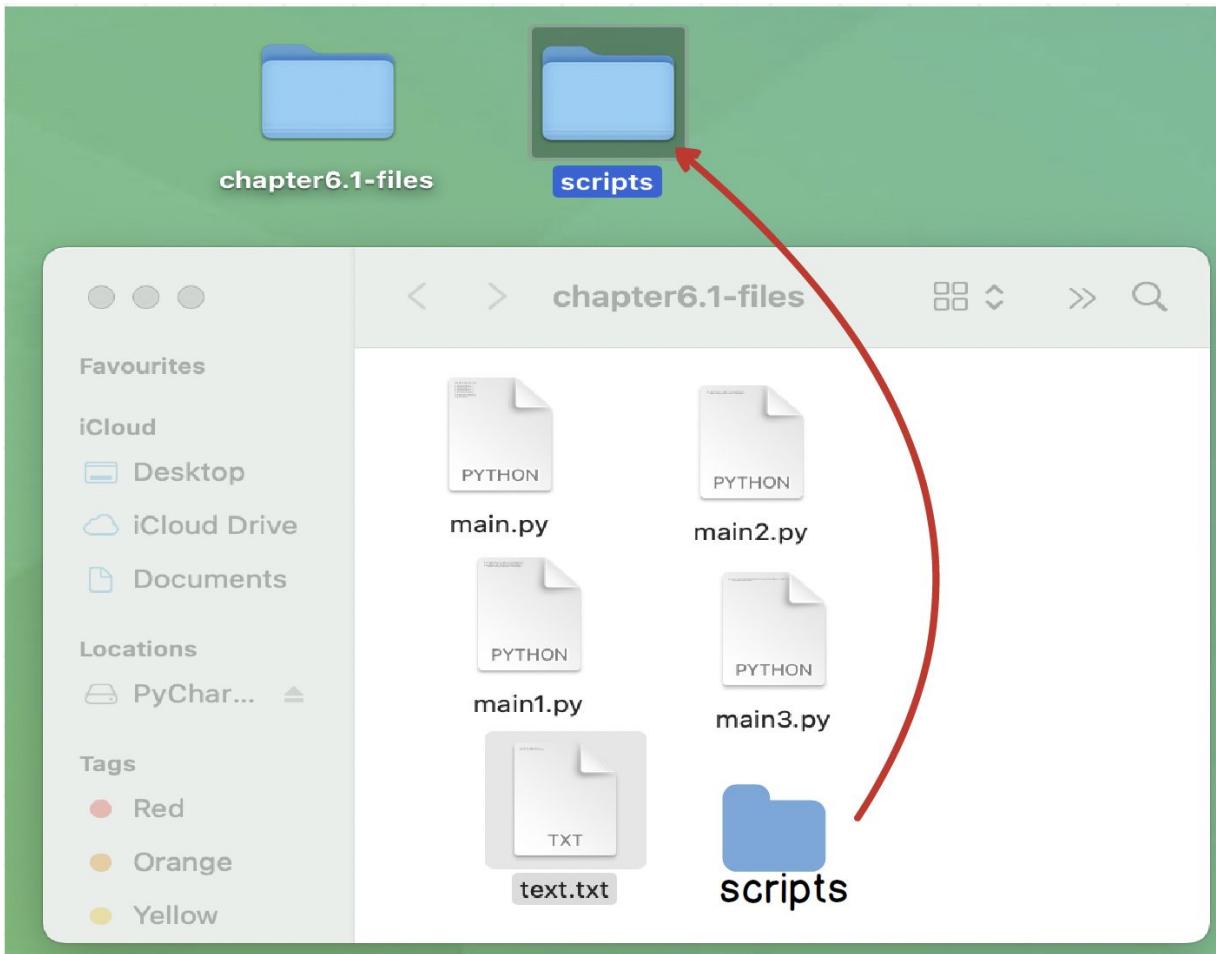
```
with open('./scripts/readme.txt', mode='r') as new_text_file:  
  print(new_text_file.read())
```

The dot and forward slash at the front means start looking from the current folder and that is what I have always used when I'm dealing with the file paths. Another important syntax is double dots and forward slash which means move up one directory from the current directory:

```
with open('../scripts/readme.txt', mode='r') as new_text_file:
```

```
print(new_text_file.read())
```

This will mean look for a folder called scripts that is outside the 'chapter6.1-files' current directory. In order for this to work, I will drag and drop the scripts folder from chapter6.1 to desktop like in the figure below and use the double dots forward slash to load the file:



Let us run the main3 file and read the content :

```
12  # one level up
13  with open('../scripts/readme.txt', mode='r') as new_text_file:
14      print(new_text_file.read())
15
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

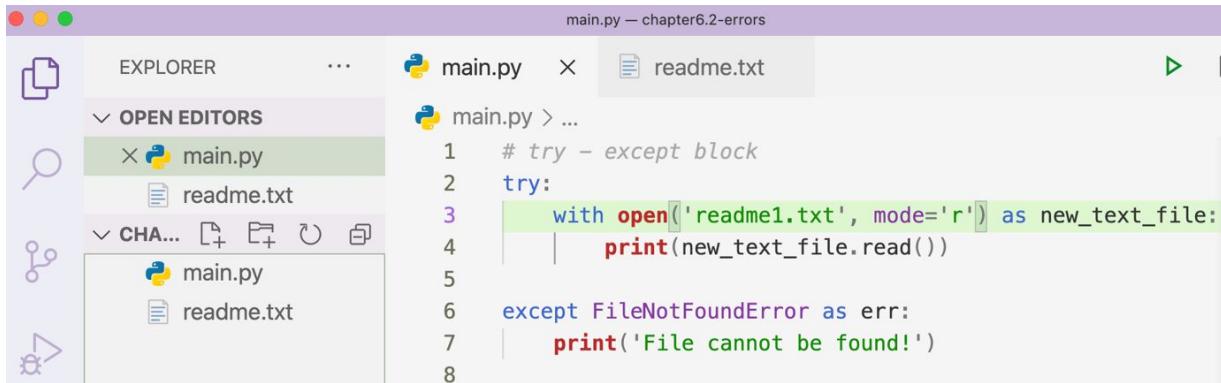
```
/usr/local/bin/python3 /Users/rick/Desktop/chapter6.1-files/main3.py
rick@Ristes-iMac chapter6.1-files % /usr/local/bin/python3 /Users/rick/I
File Paths Read Me!
```

In Python, since version 3.4, we have something called **pathlib** which stands for Object-oriented filesystem paths. This library will allow us to manipulate the paths so we can use the same path syntax on every operating system. You can read more about **pathlib** and how to use it from the official python documentation:

<https://docs.python.org/3/library/pathlib.html>

## Try – Except block for error handling

Try – Except block is perfect to catch and handle errors that can occur during the I/O operations. Whatever code you want to test should be wrapped inside the try-block and the except block is where you catch and handle the errors if any. In JavaScript, we have the try-catch block and it uses the exact same logic. The same concept is in many other programming languages. I have created another folder called chapter6.2-errors and inside I copied the readme.txt file and main.py file. Now let's try out the try-except-block:



The screenshot shows a code editor window titled "main.py — chapter6.2-errors". The left sidebar has icons for Explorer, Search, and Share. The "OPEN EDITORS" section shows two files: "main.py" and "readme.txt". The main editor area contains the following Python code:

```
1 # try - except block
2 try:
3     with open('readme1.txt', mode='r') as new_text_file:
4         print(new_text_file.read())
5
6 except FileNotFoundError as err:
7     print('File cannot be found!')
```

As you can see, we have everything indented correctly and inside the try-block is where we will always put the code we want to test or try out, and in the except block is where we will catch the errors and so far we have seen the FileNotFoundError when the file we were trying to open does not exist. Here is the complete code because sometimes the images can get blurry:

```
# try - except block
try:
    with open('readme1.txt', mode='r') as new_text_file:
        print(new_text_file.read())

except FileNotFoundError as err:
    print('File cannot be found!')
```

If we run this file, the Interpreter will see that we have a try-except block and it will go inside and try and run the code. If we look at the open () method, we can see that the file readme1.txt doesn't exist in the current directory but the readme.txt file exists. There will be an error thrown in the try-block and that error will be handled by the except-block where we just print the error message. Python comes with pre-defined default error messages for different types of errors. Now if we want to access the default error message, we can use a variable that is declared after the 'as' keyword (err). Think of this variable as a container for storing the default error message and instead of printing the custom message we can print the 'err' variable. We can name this variable differently if we want:

```
# try - except block
try:
```

```
with open('readme1.txt', mode='r') as new_text_file:  
    print(new_text_file.read())  
  
except FileNotFoundError as err:  
    print(err)
```

Let's try to run the code and observe the output:

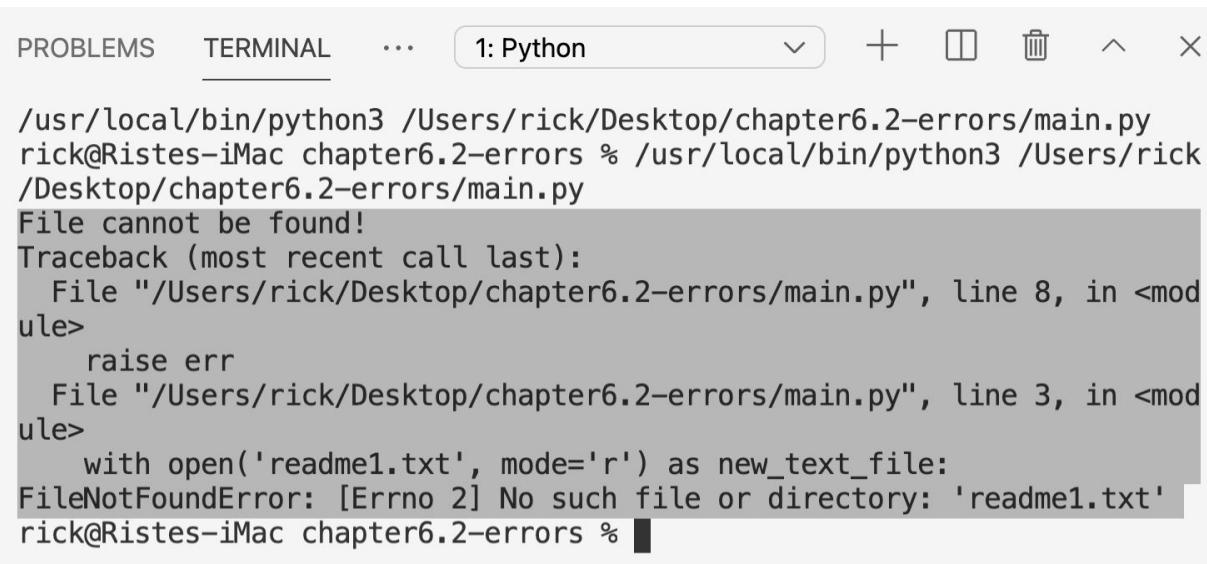


```
PROBLEMS TERMINAL ... 1: Python + □ └ ━ ^ ×  
  
/usr/local/bin/python3 /Users/rick/Desktop/chapter6.2-errors/main.py  
rick@Ristes-iMac chapter6.2-errors % /usr/local/bin/python3 /Users/rick  
/Desktop/chapter6.2-errors/main.py  
File cannot be found!  
rick@Ristes-iMac chapter6.2-errors %
```

As you can see, whatever we put in the except block was printed because there was an error opening and reading this file. There is something else that can be done in the except block and that is to raise the current error like this:

```
# try - except block  
try:  
    with open('readme1.txt', mode='r') as new_text_file:  
        print(new_text_file.read())  
  
except FileNotFoundError as err:  
    print('File cannot be found!')  
    raise err
```

The output now will be slightly changed and more informative:



A screenshot of a terminal window titled "1: Python". The terminal shows the following output:

```
/usr/local/bin/python3 /Users/rick/Desktop/chapter6.2-errors/main.py
rick@Ristes-iMac chapter6.2-errors % /usr/local/bin/python3 /Users/rick/Desktop/chapter6.2-errors/main.py
File cannot be found!
Traceback (most recent call last):
  File "/Users/rick/Desktop/chapter6.2-errors/main.py", line 8, in <module>
    raise err
  File "/Users/rick/Desktop/chapter6.2-errors/main.py", line 3, in <module>
    with open('readme1.txt', mode='r') as new_text_file:
FileNotFoundException: [Errno 2] No such file or directory: 'readme1.txt'
rick@Ristes-iMac chapter6.2-errors %
```

Another usual error that occurs is the `IOError`. This error can occur during any of the modes, like reading and writing to a file. Any problem performing the IO operations will throw this error:

```
# try - except block
try:
    with open('readme1.txt', mode='r') as new_text_file:
        print(new_text_file.read())

except FileNotFoundError as err:
    print('File cannot be found!')
    raise err
except IOError as err1:
    print('IOError!')
    raise err1
```

Other common errors in Python are:

- `ImportError` – This error is raised when the import statement fails to fetch the module
- `IndexError` – This error is raised when a sequence (list, tuple) index is out of the defined range

- `TypeError` – This error is raised when we try to use operations or functions to an object from a type that is not appropriate
- `NameError` – This error is raised when the local or global name cannot be found.
- `KeyError` – This error is raised when we try to access an item from a dictionary with a key that cannot be found

This is how we can handle different errors and I hope that you have now learned how to handle any error you might get when working with files.

## **Summary**

Congratulations! We have learned a lot in this chapter. I'm confident that now you can perform any of the I/O operations and handle the errors using the try and except block.

# Chapter 7 – Error Handling

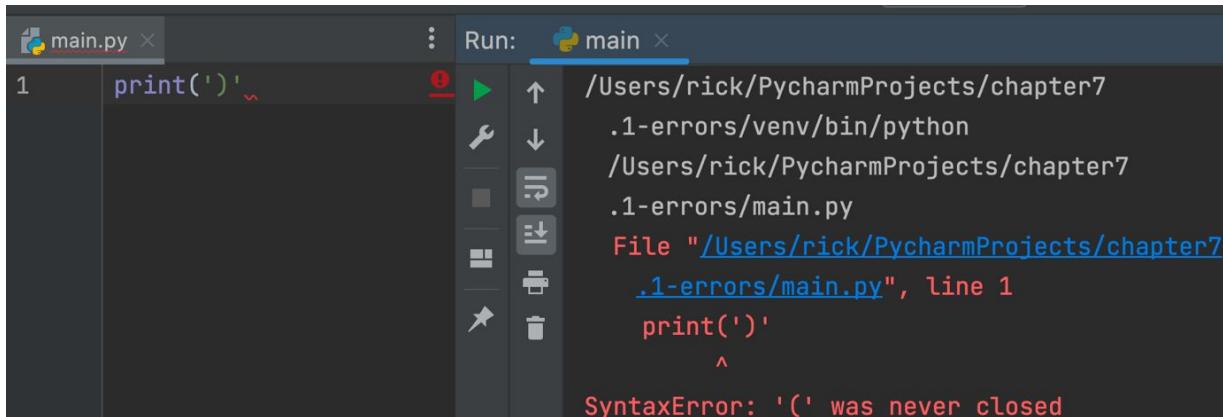
In this final chapter, we will learn how we can handle errors that we might get in our code. Being a programmer is challenging, not because of the amount of code you need to write, but because you need to write good quality code without errors. Code without errors is something we should always strive to achieve, but I will tell you this, it's not that simple and errors can occur anytime. The key point is what we need to do after we have identified those errors and what mechanism we should use to handle them so we can avoid breaking our program. Let's start learning how to deal with these errors.

## Errors in Python

In our code so far, we have seen few errors and I have made those errors on purpose because you should not be afraid of them. The most important part is to learn what they mean and after you identify their meaning you need to take appropriate actions to handle them. For example, the most basic print function if not written properly will throw us an error:

