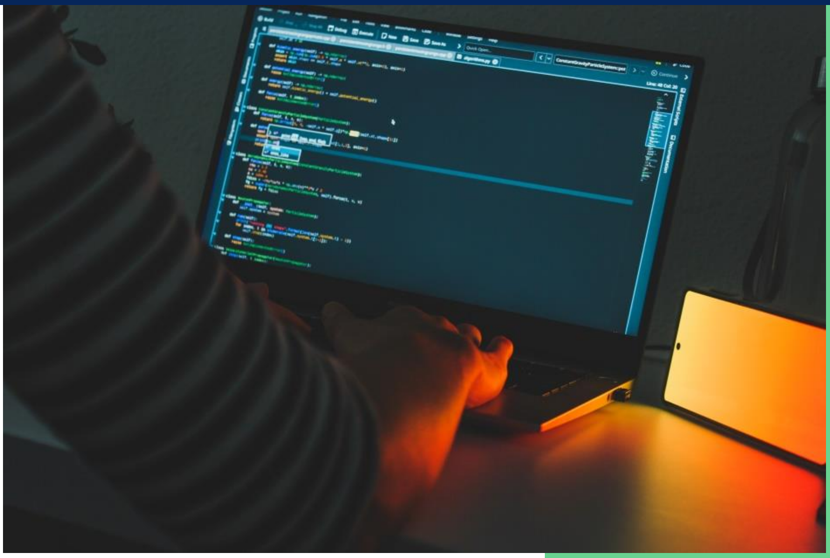
Co-funded by the Erasmus+ Programme of the European Union

We're on the Web!

Visit us at:

https://makers-project.eu

GETTING STARTED WITH PYTHON



Year of publication: **2022**

Editor: **Gergana Cisarova-Dimitrova**

MAKER SCHOOLS
3D Design for Education

Project "MAKER SCHOOLS: Enhancing Student Creativity and STEM Engagement by Integrating 3D Design and Programming into Secondary School Learning" (Agreement no. 2020-1-BG01-KA201-079274)

MAKER SCHOOLS
3D Design for Education

Contents

Co-funded by the Erasmus+ Programme of the European Union

Co-funded by the Erasmus+ Programme of the European Union

Why learn Python?

Python is a programming language released in 1991. It is popular and easy to learn and is currently in high demand on the job market. Its popularity has been rising because it is widely used in data science, artificial intelligence and machine learning. If you wish to work on artificial intelligence and machine learning in the future, learning Python is a must for you. Python is versatile and can be used both for simple and for very complex tasks. It can be used for developing the back-end part of a web application (on a server), but it cannot be used to develop the front-end (what the user sees and interacts with on the browser). You can use Python to connect to databases. Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.).

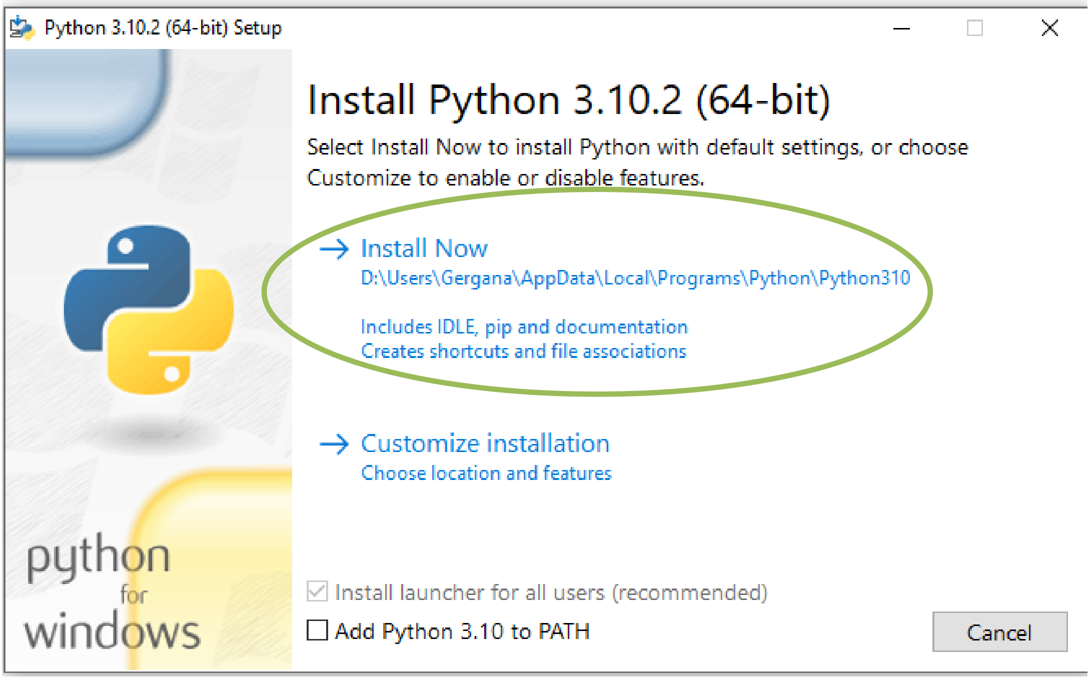
Getting started

Install an Integrated Development Environment (IDE)

Before you start learning Python, you first need to install Python itself if your computer does not already have it installed (most Windows computers will not).

Download the last version suitable for your operational system from https://www.python.org/downloads/. If you wish to directly download the latest version for Windows as of January 2022, click here.

Start the standard installation as in the screenshot below and follow the instructions.


Python 3.10.2 (64-bit) Setup

Install Python 3.10.2 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

→ Install Now
D:\Users\Gergana\AppData\Local\Programs\Python\Python310

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

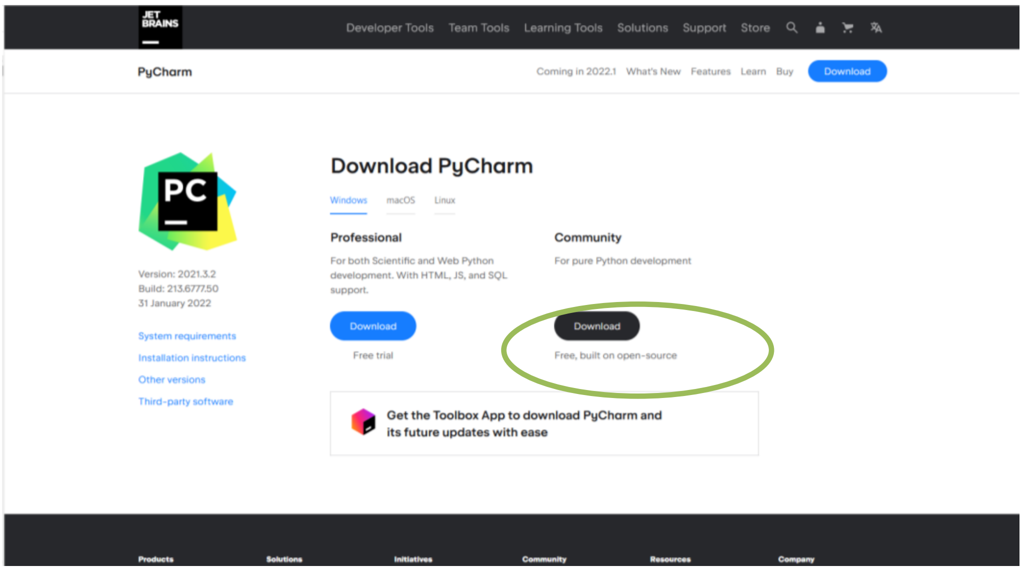
☑ Install launcher for all users (recommended)
☐ Add Python 3.10 to PATH

Python can be run even from the command-line console in Windows. However, this would not be a pleasant experience. In order to work easily with Python, it is recommended to choose and install an Integrated Development Environment (IDE) - a specialized software that allows you to compile and run Python scrips, create and save files and projects, etc. There are many IDE options that you can easily find on the internet. The choice of IDE is usually a matter of personal preferences. If you are already using one popular IDE for another language (e.g. Visual Studio Code or any JetBrains IDE) you are typically used to its interface and would use the same for Python if it is supported. For learning Python, we recommend using Visual Studio Code or PyCharm Community (Community is the free version and there is also a paid version with more features).

Co-funded by the Erasmus+ Programme of the European Union

PyCharm can be a great help for beginners as it suggests corrections to typical mistakes and has powerful features for suggesting code while you type. The disadvantages are that the program takes a lot of memory and it requires some practice to learn how to use all the useful features. Once you do learn, however, writing Python code with PyCharm will be faster and easier than with some other IDEs. We recommend using the Thonny IDE if you are using Python for 3D design due to the possibility to integrate it with 3D software (see the Module *Using Python for Procedural 3D Content Generation for 3D Printing*). Below, we guide you through the installation of PyCharm Community on Windows.

Download the PyCharm Community installation file from https://www.jetbrains.com/pycharm/download/#section=windows.


JET BRAINS
Developer Tools Team Tools Learning Tools Solutions Support Store
PyCharm
Coming in 2022.1 What's New Features Learn Buy Download
Windows macOS Linux
System requirements
Installation instructions
Other versions
Third-party software
Products Solutions Initiatives Community Resources Company

Download PyCharm

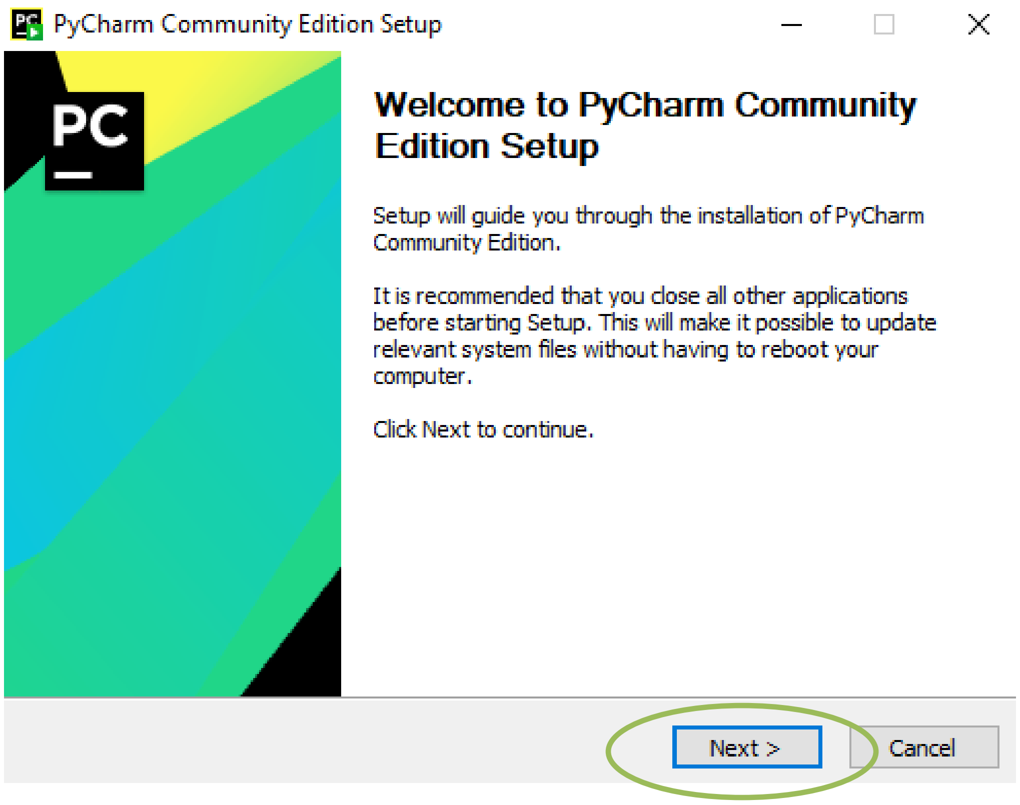
Version: 2021.3.2
Build: 213.6777.50
31 January 2022

Professional
For both Scientific and Web Python development. With HTML, JS, and SQL support.
Download
Free trial

Community
For pure Python development
Download
Free, built on open-source

Get the Toolbox App to download PyCharm and its future updates with ease

Once you start the installation, the following window will open.



PyCharm Community Edition Setup

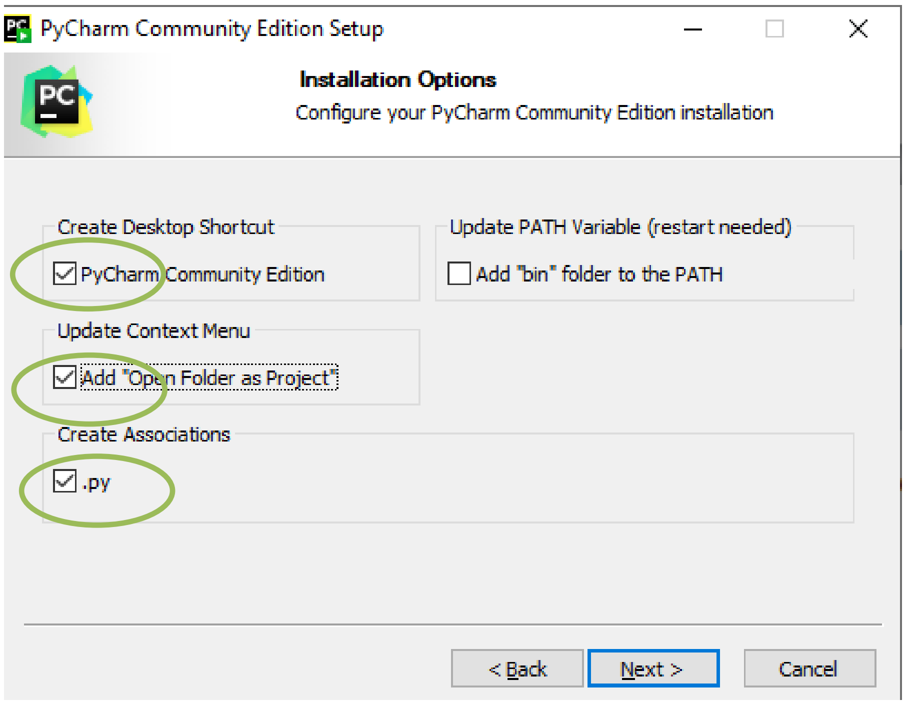
Welcome to PyCharm Community Edition Setup

Setup will guide you through the installation of PyCharm Community Edition.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

After clicking "Next" twice, you will encounter the following installation window. Select the options as shown in the screenshot below:



PyCharm Community Edition Setup

Installation Options

Configure your PyCharm Community Edition installation

Create Desktop Shortcut
- [✓] PyCharm Community Edition

Update PATH Variable (restart needed)
- [ ] Add "bin" folder to the PATH

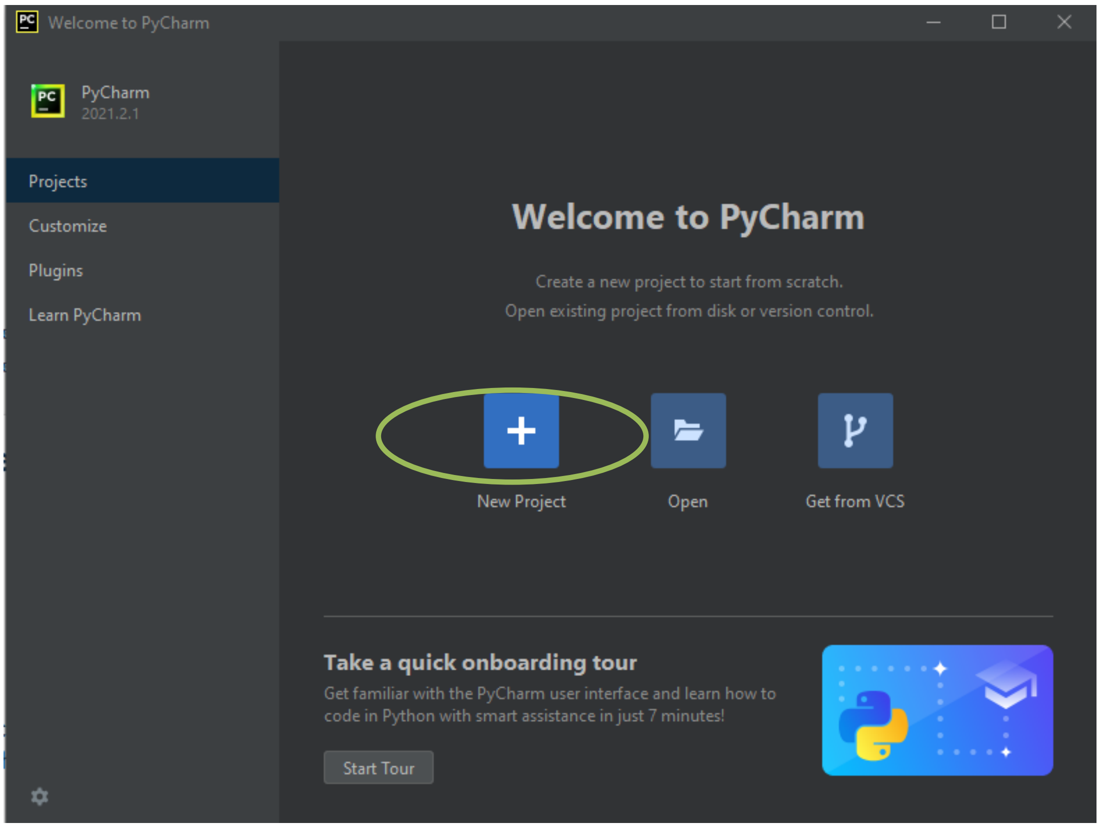
Update Context Menu
- [✓] Add "Open Folder as Project"

Create Associations
- [✓] .py

Finalize the installation as suggested by the Installer. You are now ready to start using the IDE. You can choose between dark and light modes and different color schemes any time, according to personal preference. Press Ctrl+Alt+S to open the IDE settings and select Editor | Color Scheme. Whenever you have trouble remembering how to use some features in this IDE, search the internet. The IDE has a lot of useful features accessible through key combinations.

Co-funded by the Erasmus+ Programme of the European Union

Create a project in PyCharm



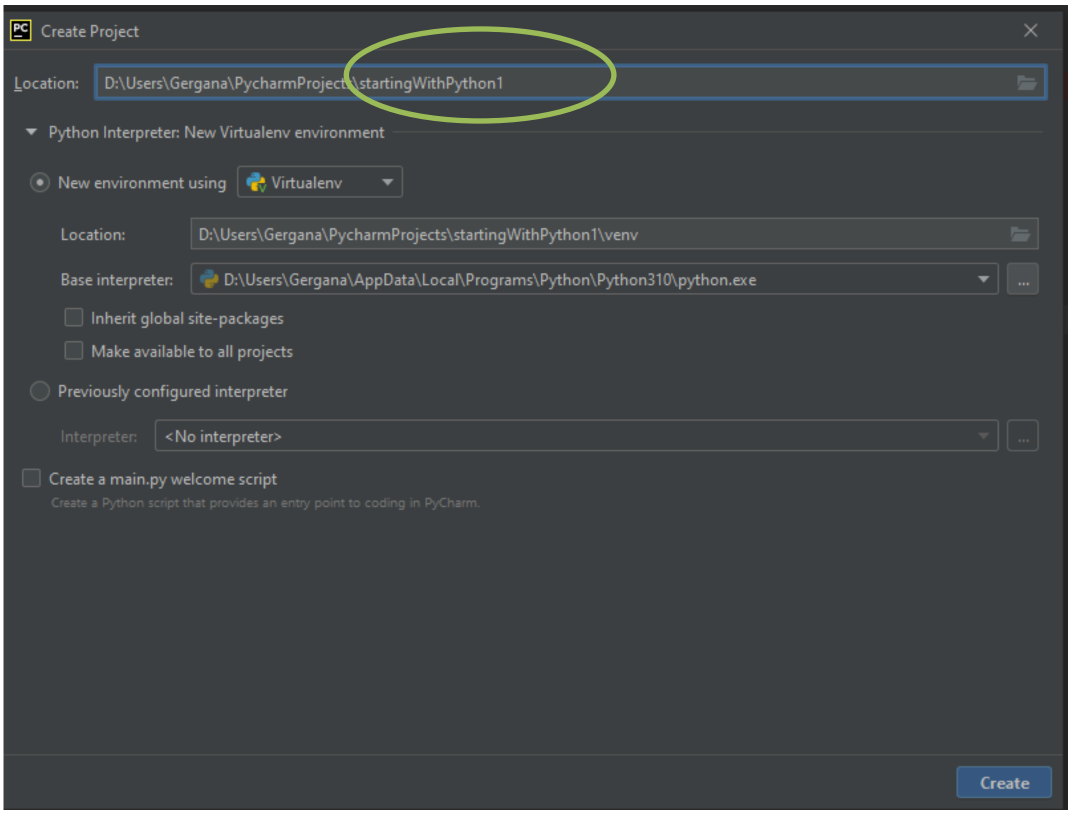
Welcome to PyCharm

Create a new project to start from scratch.
Open existing project from disk or version control.

Take a quick onboarding tour

Get familiar with the PyCharm user interface and learn how to code in Python with smart assistance in just 7 minutes!

In the window that opens, write the name of your project, leave everything else as suggested (although you may wish to unclick the creation of the main.py welcome script), then click "Create".



Location: D:\Users\Gergana\PycharmProjects\startingWithPython1

Python Interpreter: New Virtualenv environment

New environment using: Virtualenv

Location: D:\Users\Gergana\PycharmProjects\startingWithPython1\venv

Base interpreter: D:\Users\Gergana\AppData\Local\Programs\Python\Python310\python.exe

Inherit global site-packages

Make available to all projects

Previously configured interpreter

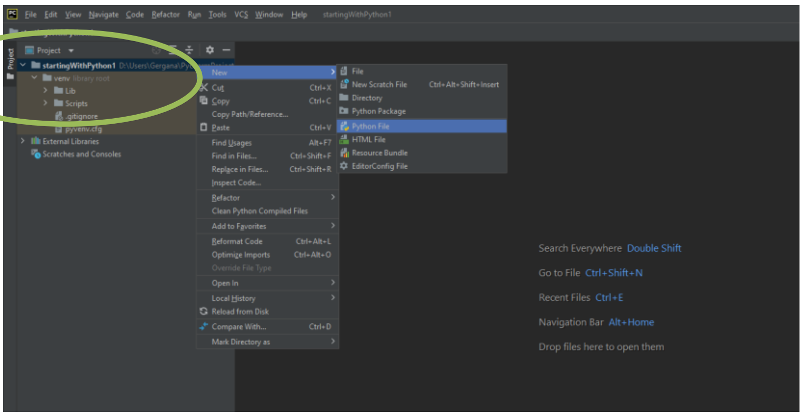
Interpreter: <No interpreter>

Create a main.py welcome script

Create a Python script that provides an entry point to coding in PyCharm.

Co-funded by the Erasmus+ Programme of the European Union

Once the project is created, create a new python file by right-clicking on the name of the project -> New -> Python File. Give your file a name, e.g. example.py.



Register on HackerRank to practice

In order to be able to solve exercises and to check if they are solved correctly, register on HackerRank: https://www.hackerrank.com/dashboard.

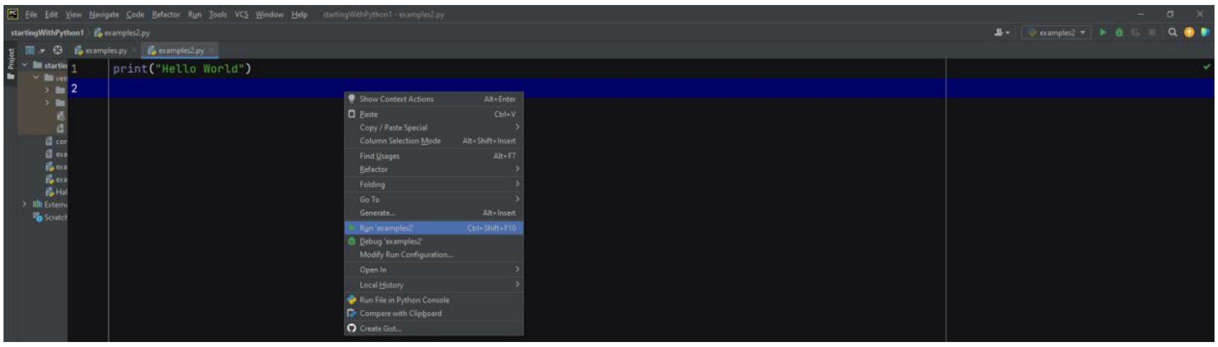
In the *Prepare* section, you will find many challenges, starting from the simplest exercises to much more complicated ones. Getting acquainted with HackerRank may even help you later on with job interviews as applicants are sometimes given tasks in HackerRank to solve.

Create your first program

In the Python file you created, type the following command

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

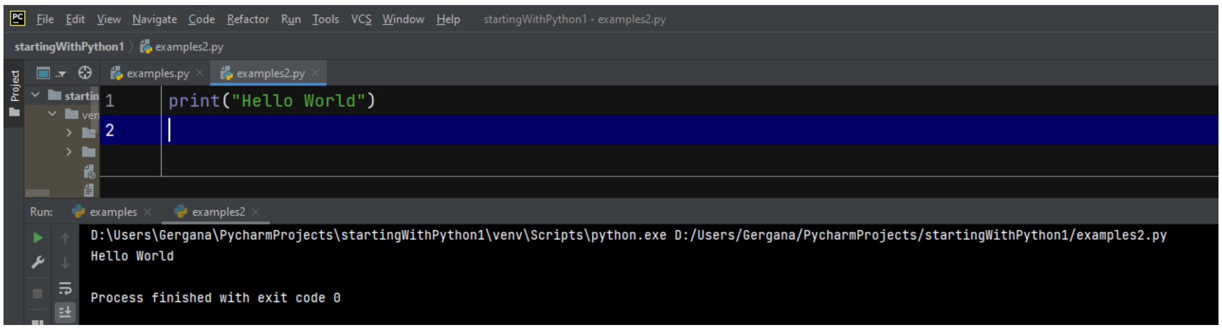
To run the program, right-click anywhere on the file window and click “Run”. Alternatively, use the Ctrl + Shift + F10 key combination.



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

File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples2.py
startingWithPython1 examples2.py
examples.py examples2.py
Show Context Actions Alt+Enter
Paste Ctrl+V
Copy / Paste Special
Column Selection Mode Alt+Shift+Insert
Find Usages Alt+F7
Refactor
Folding
Go To
Generate... Alt+Insert
Run 'examples2' Ctrl+Shift+F10
Debug 'examples2'
Modify Run Configuration...
Open In
Local History
Run File in Python Console
Compare with Clipboard
Create Gist...

As you can see, your first program wrote *Hello, World!* on the console in the bottom of the working window.