```
# 1 SyntaxError  
  
print()
```

If we run this file:

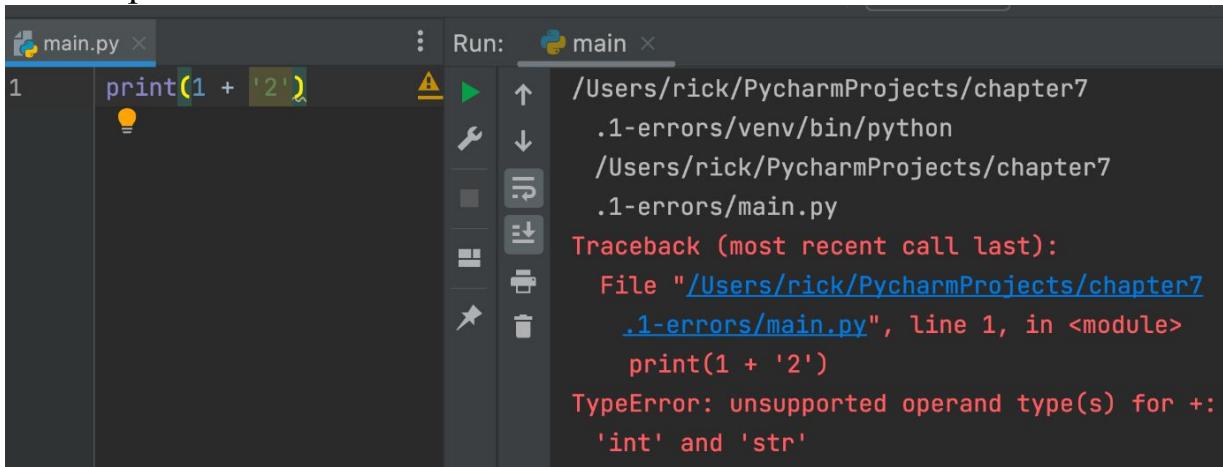


The screenshot shows a PyCharm interface. On the left is the code editor with a file named 'main.py' containing the line 'print(''. The right side shows the 'Run' tool window with the current run configuration set to 'main'. The output pane displays the error message: 'File "/Users/rick/PycharmProjects/chapter7.1-errors/main.py", line 1 print('' SyntaxError: '(' was never closed'. The status bar at the bottom also shows the error message.

As you can see, we have a **SyntaxError** because we haven't closed the print function as it should. Another everyday error that can occur is the **TypeError** that is raised by the Python Interpreter because we are trying to use two different data types that are not compatible with each other:

```
# 2 TypeError  
  
print(1 + '2')
```

The output:



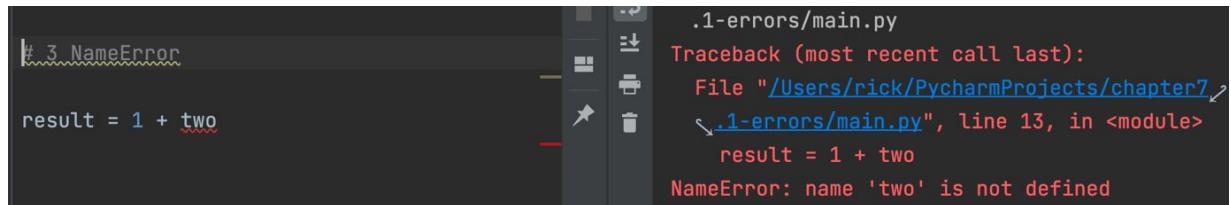
The screenshot shows a PyCharm interface. On the left is the code editor with a file named 'main.py' containing the line 'print(1 + '2')'. The right side shows the 'Run' tool window with the current run configuration set to 'main'. The output pane displays the error message: 'File "/Users/rick/PycharmProjects/chapter7.1-errors/main.py", line 1, in <module> print(1 + '2') TypeError: unsupported operand type(s) for +: 'int' and 'str''. The status bar at the bottom also shows the error message.

Another error that you might get is called the name error (NameError). I have listed these errors in the previous sections but let me give you another example:

```
# 3 NameError
```

```
result = 1 + two
```

As you can see, we are trying to calculate the integer value with a variable that is not being defined. The output:



The screenshot shows a PyCharm code editor with a dark theme. On the left, there is a code editor window containing the following Python code:

```
# 3 NameError
result = 1 + two
```

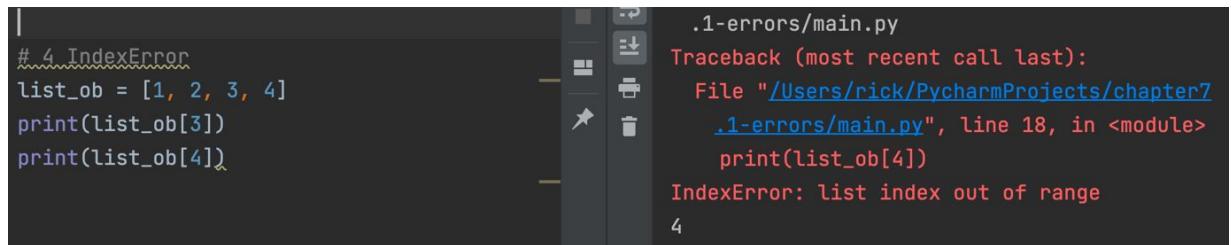
To the right of the code editor is a terminal window showing the output of running the script. The terminal output includes:

```
.1-errors/main.py
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/chapter7/.1-errors/main.py", line 13, in <module>
    result = 1 + two
NameError: name 'two' is not defined
```

The index error is an error that is common when we have a list data type and we try to access an element with an index that is out of the list range. Example:

```
# 4 IndexError
list_ob = [1, 2, 3, 4]
print(list_ob[3])
print(list_ob[4])
```

Output:



The screenshot shows a PyCharm code editor with a dark theme. On the left, there is a code editor window containing the following Python code:

```
# 4 IndexError
list_ob = [1, 2, 3, 4]
print(list_ob[3])
print(list_ob[4])
```

To the right of the code editor is a terminal window showing the output of running the script. The terminal output includes:

```
.1-errors/main.py
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/chapter7/.1-errors/main.py", line 18, in <module>
    print(list_ob[4])
IndexError: list index out of range
4
```

Similarly, we can do innocent mistakes when working with dictionaries. Usually, it can happen that the key we are trying to access from a dictionary is not there. This will raise KeyError and here is one example where we misspelled the ‘model’ keyword:

```
# 5 KeyError
```

```
car_dict = {  
    'car': 'Ford',  
    'model': 'Mustang',  
    'engine': '5.0',  
}  
  
print(car_dict['car'])  
print(car_dict['Model'])
```

Output:

```
Traceback (most recent call last):  
  File "/Users/rick/PycharmProjects/chapter7  
    .1-errors/main.py", line 28, in <module>  
      print(car_dict['Model'])  
KeyError: 'Model'  
Ford
```

From the figure, I have used the key ‘Model’ instead of the key ‘model’ in all lowercase letters. This will throw a KeyError because there is no key in the dictionary named ‘Model’.

The next error is called ZeroDivisionError and this will happen when we try to divide a number with zero, and for some of you, this might be funny but believe me, these errors can happen more frequently than you think:

```
# 6 ZeroDivisionError  
  
print(3 / 0)
```

The error will be:

```
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/chapter7
    .1-errors/main.py", line 31, in <module>
      print(3 / 0)
ZeroDivisionError: division by zero
```

You can read about the different types of exceptions and what they mean if you visit the following Python documentation:

<https://docs.python.org/3/library/exceptions.html>

These errors will crash our program and will not let the rest of the code finish its execution. At the moment, we have few lines of code and if the error happens in the first line, the code below will never be executed like in the following example:

```
print(1 + '2')

if 5 > 3:
    print('5 is bigger than 3')
```

Although the if statement condition is true, the second print will never be executed because the error that will halt our program happened in the first line. The errors that will stop our code from execution are known as **exceptions**. The Interpreter will go line by line and when he sees this kind of problem, it will raise an exception causing the program to stop. This is why we need to learn how to **handle errors**.

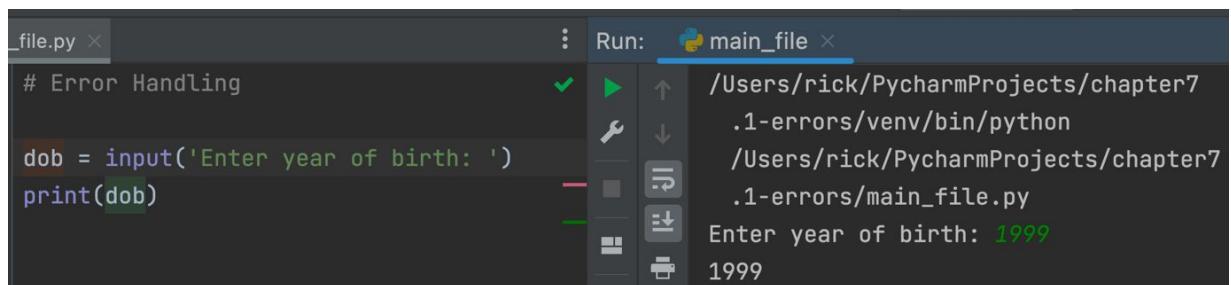
## Error Handling

This mechanism will allow us to continue executing the Python script even though there has been an error somewhere. To explain the error handling, I will create another file and prompt the user to enter their date of birth:

```
# Error Handling

dob = input('Enter year of birth')
print(dob)
```

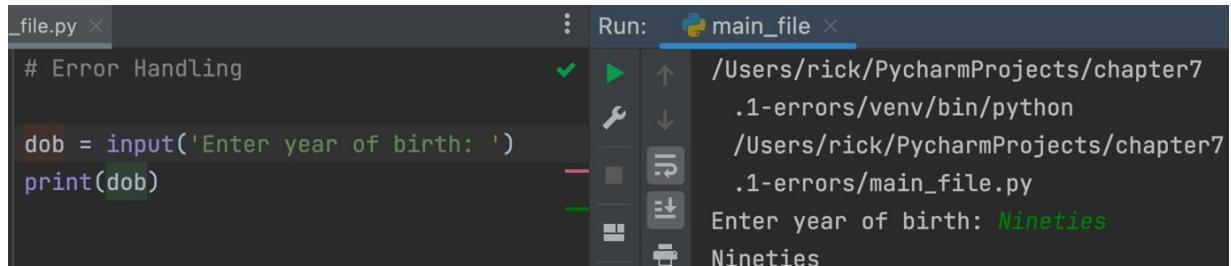
If we enter 1999 and press return or enter, we will get the output we wanted:



```
_file.py × Run: main_file ×
# Error Handling
dob = input('Enter year of birth: ')
print(dob)

Run: main_file ×
/Users/rick/PycharmProjects/chapter7
    .1-errors/venv/bin/python
/Users/rick/PycharmProjects/chapter7
    .1-errors/main_file.py
Enter year of birth: 1999
1999
```

As you can see, the program worked just fine, but let's run it again and this time enter a text/string instead of a number:



```
_file.py × Run: main_file ×
# Error Handling
dob = input('Enter year of birth: ')
print(dob)

Run: main_file ×
/Users/rick/PycharmProjects/chapter7
    .1-errors/venv/bin/python
/Users/rick/PycharmProjects/chapter7
    .1-errors/main_file.py
Enter year of birth: Nineties
Nineties
```

It worked perfectly for strings as well but is this the output we wanted? The answer is no because we wanted to get the year as a number and maybe later, we can do some further mathematical calculations, for example, calculating if you are eligible to get a driver's license. So, what is the solution? We can convert the input into integers and by doing this, we will always get integers as output not strings:

```
# Error Handling

dob = int(input('Enter year of birth: '))
print(dob)
```

If we run this code with an integer, there will be no problems but if the user inputs a string again, this is what we will get:

```
Enter your of birth: Nineties
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/chapter7
    .1-errors/main_file.py", line 3, in <module>
      dob = int(input('Enter your of birth: '))
ValueError: invalid literal for int() with base
  10: 'Nineties'
```

From the figure above, you can see that our code is now throwing a **ValueError** and this is not what we wanted. Just as in the I/O chapter, we can use the **try-except** block to handle errors. We put the code that needs to be executed in the **try** block but if there are any errors, the **except** block will handle them. Make sure you are using the correct indentation:

```
# Error Handling
try:
    dob = int(input('Enter year of birth: '))
    print(dob)

except:
    print('You have entered invalid value, please'
          ' enter a valid number!')
```

Here is what will happen if the user inputs a string instead of a number:

```
main_file ×
/Users/rick/PycharmProjects/chapter7
.1-errors/venv/bin/python
/Users/rick/PycharmProjects/chapter7
.1-errors/main_file.py
Enter your date of birth: Nineties
You have entered invalid value, please enter a
valid number!
```

As you can see from the figure above, we didn't get the error that will block and terminate the entire program, instead, we are notifying the user what he/she has done wrong and what he/she needs to do next. Now even if the user makes a mistake, the Python interpreter will continue its work and execute the rest of the statements. Let's add a new print function after the try-except block and confirm that the program will continue its normal execution:

```
_file.py ×
# Error Handling
try:
    dob = int(input('Enter year of birth: '))
    print(dob)

except:
    print('You have entered invalid value, please'
          ' enter a valid number!')

print('This is printed after try-except loop!')
```

```
Run: main_file ×
/Users/rick/PycharmProjects/chapter7
.1-errors/venv/bin/python
/Users/rick/PycharmProjects/chapter7
.1-errors/main_file.py
Enter year of birth: Nineties
You have entered invalid value, please enter a
valid number!
This is printed after try-except loop!
Process finished with exit code 0
```

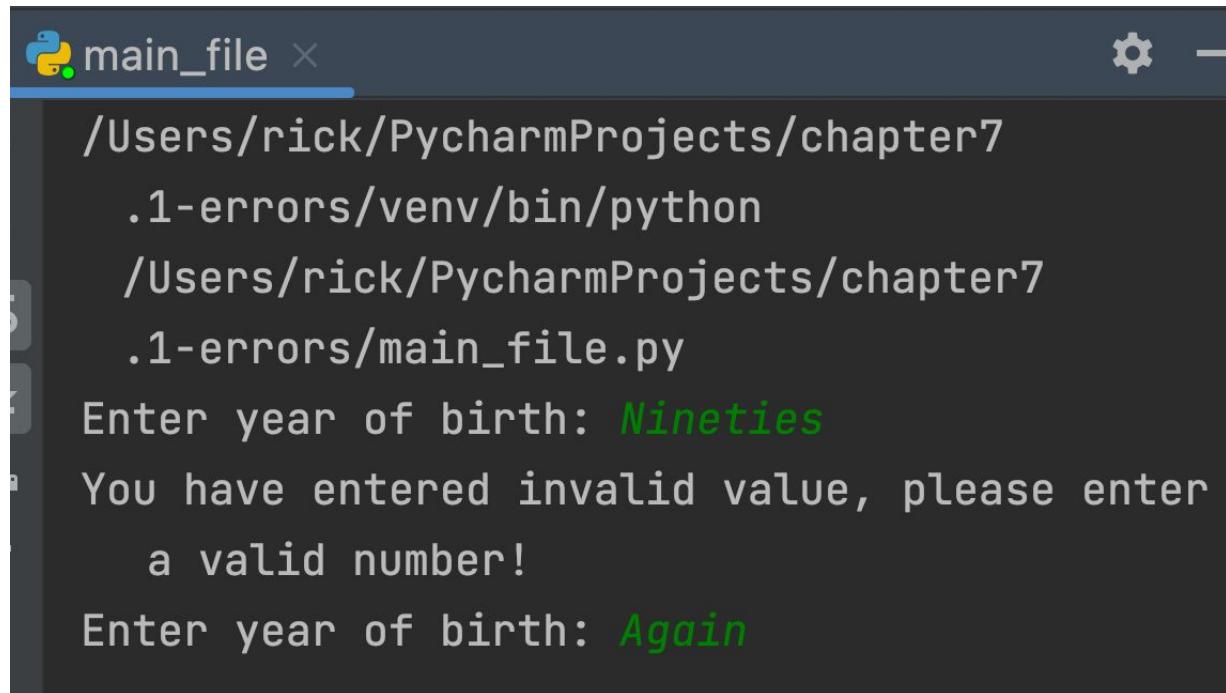
Great! No more red errors in our console but our code is far from perfect. For example, the user inputs a string, and this error will be caught and handled in the except block, but if the user wants to re-enter a new number to fix the previous mistake, he/she will need to run the file again. To fix this, you can simply use a while-loop and set the condition to True, and wrap the entire try-except block like this:

```
# Error Handling
while True:
    try:
        dob = int(input('Enter year of birth: '))
        print(dob)

    except:
        print('You have entered invalid value, please'
              ' enter a valid number!')

print('This is printed after while loop!')
```

If we run the code now and if user enters a string this will happen:



```
main_file ×
/Users/rick/PycharmProjects/chapter7
    .1-errors/venv/bin/python
/Users/rick/PycharmProjects/chapter7
    .1-errors/main_file.py
Enter year of birth: Nineties
You have entered invalid value, please enter
a valid number!
Enter year of birth: Again
```

Great! Our logic works! Well, it works but not perfectly because this will continue prompting the user to enter a value even if the user entered a number:

```
| Enter year of birth: Nineties
| You have entered invalid value, please enter
|   a valid number!
| Enter year of birth: Again
| You have entered invalid value, please enter
|   a valid number!
| Enter year of birth: 99
| 99
| Enter year of birth: 1221
| 1221
| Enter year of birth: |
```

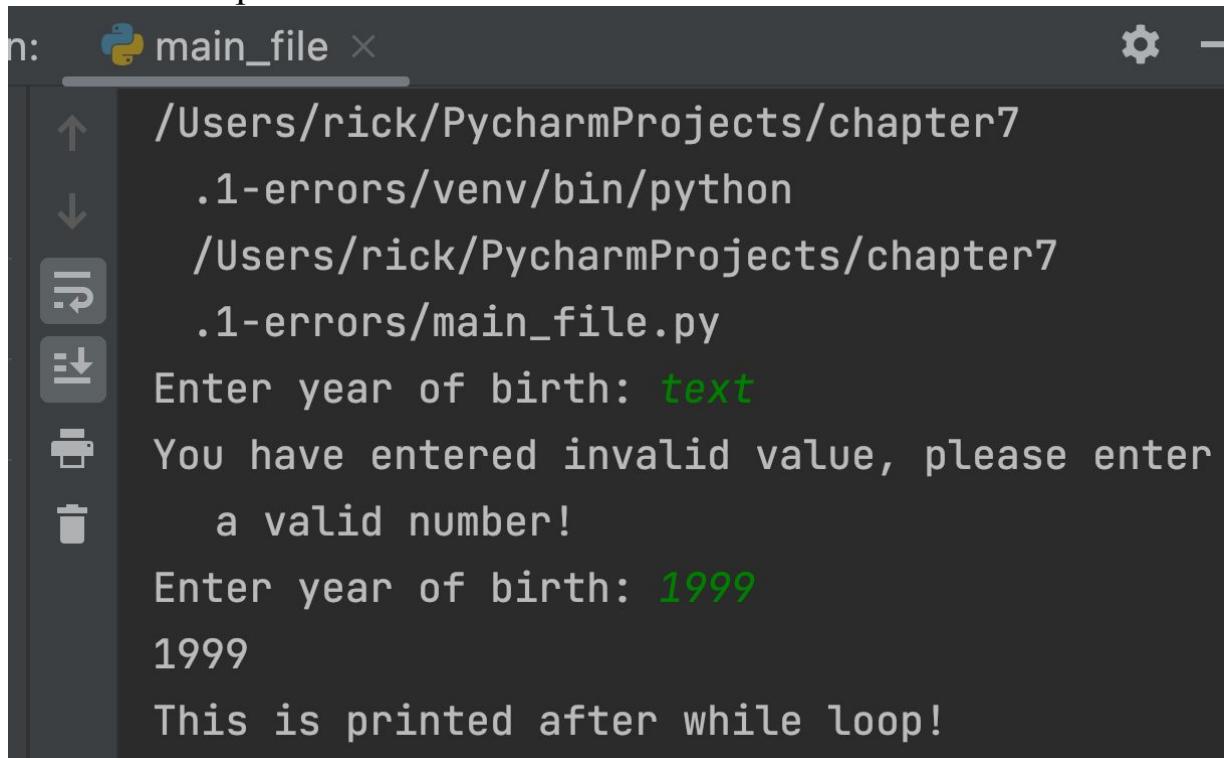
We can fix this by adding an else clause after the except block so we can break from this loop using the break statement:

```
# Error Handling
while True:
    try:
        dob = int(input('Enter year of birth: '))
        print(dob)

    except:
        print('You have entered invalid value, please'
              ' enter a valid number!')
    else:
        break

print('This is printed after while loop!')
```

This is the output now:



The screenshot shows a PyCharm terminal window titled "main\_file". The terminal displays the following text:  
/Users/rick/PycharmProjects/chapter7  
.1-errors/venv/bin/python  
/Users/rick/PycharmProjects/chapter7  
.1-errors/main\_file.py  
Enter year of birth: **text**  
You have entered invalid value, please enter  
a valid number!  
Enter year of birth: **1999**  
**1999**  
This is printed after while loop!

As you can see, after the user enters a number, the while loop will be over.

## Except block

As we saw in the I/O chapter, the except block can accept different built-in errors and we can print them. Python gives us the option to specify what types of errors we want the except block to handle. Imagine that after the user enters the year of birth, we want to find how old the user is by subtracting the current year from the year of birth. To do this, we need to use the **datetime** module that allows us to use dates and times, therefore I would like you to visit the Python documentation in your free time and go over the basic functions that this module offers:

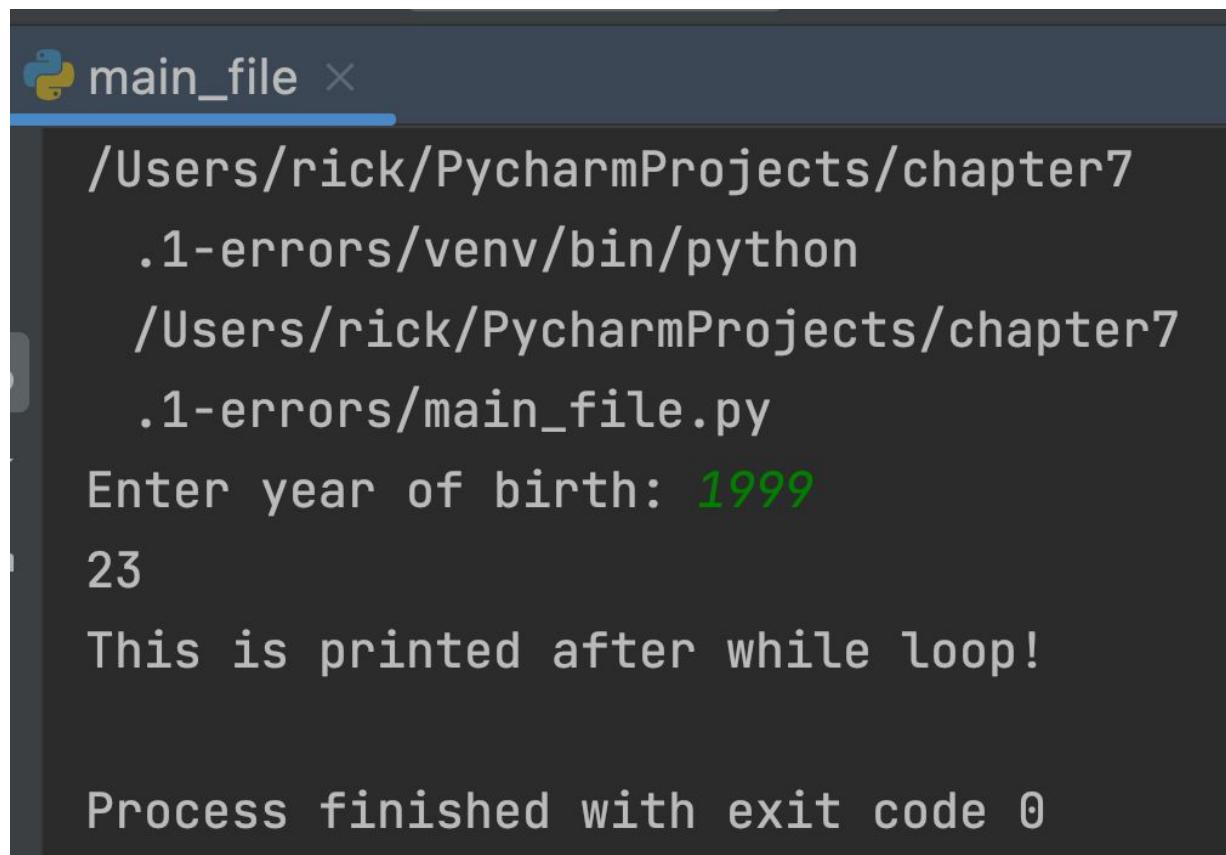
<https://docs.python.org/3/library/datetime.html>

Here is the code:

```
# Error Handling
from datetime import date
current_year = date.today().year
```

```
while True:  
    try:  
        dob = int(input('Enter year of birth: '))  
        print(current_year - dob)  
    except:  
        print('You have entered invalid value, please'  
              ' enter a valid number!')  
    else:  
        break  
  
print('This is printed after while loop!')
```

If we run the file and enter 1999, we would get this output (you might get a different value from 23 based on the year you are reading this book):



```
main_file ✘  
/Users/rick/PycharmProjects/chapter7  
.1-errors/venv/bin/python  
/Users/rick/PycharmProjects/chapter7  
.1-errors/main_file.py  
Enter year of birth: 1999  
23  
This is printed after while loop!  
Process finished with exit code 0
```

This works and now you can even change the except block to handle only ValueError errors like this:

```
except ValueError as err:  
    print('You have entered invalid value, please'  
          ' enter a valid number!')
```

We can do a lot of calculations in the try block and if any errors happened, they will be handled by the except block, but in our case, we only asked it to handle the ValueError errors. Let's find out if our application can handle ZeroDivisionError:

```
# Error Handling  
from datetime import date  
current_year = date.today().year  
  
while True:  
    try:  
        dob = int(input('Enter year of birth: '))  
        print(current_year - dob)  
        print(current_year / dob)  
    except ValueError as err:  
        print('You have entered invalid value, please'  
              ' enter a valid number!')  
    else:  
        break  
  
print('This is printed after while loop!')
```

I deliberately divided the current year with the year of birth because if the user now inserts zero (0) as a year, this should raise the ZeroDivisionError:

```
main_file ×
```

```
/Users/rick/PycharmProjects/chapter7
    .1-errors/venv/bin/python
    /Users/rick/PycharmProjects/chapter7
        .1-errors/main_file.py