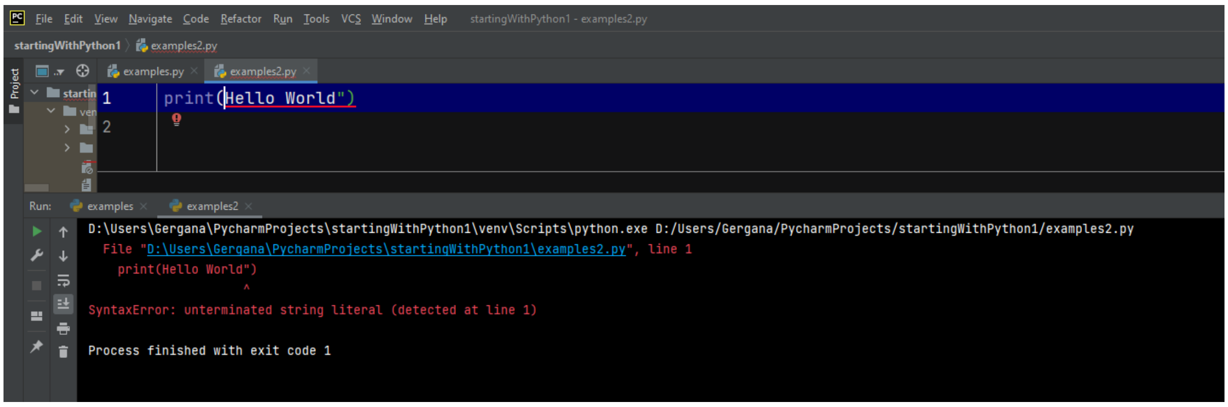
```
print("Hello World")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Hello World

Process finished with exit code 0
```

File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples2.py
startingWithPython1 examples2.py
examples.py examples2.py
Run: examples examples2

Co-funded by the Erasmus+ Programme of the European Union

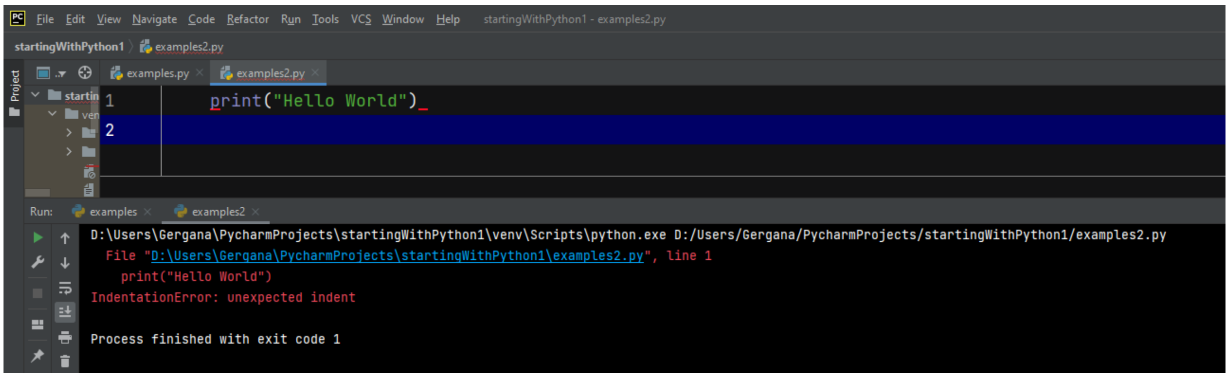
If you have made any mistake in the syntax e.g. you have forgotten the opening or closing quotation mark, the console will tell you so, and the exit code will be 1, instead of 0.


File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples2.py
startingWithPython1 examples2.py
examples.py examples2.py
Run: examples examples2

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

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
  File "D:\Users\Gergana\PycharmProjects\startingWithPython1\examples2.py", line 1
    print(Hello World")
                     ^
SyntaxError: unterminated string literal (detected at line 1)

Process finished with exit code 1
```

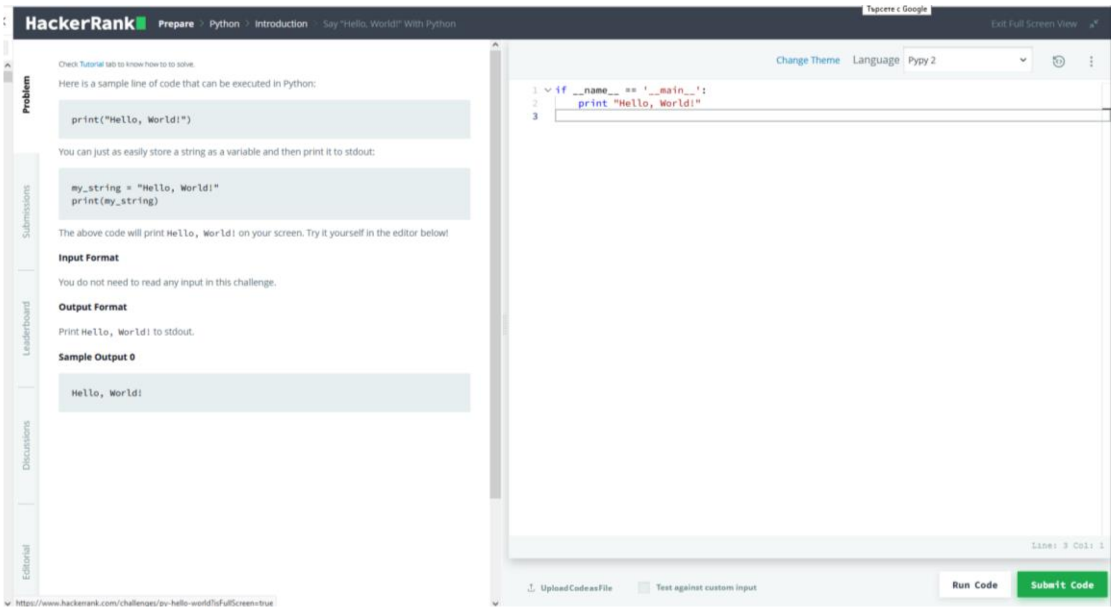
If the mistake is not in the syntax but the indentation before the command, the console will also let you know by showing an Indentation Error.


File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples2.py
startingWithPython1 examples2.py
examples.py examples2.py
Run: examples examples2

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

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
  File "D:\Users\Gergana\PycharmProjects\startingWithPython1\examples2.py", line 1
    print("Hello World")
IndentationError: unexpected indent

Process finished with exit code 1
```

Now you can try to also test your program in HackerRank. Find the "Hello, World!" challenge, and submit your code.



Check Tutorial tab to know how to to solve.

Here is a sample line of code that can be executed in Python:

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

You can just as easily store a string as a variable and then print it to stdout:

```
my_string = "Hello, World!"
print(my_string)
```

The above code will print `Hello, World!` on your screen. Try it yourself in the editor below!

Input Format

You do not need to read any input in this challenge.

Output Format

Print `Hello, World!` to stdout.

Sample Output 0

```
Hello, World!
```

```
if __name__ == '__main__':
    print "Hello, World!"
```

Co-funded by the Erasmus+ Programme of the European Union

Variables and datatypes

Similarly to other programming languages, Python programming relies on variables, which are containers for storing data values. Variables have a name, a type and a value. There are seven built-in datatypes in Python:

- Binary types: memoryview, bytearray, bytes
- Boolean type: bool
- Set types: frozenset, set
- Mapping type: dict
- Sequence types: range, tuple, list
- Numeric types: complex, float, int
- Text type: str

There is no command for declaring a variable in Python, the variable is declared by assigning a value to it. The assignment is done with a simple = sign.

```
var = 5
```

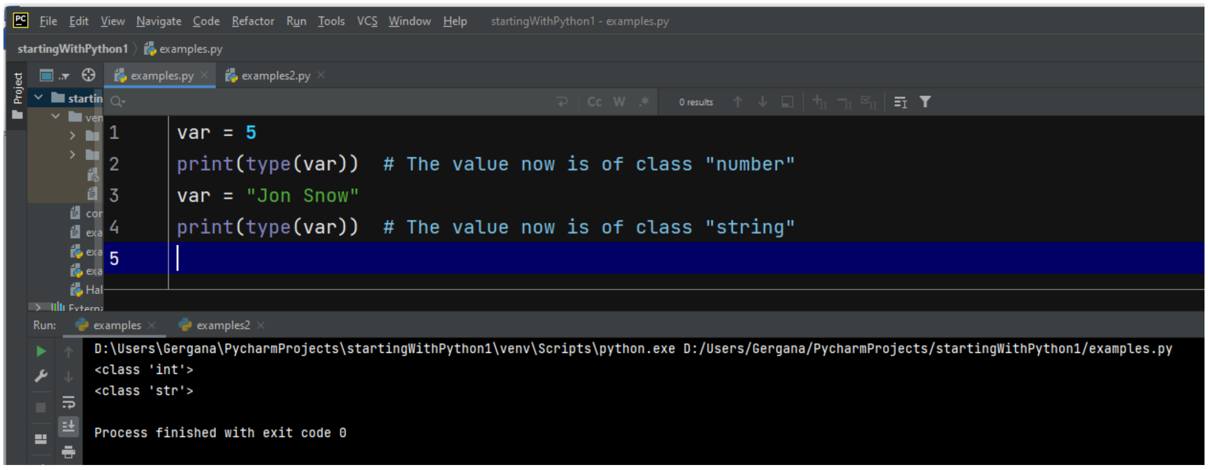
The variable named var is assigned a value 5, and hence it is a numeric type.

A chain assignment is possible, where the same value can be assigned to several variables.

```
x = y = z = 15
```

The three different variables x, y and z now all have a value 15.

In some languages, such as Java, a variable is declared to have a specific datatype (e.g. a string), and while we can assign different values to this variable during its lifetime in the program, this value has to be of this specific datatype. Python is a dynamic language and this is not necessary. The datatype is not explicitly declared and it depends only on the value we assign and re-assign to the variable. If a value of another type is assigned to an already declared variable, the datatype will change. In the example below, notice how the class types printed on the console are changing as we assign a new value.



File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples.py
startingWithPython1 > examples.py
examples.py
examples2.py
Run: examples examples2

```
var = 5
print(type(var))  # The value now is of class "number"
var = "Jon Snow"
print(type(var))  # The value now is of class "string"
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
<class 'int'>
<class 'str'>

Process finished with exit code 0
```

Co-funded by the Erasmus+ Programme of the European Union

Starting with the basics

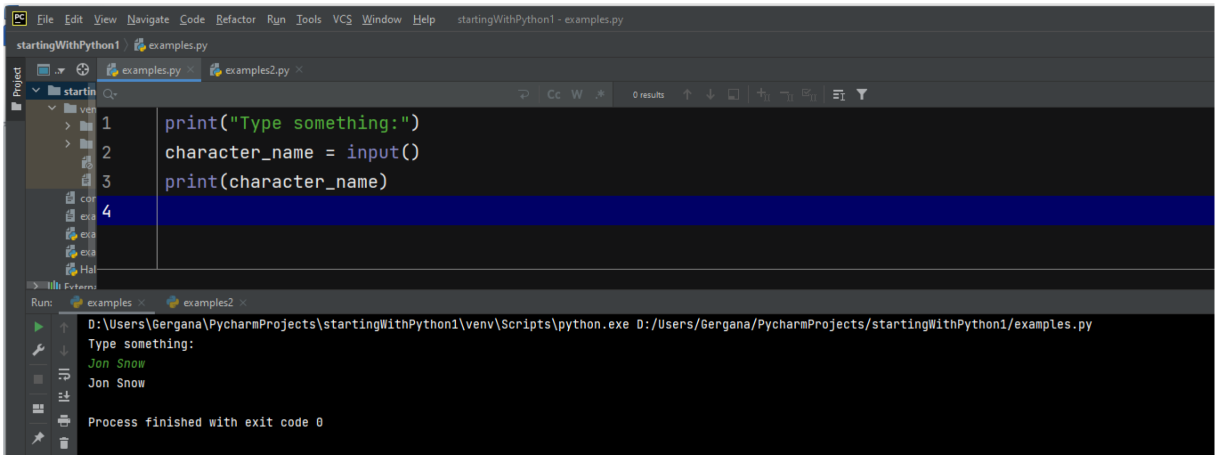
Working with the console

The console works only with text. Whatever we print on the console is converted into text. In the example above you already saw how we print on the console, namely with the print(*our variable*) command. Try it again with the code below and observe the result on the console.

```
var = 5
print(var)
```

We can also read whatever we have typed on the console and use it in our program. This is done with the input() command. We can assign the value of the input to a variable and use it in the program.

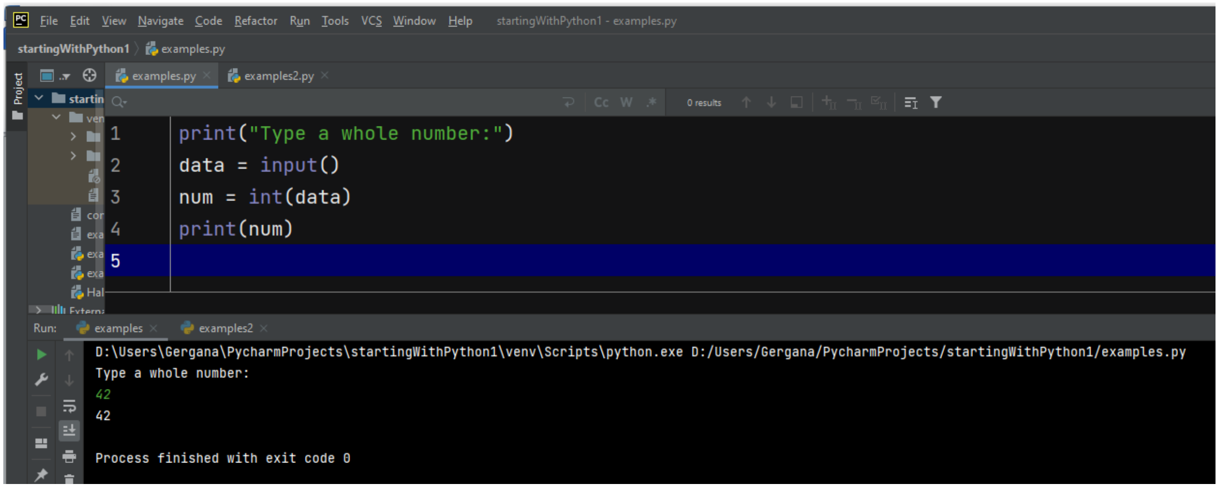
The code we show in the example below will read what the user writes on the console and then print it back on the console (the text we input is in green, the printed text is standard grey).



```
print("Type something:")
character_name = input()
print(character_name)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Type something:
Jon Snow
Jon Snow

Process finished with exit code 0
```

In a way, this was easy because we input text ("Jon Snow") and Python automatically assigned it to a *str* variable because it is a console input. But what if we want to read a number from the console and assign it to a numeric variable. Since the data we input in the console is always text, we need to convert it (cast it) into another datatype if we want to assign a numeric value to the variable. Before we cast text to another datatype, we need to be sure the casting will be possible. We cannot cast a text like "Jon Snow" to a numeric value. In the example below the variable *num* will be a numeric and can be used as such.

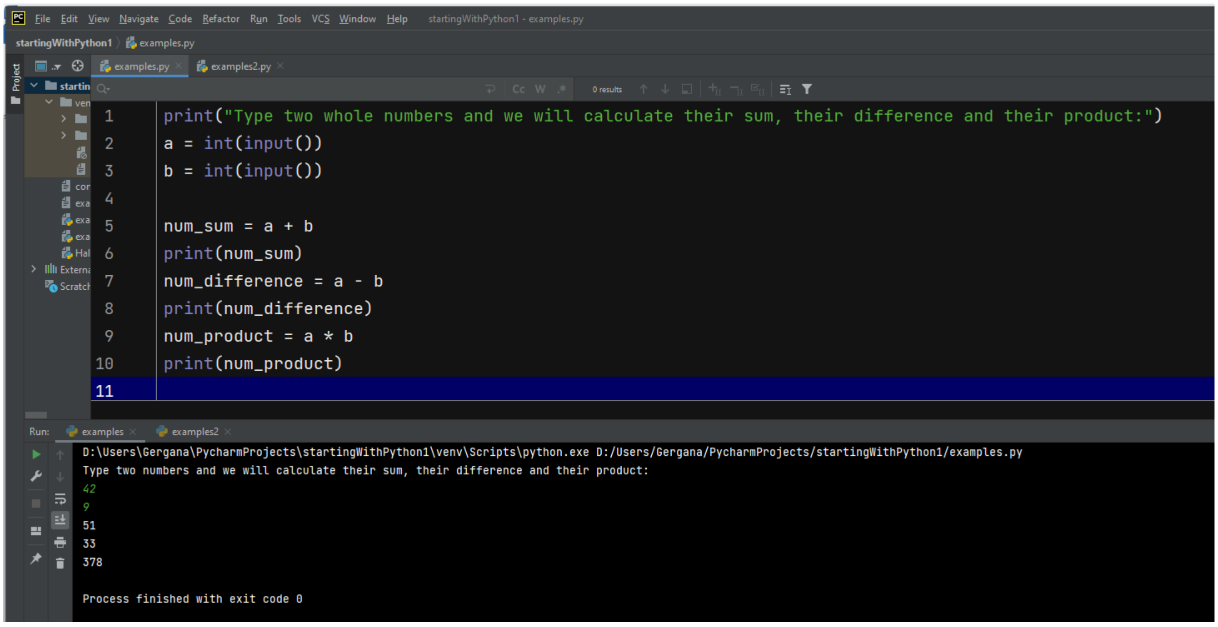


```
print("Type a whole number:")
data = input()
num = int(data)
print(num)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Type a whole number:
42
42

Process finished with exit code 0
```

Co-funded by the Erasmus+ Programme of the European Union

Let us see another example below. This program will read two integers we type on the console and will perform some basic arithmetic operations with them, namely addition (with the + operator), subtraction (with the – operator) and multiplying (with the * operator). We assign the results of the arithmetic operations to variables and then we can print them. Below is the result we will get. Remember, the input we typed is shown in green, and the result is shown in grey.



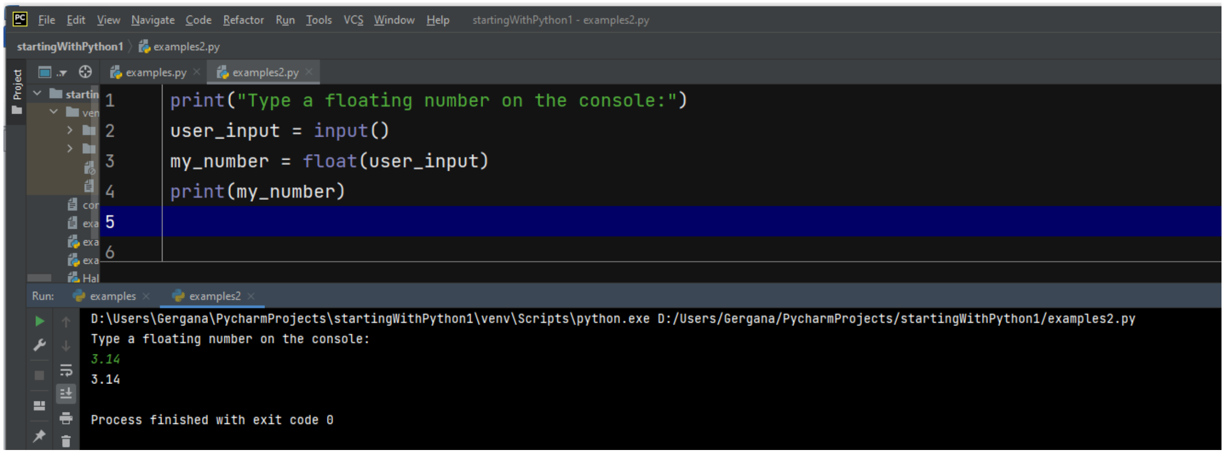
```python
print("Type two whole numbers and we will calculate their sum, their difference and their product:")
a = int(input())
b = int(input())

num_sum = a + b
print(num_sum)
num_difference = a - b
print(num_difference)
num_product = a * b
print(num_product)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Type two numbers and we will calculate their sum, their difference and their product:
42
9
51
33
378

Process finished with exit code 0
```

Tip on useful PyCharm functionality:

In longer programs in PyCharm you can use the Ctrl+Alt+L key combination to quickly format all your code automatically and minimize indentation errors.

We may also want to read floating numbers from the console. We can do that by casting the input to float. Below we show how to read the number 3.14 from the console and then print it back on the console.

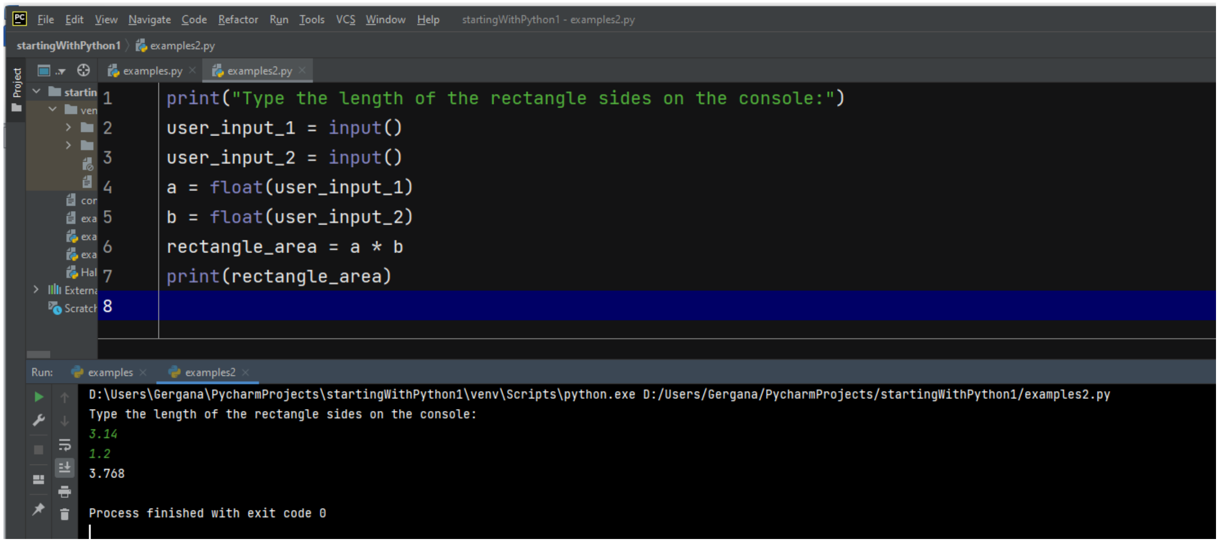


```python
print("Type a floating number on the console:")
user_input = input()
my_number = float(user_input)
print(my_number)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Type a floating number on the console:
3.14
3.14

Process finished with exit code 0
```

In the following example below we read two floating numbers typed by the user, which represent the length of the two sides of a rectangle. We then calculate the area of the rectangle (S=a*b). In order to read two floating numbers typed on the console one after the other, we need to create two variables with the input() function. The program will wait for the user to input as many numbers as we have planned to read and only then it will proceed to make the calculations. In our example, the area of a rectangle with sides 3.14 and 1.2 is 3.768.

Co-funded by the Erasmus+ Programme of the European Union


File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples2.py
startingWithPython1 examples2.py
examples.py
examples2.py
Run: examples examples2

```python
print("Type the length of the rectangle sides on the console:")
user_input_1 = input()
user_input_2 = input()
a = float(user_input_1)
b = float(user_input_2)
rectangle_area = a * b
print(rectangle_area)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Type the length of the rectangle sides on the console:
3.14
1.2
3.768

Process finished with exit code 0
```

Tip on naming variables in Python:

By convention, Python variables are written as lowercase single letter, or lowercase word, or a combination of lowercase words. Also by convention, once you are working in the software industry, you will be expected to name your variables in English in order to facilitate work in multinational teams. If more than one word is used, we separate words with underscores to improve readability. This is called *snake case* style of naming. Using more than one word in the variable name is a common practice when you want to describe better what the variable is. This will help the next coder to understand your code better. For example, it will allow you to differentiate between rectangle_side and triangle_side in a more complex program.

The *snake case* style is different from other styles used in other programming languages. For example, in Java (which you may be familiar with from your Computer Science classes) the convention is to use *camel case*, which would translate to rectangleSide and triangleSide.

Following the naming convention is not compulsory for the Python programming environment, so your program will run regardless of what you name your variables. However, following the naming conventions will be expected once you start working in the industry or if you are sharing your code with other coders, so it is advisable to get used to them already while studying different programming languages.

Arithmetic operations

We have already used three of the 4 main operators for arithmetic operations in Python when we exercised reading numbers from the console (see above):

+	Addition
-	Subtraction
*	Multiplication

Now we only need to learn more about division. You can use three different operators for division in Python:

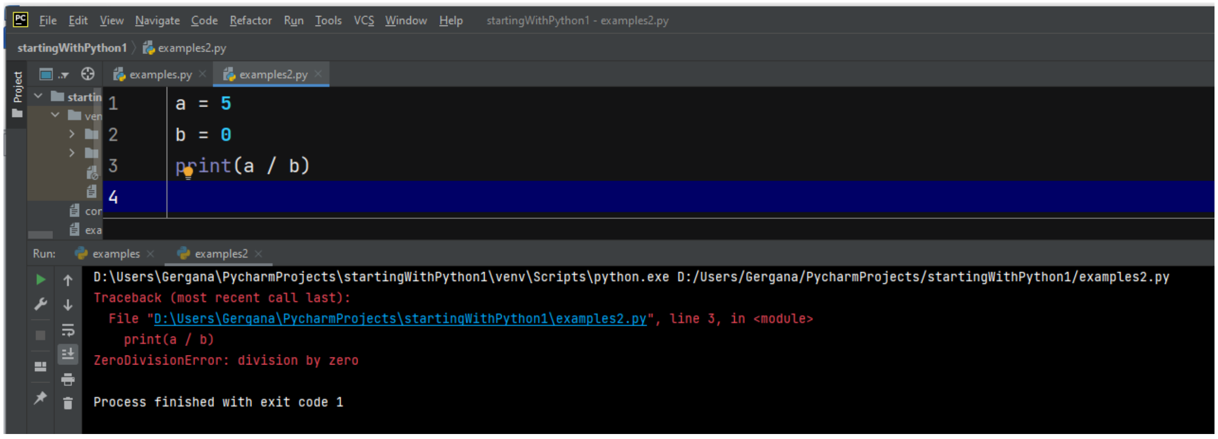
/	Regular division which will return the exact product of dividing two numbers, and it will always be a floating number. Thus, the division 5/2 will return 2.5, while 4/2 will return

Co-funded by the Erasmus+ Programme of the European Union

	2.0.
//	Integer division (also known as floor division) which will divide the two numbers but will round *down* the result to an integer (hence the name "floor"). Thus, the division 5//2 will return 2

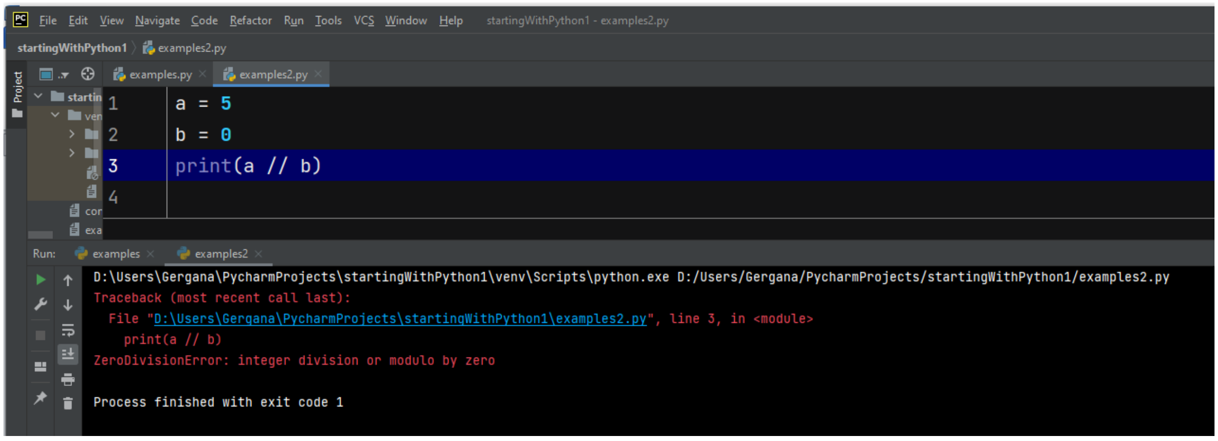
Now try yourself in PyCharm how these two operators work.