Enter year of birth: 0
2022
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/chapter7
    .1-errors/main_file.py", line 9, in
      <module>
        print(current_year / dob)
ZeroDivisionError: division by zero
```

As you can see, the except block does not handle ZeroDivisionError as we thought it would, so what is the solution? Well, we can create another except block to handle ZeroDivisionError like this:

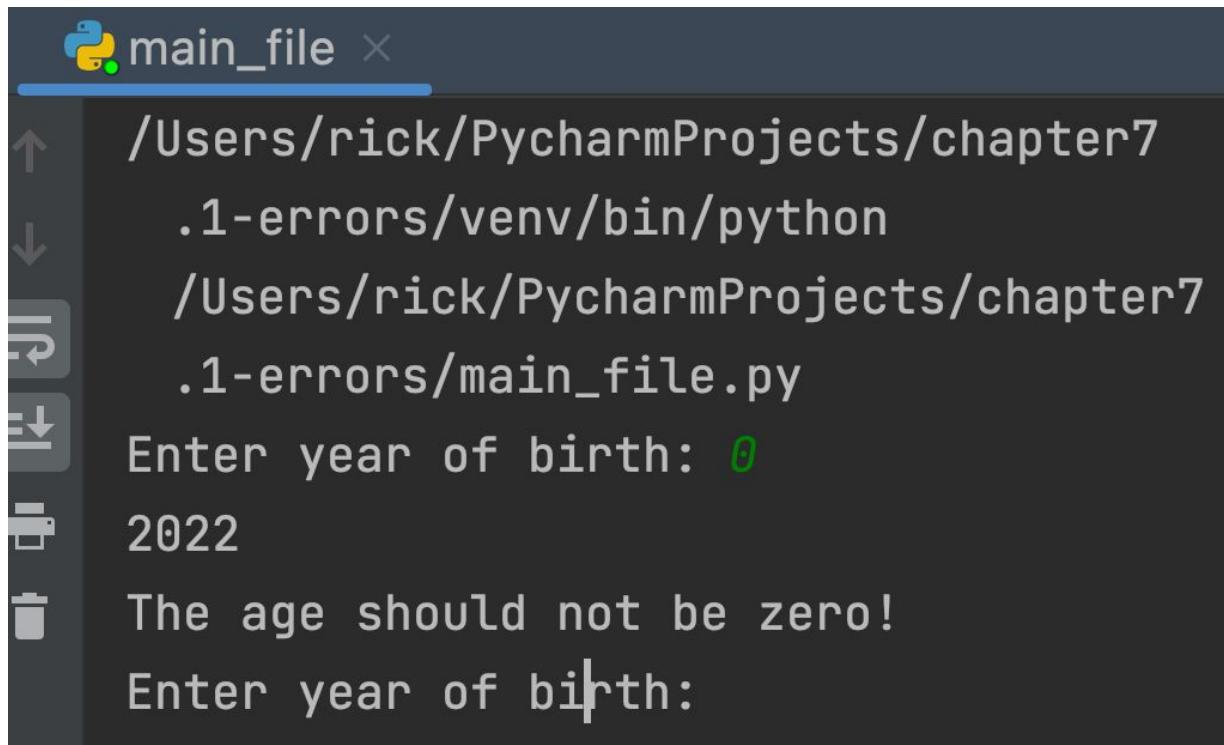
```
# Error Handling
from datetime import date
current_year = date.today().year

while True:
    try:
        dob = int(input('Enter year of birth: '))
        print(current_year - dob)
        print(current_year / dob)
    except ValueError as err:
        print('You have entered invalid value, please')
```

```
' enter a valid number!')
except ZeroDivisionError as err:
    print('The age should not be zero!')
else:
    break

print('This is printed after while loop!')
```

If we run the application now and a user enters a zero (0), this should be handled now by the second except block:



The screenshot shows a PyCharm interface with a terminal window. The terminal title is "main\_file". The working directory is "/Users/rick/PycharmProjects/chapter7.1-errors/venv/bin/python". The file being run is "main\_file.py". The terminal output shows the following interaction:

```
Enter year of birth: 0
2022
The age should not be zero!
Enter year of birth:
```

Let's create another function that will divide two numbers. This is the code for the division function:

```
# Error Handling

def div_numbers(a, b):

    return a / b
```

```
result = div_numbers(10, '2')
print(result)
```

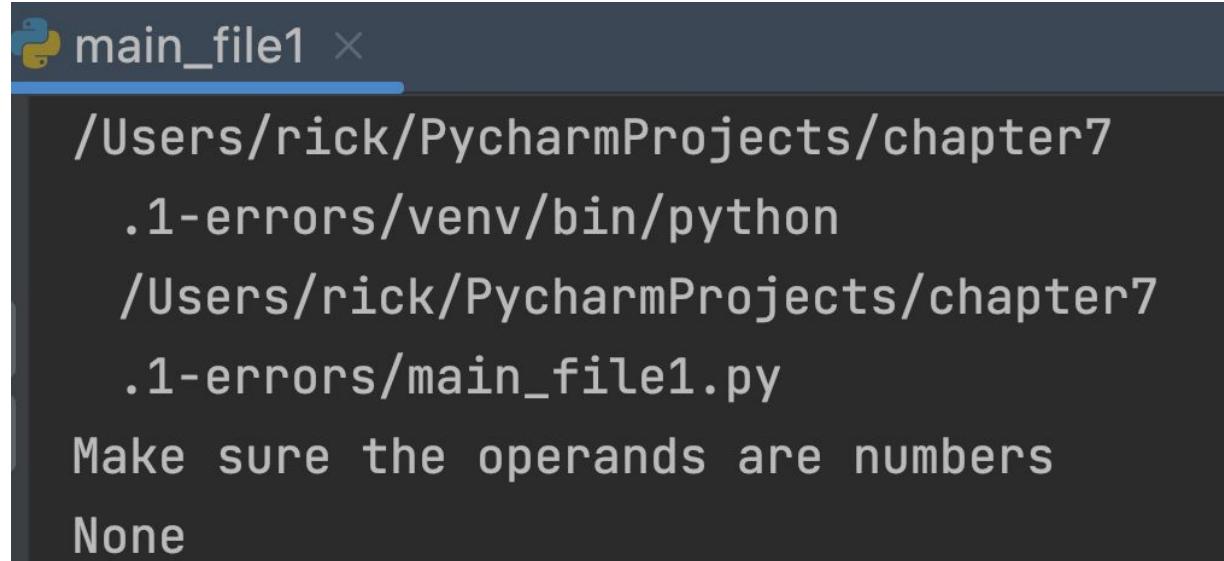
If we run the function as it is, we will get `TypeError` because the second operand we are passing into the `div_numbers` function is from type string. To handle this, we can simply use the try-except block directly in the function:

```
# Error Handling

def div_numbers(a, b):
    try:
        return a / b
    except TypeError:
        print('Make sure the operands are '
              'numbers')

result = div_numbers(10, '2')
print(result)
```

The output will be the `print` message that tells us that at least one of the operands we passed into the function is not number:



The screenshot shows a PyCharm interface with a terminal window titled "main\_file1". The terminal displays the following text:

```
/Users/rick/PycharmProjects/chapter7  
.1-errors/venv/bin/python  
/Users/rick/PycharmProjects/chapter7  
.1-errors/main_file1.py  
Make sure the operands are numbers  
None
```

In the I/O chapter, I mentioned that we can print the built-in Python error message if we assigned it to a variable and print that variable which will be the error object itself:

```
except TypeError as err:  
    print('Make sure the operands are '  
        'numbers' + err)
```

Now if we run the code with the same operands, you will see our custom print message plus the entire built-in error that comes from Python:

```
main_file1 ×

/Users/rick/PycharmProjects/chapter7.1-errors/venv/bin/python
/Users/rick/PycharmProjects/chapter7.1-errors/main_file1.py
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/chapter7.1-errors/main_file1.py",
    line 5, in div_numbers
      return a / b
TypeError: unsupported operand type(s) for /: 'int' and 'str'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/Users/rick/PycharmProjects/chapter7.1-errors/main_file1.py",
    line 11, in <module>
      result = div_numbers(10, '2')
  File "/Users/rick/PycharmProjects/chapter7.1-errors/main_file1.py",
    line 7, in div_numbers
      print('Make sure the operands are ')
TypeError: can only concatenate str (not "TypeError") to str
```

We as developers sometimes need this error message so we can truly know what is happening because if we have thousands of lines of code, this will be hard to find, but now from the error message, we can see that the problem is located on line 5. We can even use the ‘f’ string like this:

```
except TypeError as err:
    print(f'Make sure the operands are numbers {err}')
```

Let’s save the file and run it again:

```
main_file1 ✘
/Users/rick/PycharmProjects/chapter7
    .1-errors/venv/bin/python
    /Users/rick/PycharmProjects
    /chapter7.1-errors/main_file1.py
Make sure the operands are numbers
unsupported operand type(s) for /:
'int' and 'str'
```

Let's test what will happen if the second parameter in the function is zero:

```
result = div_numbers(10, 0)
print(result)
```

This will make the Interpreter throw a ZeroDivisionError:

```
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects
    /chapter7.1-errors/main_file1
    .py", line 11, in <module>
    result = div_numbers(10, 0)
  File "/Users/rick/PycharmProjects
    /chapter7.1-errors/main_file1
    .py", line 5, in div_numbers
    return a / b
ZeroDivisionError: division by zero
```

This means that we need to handle this error but instead of writing multiple **except** blocks, let's do this instead:

```
except (TypeError, ZeroDivisionError) as err:
    print(f"We got an error: {err}")
```

The output will be:

```
: main_file1 ×
  ↑
  ↓
  ⏪
  ⏴
  🖨
  ⏷
  None
```

/Users/rick/PycharmProjects/chapter7  
.1-errors/venv/bin/python  
/Users/rick/PycharmProjects  
/chapter7.1-errors/main\_file1.py  
We got an error: division by zero  
None

Here is the entire code now:

```
# Error Handling

def div_numbers(a, b):
    try:
        return a / b

    except (TypeError, ZeroDivisionError) as err:
        print(f"We got an error: {err}")

result = div_numbers(10, 0)
print(result)
```

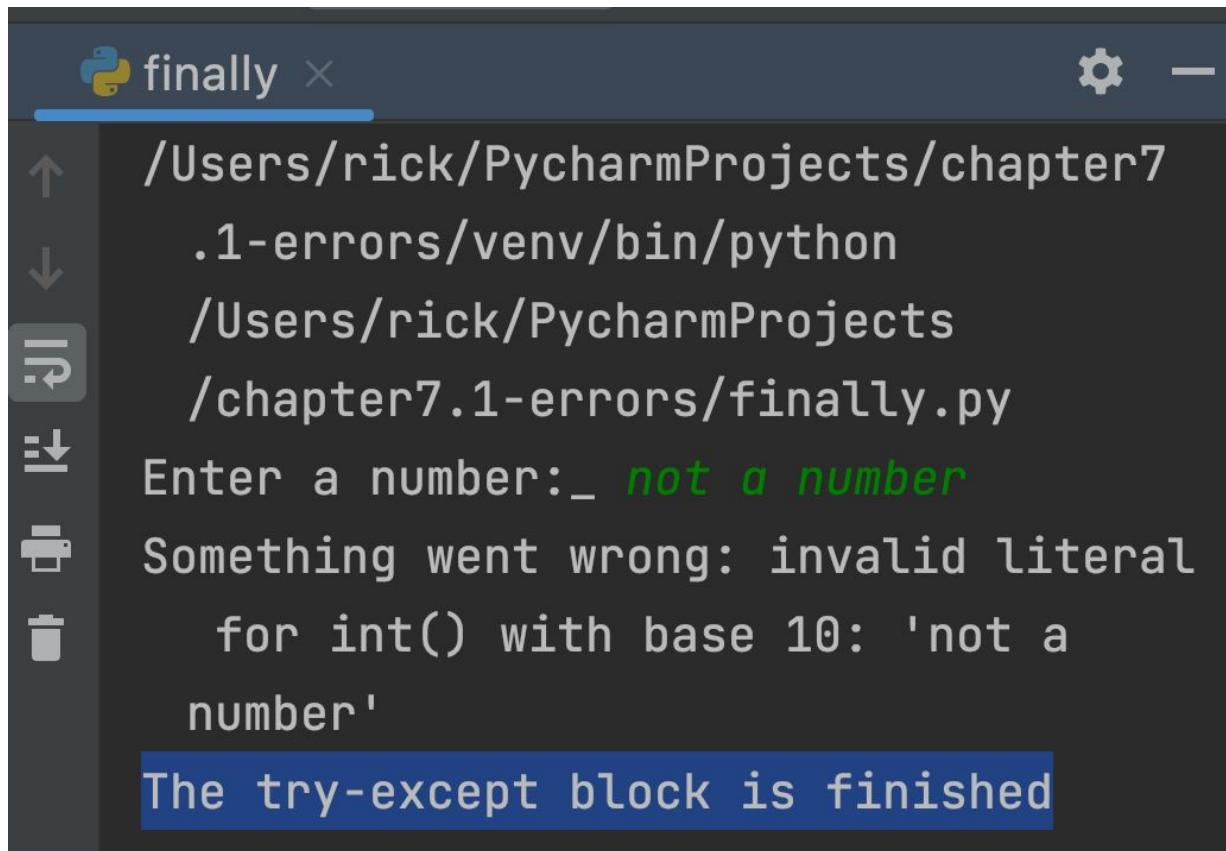
## Finally block

We have seen the try-except-else block but there is one more block we can use when we are dealing with exceptions. This block is called **finally** and it will run at the end after everything else has been completed. Here is one example of try-catch-else-finally block:

```
# Error Handling finally

try:
    number = int(input('Enter a number:_ '))
    number >= 3
except (TypeError, ValueError) as err:
    print(fSomething went wrong: {err}')
else:
    print(fEverything is ok and the result'
          f is {number * 3}')
finally:
    print("The try-except block is finished")
```

If we run this with a string instead of a number, then the except block should run and handle the errors:



The screenshot shows the PyCharm interface with a terminal window. The terminal title is 'finally'. The command entered is '/Users/rick/PycharmProjects/chapter7.1-errors/venv/bin/python /Users/rick/PycharmProjects/chapter7.1-errors/finally.py'. The terminal output shows the user being prompted to enter a number, which they type as 'not a number'. This triggers the exception handling code, resulting in the message 'Something went wrong: invalid literal for int() with base 10: 'not a number''. Finally, the message 'The try-except block is finished' is printed.