Remember that division by 0 is not allowed and will produce a ZeroDivision error and the program will terminate. This is important to keep in mind in your program if the variable that you use could potentially receive the value 0. Below we show you examples of division with 0.



```
a = 5
b = 0
print(a / b)
```

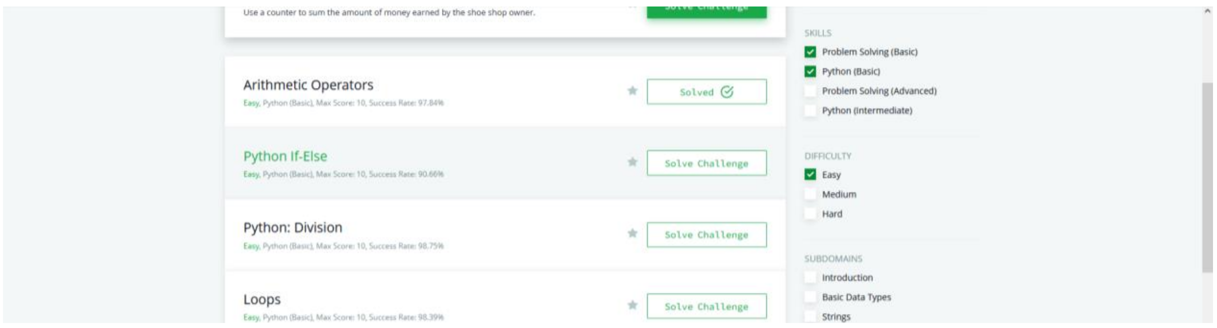
```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Traceback (most recent call last):
  File "D:\Users\Gergana\PycharmProjects\startingWithPython1\examples2.py", line 3, in <module>
    print(a / b)
ZeroDivisionError: division by zero

Process finished with exit code 1
```



```
a = 5
b = 0
print(a // b)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Traceback (most recent call last):
  File "D:\Users\Gergana\PycharmProjects\startingWithPython1\examples2.py", line 3, in <module>
    print(a // b)
ZeroDivisionError: integer division or modulo by zero

Process finished with exit code 1
```

With this knowledge, you are ready to log into HackerRank and solve the Arithmetic Operations Challenge and the Python: Division Challenge (make sure you have ticked the *Easy* difficulty level).



Use a counter to sum the amount of money earned by the shoe shop owner.

Arithmetic Operators
Easy, Python (Basic), Max Score: 10, Success Rate: 97.84%
Solved

Python If-Else
Easy, Python (Basic), Max Score: 10, Success Rate: 90.66%
Solve Challenge

Python: Division
Easy, Python (Basic), Max Score: 10, Success Rate: 98.75%
Solve Challenge

Loops
Easy, Python (Basic), Max Score: 10, Success Rate: 98.39%
Solve Challenge

SKILLS
✓ Problem Solving (Basic)
✓ Python (Basic)
Problem Solving (Advanced)
Python (Intermediate)

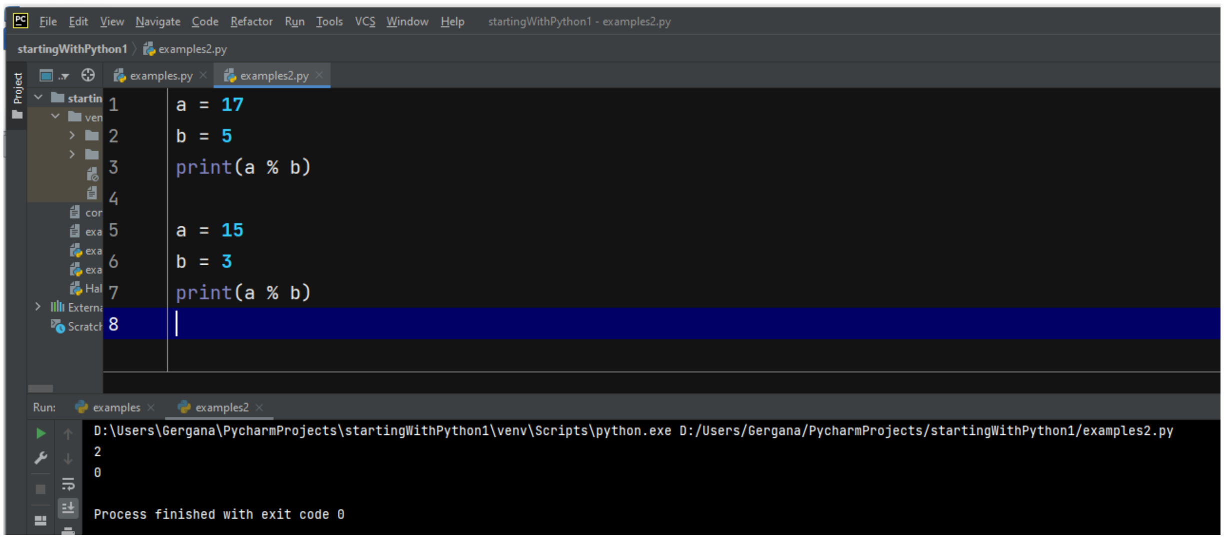
DIFFICULTY
✓ Easy
Medium
Hard

SUBDOMAINS
Introduction
Basic Data Types
Strings

And now let us learn about the last type of division possible in Python.

Co-funded by the Erasmus+ Programme of the European Union

%	Modulus division which will divide the numbers but will not return the product of the division. It will only return the remainder of an integer division. Thus, the division 5%2 will return 1 because 2 is contained in 5 twice (2*2=4) but there is a remainder of 1 (4+1=5). On the other hand 4%2 will return 0, because 2 is contained in 4 twice (2*2=4) and there is no remainder. The modulo operator can be very handy when combined with integer division in cases when we need to disintegrate numbers into hundreds, tens and ones, or when we need to know if the number is even or odd.


File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples2.py
startingWithPython1 examples2.py
examples.py examples2.py
Run: examples examples2

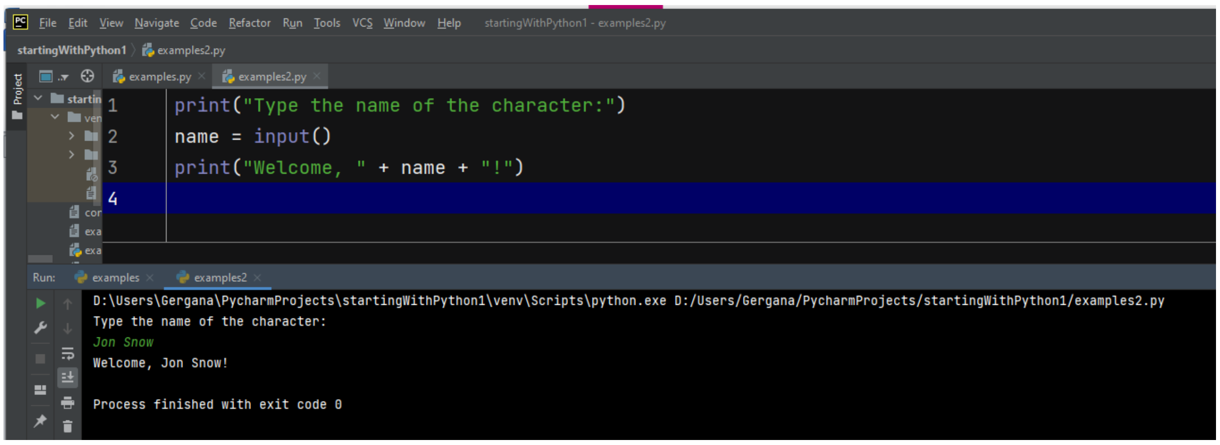
```
a = 17
b = 5
print(a % b)

a = 15
b = 3
print(a % b)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
2
0

Process finished with exit code 0
```

Like with the integer and regular division, we cannot perform modulus division by 0 and we will get a ZeroDivision error, which will terminate the whole program.

Printing on the console

We can use the variables we read from the console to print more complex texts. For example, the following code will print "Welcome, [*name provided by the user*]!". We use the + operator to combine the different parts of the text with the input variable. Be careful to add an empty space after "Welcome," or else you will print "Welcome,[*name provided by the user*]!".


File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples2.py
startingWithPython1 examples2.py
examples.py examples2.py
Run: examples examples2

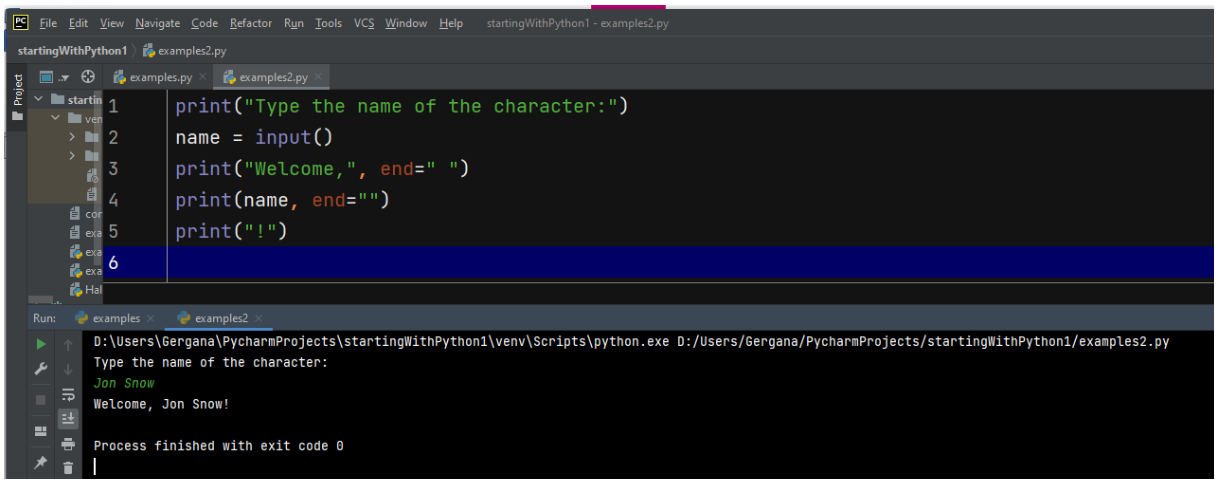
```
print("Type the name of the character:")
name = input()
print("Welcome, " + name + "!")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Type the name of the character:
Jon Snow
Welcome, Jon Snow!

Process finished with exit code 0
```

By default the print() command prints a new line after the printed text. However, you can change this by indicating that the end of the print should be for example an empty space or no space at all. You can do this by assigning the desired print ending to the end variable and inputting it as an additional argument in the print() function. In the example below, we will print "Welcome, [*name*

Co-funded by the Erasmus+ Programme of the European Union

provided by the user]!" using this approach. The disadvantage of this approach is that we call the print() function 3 times.



```
print("Type the name of the character:")
name = input()
print("Welcome,", end=" ")
print(name, end="")
print("!")
```

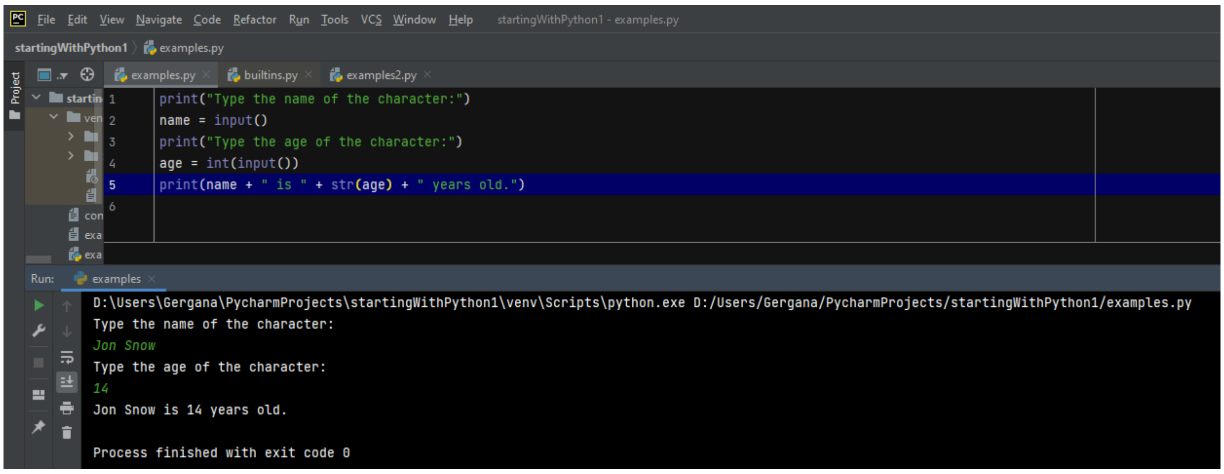
```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Type the name of the character:
Jon Snow
Welcome, Jon Snow!

Process finished with exit code 0
```

Tip on useful PyCharm functionality:

In PyCharm, you can quickly duplicate one line by placing your cursor anywhere over the line you want to duplicate and pressing the Ctrl+D key combination. The code will be duplicated right below the selected line. If you select a piece of code and then press Ctrl+D, you will duplicate the selection. This helps to write code faster if the code is repetitive.

It is a bit more complicated when we want to print variables that are numbers. In that case, to convert the number into text, we use the str() function and input the name of the number variable as an argument (by placing it inside the brackets of the function). It works for both integers and floating numbers, as long as the input is cast to floating number by using float(input()) .

```
age = int(input())
str(age)
```



```
print("Type the name of the character:")
name = input()
print("Type the age of the character:")
age = int(input())
print(name + " is " + str(age) + " years old.")
```

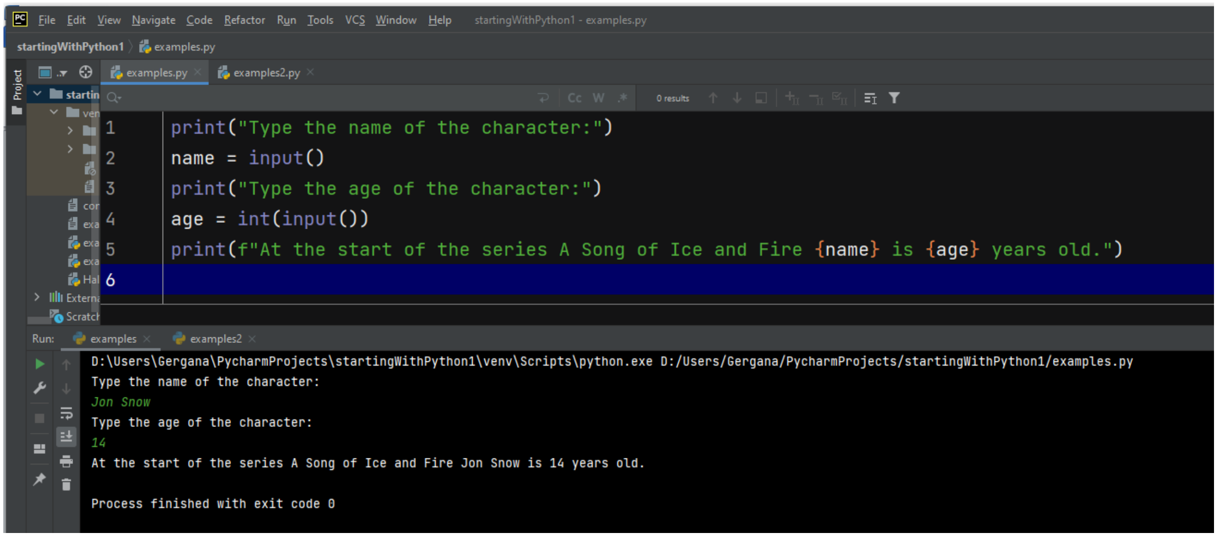
```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Type the name of the character:
Jon Snow
Type the age of the character:
14
Jon Snow is 14 years old.

Process finished with exit code 0
```

Tip on simplifying code:

We do not need to first assign the input() result to a variable and then pass this variable as an argument in the print() function. We can directly pass input() as an argument into the print() function: i.e. print(input())

Co-funded by the Erasmus+ Programme of the European Union

Using the + operator for printing is intuitive but the code ends up being quite cluttered. While it does not matter in this small example, imagine that you need to combine the text from 10 variables… Fortunately, there is a more streamlined way which is based on formatting the output string (also known as *string interpolation*). Starting from Python 3.6, string interpolation can be performed by placing the f symbol in front of the text string, and placing text and all the variables' placeholders between the quotation marks of the string. The placeholder for a variable is made up of the variable name, placed inside curly brackets, namely *{variable name}*. To understand the approach, see the example below. We no longer need to worry about the + operator or to add empty spaces. The variable names in the curly brackets will be neatly replaced with the user input from the console. There are also other ways of formatting text, which we will deal with in the Section "Working with Text


```python
print("Type the name of the character:")
name = input()
print("Type the age of the character:")
age = int(input())
print(f"At the start of the series A Song of Ice and Fire {name} is {age} years old.")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Type the name of the character:
Jon Snow
Type the age of the character:
14
At the start of the series A Song of Ice and Fire Jon Snow is 14 years old.

Process finished with exit code 0
```

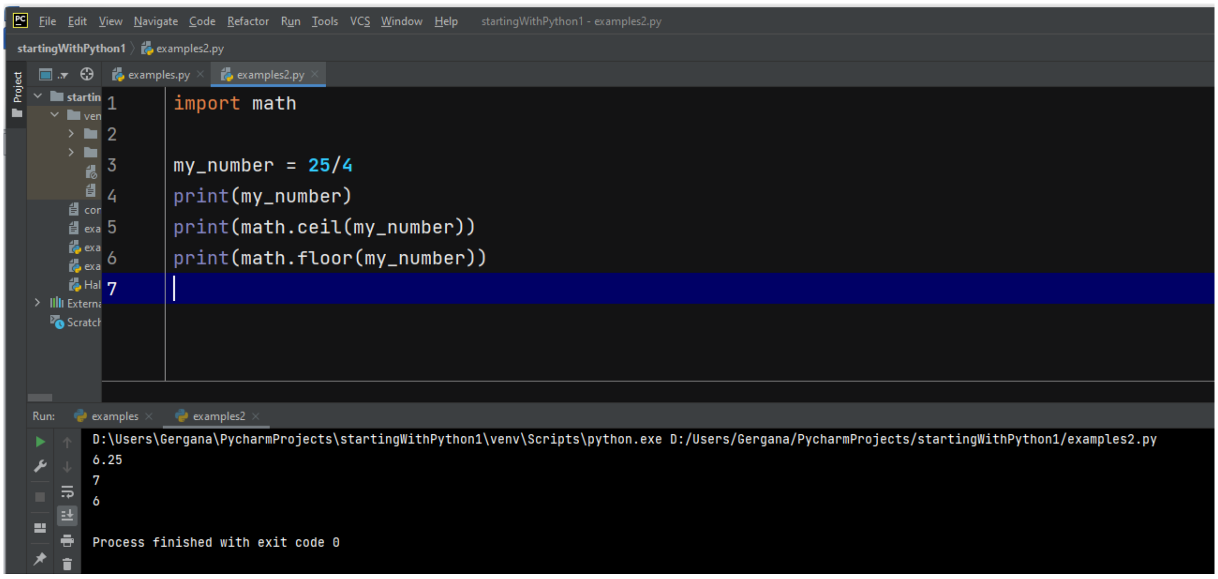
Importing libraries

Python has a lot of libraries that contain a myriad of useful complex functions that are already implemented and can simply be called and used in our programs. To do this, we need to only import them in our program. This is done in the beginning of the Python file by using the import keyword followed by the name of the library.

```
import math
```

This simple statement will load all the standard mathematical constants and functions available in the math library and we will be able to use them in our program.

In the example below we import the *math* library and we call two of its functions – ceil() and floor(). They are called by adding their name after math with a dot notation: math.ceil(*our number*) rounds our floating number *up* to the nearest integer, while math.floor(*our number*) rounds our floating number *down* to the nearest integer.

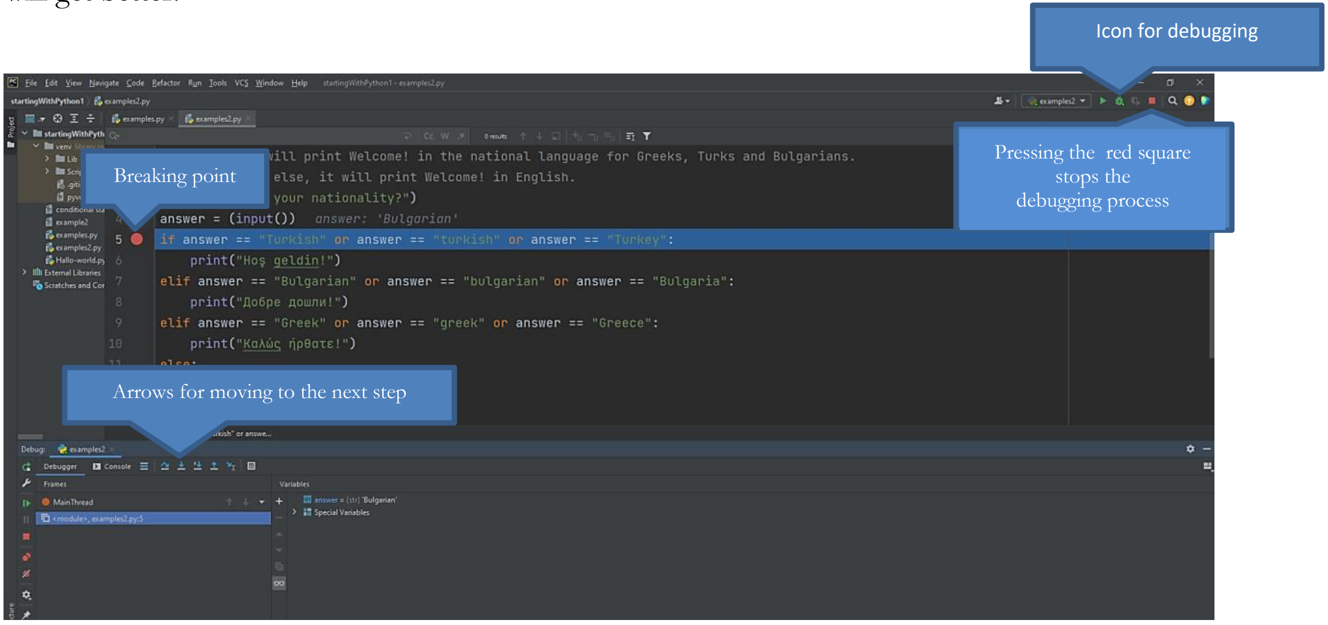


```
import math

my_number = 25/4
print(my_number)
print(math.ceil(my_number))
print(math.floor(my_number))
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
6.25
7
6

Process finished with exit code 0
```

Debugging

When writing complex programs, we often need to be able to trace the execution of the program step by step in order to find out where our algorithm is wrong. We can do this by restarting our program not with the green arrow but with the icon that looks like a bug. In order to debug the program, we need to specify a breaking point in the program, after which PyCharm will show us the step-by-step execution. If we do not specify the break point, the program will run as usually. We usually specify the breaking point in the place in the program right before we expect the mistake to be. The breaking point is placed by simply clicking on the grey space on the left of the program, as shown in the screenshot below. Once you start the debugging process, information will be shown in the console to help you understand what happens with the different variables and determine where the algorithm is wrong. You can move forward and backward in the consecutive steps with the arrows above this information. To observe step-by-step the execution of a separate code block, you need to choose "Step into" (it is very useful when you are debugging loops, which we will learn soon). To move to the next block, you need to choose "Step over". Learning how to effectively debug usually takes some time, but with practice, you will get better.