```
/Users/rick/PycharmProjects/chapter7.1-errors/venv/bin/python /Users/rick/PycharmProjects/chapter7.1-errors/finally.py
Enter a number:_ not a number
Something went wrong: invalid literal
for int() with base 10: 'not a
number'
The try-except block is finished
```

But even in a case of an error, we can see that the finally block is executed and the message it's printed. The finally keyword-block is used to define a block of code that will run when the try-except-else block is finished.

Whatever code we have in the finally block will run no matter what we have in the previous blocks.

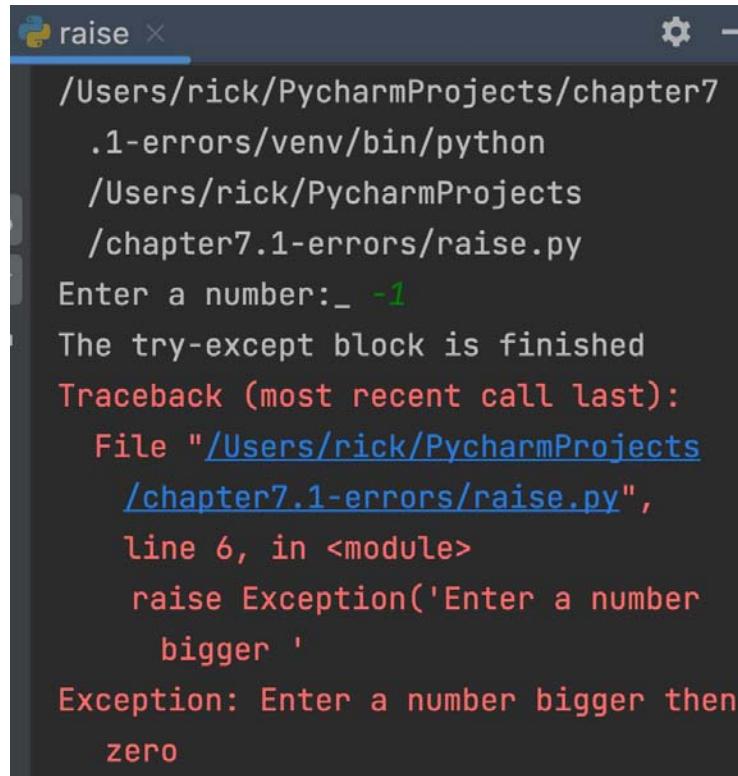
## Raise an Exception

Sometimes as a developer, you need to **throw** an exception if a specific condition occurs in your code and Python allows us to **raise** our own exception using the keyword **raise**. Please consider the following example:

```
# Error Handling finally

try:
    number = int(input('Enter a number: '))
    if number < 0:
        raise Exception('Enter a number bigger '
                        'then zero')
    number >= 3
except (TypeError, ValueError) as err:
    print(f'Something went wrong: {err}')
else:
    print(f'Everything is ok and the result'
          f' is {number * 3}')
finally:
    print("The try-except block is finished")
```

As you can see in the if-statement, we manually **raise** an exception with a message. If we run the same code with -1, this is what we will get:

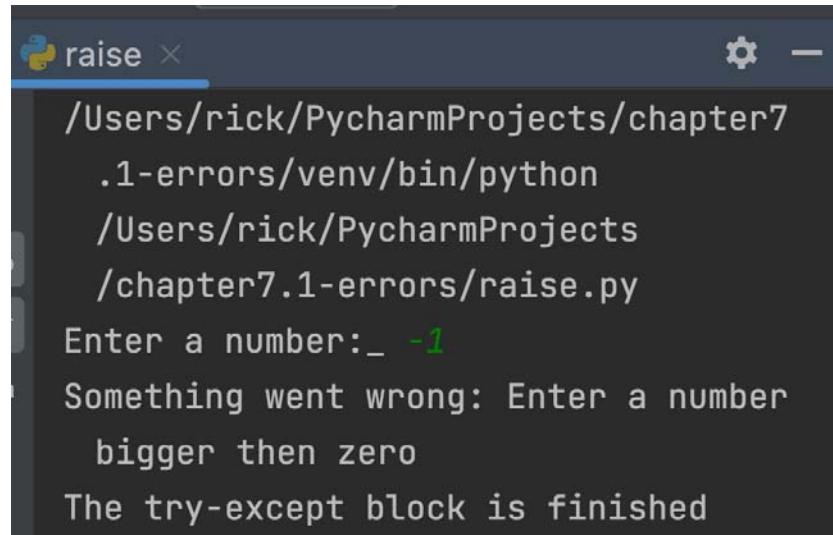


```
/Users/rick/PycharmProjects/chapter7
    .1-errors/venv/bin/python
    /Users/rick/PycharmProjects
    /chapter7.1-errors/raise.py
Enter a number:_ -1
The try-except block is finished
Traceback (most recent call last):
  File "/Users/rick/PycharmProjects
    /chapter7.1-errors/raise.py",
    line 6, in <module>
      raise Exception('Enter a number
                      bigger '
Exception: Enter a number bigger than
          zero
```

From the figure above, we can see that the **finally**-statement is executed at the end. If we don't want the error to be like this, then we can list this error in the **except-block** by adding **Exception**:

```
except (TypeError, ValueError, Exception) as err:
    print(f'Something went wrong: {err}')
```

And if we run the same code again with a value of -1, we will get the following output:



A screenshot of a PyCharm terminal window titled "raise". The terminal shows the following command-line interaction:

```
/Users/rick/PycharmProjects/chapter7  
.1-errors/venv/bin/python  
/Users/rick/PycharmProjects  
/chapter7.1-errors/raise.py  
Enter a number:_ -1  
Something went wrong: Enter a number  
bigger then zero  
The try-except block is finished
```

Great! Now we know how you can raise your own errors or **Exceptions**. We can raise any specific built-in Python errors like **TypeError** if we list them in the **except block**. This is everything I wanted to teach you in this chapter and you should know that errors are inevitable in any programming but a good programmer will know how to handle these errors without crashing the application.

## **Summary**

Congratulations! I'm always sad when I finish the final chapter. Error handling is a very important concept and most of the time, it's overlooked by programmers and that was why I decided to make it the last chapter. We have done a lot in this book, we started from scratch and most of you I assume knew nothing about Python but now that is no longer the case. From here your path is very simple because now you have enough knowledge to continue learning new and more advanced Python features. This was my first book about Python and I hope you liked it because very soon I will announce my second one which will be much more advanced than this one. In the introduction chapter, I mentioned that I need you to connect with me on any of the social media because very often I lower my books to below one dollar so everyone can get them basically for free. Before I say goodbye, I would like you to check the Projects section below and attempt to write some of the coding challenges on your own. We had a fun ride together and now it's time to go our separate ways.

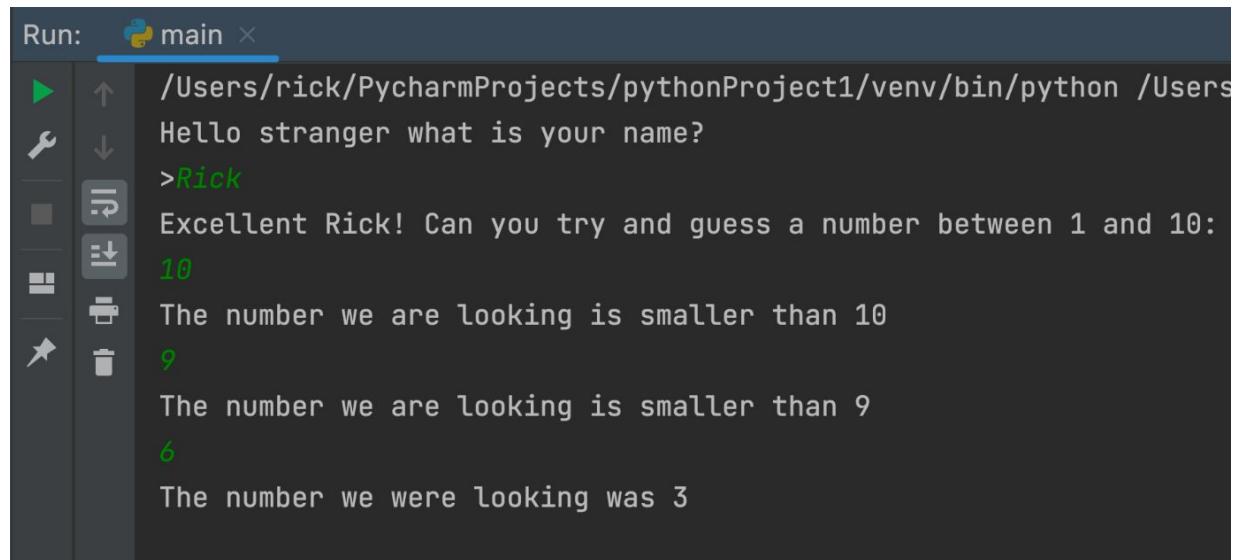
# Projects

In this section, we will have a few projects that will help you understand how Python works. You should try and write the coding challenges yourself but if you cannot, it's not a problem because the final code will be supplied at the end in the appendix section. Please read the projects instructions carefully and try to write the code:

## Project\_1: Guess the Random Number

This project is a mini-interactive game that you need to write in Python. The computer will create a random number between 1 and 10 and the user will be given an option to guess it in 3 attempts. In each of the attempts, the program must guide the user and provide minimal information if the random number is higher or lower than the computer random number.

This is the output when the user tries to guess the number in three attempts but fails:



The screenshot shows the PyCharm IDE's run interface. The 'Run' tab is selected, and the project name 'main' is shown. The terminal window displays the following interaction:

```
Run: main ×
▶ /Users/rick/PycharmProjects/pythonProject1/venv/bin/python /Users/rick/PycharmProjects/pythonProject1/main.py
Hello stranger what is your name?
>Rick
Excellent Rick! Can you try and guess a number between 1 and 10:
10
The number we are looking is smaller than 10
9
The number we are looking is smaller than 9
6
The number we were looking was 3
```

This is the output when the user guesses the number:

```
main ×  
/Users/rick/PycharmProjects/pythonProject1/venv/bin/python /Users/  
Hello stranger what is your name?  
>Rick  
Excellent Rick! Can you try and guess a number between 1 and 10:  
10  
The number we are looking is smaller than 10  
4  
The number we are looking is smaller than 4  
2  
You guessed the number in your 3-nd attempt!
```

## Coding hints

There are different ways you can write this program and there is no perfect solution. This means that everyone can come up with different working solutions. The following tips that I have included will contain bits and pieces of the final code so you can change any variables that I have created and name them as you want. I mentioned earlier that there is no single solution to this problem. The first tip is that you need to import a module that will help you to generate a random number. The module is called **random** and here is the link from Python documentation that will explain how to use this module (the most important part for you to read in the **Functions for integers** from the documentation):

<https://docs.python.org/3/library/random.html>

The second hint is to use the input function so the user can enter a number but be careful because the user inputs will not be in a number form, they are always Strings. This means that the user input needs to be converted from String to Integer. The third hint is to use a while loop that will count how many times the user has attempted to guess the random number. Before the while loop, you need some sort of counter variable like this:

```
number_of_guesses = 1
```

This variable will be the actual condition in the while loop:

```
while number_of_guesses < 3:
```

Inside the while loop, you need to increment the number\_of\_guesses variable by 1 so it will keep track of how many times the user attempted to guess the number. Inside you will have a couple of if statements, for example, one will be if the user entered a number that is bigger than the computer random number:

```
if player_number > the_number:
```

You will need another if statement that will check the opposite. Inside the if statements, you need to print the message saying what the user should do next and again use the input function to initiate the user to enter a new number. After each fail, we need to give the user a chance to enter the number again, therefore the input function is a must. The final third if-statement should be if in case the user entered a number that is the same as the computer random number. In this case, you should use one of the following statements (please think about which statement you should use when the user guessed the number):

- Break
- Pass
- continue

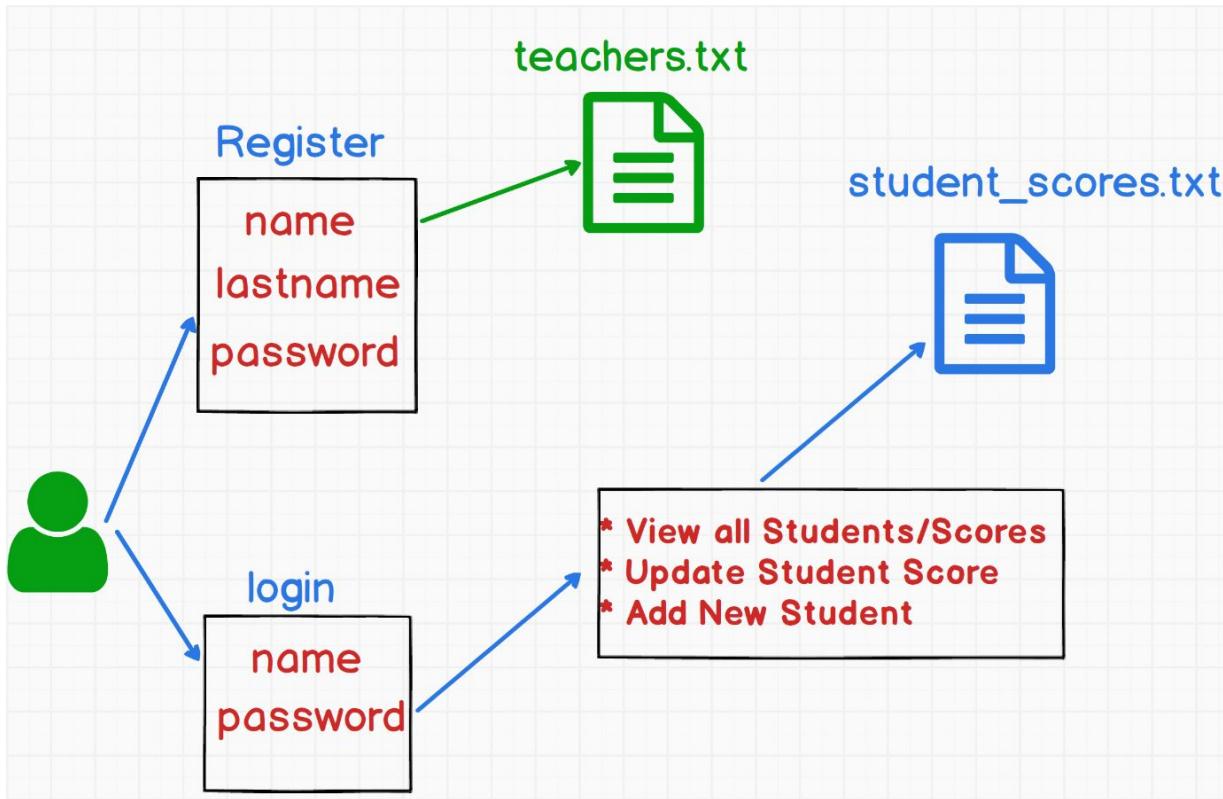
After the while loop, you can have another if-else statement where you mentioned how many times the user guessed the number or if the user didn't guess the number, then you need to print what the random number was. If you can't solve it on your own, the answer will be in Appendix A, Project\_1: Guess the Random Number Solution.

## **Project\_2: Teacher, Student App**

This project will be a bit harder compared to the first one. The project will be about coding a simple program that will allow a teacher to be created. Once the teacher is created, he/she can login into the system and view all students and their scores. Also, the teacher can add a new student and update students' scores. The student data will come from a file called `student_scores.txt`. Before we dive into this project, these are some of the things this project will include:

- I/O Files
- Hashing Passwords
- Teacher Login
- Teacher Registration
- Adding new Students
- Updating Students
- Removing files
- Updating files
- Using the datetime module

This is what the application we are going to write will look like:



As shown in the figure above, we have a teacher that will need to register first. After the registration, the details will be stored in the teachers.txt file. The teacher can then login with their name and password and will be able to:

- View all students and their scores
- Update student score
- Add new student

The student details are located in the student\_scores.txt file. Each of the teacher's actions for the students will be recorded in the student\_scores.txt. I will guide you step by step on how to write this app but if you feel confident you can do it on your own, then start coding. Before you attempt to write your code, these are some features that we haven't talked about in this book and you will need them to finish this app:

- Remove and rename files
- Password hashing using bcrypt package from pypi.org

- Datetime module to create current date and time

## Let's start with removing and renaming files

To remove and rename files, you can use the module called OS. This module provides a few functions that you can use to interact with the file system. You are going to use two methods only, the remove and rename methods. The os.remove() method is used to remove or delete a file path. Syntax:

```
remove(file.txt)
```

The second method is called rename and it's used to rename the existing file:

```
rename('old_file_name.txt', new_file_name.txt)
```

This means we will rename the old\_file\_name.txt to new\_file\_name.txt

## Password hashing

This application will not be a real register and login system but I wanted to give you an app where you install some packages and this was a perfect opportunity. You will use the pypi.org website where we can find and install a package called bcrypt:



<https://pypi.org/project/bcrypt/>

This is good for password hashing and for example, the teacher will try to register with the password ‘MyPassword’ as a string. This package will take this string and using the hashing function will turn it into something:

```
b'$2b$12$fwBh8IUb4pC917unYjStSuwmJXyOlWH2jCXKOazcXikZS7J1wzF9q'
```

We can use another function from bcrypt that will compare the original password with the hashed password so it can let the teacher login to the system. This application is very simple but when working on larger applications, the security and password hashing is very, very important. To install this package, you will need to use the:

```
pip install bcrypt
```

or

```
pip3 install bcrypt
```

If you are using the PyCharm IDE, you can create a new project and go the PyCharm> Preferences then find the project name, select the interpreter, and install the package:



## **Datetime module**

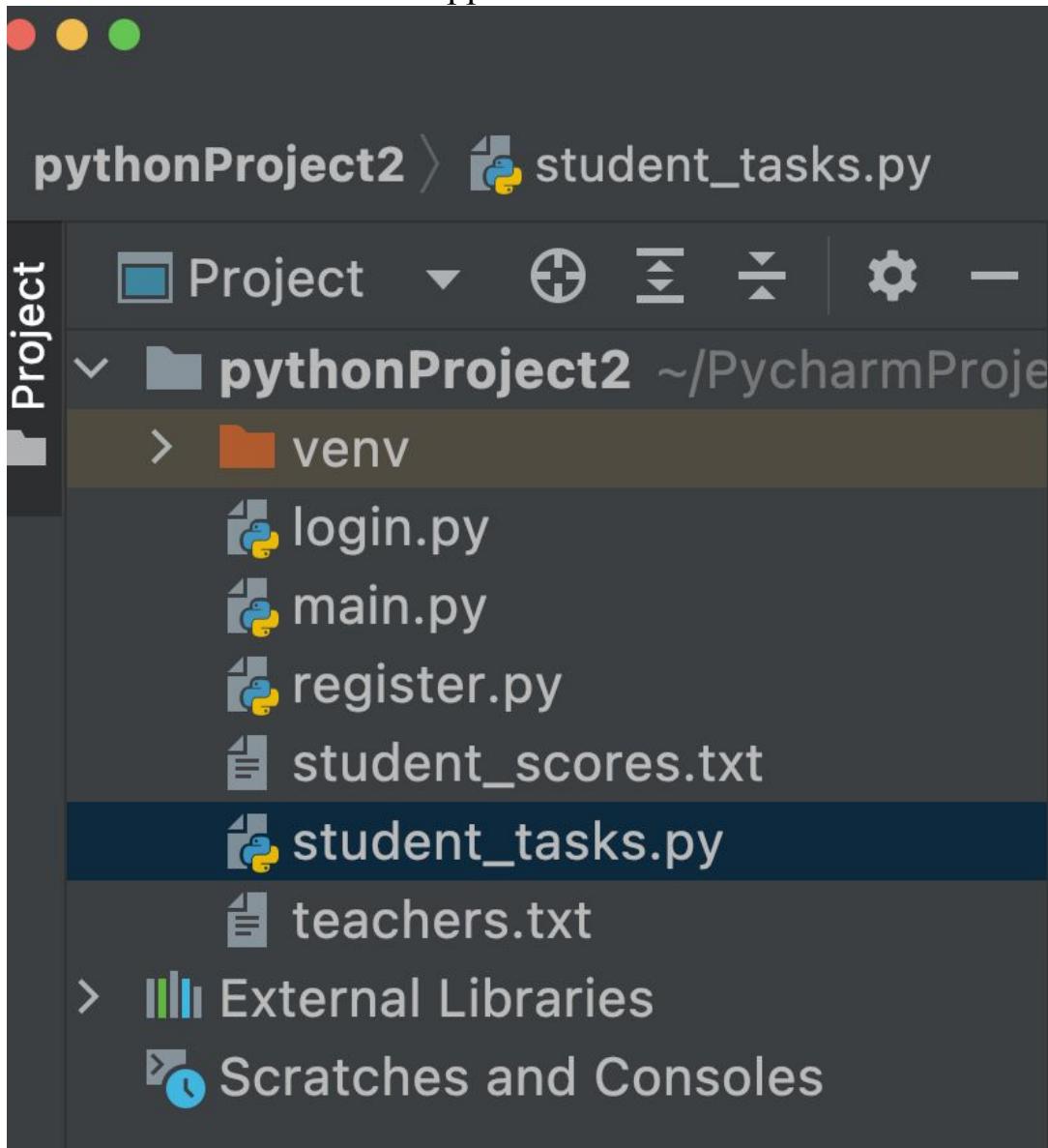
You can read about the date time here:

<https://docs.python.org/3/library/datetime.html>

We need datetime module because every time the teacher manipulates the student file, it should record the time or you can create some sort of timestamp. Please read the next section where I will provide a few screenshots from the app I have built for this project.

## Screenshots and files

To create this app, I would like to include a few screenshots so you can get an idea of how to write this app. In this project, I have created four Python files and 2 text files. The Python files are login, main, register, and student\_tasks. The text files are student\_scores and teachers. I have used the PyCharm to create and run this app:



Here is what I get when we run the main file:

```
main ×  
/Users/rick/PycharmProjects/pythonProject2/  
Welcome to Teacher Panel  
What do you want to do (select a number):  
1) Login  
2) Register  
3) Exit  
Select a number  
>
```

The teacher can choose to Login, Register, and Exit by selecting one of the numbers on the left side, for example, if the teacher needs to Login, it should press 1. Let's start with the simplest one the number 3 that will make the app terminate:

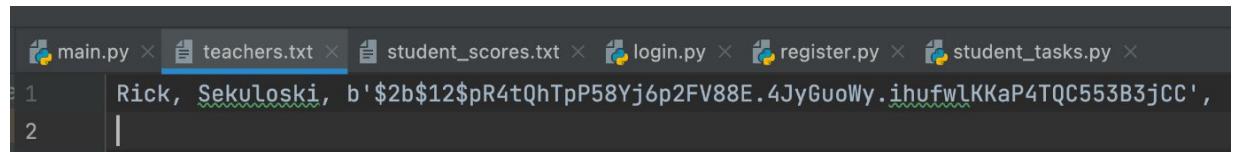
```
main ×  
/Users/rick/PycharmProjects/pythonProject2/  
Welcome to Teacher Panel  
What do you want to do (select a number):  
1) Login  
2) Register  
3) Exit  
Select a number  
>  
3  
  
Process finished with exit code 0
```

As you can see, the app is terminated. Let's now run the app again and Register the new teacher by selecting the number 2:

```
/Users/rick/PycharmProjects/pythonProject2/venv/bin/
Welcome to Teacher Panel
What do you want to do (select a number):
1) Login
2) Register
3) Exit
Select a number
>
2
***** Teacher Registration *****

Your first name
Rick
Your last name
Sekuloski
Your login password
password|
```

If we press enter/return the same menu will appear again but if we open the teachers.txt file, this is what we should have:

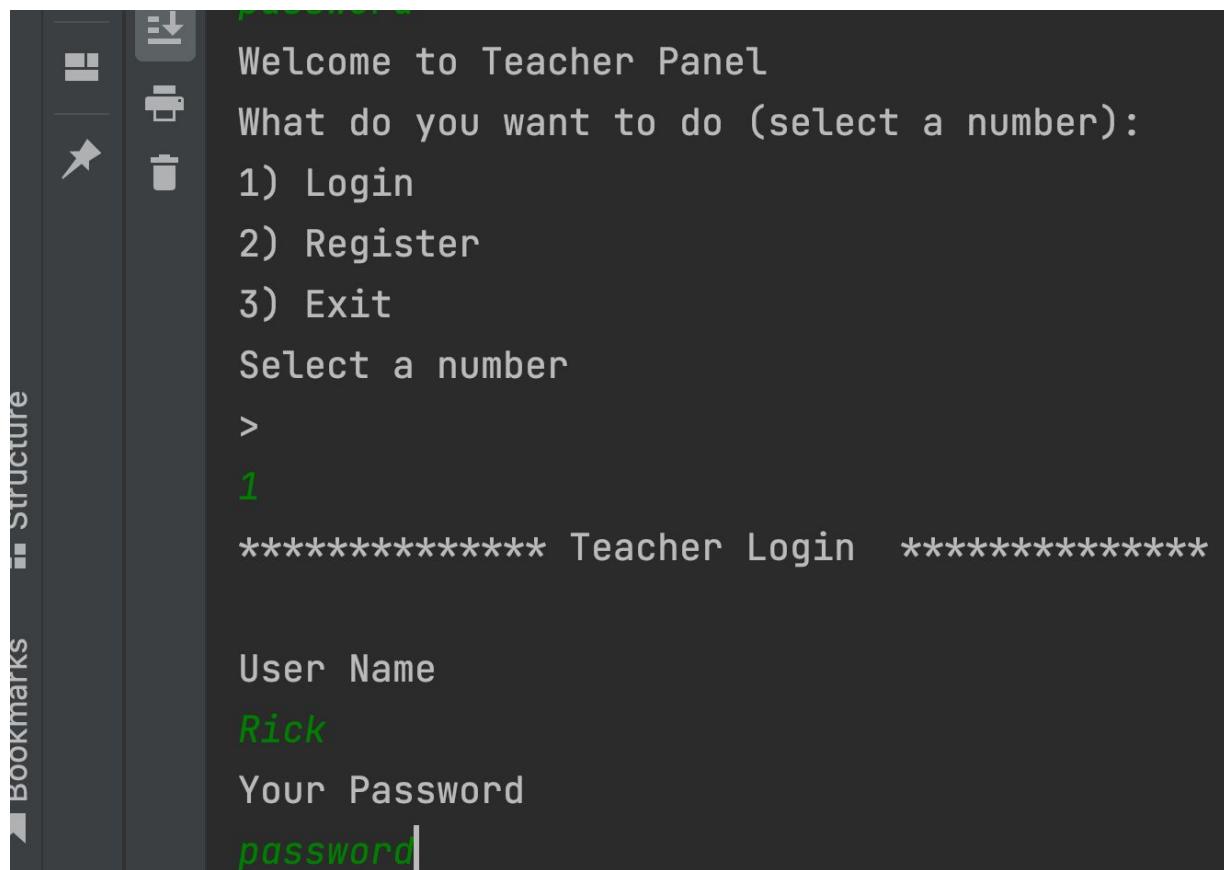


The screenshot shows a terminal window with several tabs at the top: main.py, teachers.txt, student\_scores.txt, login.py, register.py, and student\_tasks.py. The teachers.txt tab is active, displaying the following content:

```
Rick, Sekuloski, b'$2b$12$pR4tQhTpP58Yj6p2FV88E.4JyGuoWy.ihufwlKKaP4TQC553B3jCC',
```

The new teacher is created with first name, last name, and the hashed password. They are all comma-separated values.

Now since the teacher is created, we can try to login:



The screenshot shows a terminal window with a dark theme. On the left, there are icons for file operations: download, print, and delete. Below these are 'Bookmarks' and 'Structure' buttons. The main area displays the following text:

```
Welcome to Teacher Panel
What do you want to do (select a number):
1) Login
2) Register
3) Exit
Select a number
>
1
***** Teacher Login *****

User Name
Rick
Your Password
password|
```

The text is white on a black background, with the selected option '1) Login' highlighted in green.

If we press return or enter, it should get a new menu where we can add a new student, update student scores, and get student scores:

```
***** Teacher Login *****

User Name
Rick

Your Password
password

Welcome back teacher Rick
What you want to do (select a number):
1) Get Student Scores
2) Update Student Score
3) Enroll New Student
4) Exit
>
```

Because the student's file is empty let's select option number 3 and create a new student:

```
Welcome back teacher Rick
What you want to do (select a number):
1) Get Student Scores
2) Update Student Score
3) Enroll New Student
4) Exit
>
3
Enroll new student
Student name
Jason
Student score
134
```

After we press enter or return, let's check the student\_scores.txt file:

```
main.py × student_scores.txt × login.py × register.py × student_tasks.py ×
1
2 Jason, 134, Added by Teacher Rick at:, 08/19/2022, 19:28:58
```

As you can see, we have the student's name, score, the teacher's name, and the timestamp. After this, you can use option 2 to update the student score from 134 to 199:

```
Welcome back Teacher Rick  
What you want to do (select a number):  
1) Get Student Scores  
2) Update Student Score  
3) Enroll New Student  
4) Exit  
>  
2  
Input student details  
Student name  
Jason  
New student score  
199
```

If we press enter/return, we don't need to go check the student\_scores file because we can use option 1 from the menu and it will list everything inside that file including the changes we have made:

```
What you want to do (select a number):  
1) Get Student Scores  
2) Update Student Score  
3) Enroll New Student  
4) Exit  
>  
1  
  
Jason, 199, Updated by Teacher:Rick, 08/19/2022, 20:00:19
```

As we can see, the Jason score is being updated from 134 to 199 and just to confirm the student\_scores.txt file is also updated:

```
1
2 Jason, 199, Updated by Teacher:Rick, 08/19/2022, 20:00:19
3
```

That is everything this application does. Next, I will provide from each of the files the starter code so you can just code the main functions:

## Starter code with comments on what code you should add

Main.py file:

```
# import the teacher_login function from login

# import teacher_register from register

# import get_student_scores, update_student_score, add_new_student from student_tasks

# This variable will make main menu to appear
# as long as the while loop is evaluated to true
# the while loop will stop when user selects number 3
teacher_selection = 0

while teacher_selection != 3:
    print(f'Welcome to Teacher Panel \n'
          f'What do you want to do (select a number): \n'
          f'1) Login\n'
          f'2) Register\n'
          f'3) Exit\n'
          f'Select a number \n'
          f'>')
    teacher_selection = int(input())
    if teacher_selection == 2:
        print('***** Teacher Registration ***** \n')
        print('Your first name')
```