Icon for debugging
Breaking point
Pressing the red square stops the debugging process
Arrows for moving to the next step

```
will print Welcome! in the national language for Greeks, Turks and Bulgarians.
else, it will print Welcome! in English.
your nationality?")
answer = (input())   answer: 'Bulgarian'
if answer == "Turkish" or answer == "turkish" or answer == "Turkey":
    print("Hoş geldin!")
elif answer == "Bulgarian" or answer == "bulgarian" or answer == "Bulgaria":
    print("Добре дошли!")
elif answer == "Greek" or answer == "greek" or answer == "Greece":
    print("Καλώς ήρθατε!")
else:
```

```
answer = {str} 'Bulgarian'
```

Co-funded by the Erasmus+ Programme of the European Union

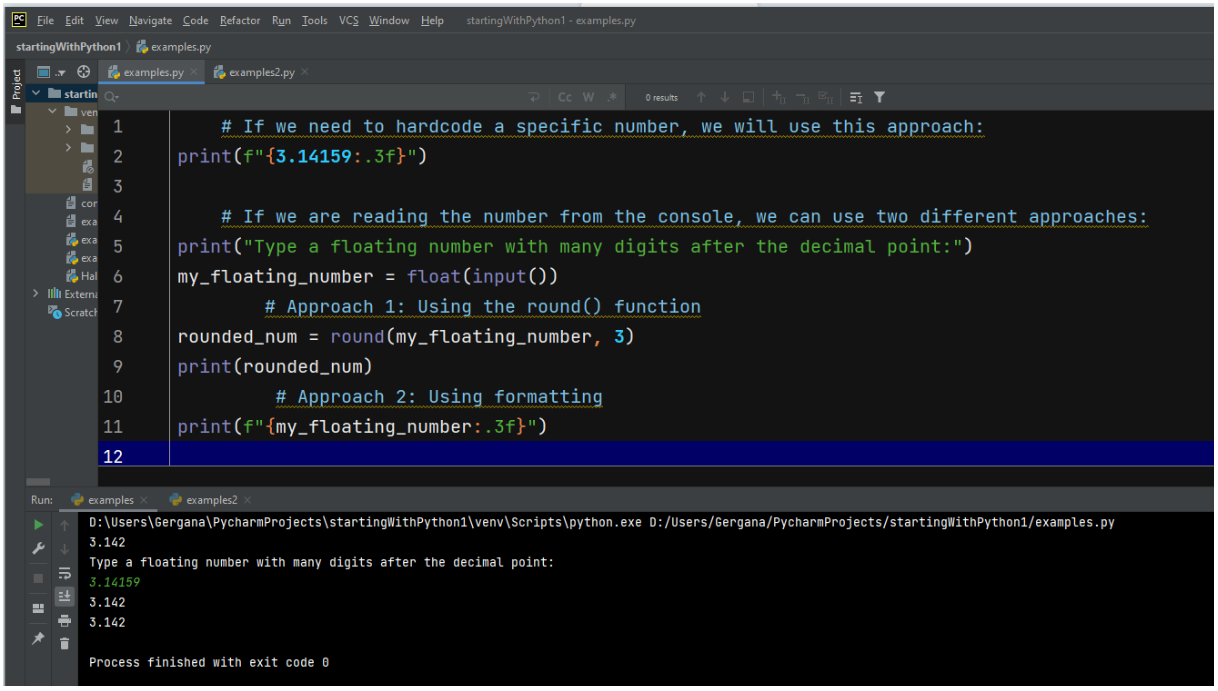
Working with numbers

Rounding numbers

Often in programs, we need to round numbers. We already learned about how we could round numbers up or down to the nearest integer. More often, however, we need to return the number with a specified number of decimal places (with a specific number of digits after the decimal point). In Python this can be achieved in two ways, and the choice depends on the program and what we do with the output. The first option is to use the round(*my number, number of decimal places*) function, in which we need to pass as arguments the floating number and the number of decimal places that we want to show. The second option is to use additional formatting during string interpolation. This is shown below:

```
f"{my_floating_number: .2f}"
```

In the above example, we can replace “2” with any number, depending on how many decimals we want to show. Below you can see an example of formatting a specified number, and an example of formatting a number passed in by a variable. Both approaches print the rounded *pi* number.



```
    # If we need to hardcode a specific number, we will use this approach:
print(f"{3.14159:.3f}")

    # If we are reading the number from the console, we can use two different approaches:
print("Type a floating number with many digits after the decimal point:")
my_floating_number = float(input())
        # Approach 1: Using the round() function
rounded_num = round(my_floating_number, 3)
print(rounded_num)
         # Approach 2: Using formatting
print(f"{my_floating_number:.3f}")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
3.142
Type a floating number with many digits after the decimal point:
3.14159
3.142
3.142

Process finished with exit code 0
```

Absolute value

Sometimes in programs we need to use the absolute value of numbers – the value without regard to the sign (i.e. the non-negative value of a number). This can be done using the abs(*my number*) function. In this way, abs(-5) will be equal to abs(5). Try it yourself in PyCharm.

Conditional statements

Conditional statements are at the heart of programming algorithms as they allow us to make decisions about how the program will proceed and what the program will do based on whether a certain condition is True or False.

Co-funded by the Erasmus+ Programme of the European Union

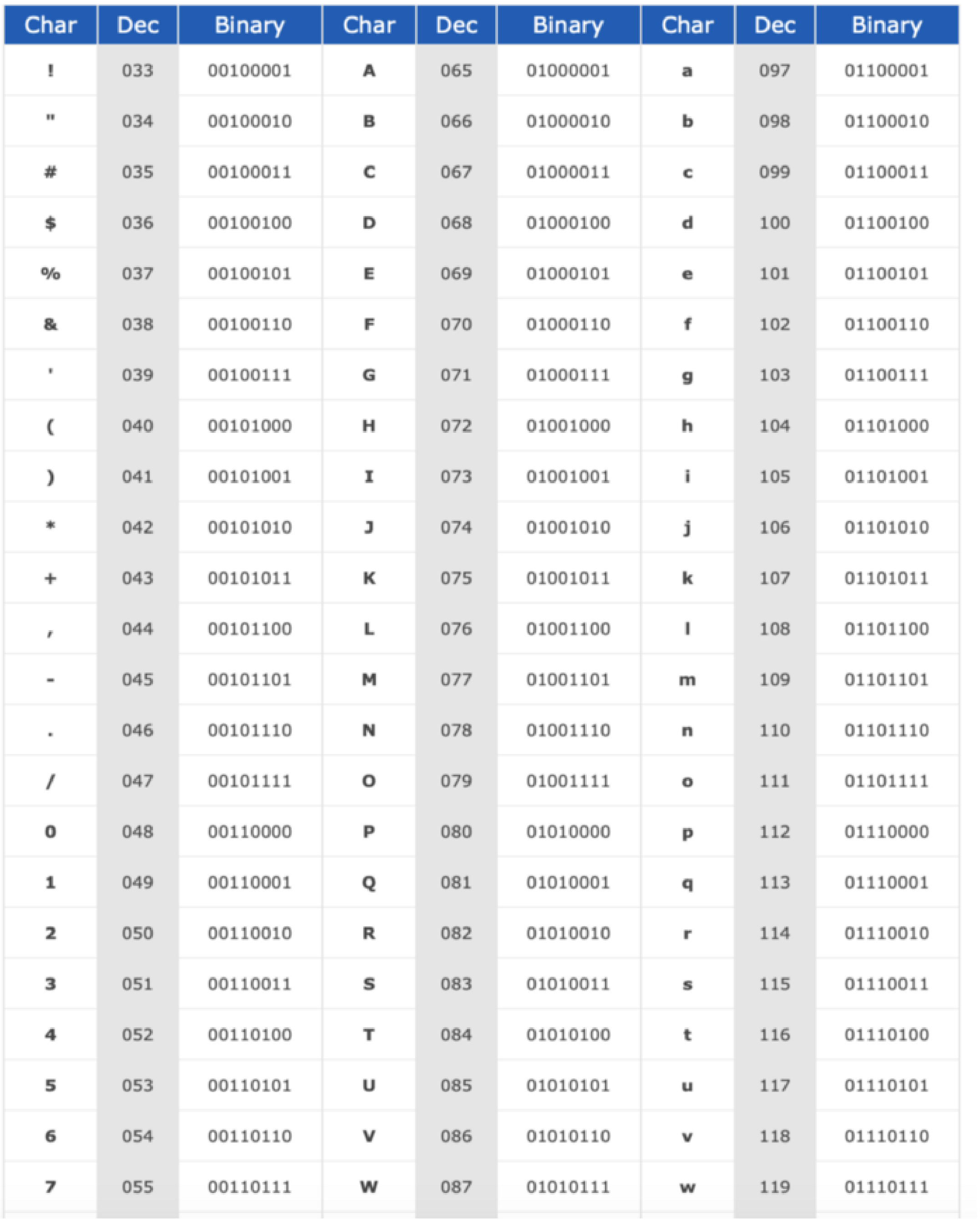
Checking if a condition is True or False

In order to decide if a condition is True or False, we have to perform some checks that compare values. The basic operators for such comparisons are:

Operator	Name	Explanation	Examples *Assume that:* *a=20* *b=40*
==	Equal	Returns True if two values are equal and False if they are different	print(a==b) #False print(2*a==b) #True
!=	Not equal	Returns True if two values are different and False if they are equal	print(a!=b) # True print(a!=2*b) # True print(2*a!=b) #False
>	Greater than	Returns True if the first value is greater than the second value	print(a>b) # False print(b>a) #True print(2*a>b) # False
>=	Greater than or equal to	Returns True if the first value is greater than or equal to the second value	print(a>=b) # False print(b>=a) #True print(2*a>=b) # True
<	Less than	Returns True if the first value is less than the second value	print(a<b) # True print(b<a) # False print(2*a<b) # False
<=	Less than or equal to	Returns True if the first value is less than or equal to the second value	print(a<=b) # True print(b<=a) # False print(2*a<=b) # True
in	Item present in a sequence	Evaluates if the specified item is in a given sequence	'o' **in** 'Jon' # True

Comparing numbers is similar to what we learn in basic Mathematics. However, it is also possible to use these operators to compare different type of values, such as dates and strings. String comparison in Python is done based on the characters in both strings, and the characters are compared one by one. When different characters are found, Python compares their ASCII values. To grasp this, you should be aware that a string is a sequence of characters (symbols). Computers, however, do not understand characters – they store and manipulate all data as 0s and 1s. Therefore, a character is identified by a binary number, as shown in the ASCII table below – for each character we have a specific number.

Co-funded by the Erasmus+ Programme of the European Union

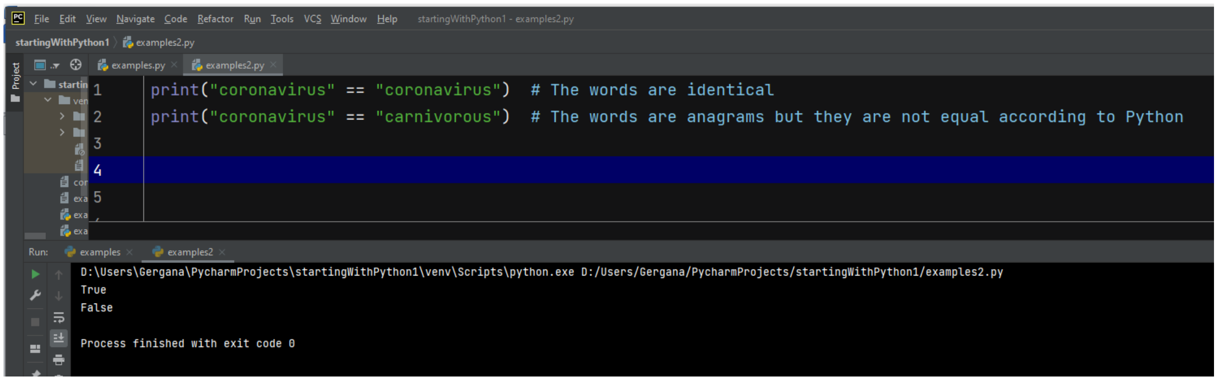


Char	Dec	Binary	Char	Dec	Binary	Char	Dec	Binary
!	033	00100001	A	065	01000001	a	097	01100001
"	034	00100010	B	066	01000010	b	098	01100010
#	035	00100011	C	067	01000011	c	099	01100011
$	036	00100100	D	068	01000100	d	100	01100100
%	037	00100101	E	069	01000101	e	101	01100101
&	038	00100110	F	070	01000110	f	102	01100110
'	039	00100111	G	071	01000111	g	103	01100111
(	040	00101000	H	072	01001000	h	104	01101000
)	041	00101001	I	073	01001001	i	105	01101001
*	042	00101010	J	074	01001010	j	106	01101010
+	043	00101011	K	075	01001011	k	107	01101011
,	044	00101100	L	076	01001100	l	108	01101100
-	045	00101101	M	077	01001101	m	109	01101101
.	046	00101110	N	078	01001110	n	110	01101110
/	047	00101111	O	079	01001111	o	111	01101111
0	048	00110000	P	080	01010000	p	112	01110000
1	049	00110001	Q	081	01010001	q	113	01110001
2	050	00110010	R	082	01010010	r	114	01110010
3	051	00110011	S	083	01010011	s	115	01110011
4	052	00110100	T	084	01010100	t	116	01110100
5	053	00110101	U	085	01010101	u	117	01110101
6	054	00110110	V	086	01010110	v	118	01110110
7	055	00110111	W	087	01010111	w	119	01110111

Source: https://towardsdatascience.com/processing-text-with-unicode-in-python-eacc226886cb

The comparison using the == and the != operators is quite intuitive. Two strings will be equal only if all their characters are identical and are identically positioned. They will be different if there is a single different character or the positioning of the characters is different. The comparison is case sensitive because capital letters have different ASCII values than small letters. Anagrams - words containing the same characters but in a different order will not be equal because characters are compared one by one and positioning matters. See some examples below.

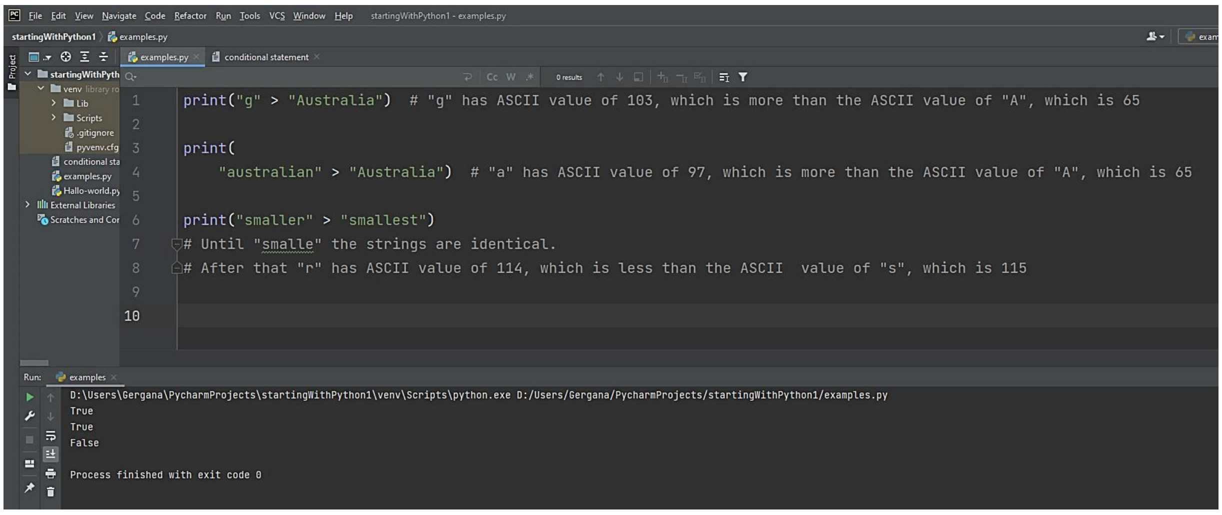
Co-funded by the
Erasmus+ Programme
of the European Union



```
print("coronavirus" == "coronavirus")  # The words are identical
print("coronavirus" == "carnivorous")  # The words are anagrams but they are not equal according to Python
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
True
False

Process finished with exit code 0
```

When using the >, <, >= and <= operators, the character with lower ASCII value is considered to be smaller. Python will compare the ASCII values of the first diverging characters in the two strings, and this value will decide which string is greater than the other. The values of the characters after that make no difference. See the examples below.



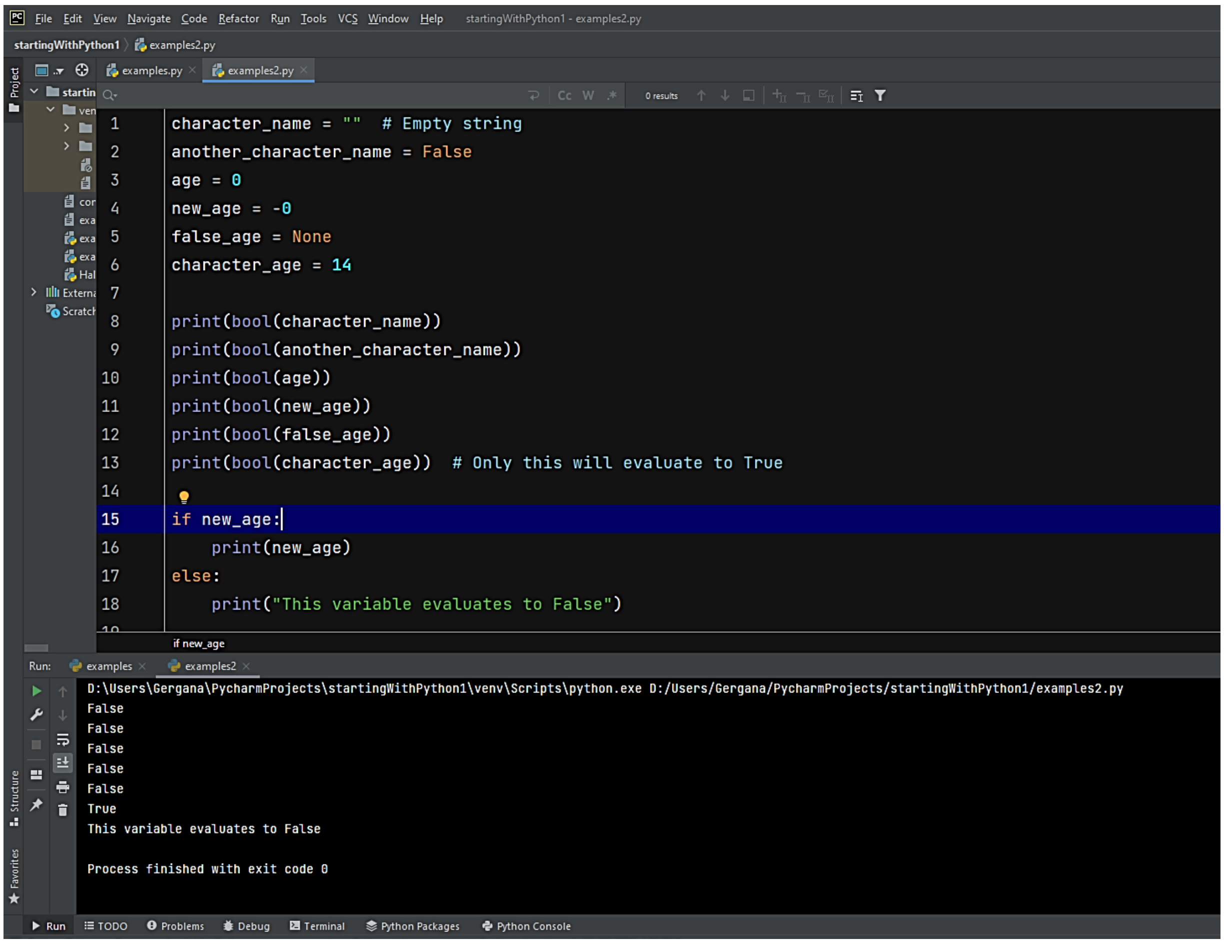
```
print("g" > "Australia")  # "g" has ASCII value of 103, which is more than the ASCII value of "A", which is 65

print(
    "australian" > "Australia")  # "a" has ASCII value of 97, which is more than the ASCII value of "A", which is 65

print("smaller" > "smallest")
# Until "smalle" the strings are identical.
# After that "r" has ASCII value of 114, which is less than the ASCII  value of "s", which is 115
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
True
True
False

Process finished with exit code 0
```

Tip on True and False values of variables:

We can also evaluate a variable itself to True or False. If the variable has a value, it will always evaluate as True. If it is declared as None, 0, -0, "", ' ' or False, then it will evaluate as False. Thus we can use a variable to obtain a Boolean or we can use it in conditional statements (we will learn about conditional statements in the very next section, so if you do not understand this, go back to it. However, conditional statements are very similar in all programming languages). See below.

Co-funded by the Erasmus+ Programme of the European Union

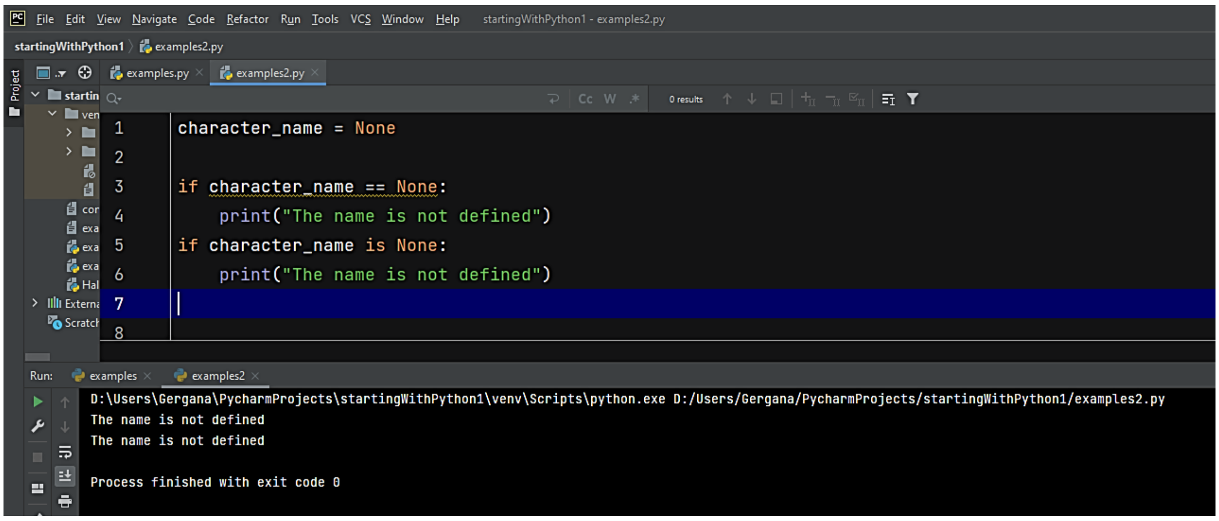


```
character_name = ""  # Empty string
another_character_name = False
age = 0
new_age = -0
false_age = None
character_age = 14

print(bool(character_name))
print(bool(another_character_name))
print(bool(age))
print(bool(new_age))
print(bool(false_age))
print(bool(character_age))  # Only this will evaluate to True

if new_age:
    print(new_age)
else:
    print("This variable evaluates to False")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
False
False
False
False
False
True
This variable evaluates to False

Process finished with exit code 0
```

Tip on the None keyword:

None is keyword, which defines a null value or no value. We can assign None to a variable. In a conditional statement we can also check if a variable is None. To do this we compare the variable and None either with the = operator or with the keyword is. The preferred syntax is the is keyword.



```
character_name = None

if character_name == None:
    print("The name is not defined")
if character_name is None:
    print("The name is not defined")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
The name is not defined
The name is not defined

Process finished with exit code 0
```

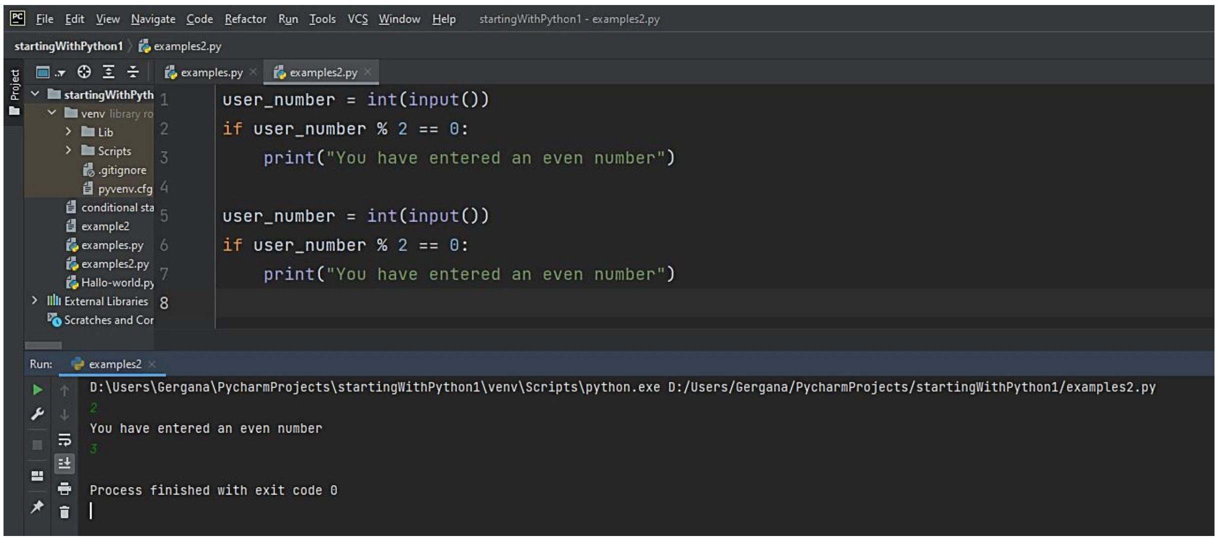
Using conditional statements

Once we are able to decide if a condition is True or False, we can construct detailed programming logic based on simple algorithms. To do this, we use of the following:

Co-funded by the Erasmus+ Programme of the European Union

- if statement
- if-else statement
- if-elif-else ladder
- nested if-else statements

When we use the if statement, we check the validity of the condition and on the basis of this we decide whether a block of code will be executed. Let us say that we want the user to enter a number and we notify them if the number is even. We can do this by using the modulus division by 2. An even number would return no remainder, i.e. the result will be 0. Bellow you can see the detailed syntax. Note that we use if and pass in it a statement that checks if a condition is True or False. After this statement, we add colon and, on a new line, with an indent, we write the bloc of code that we want to execute if the condition is True. If the condition is False, the code specified after the colon will not be executed. In the example below, we first enter an even number on the console, and we get the message. Then we enter an odd number, and we get no message.

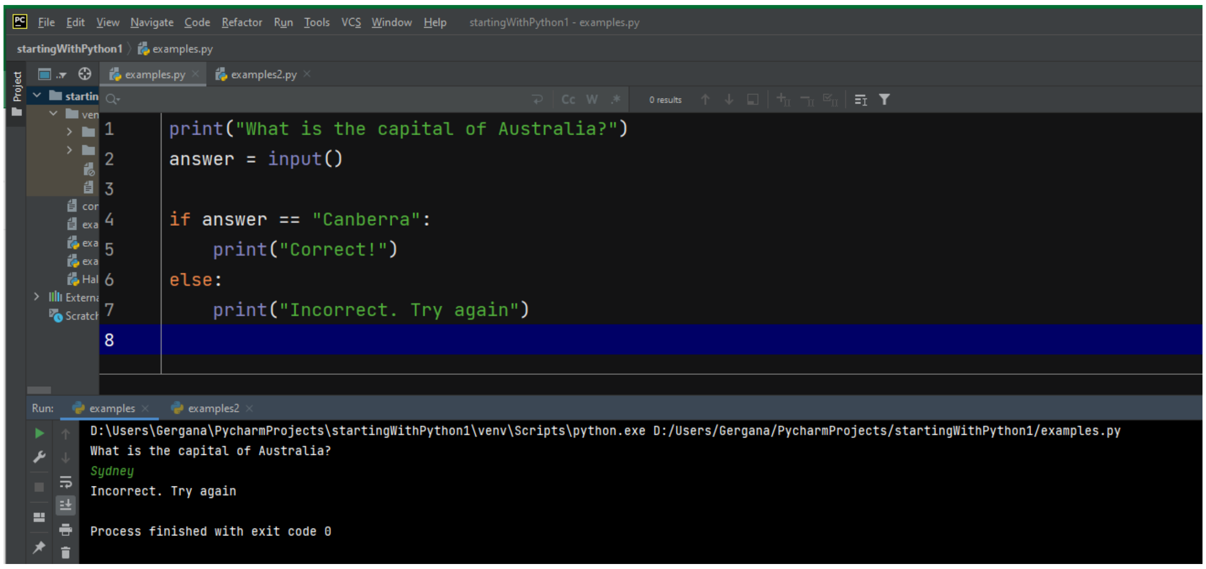


```python
user_number = int(input())
if user_number % 2 == 0:
    print("You have entered an even number")

user_number = int(input())
if user_number % 2 == 0:
    print("You have entered an even number")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
2
You have entered an even number
3

Process finished with exit code 0
```

More often than not, however, when a condition is not True, we would like to execute a different code block. If this is indeed the case, we should use the if-else statement. The code block after the else statement will execute if the condition is False. In the example below, let us create a simple quiz game where the user is asked to input the capital of Australia. If the answer is correct, we will show “Correct!”. If not, we will show “Incorrect. Try again”. Try the correct answer yourself. Below we show what happens if we give an incorrect answer.