```

teacher_name = input(")
print('Your last name')
teacher_lastname = input(")
print('Your login password')
teacher_password = input(")
# call the teacher_register() function with 3 arguments
elif teacher_selection == 1:
    print('***** Teacher Login ***** \n')
    print('User Name')
    teacher_username = input(")
    print('Your Password')
    teacher_password = input(")
    # call the teacher_login function with 2 arguments
    # store the result of teacher_login in the variable bellow login_result
    login_result = 0

while login_result != -1:
    print(f'Welcome back teacher {teacher_username} \n'
          f'What you want to do (select a number): \n'
          f'1) Get Student Scores \n'
          f'2) Update Student Score \n'
          f'3) Enroll New Student \n'
          f'4) Exit \n'
          f'>')
    teacher_login_input = int(input("))
    if teacher_login_input == 4:
        break
    if teacher_login_input == 1:
        pass
        # delete the pass statement above
        # use the print function to print the get_student_scores() data

    elif teacher_login_input == 2:
        print('Input student details')
        print('Student name')
        student_name = input(")
        print('New student score')

```

```

student_score = int(input())
# Call the update_student_score with 3 arguments
# As arguments use the variables declared above
# from the user input.
# Don't forget the teacher_username should be the last
# variable

elif teacher_login_input == 3:
    print('Enroll new student')
    print('Student name')
    student_name = input()
    print('Student score')
    student_score = int(input())
    # call add_new_student function with 3 arguments
    # remember the last argument is the teacher_username

else:
    print('You have entered a wrong choice')

```

## Login.py file:

```

# import bcrypt, after you have installed the bcrypt
# package and the import bcrypt will not cause errors
import bcrypt

# write login function that have 2 parameters :
# the teacher name and password
def teacher_login(user_name, password):
    # Encode method returns an encoded version of the given string.
    # If you read the bcrypt documentation you need the string
    # to be encoded in order to be hashed or verified
    new_password = password.encode('utf-8')
    try:
        pass
        # Delete the above pass statement.
        # In the try-except block you need to try

```

```
# to open the teachers.txt file using the mode=r  
# example:  
  
# with open('teachers.txt', mode='r') as teachers:  
  
# Use the for-in loop to loop through all of the  
# the lines in the teachers.txt file.  
# At the moment there is only one, but  
# if there are more teachers registered,  
# the teachers.txt file will contain multiple  
# lines.  
# Important you can use split() method that will  
# split the string into a list.  
# The syntax is split(separator, maxsplit)  
# This is the hardest function and here is a  
# helper code:  
  
# for teacher in teachers:  
#     result = teacher.split(',')  
  
# After this you need to find if the teacher  
# with the user_name passed as parameter exists  
# in one of the lines. You can use the  
# keyword 'in' to check if the user_name  
# can be found in one of the lines.  
# If the user is found then the password that  
# is passed and encoded matches  
# the hashed password stored in the file.  
# For this you will need to use the  
# bcrytp.checkpw method that will compare  
# the passed password with the hashed password from  
# teachers.txt file.  
# If there is no match of the user_name  
# this means that the teacher is not registered  
# so print the following message:  
# print(f'Sorry you are not registered as a teacher \n'  
#     f' please contact the administration for access')
```

```
# After the print message use the return statement to  
# return -1
```

```
except FileNotFoundError as err:  
    print('File cannot be found!')
```

## Register.py

```
import bcrypt  
  
# register new teacher using user_name, user_lastname and password  
def teacher_register(user_name, user_lastname, password):  
    # try - except block  
    try:  
        # here you will need to call the hash_password()  
        # function that will return  
        # a password that is hashed using bcrypt.  
        # You don't need to know about the hash_password()  
        # function at the moment just follow the instructions:  
        # 1-st) Pass the password parameter in the hash_password(password).  
        # 2-nd) Store whatever is returned from the above function  
        # in a variable called password_hashed  
  
        with open('teachers.txt', mode='a') as new_text_file:  
            pass  
            # Delete pass.  
            # Use the new_text_file to write to the teachers.txt file  
            # Remember you need to follow the following format:  
            # user_name, last_name, password_hashed, \n  
            # when you are done writing to teachers.txt file return a formatted  
            # string like this:  
            # f'{user_name} , Your registration is successful, go back and log in!'  
            # Make sure the return is in the same level as the with-open block.  
    except FileNotFoundError as err:  
        print('File cannot be found!')
```

```
# hashing function to hash the above password
def hash_password(user_password):
    # encode the password
    bytePwd = user_password.encode('utf-8')
    # Generate salt
    mySalt = bcrypt.gensalt()
    # Hash password
    user_password_hashed = bcrypt.hashpw(bytePwd, mySalt)
    return user_password_hashed
```

## student\_tasks.py

```
from os import remove, rename
from datetime import datetime

# This is function to get the student scores
def get_student_scores():
    try:
        with open('student_scores.txt', mode='r') as students:
            pass
        # you can use the simple read() function to return
        # all the students like we did in the book.

    except FileNotFoundError as err:
        print('File cannot be found!')

# This is a function to update the user score based on 3 parameters
def update_student_score(student_name, student_score, teacher_username):
    try:
        # To create the timestamp I have used:
        # now = datetime.now()
        # and date_time = now.strftime("%m/%d/%Y, %H:%M:%S")
        # If you like to create something different you are welcome
        # to try.
        # Instead of nested with open I have used the old open method
        # but we need to close it at the end
        # tmp stands for the temporary file with mode write
```

```
tmp = open('student_scores.tmp', mode='w')
# This will open the original file with read mode
original_file = open('student_scores.txt', mode='r')
# The idea is to update the student_score in a new file
# and then to rename the temporary file with the changes
# as the original file
for line in original_file:
    content = line.split(',')
    if student_name in content:
        pass
        # Delete the pass statement.
        # If the student_name is found in the content then
        # you need to write whatever you have from original file
        # into the temporary file using the tmp.write()
        # f{student_name}, {student_score}, Updated by Teacher:{teacher_username},
        # {date_time}\n'
    else:
        pass
        # delete the pass
        # You can write the entire line in the tmp file
    # You need to close the tmp and original_file
    # Next is to remove the student_scores.txt using the remove function
    # Next is to rename the student_scores.tmp to student_scores.txt

except FileNotFoundError as err:
    print('File cannot be found!')


def add_new_student(student_name, student_score, teacher_username):
    try:
        pass
        # Delete the pass statement above
        # Use the now and date_time from the function above
```

```
with open('student_scores.txt', mode='a') as new_text_file:  
    pass  
    # Delete the pass above  
    # write to file using the write function the following string:  
    #f\n{student_name}, {student_score}, Added by Teacher {teacher_username} at:,  
    {date_time}'  
    # print message f'Student with the {student_name} successfully added in the system '  
  
except FileNotFoundError as err:  
    print('File cannot be found!')
```

The solution will be in Appendix A Project\_2: Teacher, Student App Solution. All of the project files are in the folder pythonProject2\_starter.

# Appendix A – Project Solutions

## Project\_1: Guess the Random Number Solution:

You can run the following code and try out the random number game:

```
import random

the_number = random.randint(1, 10)

player = input('Hello stranger what is your name? \n>')
print(f'Excellent {player}! Can you try and guess a number between 1 and 10:')
player_number = int(input(""))

number_of_guesses = 1
while number_of_guesses < 3:
    number_of_guesses += 1
    if player_number < the_number:
        print(f'The number we are looking for is greater than {player_number}')
        player_number = int(input(""))
    if player_number > the_number:
        print(f'The number we are looking for is smaller than {player_number}')
        player_number = int(input(""))
    if player_number == the_number:
        break

if player_number == the_number:
    if number_of_guesses == 1:
        print(f'You guessed the number in your {number_of_guesses}-st attempt!')
    elif number_of_guesses == 2:
        print(f'You guessed the number in your {number_of_guesses}-nd attempt')
    else:
        print(f'You guessed the number in your {number_of_guesses}-rd attempt!')
```

```
else:  
    print(f'The number we were looking was {the_number}')
```

## Project\_2: Teacher, Student App Solution

### Main.py

```
from login import teacher_login  
from register import teacher_register  
from student_tasks import get_student_scores, update_student_score, add_new_student  
  
teacher_selection = 0  
  
while teacher_selection != 3:  
    print(f'Welcome to Teacher Panel \n'  
        f'What do you want to do (select a number): \n'  
        f'1) Login\n'  
        f'2) Register\n'  
        f'3) Exit\n'  
        f'Select a number \n'  
        f'>')  
    teacher_selection = int(input(''))  
    if teacher_selection == 2:  
        print('***** Teacher Registration ***** \n')  
        print('Your first name')  
        teacher_name = input()  
        print('Your last name')  
        teacher_lastname = input()  
        print('Your login password')  
        teacher_password = input()  
        teacher_register(teacher_name, teacher_lastname, teacher_password)  
    elif teacher_selection == 1:  
        print('***** Teacher Login ***** \n')  
        print('User Name')
```

```
teacher_username = input()
print('Your Password')
teacher_password = input()
login_result = teacher_login(teacher_username, teacher_password)

while login_result != -1:
    print(f'Welcome back teacher {teacher_username} \n'
          f'What you want to do (select a number): \n'
          f'1) Get Student Scores \n'
          f'2) Update Student Score \n'
          f'3) Enroll New Student \n'
          f'4) Exit \n'
          f'>')
    teacher_login_input = int(input())
    if teacher_login_input == 4:
        break
    if teacher_login_input == 1:
        print(get_student_scores())
    elif teacher_login_input == 2:
        print('Input student details')
        print('Student name')
        student_name = input()
        print('New student score')
        student_score = int(input())
        update_student_score(student_name, student_score, teacher_username)

    elif teacher_login_input == 3:
        print('Enroll new student')
        print('Student name')
        student_name = input()
        print('Student score')
        student_score = int(input())
        add_new_student(student_name, student_score, teacher_username)
    else:
        print('You have entered a wrong choice')
```

## login.py

```
import bcrypt

def teacher_login(user_name, password):
    new_password = password.encode('utf-8')
    try:
        with open('teachers.txt', mode='r') as teachers:
            for teacher in teachers:
                result = teacher.split(',')
                if user_name in result:
                    striped_string = result[2][2:-1]
                    if bcrypt.checkpw(new_password, striped_string.encode('utf-8')):
                        return True
    print(f'Sorry you are not registered as a teacher \n'
          f'please contact the administration for access')
    return -1

except FileNotFoundError as err:
    print('File cannot be found!')
```

## student\_tasks.py

```
from os import remove, rename
from datetime import datetime

def get_student_scores():
    try:
        with open('student_scores.txt', mode='r') as students:
            return students.read()
    except FileNotFoundError as err:
        print('File cannot be found!')
```

```

def update_student_score(student_name, student_score, teacher_username):
    try:
        now = datetime.now()
        date_time = now.strftime("%m/%d/%Y, %H:%M:%S")
        tmp = open('student_scores.tmp', mode='w')
        original_file = open('student_scores.txt', mode='r')
        for line in original_file:
            content = line.split(',')
            if student_name in content:
                tmp.write(f'{student_name}, {student_score}, Updated by Teacher: {teacher_username}, {date_time}\n')
            else:
                tmp.write(line)
        tmp.close()
        original_file.close()
        remove('student_scores.txt')
        rename('student_scores.tmp', 'student_scores.txt')

    except FileNotFoundError as err:
        print('File cannot be found!')


def add_new_student(student_name, student_score, teacher_username):
    try:
        now = datetime.now()
        date_time = now.strftime("%m/%d/%Y, %H:%M:%S")
        with open('student_scores.txt', mode='a') as new_text_file:
            new_text_file.write(
                f'{student_name}, {student_score}, Added by Teacher {teacher_username} at: {date_time}')
        print(f'Student with the {student_name} successfully added in the system')
    except FileNotFoundError as err:
        print('File cannot be found!')

```

register.py

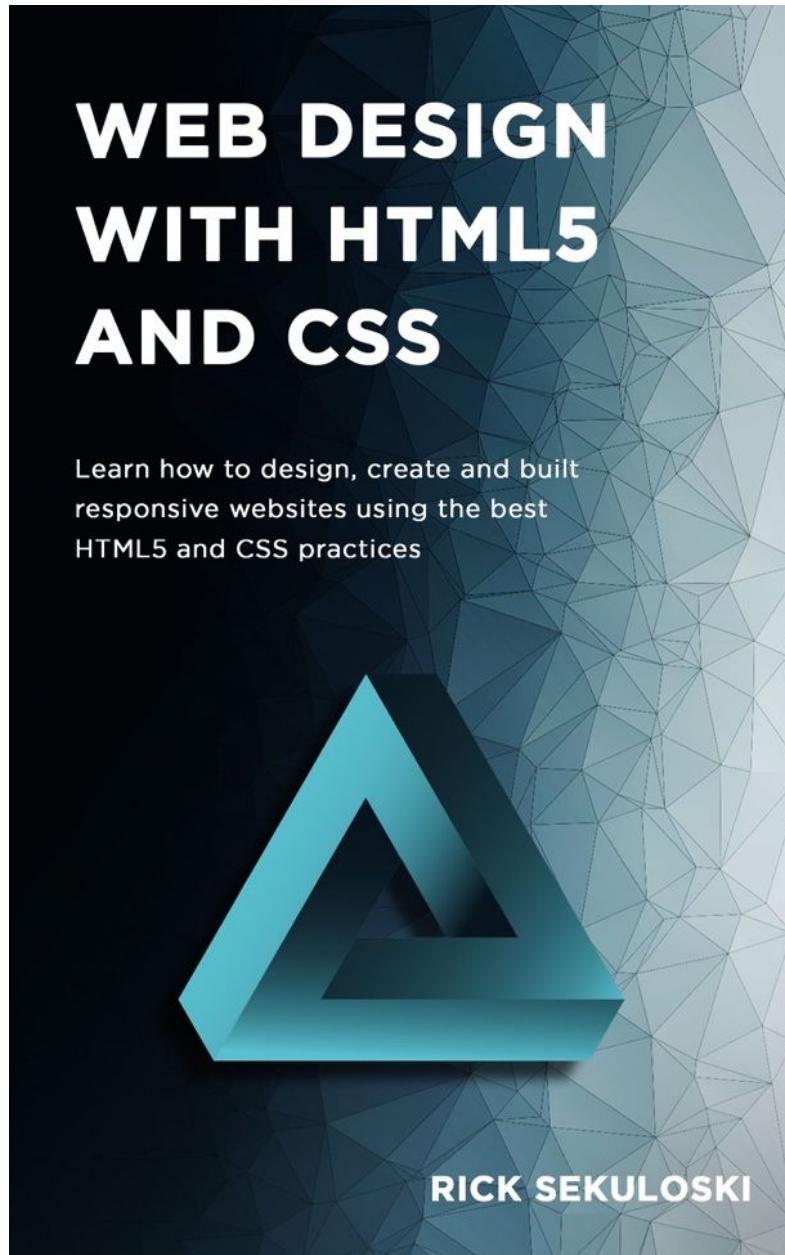
```
import bcrypt

# register new teacher
def teacher_register(user_name, user_lastname, password):
    # try - except block
    try:
        password_hashed = hash_password(password)
        with open('teachers.txt', mode='a') as new_text_file:
            new_text_file.write(f'{user_name}, {user_lastname}, {password_hashed}, \n')
        return f'{user_name} , Your registration is successful, go back and log in!'
    except FileNotFoundError as err:
        print('File cannot be found!')

# hashing function
def hash_password(user_password):
    # encode the password
    bytePwd = user_password.encode('utf-8')
    # Generate salt
    mySalt = bcrypt.gensalt()
    # Hash password
    user_password_hashed = bcrypt.hashpw(bytePwd, mySalt)
    return user_password_hashed
```

## Appendix B: My Bestseller books

If you still want to learn more about the basic to intermediate JavaScript, HTML and CSS then I suggest getting my best seller eBook:

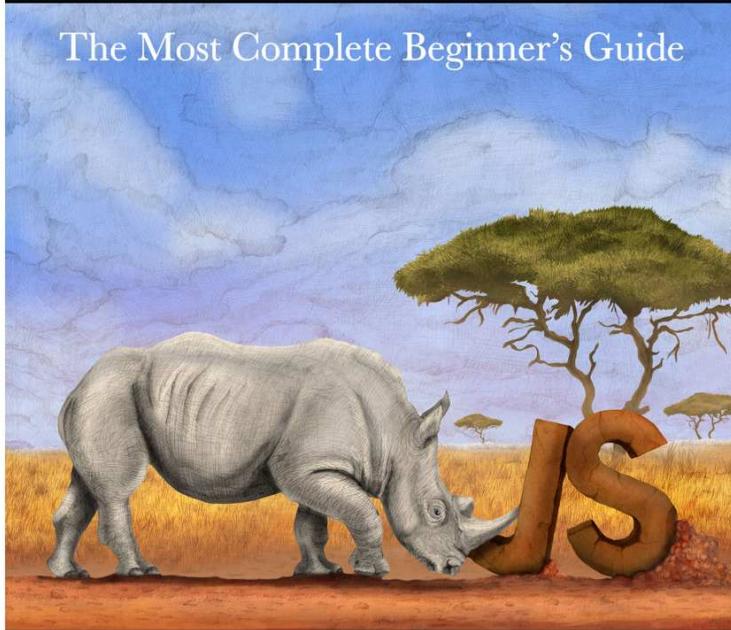


[https://www.amazon.com/Web-Design-HTML5-CSS-responsive-ebook/dp/B097DWX99H/ref=sr\\_1\\_1?](https://www.amazon.com/Web-Design-HTML5-CSS-responsive-ebook/dp/B097DWX99H/ref=sr_1_1?)

[keywords=rick+sekuloski&qid=1650700347&sprefix=rick+seku%2Caps%2C365&sr=8-1](#)

# Master Modern JavaScript Fast

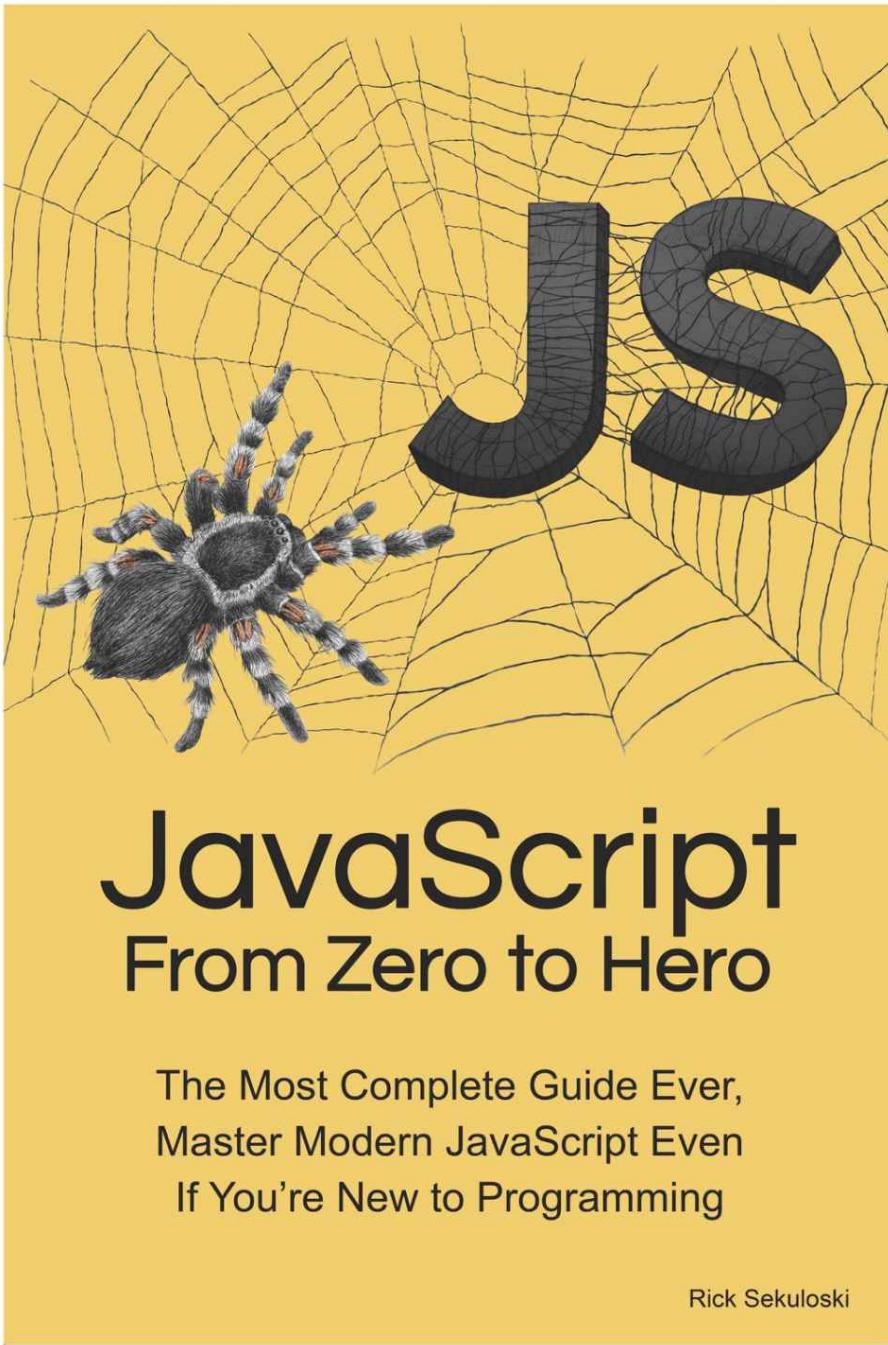
The Most Complete Beginner's Guide



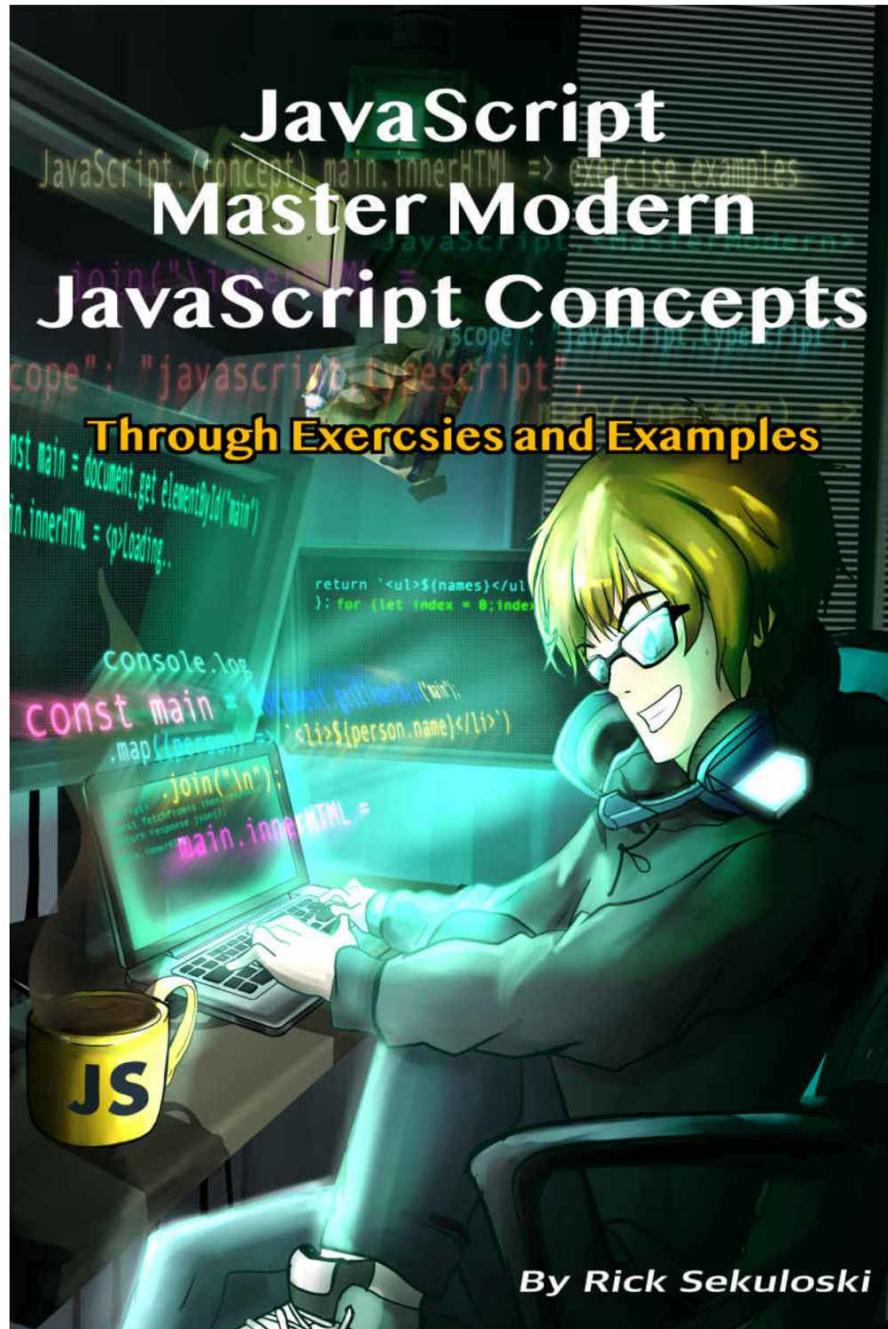
And The Weird Parts Explained

Rick Sekuloski

[https://www.amazon.com/Master-Modern-JavaScript-Fast-programming-ebook/dp/B09MB4JTMT/ref=sr\\_1\\_2?  
keywords=rick+sekuloski&qid=1650700372&suffix=rick+seku%2Caps%2C365&sr=8-2](https://www.amazon.com/Master-Modern-JavaScript-Fast-programming-ebook/dp/B09MB4JTMT/ref=sr_1_2?keywords=rick+sekuloski&qid=1650700372&suffix=rick+seku%2Caps%2C365&sr=8-2)



[https://www.amazon.com/JavaScript-Zero-Hero-Complete-Programming-ebook/dp/B09R73RWH2/ref=sr\\_1\\_3?  
keywords=rick+sekuloski&qid=1650700372&sprefix=rick+seku%2Caps%  
2C365&sr=8-3](https://www.amazon.com/JavaScript-Zero-Hero-Complete-Programming-ebook/dp/B09R73RWH2/ref=sr_1_3?keywords=rick+sekuloski&qid=1650700372&sprefix=rick+seku%2Caps%2C365&sr=8-3)



[https://www.amazon.com/JavaScript-Concepts-Exercises-Mastering-exercises-ebook/dp/B0B4W92N3R/ref=sr\\_1\\_8?crid=1D8WW0MGX5WL0&keywords=rick+sekuloski&qid=1657010993&sprefix=rick+sekulosk%2Caps%2C290&sr=8-8&asin=B0B4W92N3R&revisionId=&format=2&depth=1](https://www.amazon.com/JavaScript-Concepts-Exercises-Mastering-exercises-ebook/dp/B0B4W92N3R/ref=sr_1_8?crid=1D8WW0MGX5WL0&keywords=rick+sekuloski&qid=1657010993&sprefix=rick+sekulosk%2Caps%2C290&sr=8-8&asin=B0B4W92N3R&revisionId=&format=2&depth=1)

# **HTML and CSS**

The Most Complete  
Web Development Guide



Rick Sekuloski

[https://www.amazon.com/HTML-CSS-Complete-Development-Guide-ebook/dp/B09YNQFRYN/ref=sr\\_1\\_4?  
crid=1D8WW0MGX5WL0&keywords=rick+sekuloski&qid=165  
7010950&sprefix=rick+sekulosk%2Caps%2C290&sr=8-  
4&asin=B09YNQFRYN&revisionId=e1f1dcc3&format=1&depth  
=1](https://www.amazon.com/HTML-CSS-Complete-Development-Guide-ebook/dp/B09YNQFRYN/ref=sr_1_4?crid=1D8WW0MGX5WL0&keywords=rick+sekuloski&qid=1657010950&sprefix=rick+sekulosk%2Caps%2C290&sr=8-4&asin=B09YNQFRYN&revisionId=e1f1dcc3&format=1&depth=1)



# JavaScript Interview

## Master Your Next Interview

Land Your Dream Job with my Up-To-Date  
Interview Questions

Rick Sekuloski

[https://www.amazon.com/JavaScript-Interview-questions-Up-Date-ebook/dp/B0B5V4ZS51/ref=sr\\_1\\_9?](https://www.amazon.com/JavaScript-Interview-questions-Up-Date-ebook/dp/B0B5V4ZS51/ref=sr_1_9?)

[keywords=rick+sekuloski&qid=1661072932&sprefix=rick+sek%2Caps%2C341&sr=8-9](#)

## Appendix C: More exercises and learn more about JavaScript, HTML, and PHP

Please check out my courses if you want to learn more about JavaScript or other Web Development languages. I promise you will gain huge experience working on real-life examples.

You can find my courses on Udemy:

<https://www.udemy.com/user/riste3/>



**Ultimate PHP, Laravel, CSS & Sass!**  
**Learn PHP, Laravel & Sass**

Rick Sekuloski

**4.4 ★★★★★ (128)**

95.5 total hours • 444 lectures • All Levels

**A\$13.99 A\$109.99**



**Master JavaScript - The Most**  
**Complete JavaScript Course 2021**

Rick Sekuloski

**4.2 ★★★★★ (27)**

45.5 total hours • 181 lectures • All Levels

**A\$16.99 A\$129.99**



**Modern Web Bootcamp, Design with**  
**PHP, SASS, CSS-GRID & FLEX**

# **Appendix D: Resources**

<https://docs.python.org/3/>