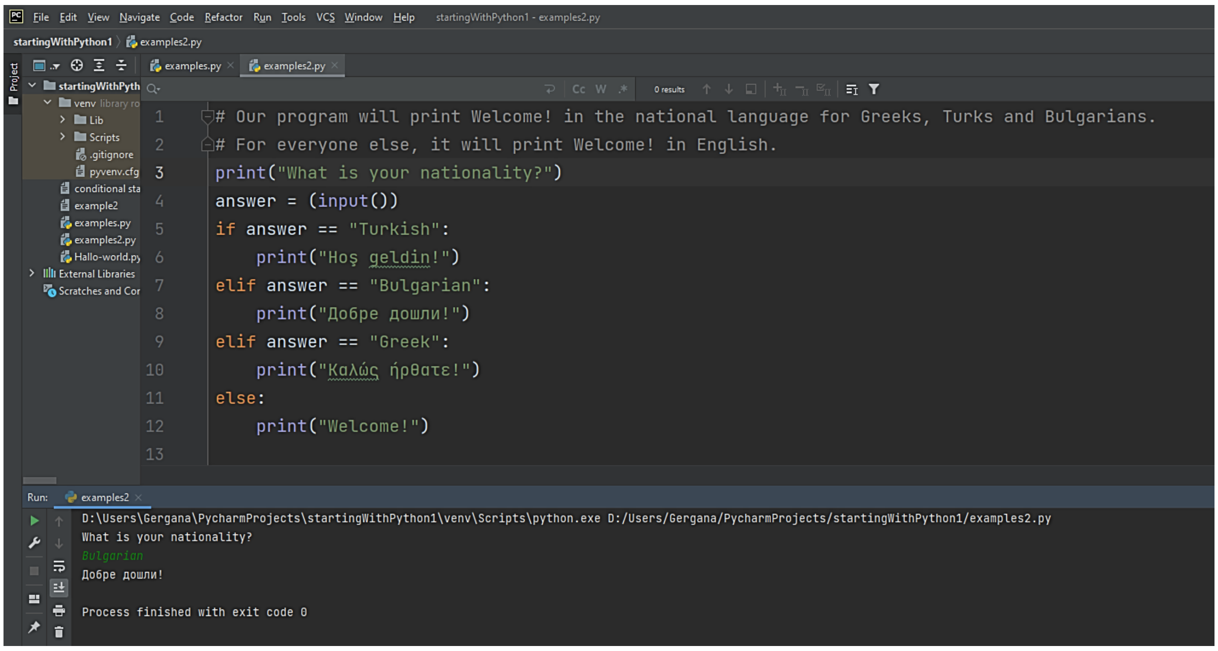
Co-funded by the Erasmus+ Programme of the European Union



```
print("What is the capital of Australia?")
answer = input()

if answer == "Canberra":
    print("Correct!")
else:
    print("Incorrect. Try again")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
What is the capital of Australia?
Sydney
Incorrect. Try again

Process finished with exit code 0
```

Of course, in more complex programs, we will have more complicated logic and we will need to execute different code based on different input. In these cases, we have to use the if-elif-else ladder. The elif statement allows us to add as many alternative conditions as we need. When one of them is True, a specific code is executed and the conditions in following elif statements will not be checked. Then again, if none of the elif conditions are true, we specify what code will be executed using the else statement. Let us say that we want some text to be shown only to people with a specific nationality – we will print "Welcome!" in different languages based on the nationality of the user. See below how this logic is executed.

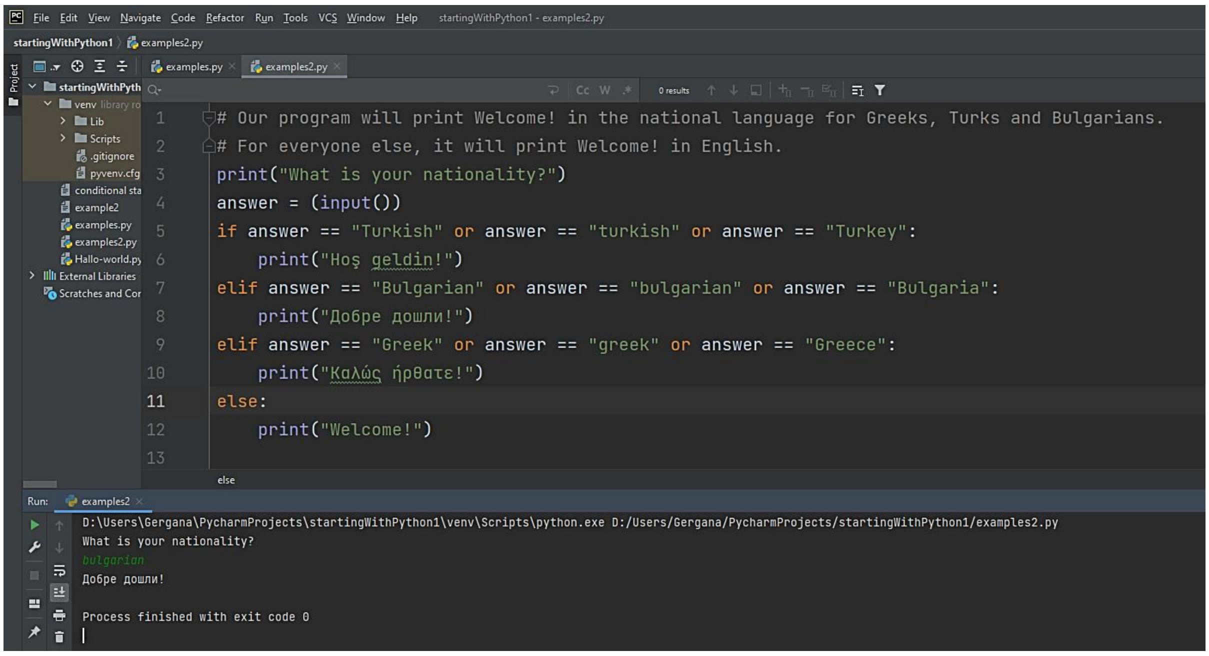


```
# Our program will print Welcome! in the national language for Greeks, Turks and Bulgarians.
# For everyone else, it will print Welcome! in English.
print("What is your nationality?")
answer = (input())
if answer == "Turkish":
    print("Hoş geldin!")
elif answer == "Bulgarian":
    print("Добре дошли!")
elif answer == "Greek":
    print("Καλώς ήρθατε!")
else:
    print("Welcome!")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
What is your nationality?
Bulgarian
Добре дошли!

Process finished with exit code 0
```

We can fine-tune the program to recognize the nationality even if someone writes it in small letters only or writes the name of the country instead. To do this, we use the or operator when evaluating whether the condition is True or False. We can allow the condition to be evaluated as True in different cases. In our example, we specify that the program will write *Welcome!* in Bulgarian if the user enters either "Bulgarian" or "bulgarian" or "Bulgaria".

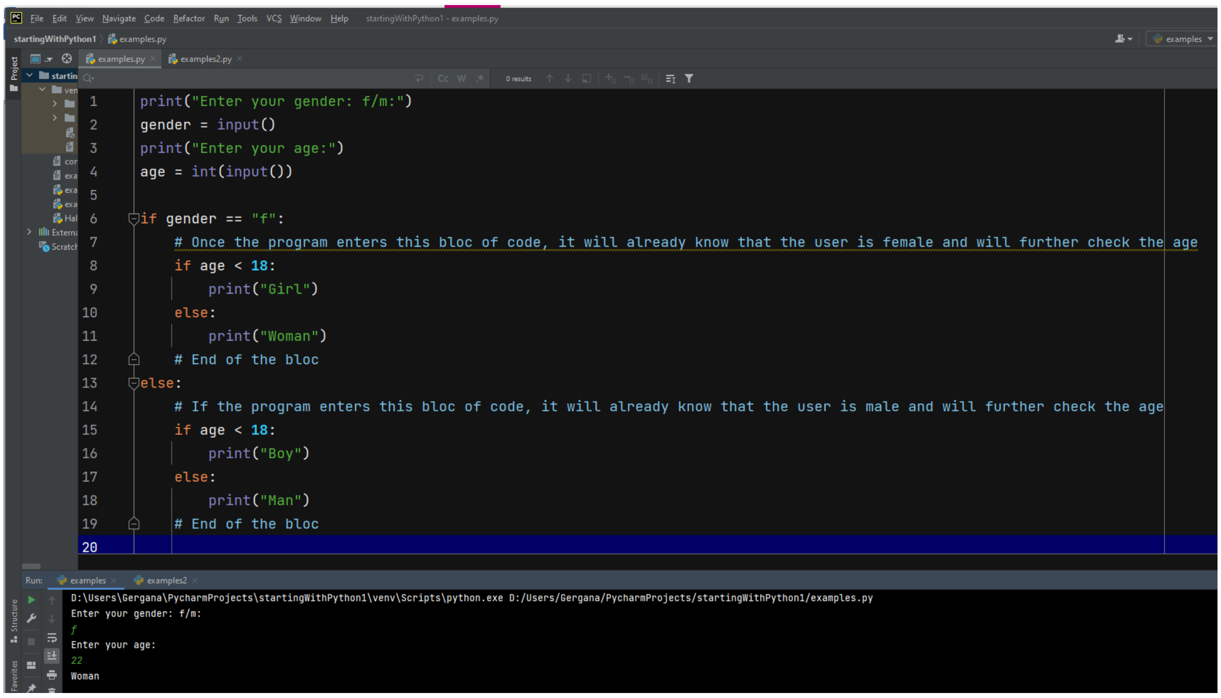
Co-funded by the Erasmus+ Programme of the European Union



```python
# Our program will print Welcome! in the national language for Greeks, Turks and Bulgarians.
# For everyone else, it will print Welcome! in English.
print("What is your nationality?")
answer = (input())
if answer == "Turkish" or answer == "turkish" or answer == "Turkey":
    print("Hoş geldin!")
elif answer == "Bulgarian" or answer == "bulgarian" or answer == "Bulgaria":
    print("Добре дошли!")
elif answer == "Greek" or answer == "greek" or answer == "Greece":
    print("Καλώς ήρθατε!")
else:
    print("Welcome!")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
What is your nationality?
bulgarian
Добре дошли!

Process finished with exit code 0
```

In some cases, we need to develop even more complex algorithms that make decisions at several levels. To achieve such complexity, we need to use nested if-else statements. Let us say we want the user to enter their gender and their age, and on that basis we will decide if they are “boy”, “girl”, “woman”, or “man”. We will first check if the user is female or male, and then within each option, we will further check the age. See the code below to understand how *if-else* statements are nested and how the program decides which code to execute.

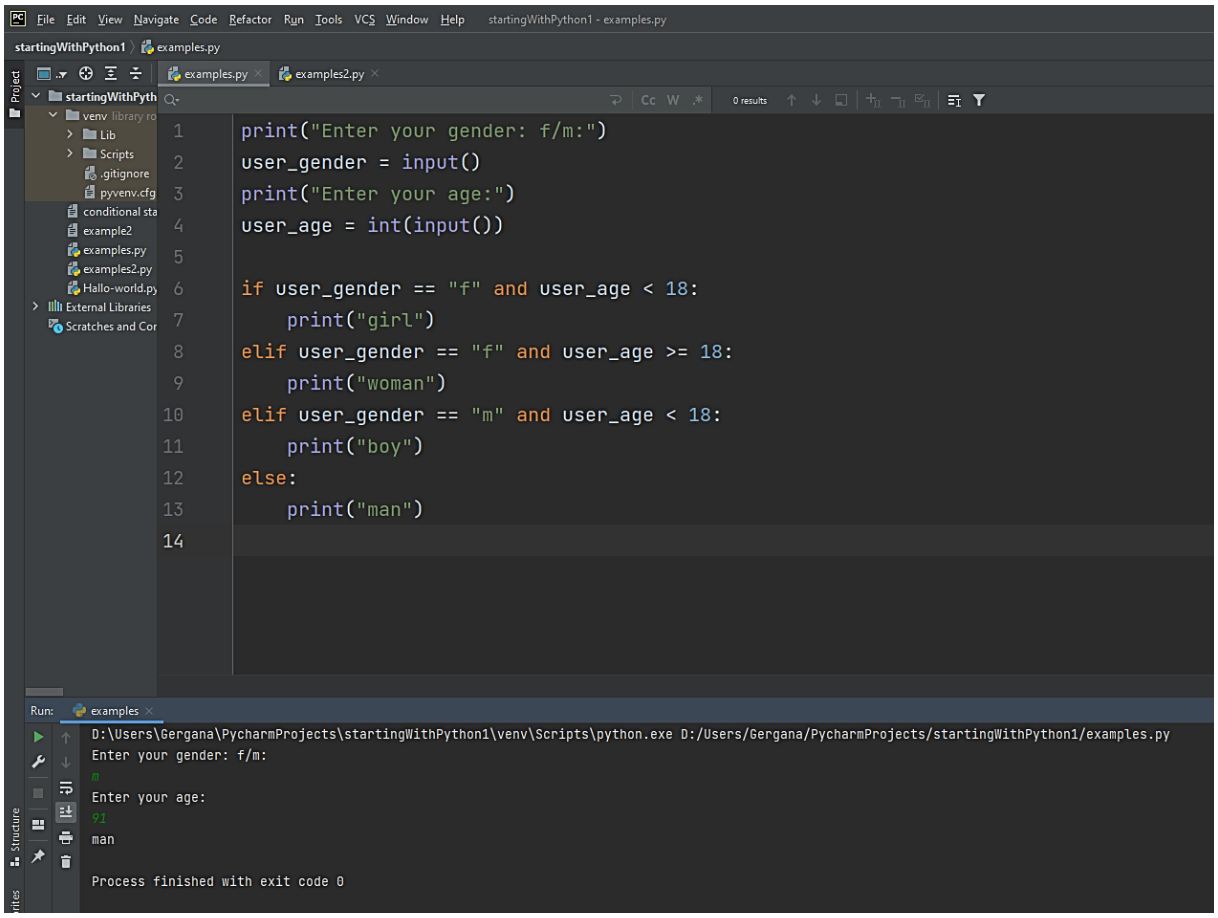


```python
print("Enter your gender: f/m:")
gender = input()
print("Enter your age:")
age = int(input())

if gender == "f":
    # Once the program enters this bloc of code, it will already know that the user is female and will further check the age
    if age < 18:
        print("Girl")
    else:
        print("Woman")
    # End of the bloc
else:
    # If the program enters this bloc of code, it will already know that the user is male and will further check the age
    if age < 18:
        print("Boy")
    else:
        print("Man")
# End of the bloc
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Enter your gender: f/m:
f
Enter your age:
22
Woman
```

Of course, we can also achieve the same without nested if-else statements, by using if-elif statement and combining the conditions with the and operator. We are showing this approach in the code below. Choosing one or the other option is entirely up to your personal preferences.

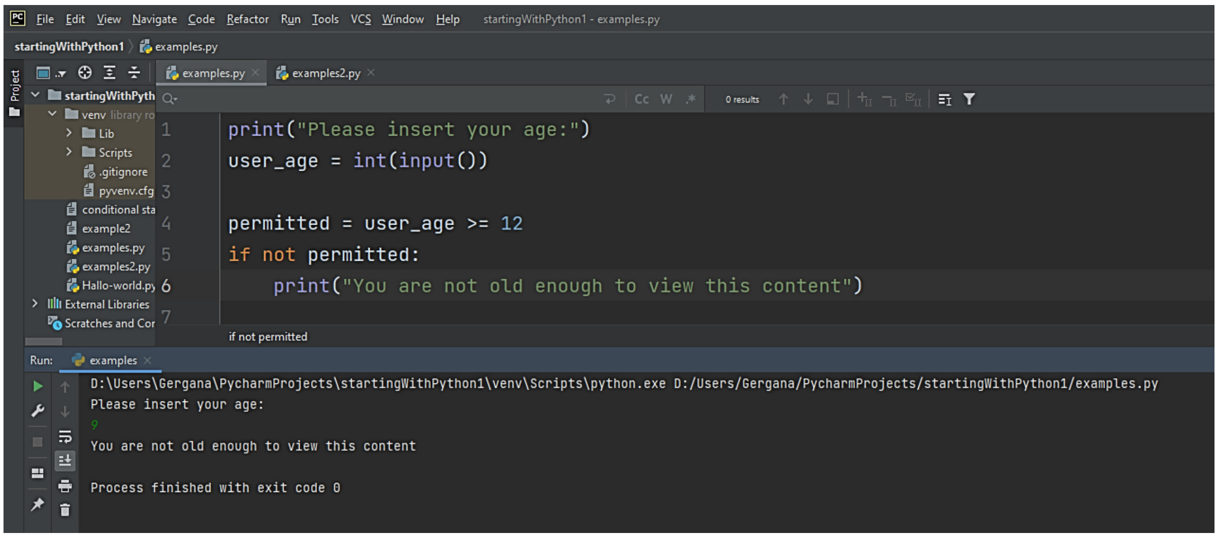
Co-funded by the Erasmus+ Programme of the European Union



```
print("Enter your gender: f/m:")
user_gender = input()
print("Enter your age:")
user_age = int(input())

if user_gender == "f" and user_age < 18:
    print("girl")
elif user_gender == "f" and user_age >= 18:
    print("woman")
elif user_gender == "m" and user_age < 18:
    print("boy")
else:
    print("man")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Enter your gender: f/m:
m
Enter your age:
91
man

Process finished with exit code 0
```

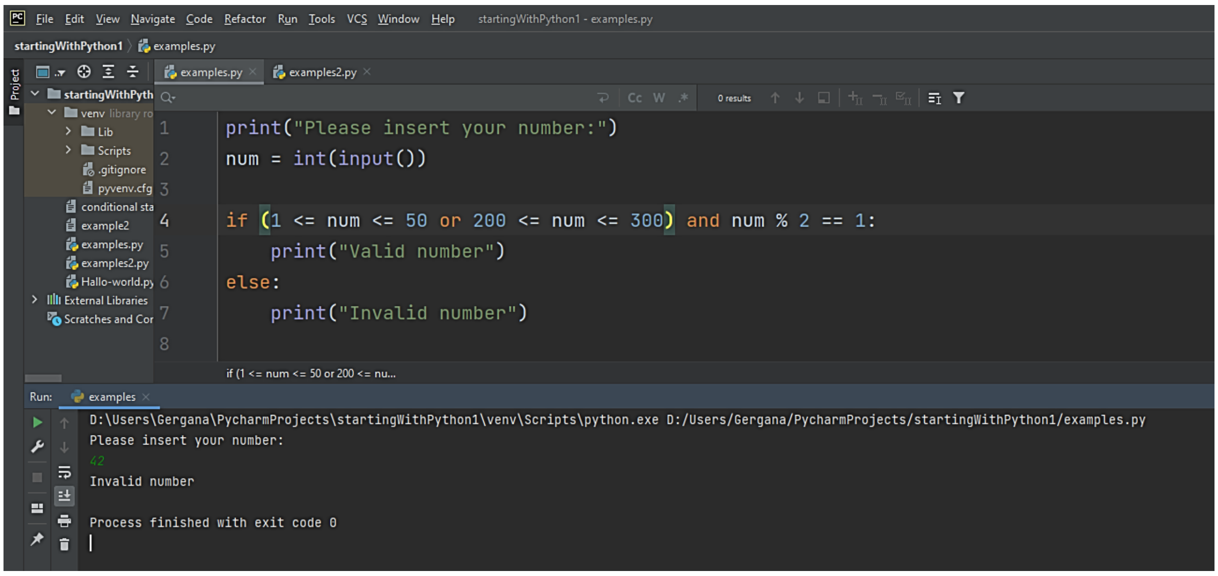
In any case, you should learn to use the logical operators and, or and not. The and operator returns True if both conditions are True. The or operator returns True if at least one of the conditions is correct. The not operator will return True if the condition is False. To see an example of the use of the or operator, see the previous example about writing "Welcome!" in different languages. Let us also give an example of the not operator. Let us write a program which will ask for the users' age and will not allow them to proceed using the program unless they are at least 12 years old.



```
print("Please insert your age:")
user_age = int(input())

permitted = user_age >= 12
if not permitted:
    print("You are not old enough to view this content")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Please insert your age:
9
You are not old enough to view this content

Process finished with exit code 0
```

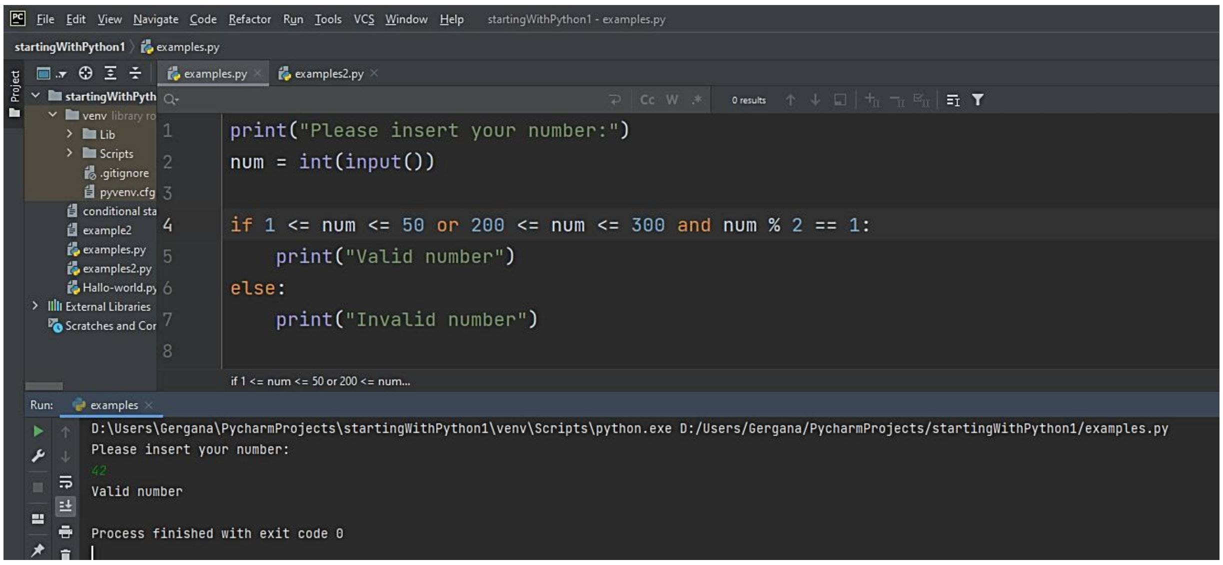
In some cases, we would need a very complex logic when some conditions need to be combined by using multiple operators and, or and not operators, and – when necessary for the logic to be executed properly – encapsulated in parentheses () to define the order in which the evaluation should take place. In the example below, we want to check if a number is valid. We define a valid number as an odd number that is either between 1 and 50 or between 200 and 300. To write this code, we first check if a) a number is between 1 and 50 or between 200 and 300, and then we check if b) it is an odd number (for an odd number modulus division by 2 will return 1). The first condition itself includes two conditions, so we will encapsulate them into parentheses before the use the and operator.



```python
print("Please insert your number:")
num = int(input())

if (1 <= num <= 50 or 200 <= num <= 300) and num % 2 == 1:
    print("Valid number")
else:
    print("Invalid number")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Please insert your number:
42
Invalid number

Process finished with exit code 0
```

Now notice that if you omit the parentheses, the result for the same number (42), will be wrong (see the code below). This is because Python will not understand the logic of the comparison and will evaluate 42 as valid just because it falls in the 1-50 range. It will only check if the number is odd if it is in the 200-300 range. Parentheses are often necessary for logic to be executed correctly and they generally improve the readability of more complex code (even if they are not necessary).



```python
print("Please insert your number:")
num = int(input())

if 1 <= num <= 50 or 200 <= num <= 300 and num % 2 == 1:
    print("Valid number")
else:
    print("Invalid number")
```

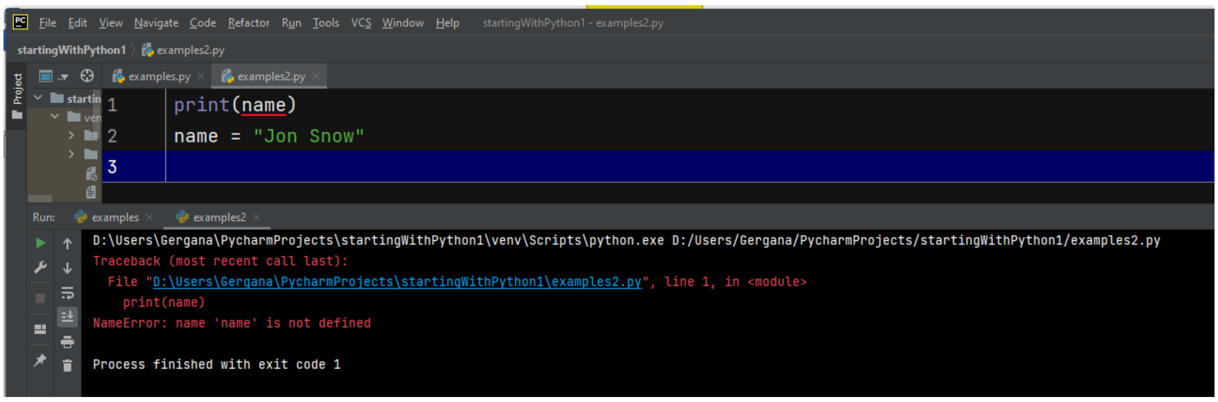
```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Please insert your number:
42
Valid number

Process finished with exit code 0
```

You are now ready to test yourself in HackerRank. Solve the Python If-Else Challenge (make sure you have enabled the "Easy" difficulty level).

Co-funded by the Erasmus+ Programme of the European Union

Initialization and lifetime of variables in Python; global vs. local variables

Variables only exist if they have been initialized somewhere in the program. In Python variables can be declared / initialized *outside a function* or *inside a function*. This determines if they have a global or a local scope. The variables that are declared outside a function (in the main program) are *global* variables and are accessible from anywhere in the program. You should make sure that the variables are declared *before* they are used. If you try to use a variable that is declared later in the program, you will get an error and the program will be terminated.

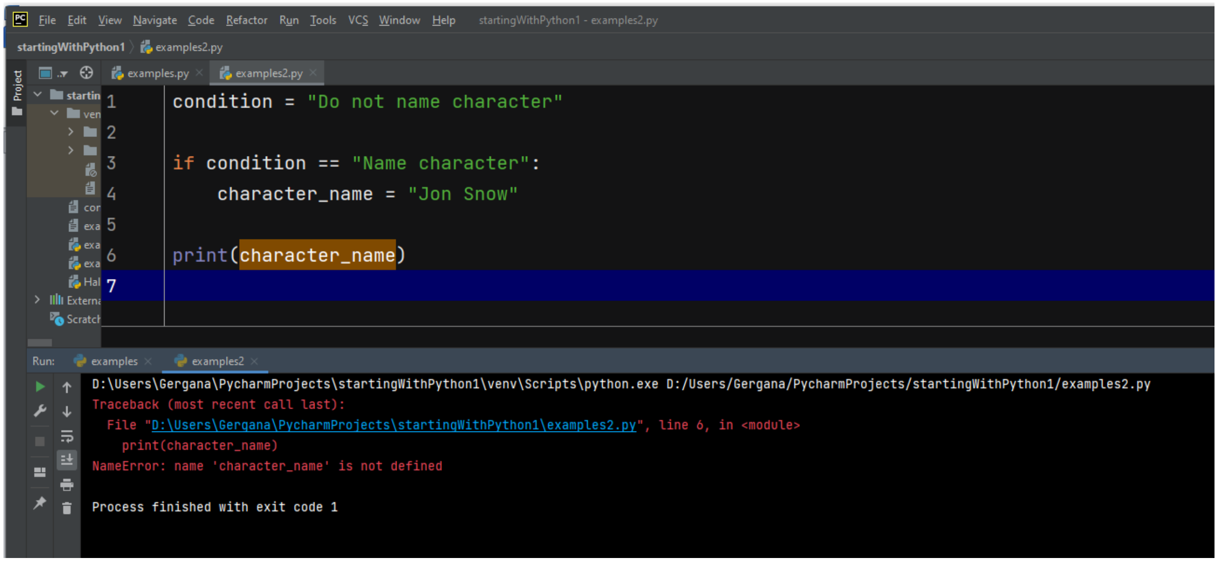


```
print(name)
name = "Jon Snow"
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Traceback (most recent call last):
  File "D:\Users\Gergana\PycharmProjects\startingWithPython1\examples2.py", line 1, in <module>
    print(name)
NameError: name 'name' is not defined

Process finished with exit code 1
```

Global variables can be changed within a particular function, but in order to do this, you need to define them with the keyword *global*.

The variables that are declared *within a function* are *local* variables. They can be accessed and used only within the function and they are erased from the memory (they cease to exist) after the function is executed. This is valid also for variables that are defined within conditional statements. In the example below, we cannot print name variable, because it will be defined only if we execute the if statement, which we do not (because the condition is False); hence, the variable name is never initialized. The IDE warns you about the possibility that your variable is not initialized by marking it in orange.



```
condition = "Do not name character"

if condition == "Name character":
    character_name = "Jon Snow"

print(character_name)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Traceback (most recent call last):
  File "D:\Users\Gergana\PycharmProjects\startingWithPython1\examples2.py", line 6, in <module>
    print(character_name)
NameError: name 'character_name' is not defined

Process finished with exit code 1
```

Loops in Python

In programs it is very often necessary to repeat the same code over and over again, with different input. For example, we may have a list of names and we may need to go through all of them to see if any of them is "Jon Snow". Instead of writing the same code 5 or 5000 times, we will use a *loop*. A loop forces the program to run code encapsulated in it as many times as we need.

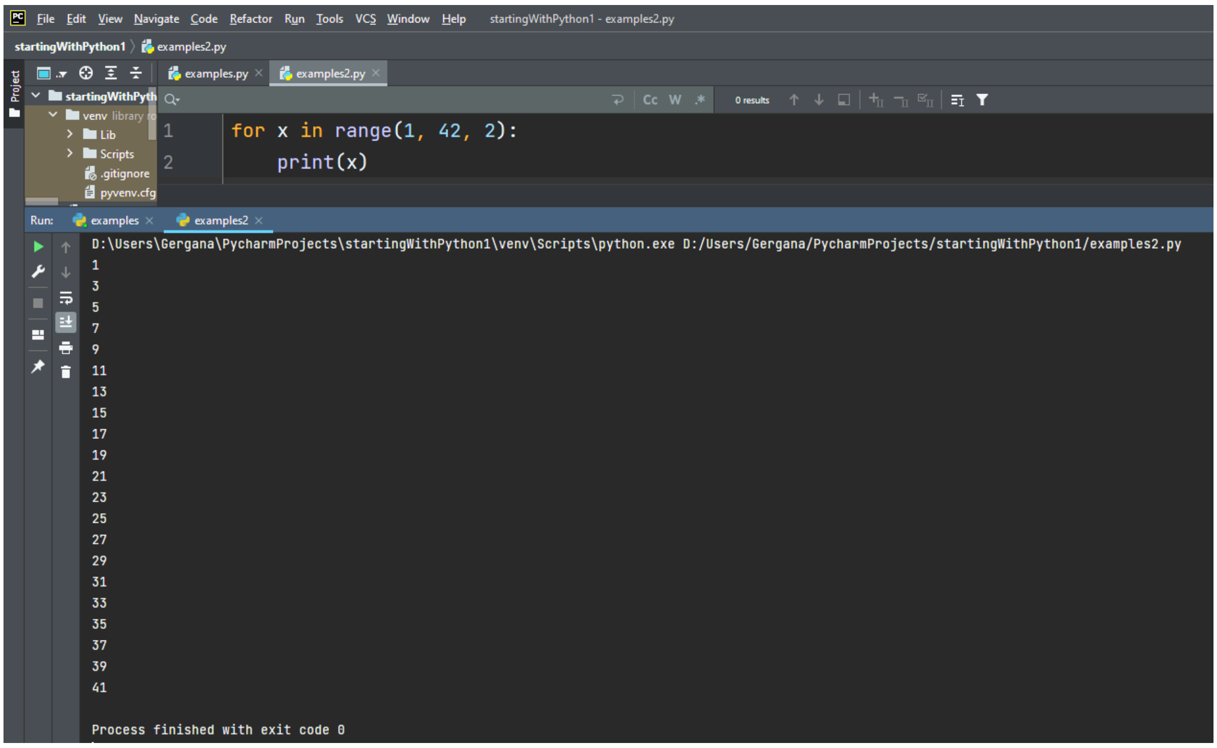
For loop

A *for* loop is the first major type of loop. It is constructed by using a *range*, which specifies how many times the code should be re-run (iterated). An example is:

```
for x in range(1, 43):
    print(x)
```

This code will print the numbers from 1 to 42 on a new line, without us needing to write print 42 times. Notice that the end of the range is excluded, so for the range 1-43, the last number that we will print is 42. Try it yourself in PyCharm.

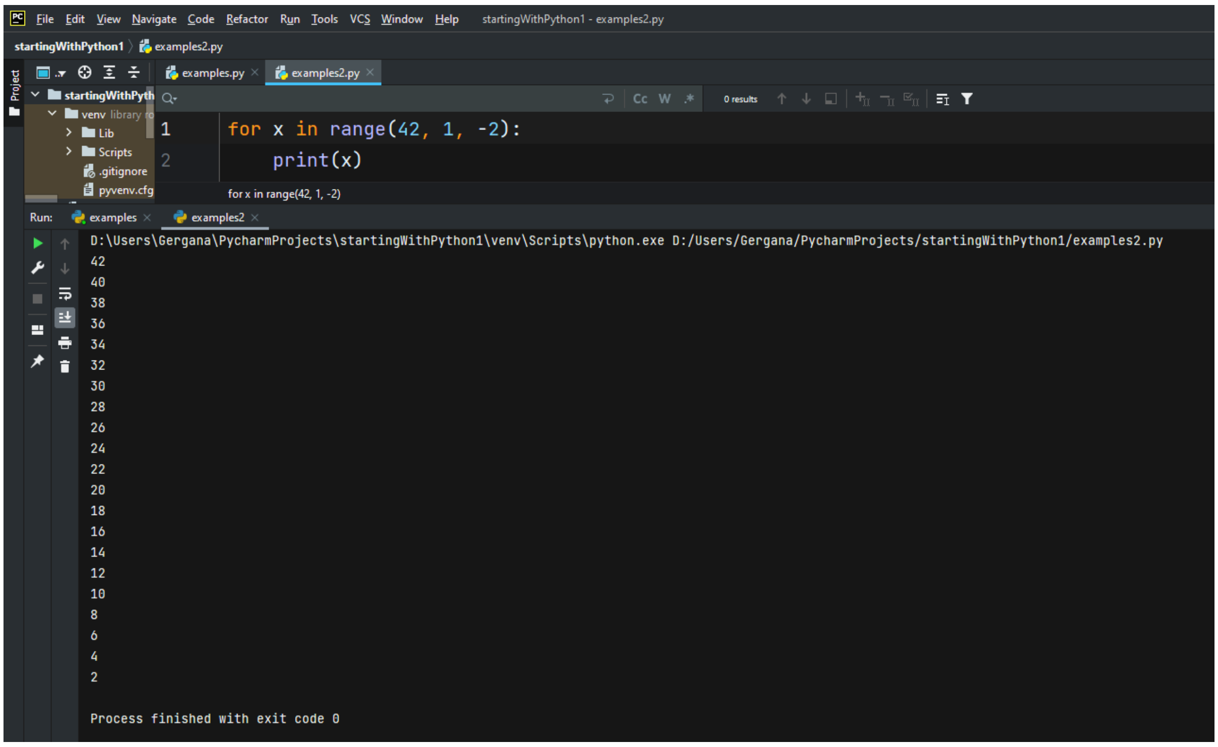
We can also specify a step for the iteration. If we want to only print every second number in the 1 - 42 range (i.e. only the odd numbers), we will add the *step* of 2 when constructing the loop. We add it after a coma following the range. See this code below.


File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples2.py
startingWithPython1 examples2.py
examples.py examples2.py
venv library
Lib
Scripts
.gitignore
pyvenv.cfg
Run: examples examples2

```
for x in range(1, 42, 2):
    print(x)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
31
33
35
37
39
41

Process finished with exit code 0
```

If we are iterating backwards, we could add a negative step of -1 or any negative step. See such a loop below, with a negative step of -2.

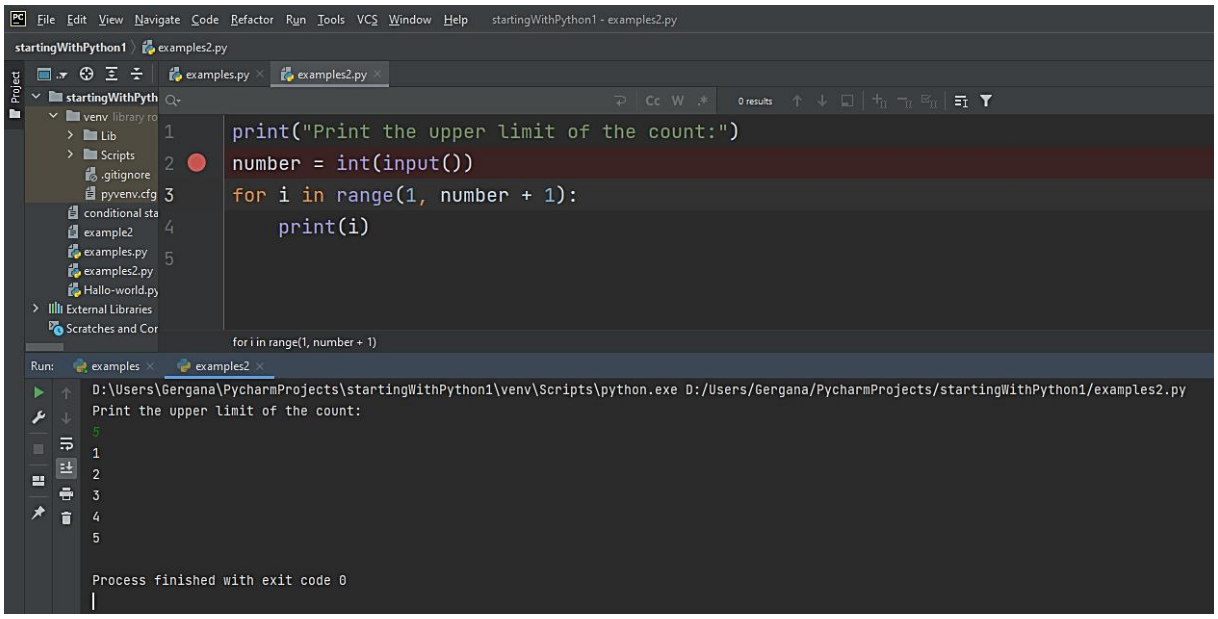

```python
for x in range(42, 1, -2):
    print(x)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
42
40
38
36
34
32
30
28
26
24
22
20
18
16
14
12
10
8
6
4
2

Process finished with exit code 0
```

More often than not, we will not know how many times we need to iterate. The number of iterations will be given by a variable in our program or by the user. It is therefore possible to define the range of the loop with a variable that will be changed or passed in by the program. In the example below we print the numbers from 1 to the number that the user types on the console. Notice that the upper limit of the range is the number of the user + 1, because the end number is excluded.

Tip on conventional naming:

The typical letter that we use for iteration in a single loop is *i* rather than the *x* that we used in our examples so far. However, the program will work with any letter or word that you choose.


```python
print("Print the upper limit of the count:")
number = int(input())
for i in range(1, number + 1):
    print(i)
```

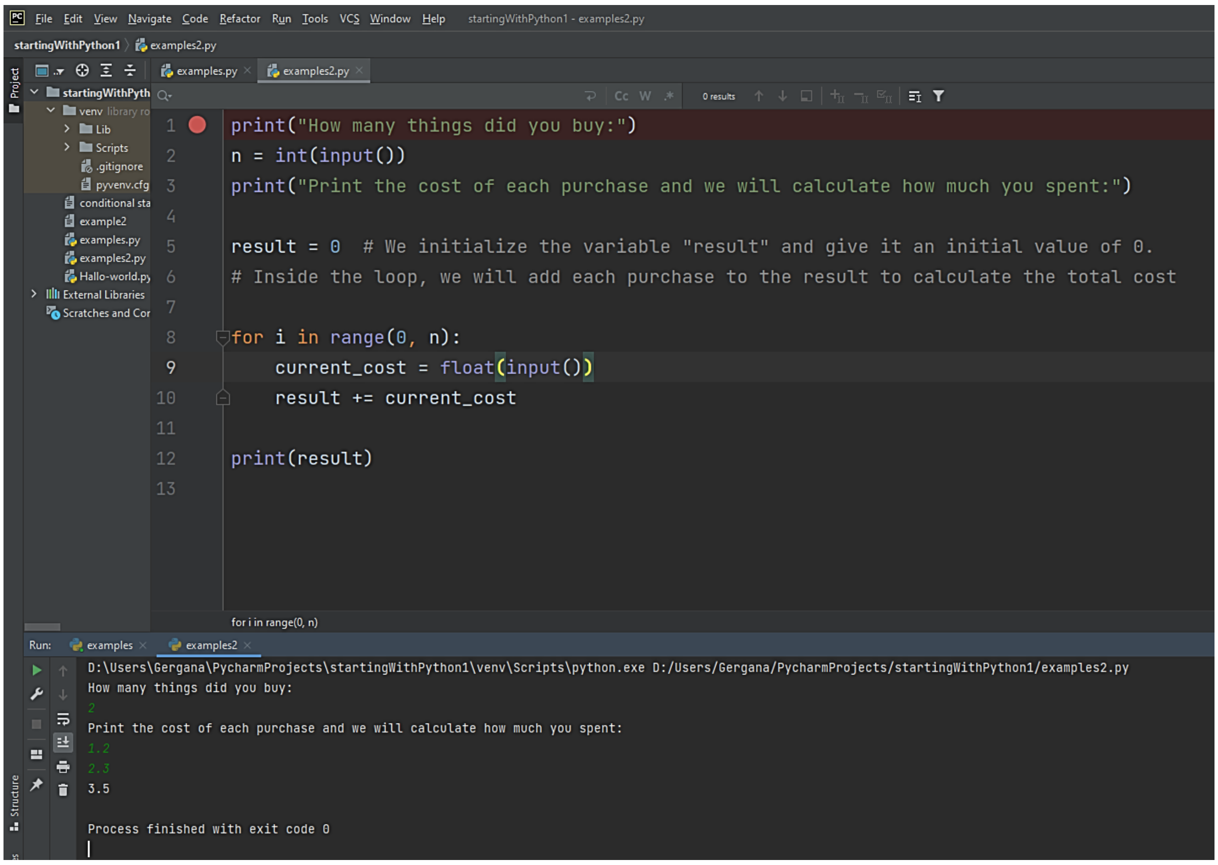
```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Print the upper limit of the count:
5
1
2
3
4
5

Process finished with exit code 0
```

Co-funded by the Erasmus+ Programme of the European Union

The best thing about *for* loops (and indeed, about any kind of loops) is that we can write very complex code inside them. This allows us to perform very complex logic multiple times, but with very little code to write.

Examples of using for loops

Let us let the user calculate their expenses by writing them one after the other. We will use a *for* loop to sum all the expenses and we will provide the total cost. Do not forget to initialize the variable representing the total cost outside of the loop in order to be able to print it later on. Notice that we started the loop at 0, instead of 1 (this is more conventional), so the end of the loop range will be n, and not n+1. Both range(1, n+1) and range(0, n) will have the same result.



```python
print("How many things did you buy:")
n = int(input())
print("Print the cost of each purchase and we will calculate how much you spent:")

result = 0  # We initialize the variable "result" and give it an initial value of 0.
# Inside the loop, we will add each purchase to the result to calculate the total cost

for i in range(0, n):
    current_cost = float(input())
    result += current_cost

print(result)
```

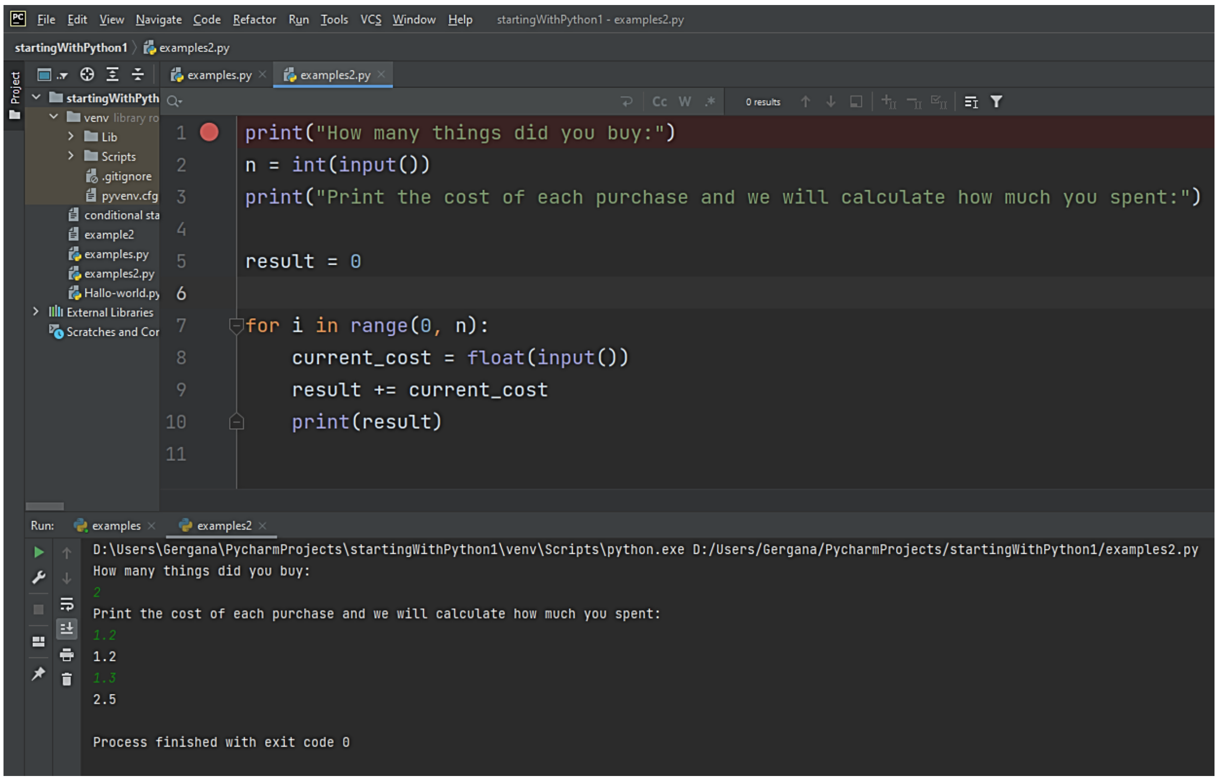
```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
How many things did you buy:
2
Print the cost of each purchase and we will calculate how much you spent:
1.2
2.3
3.5

Process finished with exit code 0
```

Tip on delineating loop code from the other code:

The indentation tells Python if a command is inside or outside of the loop. In other programming languages that you may be familiar with, the loop code is placed inside curly brackets (e.g. in Java). But in Python, special attention has to be paid to indentation. In the above example, print(result) is not indented, and therefore the print will be carried out after the program exits the loop. If you indent it to the level of the commands inside the loop (as in the example below), the program will print the result at each iteration.

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

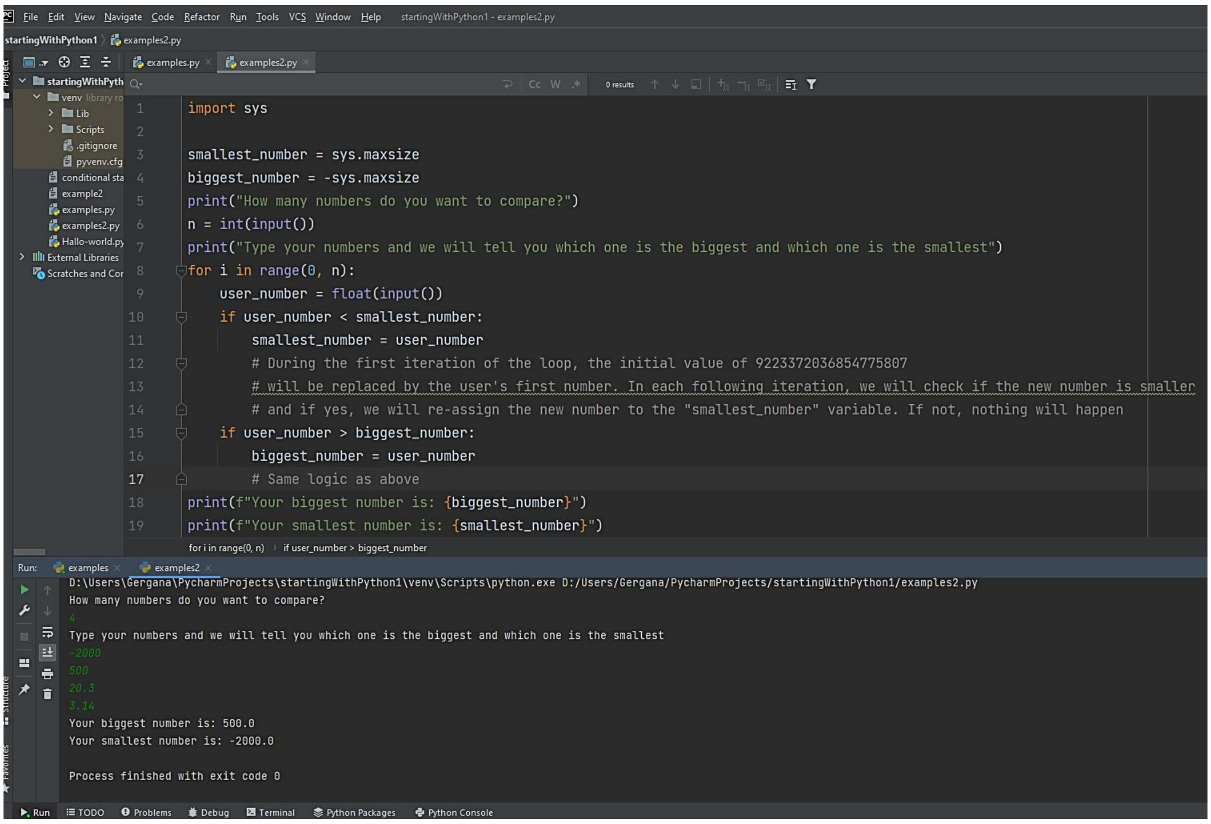

```python
print("How many things did you buy:")
n = int(input())
print("Print the cost of each purchase and we will calculate how much you spent:")

result = 0

for i in range(0, n):
    current_cost = float(input())
    result += current_cost
    print(result)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
How many things did you buy:
2
Print the cost of each purchase and we will calculate how much you spent:
1.2
1.2
1.3
2.5

Process finished with exit code 0
```

Let us now write a program that allows the user to type as many numbers as they like and the program will print the biggest and the smallest number. The program will accept both floating numbers and whole numbers, as well as both positive and negative numbers. The specificity here is how to initialize the variables that we need. We obviously need a variable for the biggest number and a variable for the smallest number, and they have to be initialized before the loop (outside of it). But how do we set the initial value of each? If you look at the code below, you will notice that each number typed by the user is compared to the currently biggest number and the currently smallest number, and if it is respectively bigger than the biggest number or smaller than the smallest number, then the current number will itself become respectively the biggest number or smallest number. Therefore, when we initialize the numbers, the biggest number should initially be the *smallest* number possible; otherwise, the user-defined number may turn out to be smaller than it and we will not replace this initial biggest value. Similarly, the smallest number should initially be the *biggest* possible number. In this way, it will be impossible to omit any user-defined number that is smaller than the initial value. In Python, the biggest possible number in a 64-bit system is 9223372036854775807, while the smallest is -9223372036854775807. For 32-bit systems the biggest number is 2147483647 and the smallest is -2147483647. These values are given by the Python function sys.maxsize(). To use this function, we need to import the sys library. See the whole code below.

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

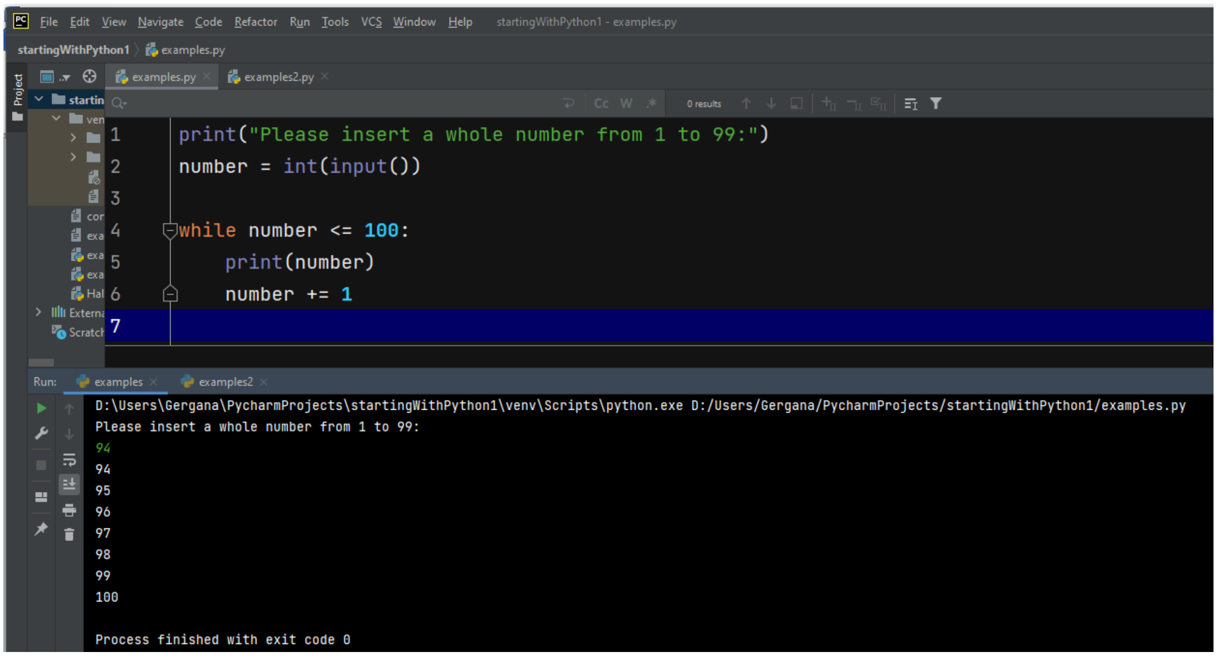


```python
import sys

smallest_number = sys.maxsize
biggest_number = -sys.maxsize
print("How many numbers do you want to compare?")
n = int(input())
print("Type your numbers and we will tell you which one is the biggest and which one is the smallest")
for i in range(0, n):
    user_number = float(input())
    if user_number < smallest_number:
        smallest_number = user_number
        # During the first iteration of the loop, the initial value of 9223372036854775807
        # will be replaced by the user's first number. In each following iteration, we will check if the new number is smaller
        # and if yes, we will re-assign the new number to the "smallest_number" variable. If not, nothing will happen
    if user_number > biggest_number:
        biggest_number = user_number
        # Same logic as above
print(f"Your biggest number is: {biggest_number}")
print(f"Your smallest number is: {smallest_number}")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
How many numbers do you want to compare?
4
Type your numbers and we will tell you which one is the biggest and which one is the smallest
-2000
500
20.3
3.14
Your biggest number is: 500.0
Your smallest number is: -2000.0

Process finished with exit code 0
```

While loop

The *for* loop requires us to specify the exact range for iteration. But what if we do not know the range or we simply want to iterate as long as some condition is valid? For the latter purpose, we use the *while* loop. The *while* loop specifies a condition that can be evaluated as True or False, and it iterates as long as this condition is True. For example, let us ask the user to input a number from 1 to 99, and then print all the numbers greater than the user's number, until we reach 100. You will notice in the code below that we repeat the print() function until the number equals 100 and, at each iteration, we increase the initial number with 1.

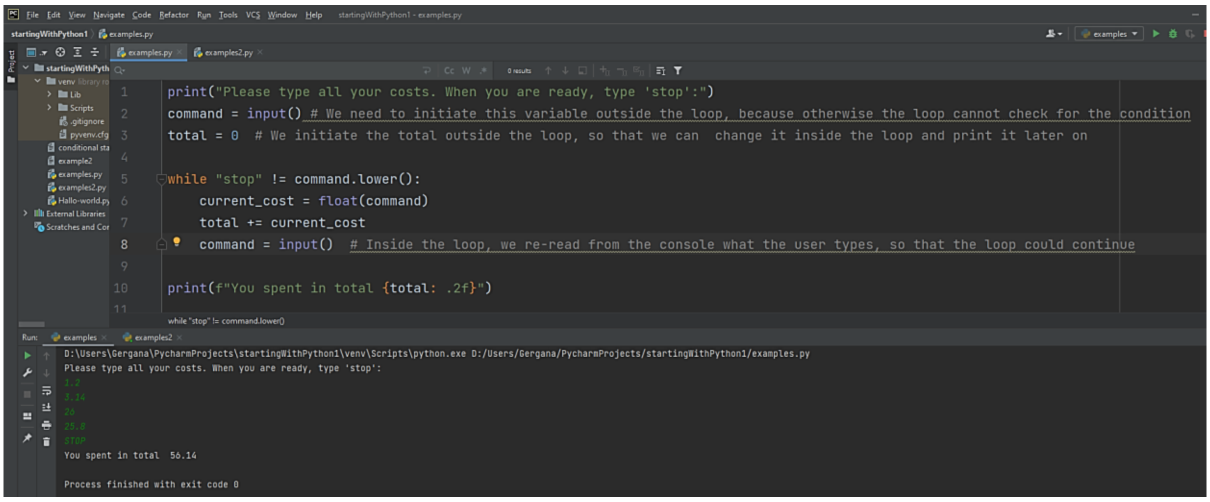


```python
print("Please insert a whole number from 1 to 99:")
number = int(input())

while number <= 100:
    print(number)
    number += 1
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Please insert a whole number from 1 to 99:
94
94
95
96
97
98
99
100

Process finished with exit code 0
```

Co-funded by the
Erasmus+ Programme
of the European Union

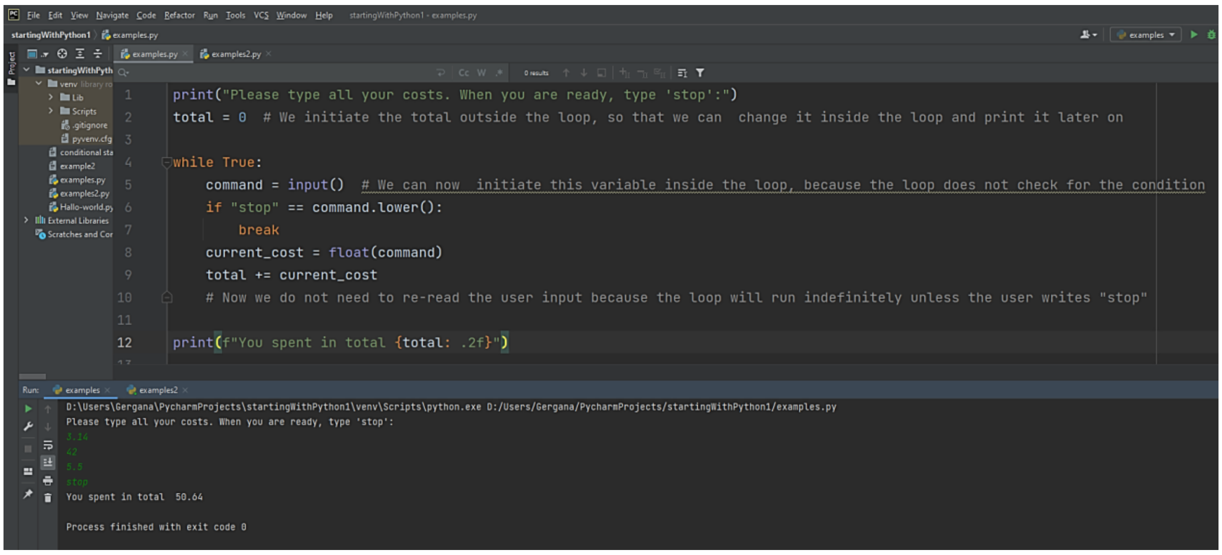
Let us now use a *while* loop to rewrite our previous program, where the user inputs their costs and we calculate the total money spent. In the previous program, we asked the users to say how many costs they will input and we used a *for* loop with a range set by the user's answer. Now, we will let the users input costs until they write "Stop". Then we will calculate the total cost and will print it on the console. Look at the code below. Read the comments about initialization of the variables. Notice that we first read the user input outside the loop to know if we need to enter the loop at all. Then inside the loop we re-read it at each iteration in order to check if the loop condition is still valid for the next iteration. Each time when we re-read the input, the variable command is assigned the new input as its value. You should also notice that we first read the input as string, and we cast it to float only after we enter the loop. This is because if we cast it immediately, as soon as the user writes "stop", the program will be unable to cast it and will crash with an error. We check if the command is "Stop" or "stop" or "STOP". We do this by comparing the two strings - the command string and "Stop" - in case-insensitive manner. We use the *my_string*.lower() command to transform the command in small letters only. Thus, even though the input in our example was "STOP" instead of "stop", the program was executed correctly.



```
print("Please type all your costs. When you are ready, type 'stop':")
command = input() # We need to initiate this variable outside the loop, because otherwise the loop cannot check for the condition
total = 0  # We initiate the total outside the loop, so that we can  change it inside the loop and print it later on

while "stop" != command.lower():
    current_cost = float(command)
    total += current_cost
    command = input()  # Inside the loop, we re-read from the console what the user types, so that the loop could continue

print(f"You spent in total {total: .2f}")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Please type all your costs. When you are ready, type 'stop':
1.2
3.14
26
25.8
STOP
You spent in total  56.14

Process finished with exit code 0
```

A variation of the *while* loop is the *while True* loop. This loop will run indefinitely until there is a command that breaks from it - the break command. We can call this command in a conditional statement, so that the loop can break when a certain condition is True. However, the same can be easily achieved with inserting the condition in the *while* loop itself, so using the *while True* loop is not usually a recommended practice (it can make the code more difficult to change). There are some use cases of the *while True* loop, though, like in the case of a network listener, where we do need an infinite loop and so using the *while True* loop is justified. In any case, you can see below the above example implemented with a *while True* loop.



```
print("Please type all your costs. When you are ready, type 'stop':")
total = 0  # We initiate the total outside the loop, so that we can  change it inside the loop and print it later on

while True:
    command = input()  # We can now  initiate this variable inside the loop, because the loop does not check for the condition
    if "stop" == command.lower():
        break
    current_cost = float(command)
    total += current_cost
    # Now we do not need to re-read the user input because the loop will run indefinitely unless the user writes "stop"

print(f"You spent in total {total: .2f}")
```

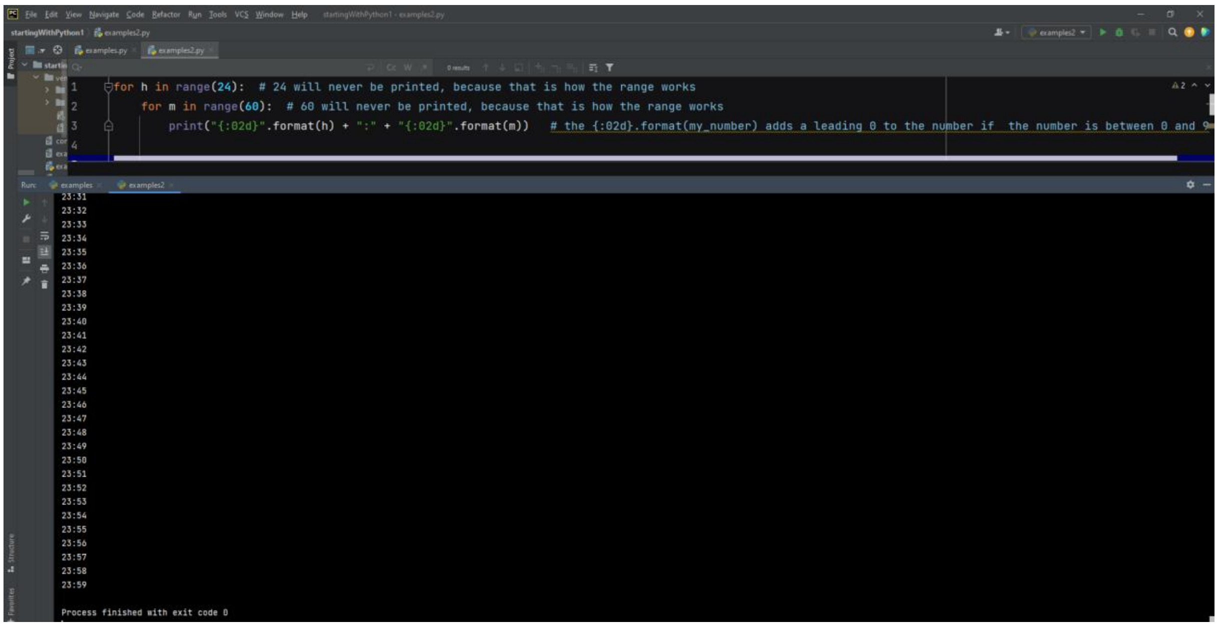
```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Please type all your costs. When you are ready, type 'stop':
3.14
42
5.5
stop
You spent in total  50.64

Process finished with exit code 0
```

You are now ready to test yourself in HackerRank. Solve the *Loops* Challenge. It is available in the "Easy" difficulty mode. Try also the *Print Function* Challenge, which will test your knowledge of both printing on the console and *for* loops.

Nested loops

For more complex algorithms, it is necessary to loop over several data sets simultaneously. To do this, you need nested loops. The iteration starts from the outer loop. Once the first value of the outer loop is loaded, the program iterates over all the values in the inner loop. Following this, the second value of the outer loop is loaded, and the program again iterates over all the values in the inner loop. This is repeated until all the values in the outer loop are iterated over.

To understand nested loops, it is best to try an example. Let us print the minutes and hours of a day using nested loops. We know how many hours there are and how many minutes there are per hour, so we can use nested *for* loops. As you know, you need 60 minutes to pass before the hour changes. Thus, the outer loop is the hours and the inner loop is the minutes.



```
for h in range(24):  # 24 will never be printed, because that is how the range works
    for m in range(60):  # 60 will never be printed, because that is how the range works
        print("{:02d}".format(h) + ":" + "{:02d}".format(m))   # the {:02d}.format(my_number) adds a leading 0 to the number if  the number is between 0 and 9
```

```
23:31
23:32
23:33
23:34
23:35
23:36
23:37
23:38
23:39
23:40
23:41
23:42
23:43
23:44
23:45
23:46
23:47
23:48
23:49
23:50
23:51
23:52
23:53
23:54
23:55
23:56
23:57
23:58
23:59

Process finished with exit code 0
```

What does this program do? It enters the outer loop and takes the first value in the 0-24 range, which is 0 (this is the hour). Once this value is loaded, the program enters the inner loop and starts iterating over all the values in the inner loop (the minutes). It will load all the values,

Co-funded by the Erasmus+ Programme of the European Union

starting from 0 and ending with 59 (the end index - the ceiling of the range - is not included, you remember). When printing, we have added a leading zero to be printed whenever the number is between 0 and 9. So, instead of 1, we will print 01. At each iteration of the inner loop, the program will print the hours and the minutes: 00:00, 00:01, 00:02……00:59. Once the program reaches 60 in the inner loop, it will exit the inner loop, enter the outer loop and load the second value in the outer loop, namely 1. Then it will iterate again over all the values in the inner loop (from 0 to 59) and will print: 01:00, 01:01, 01:02….01:59. Then it will again exit the inner loop, enter the outer loop and load the third value in the outer loop, namely 2. This will be repeated until the last iteration of the outer loop, namely 23. Once 23 is loaded, the program will iterate over 0-60 in the inner loop and will print: 23:00, 23:01, 23:02….23:59. The last printed value will be 23 hours and 59 minutes. Write the program to check the results yourself in PyCharm.

Tip on writing code more efficiently:

Notice that we did need not need to write range(0, 24). We wrote only range(24), because it is equivalent when the starting value is 0.

Tip on formatting:

A leading zero is printed by "{:02d}".format(*my-number*). This function adds a zero if the number contains only one digit. If you write "{:03d}".format(*my-number*), you will get 1 or 2 leading zeroes, so that the printed number will always have 3 digits.

Working with text

Basic functions

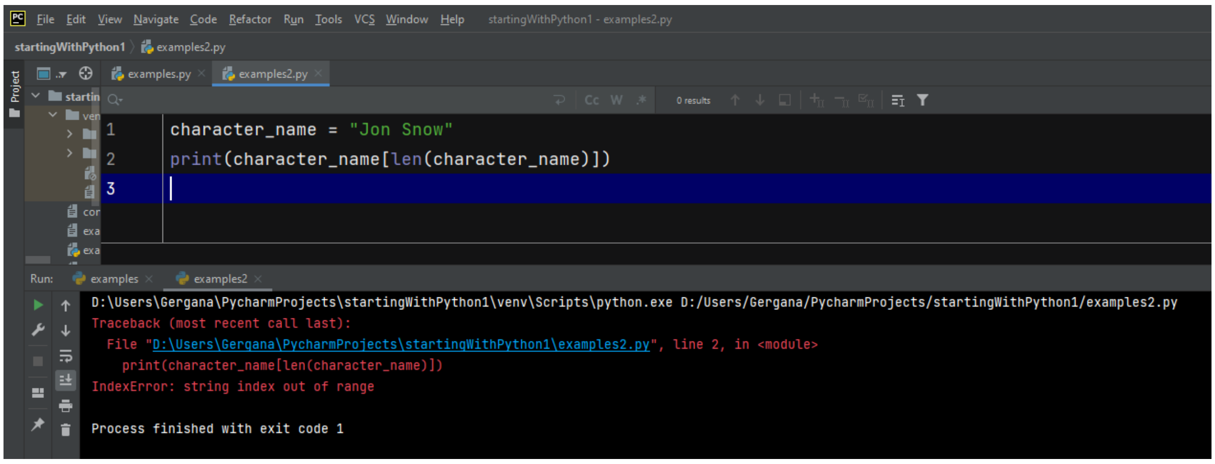
In programs we often need to work with strings (text). Python provides a lot of commands and functions to allow us to use and manipulate strings as we wish.

The len(*text*) function will return the length of a string (be it a word or a long text). Try it yourself in PyCharm.

The function *string_variable*[*i*] will return the character on the i-th index of the string variable. The index indicates the position of a character in the string. For example:

```
character_name = "Jon Snow"
print(character_name[0])
```

This will return "J". Indices are always integers. The first index of every string is always 0, so the last index is the *length of the string* – 1. Confusing the indices will lead to an "index out of range" error.


File Edit View Navigate Code Refactor Run Tools VCS Window Help
startingWithPython1 - examples2.py
startingWithPython1
examples2.py
examples.py
examples2.py
0 results
Run:
examples
examples2

```
character_name = "Jon Snow"
print(character_name[len(character_name)])
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Traceback (most recent call last):
  File "D:\Users\Gergana\PycharmProjects\startingWithPython1\examples2.py", line 2, in <module>
    print(character_name[len(character_name)])
IndexError: string index out of range

Process finished with exit code 1
```

Co-funded by the
Erasmus+ Programme
of the European Union

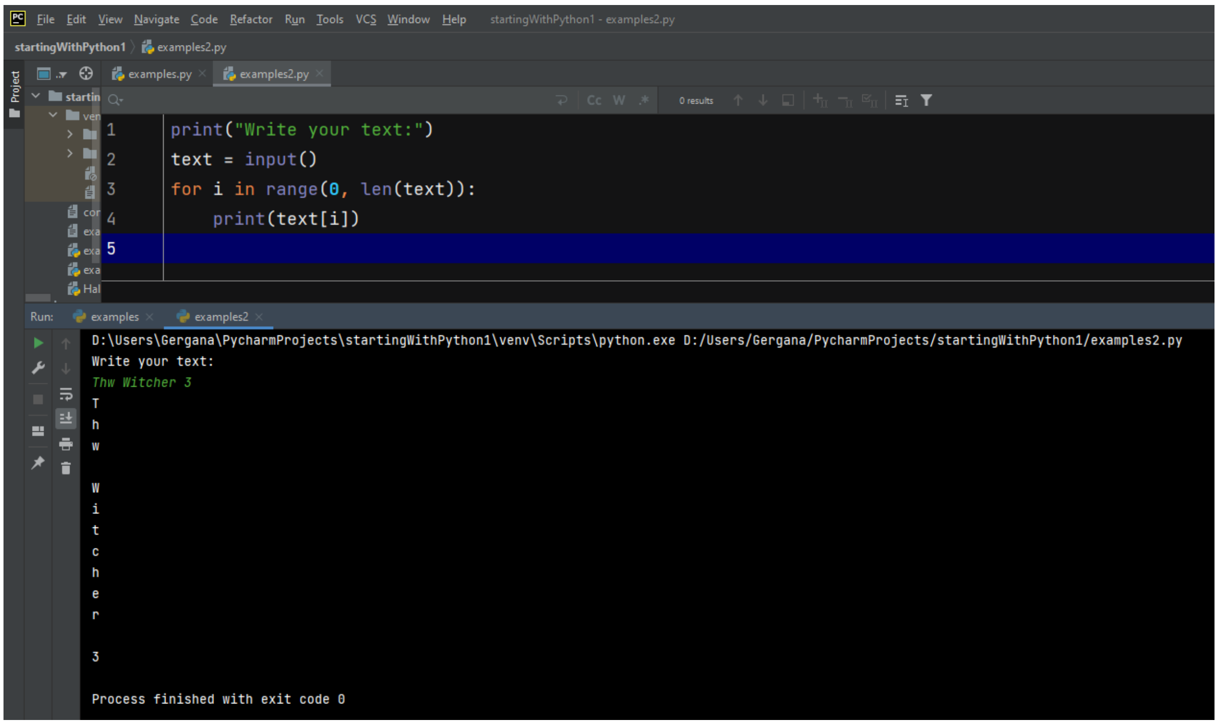
The above example ended in an error. We tried to get the last index using the length of the string. This is intuitive but wrong, because we started from 0 and not from 1. So, if you want to take the last character of the string, you should use:

```
character_name = "Jon Snow"
print(character_name[len(character_name) - 1])
```

You can use a *for* loop to iterate over all the characters in a string. For example, with the code below we write each character of "Jon Snow" on a new line. Notice that since the range of the loop does not include the last index, we do not subtract 1 from the length, as we did above. We iterate exactly to the length of the string, because we know that the loop will terminate before the last index – len(character_name) (which is out of range).

```
character_name = "Jon Snow"
for i in range(0, len(character_name)):
    print(character_name[i])
```

We can also do the same for any text input by the user on the console:

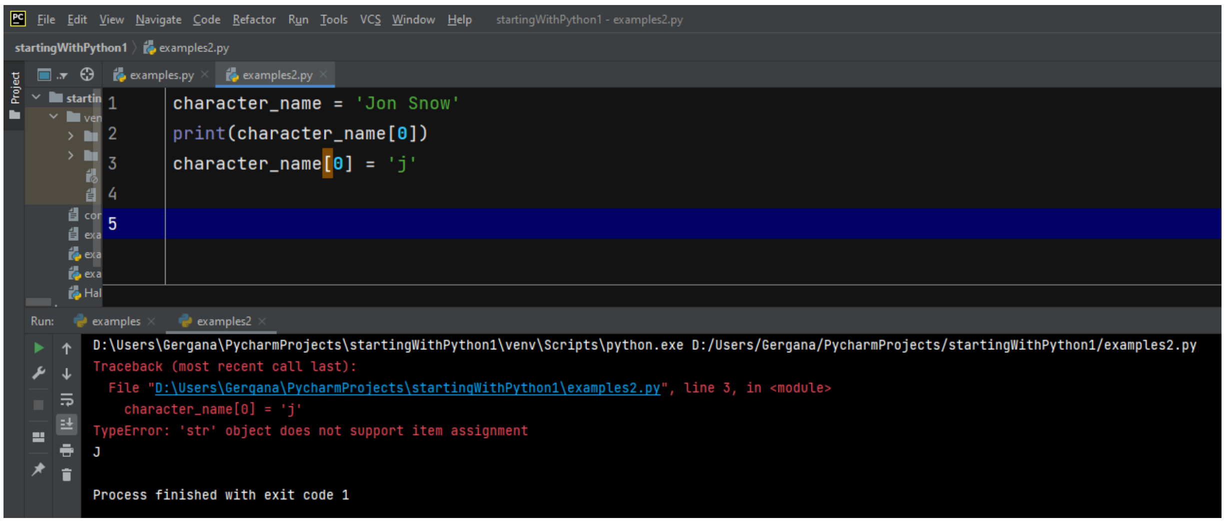


```
print("Write your text:")
text = input()
for i in range(0, len(text)):
    print(text[i])
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Write your text:
Thw Witcher 3
T
h
w

W
i
t
c
h
e
r

3

Process finished with exit code 0
```

In Python, strings are immutable and cannot be changed. You can print a specific character but you cannot change it. In the example below, we can print the first letter of the text, but when we try to change it, we get an error.

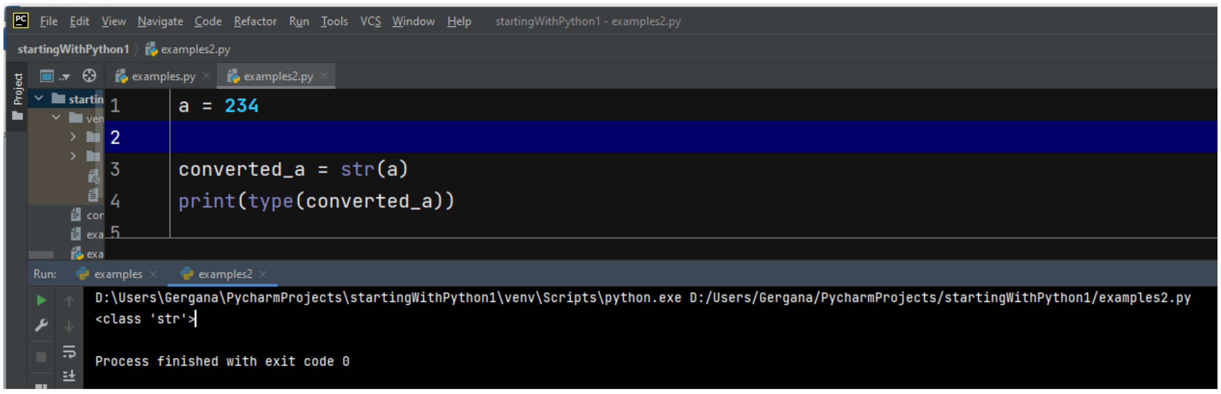
Co-funded by the Erasmus+ Programme of the European Union


```
character_name = 'Jon Snow'
print(character_name[0])
character_name[0] = 'j'
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Traceback (most recent call last):
  File "D:\Users\Gergana\PycharmProjects\startingWithPython1\examples2.py", line 3, in <module>
    character_name[0] = 'j'
TypeError: 'str' object does not support item assignment
J

Process finished with exit code 1
```

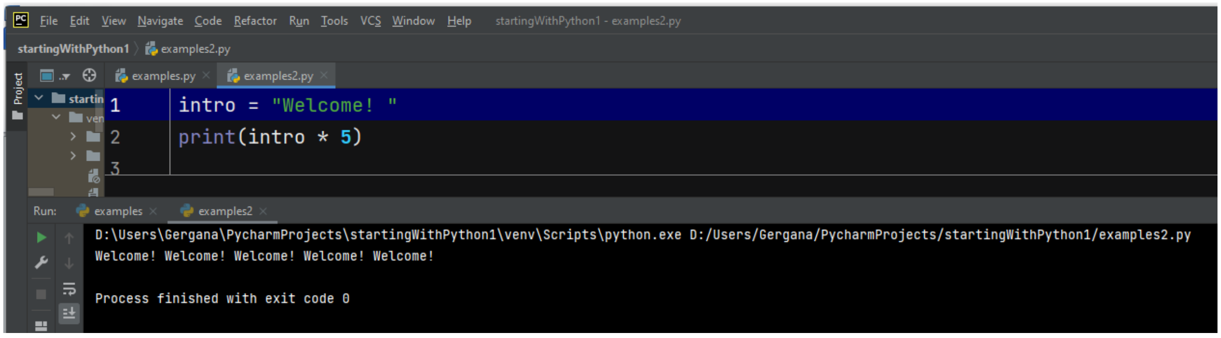
More about formatting strings

We already learned in the previous sections that when we need to convert another datatype to string, we use the str() function, and we place the variable that we need to convert inside the round brackets of the function (we pass it in as an argument). We can use this function to convert into string any built-in datatype in Python, including floating numbers, lists, tuples, dictionaries, etc. We will learn more about some of these datatypes later on, and then you will be able to try out the function. Below we show again a simple example of converting an integer.


```
a = 234

converted_a = str(a)
print(type(converted_a))
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
<class 'str'>

Process finished with exit code 0
```

We can use the * operator to repeat a string a specific number of times.

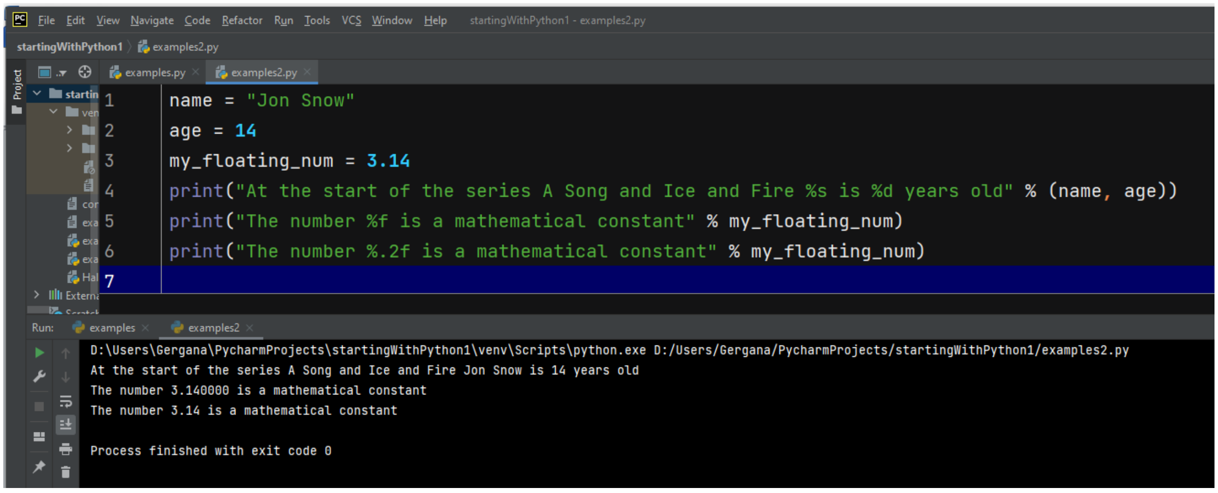

```
intro = "Welcome! "
print(intro * 5)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Welcome! Welcome! Welcome! Welcome! Welcome!

Process finished with exit code 0
```

Now let us return to working with text that includes the values of certain variables (i.e. this value is not known in advance, but is provided as the program executes). We deal with this through string formatting. We already learned about string formatting using the f"*formatted string*" method. There are other methods that you may see and use. You can format with the % operator, followed by the suffix s for string, the suffix d for integer and the suffix f for float. The

Co-funded by the Erasmus+ Programme of the European Union

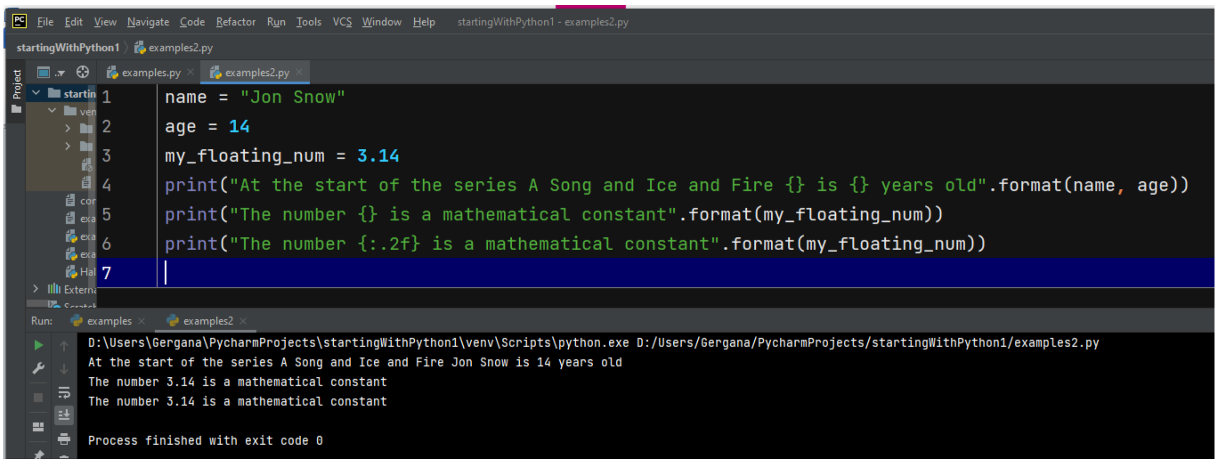
combination % and suffix serves as a placeholder for the variable, which is then provided after the string, using the syntax % *variable*. If more placeholders are used, we use % (*variable_1, variable_2*). Look at the examples below. The f suffix allows for additional formatting. If we add .2 or .3 before it, we will specify how many decimal places will be shown after the decimal point for a floating number. We can specify any number of decimal places. If we do not specify the decimal places, by default Python will extend floating numbers to 6 decimal places after the decimal point.



```
name = "Jon Snow"
age = 14
my_floating_num = 3.14
print("At the start of the series A Song and Ice and Fire %s is %d years old" % (name, age))
print("The number %f is a mathematical constant" % my_floating_num)
print("The number %.2f is a mathematical constant" % my_floating_num)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
At the start of the series A Song and Ice and Fire Jon Snow is 14 years old
The number 3.140000 is a mathematical constant
The number 3.14 is a mathematical constant

Process finished with exit code 0
```

Another way to format a string is the {} operators, followed by .format(*variable_1*, *variable_2*). This is similar to the f-string formatting we saw earlier in this Compendium. The {} operators are used as a placeholder for the variable, but instead of including the variable's name between the curly brackets, we leave the brackets empty and pass the variable in the .format() function as an argument. If several variables are provided as arguments, they need to be separated by comas. When the program is executed, the placeholders will be replaced by the actual values of the variables in the order in which the variables were passed. The floating number can be further formatter with the .xf notation, preceded by a colon, with x indicating the number of decimal places that should be shown after the decimal point. Look at the code below, which is identical to the previous example, even though we used this alternative method.



```
name = "Jon Snow"
age = 14
my_floating_num = 3.14
print("At the start of the series A Song and Ice and Fire {} is {} years old".format(name, age))
print("The number {} is a mathematical constant".format(my_floating_num))
print("The number {:.2f} is a mathematical constant".format(my_floating_num))
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
At the start of the series A Song and Ice and Fire Jon Snow is 14 years old
The number 3.14 is a mathematical constant
The number 3.14 is a mathematical constant

Process finished with exit code 0
```

Lists

In Python a list is a datatype - a collection of values that are *ordered (index supported)* and *changeable* (*mutable*). The items in the list are separated with comas and the list is written inside square brackets [].

Co-funded by the Erasmus+ Programme of the European Union

An empty list is created with empty square brackets [] or with the list() function. A list with pre-defined elements is created by placing these elements inside square brackets, separated by comas. Elements can be duplicates (they do not have to be unique).

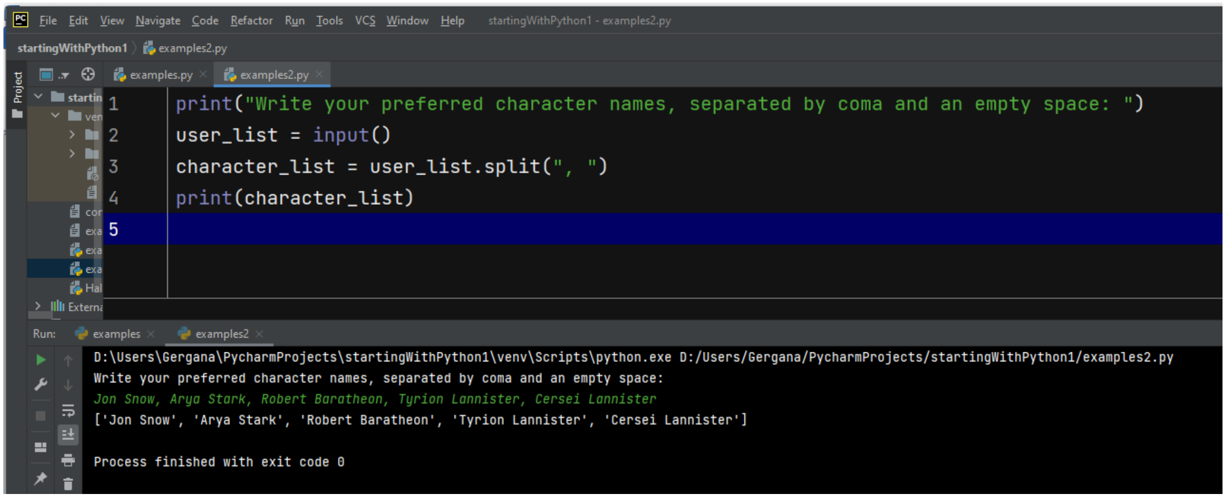
```
empty_list = []
numbers = [42, 3.14, 1, 9]
```

The elements/members of a list can be of any datatype, including other lists. In the latter case, they are called nested lists.

```
nested_numbers = [42, 3.14, 1, 9, [3.14, 0]]
```

Tip:

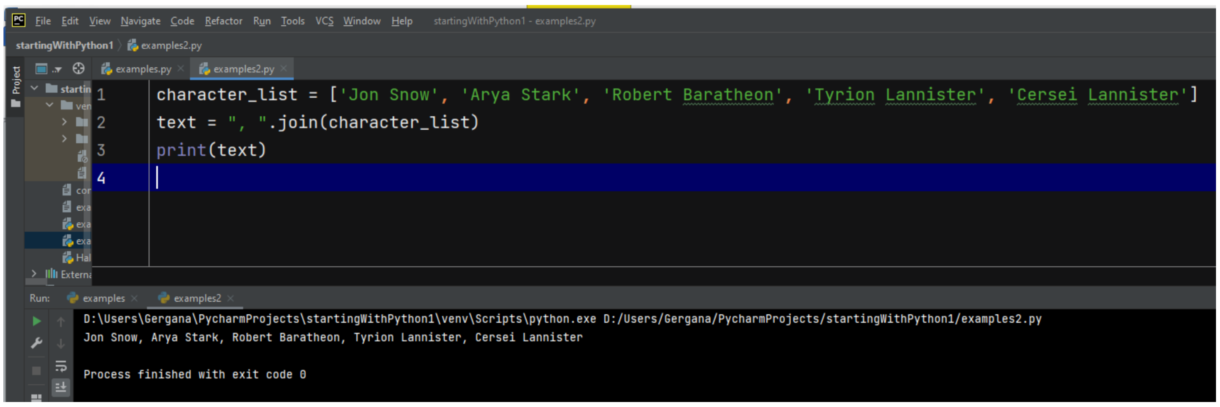
A useful approach is to *split* a text (string) and to directly create a list out of the resulting pieces. This would allow us to read text typed on the console or provided by the user and to store it inside a list, as in the example below. Splitting is achieved with the *user_text*.split(“*delimiter*”) function. You need to choose the delimiter, which helps Python recognize where one word ends and the other one starts. Any character or group of characters can be a delimiter.



```
print("Write your preferred character names, separated by coma and an empty space: ")
user_list = input()
character_list = user_list.split(", ")
print(character_list)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Write your preferred character names, separated by coma and an empty space:
Jon Snow, Arya Stark, Robert Baratheon, Tyrion Lannister, Cersei Lannister
['Jon Snow', 'Arya Stark', 'Robert Baratheon', 'Tyrion Lannister', 'Cersei Lannister']

Process finished with exit code 0
```

Conversely, we can create a text out of an existing list of strings. This is done with the *“delimiter”*.join*(list_name)* command. We cannot join elements of other datatypes unless we have converted them into strings beforehand.



```
character_list = ['Jon Snow', 'Arya Stark', 'Robert Baratheon', 'Tyrion Lannister', 'Cersei Lannister']
text = ", ".join(character_list)
print(text)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Jon Snow, Arya Stark, Robert Baratheon, Tyrion Lannister, Cersei Lannister

Process finished with exit code 0
```

Since lists are numbered, we can use indices to access a list member. Remember that indices start from 0, not from 1. Look again at this list

```
nested_numbers = [42, 3.14, 1, 9, [3.14, 0]]
```

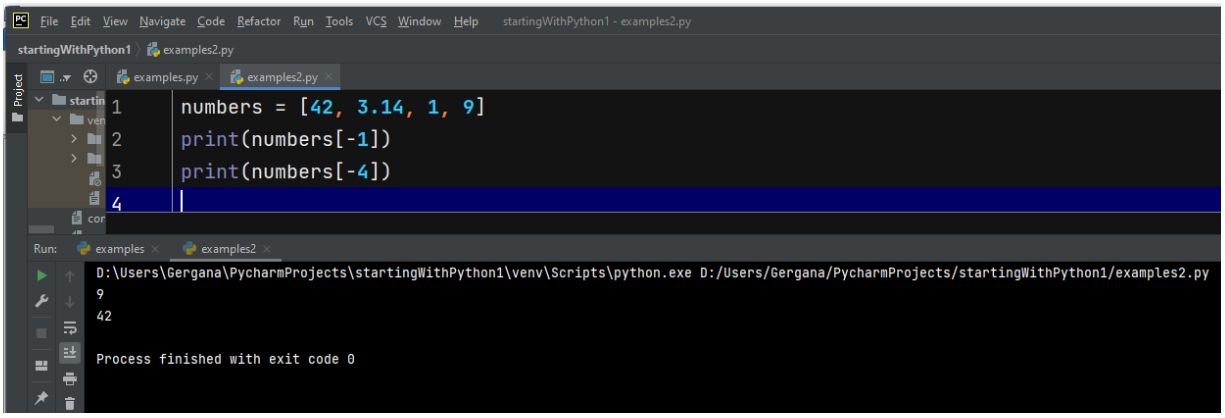
Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

The element on index 1 of the above list is 3.14, and not 42. 42 is on index 0. Similarly, the last index is equal to the *number of elements* – 1. Trying to access an index that is out of range is a common mistake in programming and would cause the program to terminate with an error. If we have nested a list inside another list, we may access the nested list's elements with nested indexing (the nested list's index in the main list comes first, and the element's index in the nested list comes second). For example, to access 0 in the above example, we need to write:

```
print(nested_numbers[4][1])  # The list [3.14, 0] is the 5th element in the nested_numbers list, and is accessed with [4]; 0 is the second element in the nested list and is accessed with [1]
```

Like with strings, it is possible to use negative indices. The -1 index accesses the last item, and the –[*length of list*] index accesses the first element of the list. See the example below.

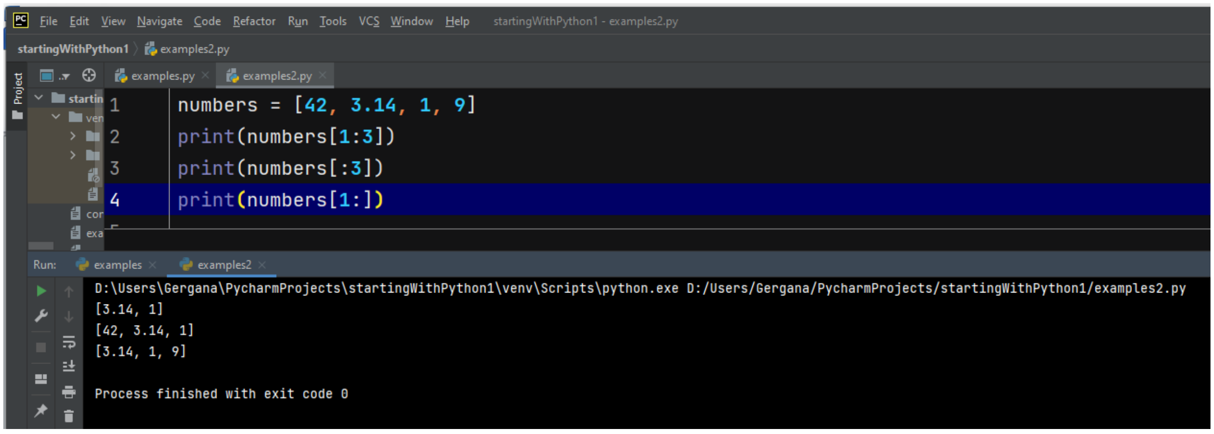


```
numbers = [42, 3.14, 1, 9]
print(numbers[-1])
print(numbers[-4])
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
9
42

Process finished with exit code 0
```

We can use slicing in lists in order to return the elements in a range from a start index to an end index. We do this by specifying the range of indices that we need. The start index in the range is accessed, but the end index is not accessed (the start index is inclusive, but the end index is exclusive).

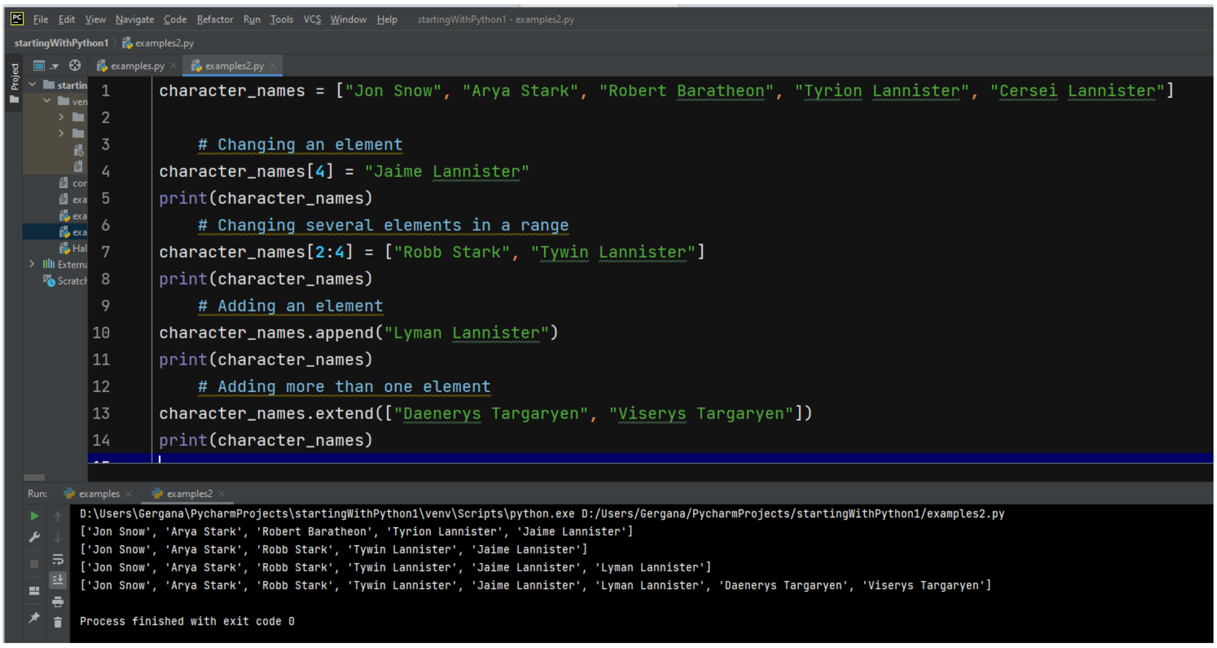
The command *list_name*[1:3] will access and print the elements on the first and second index, but not the third one. If we do not specify a start index but instead start from : (colon), followed by the end index, we will access all elements from the beginning to the *end index* - 1. If we do not specify an end index, and start from a start index, followed by : (colon) only, we will access all elements starting from the start index till the end of the list. If we use [::] or [0:] we access all the elements. See the examples below.



```
numbers = [42, 3.14, 1, 9]
print(numbers[1:3])
print(numbers[:3])
print(numbers[1:])
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
[3.14, 1]
[42, 3.14, 1]
[3.14, 1, 9]

Process finished with exit code 0
```

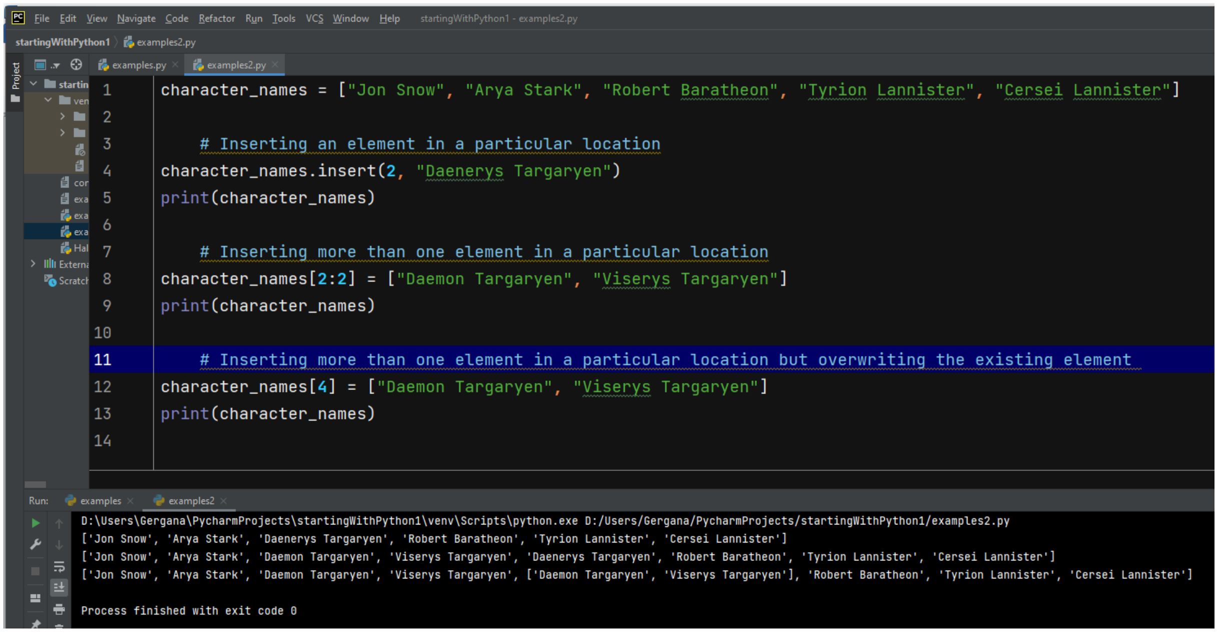
We can add, remove and change elements in the list by accessing their index. New values are assigned using the = operator. A new element is added with the *name_of_list*.append(*element*) method. The new element is always added at the tail of the list. We can use the *name_of_list*.extend(*element_1, element_2*) method to add more than one element, again at the tail of the list.



```python
character_names = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Tyrion Lannister", "Cersei Lannister"]

    # Changing an element
character_names[4] = "Jaime Lannister"
print(character_names)
    # Changing several elements in a range
character_names[2:4] = ["Robb Stark", "Tywin Lannister"]
print(character_names)
    # Adding an element
character_names.append("Lyman Lannister")
print(character_names)
    # Adding more than one element
character_names.extend(["Daenerys Targaryen", "Viserys Targaryen"])
print(character_names)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
['Jon Snow', 'Arya Stark', 'Robert Baratheon', 'Tyrion Lannister', 'Jaime Lannister']
['Jon Snow', 'Arya Stark', 'Robb Stark', 'Tywin Lannister', 'Jaime Lannister']
['Jon Snow', 'Arya Stark', 'Robb Stark', 'Tywin Lannister', 'Jaime Lannister', 'Lyman Lannister']
['Jon Snow', 'Arya Stark', 'Robb Stark', 'Tywin Lannister', 'Jaime Lannister', 'Lyman Lannister', 'Daenerys Targaryen', 'Viserys Targaryen']

Process finished with exit code 0
```

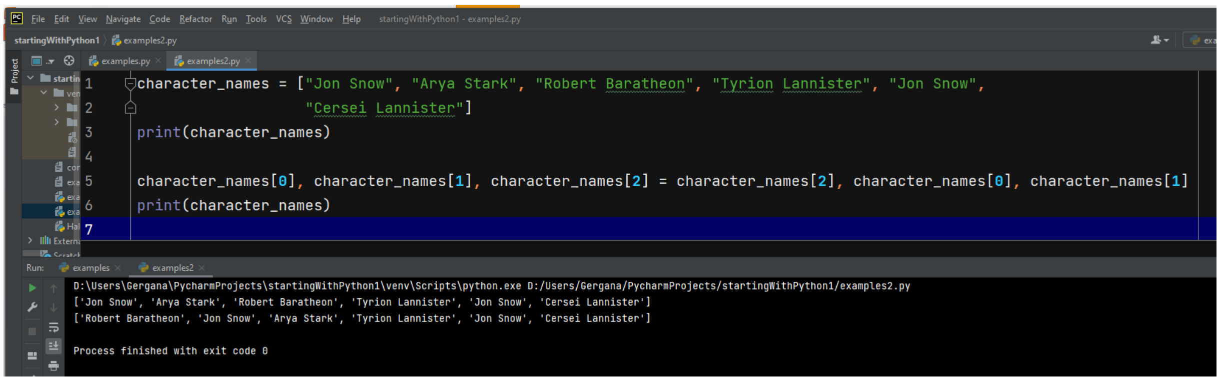
If we need to add an element at a particular index, we need to use instead the *name_of_list*.insert(*index*, *element*) method. We pass in as arguments the index on which we need to insert the new element and the new element itself. If we need to insert more than one element in a particular location, then we assign the new elements (grouped into a list) on the [index:index] position. Be careful not assign them on the [index] position, because you will overwrite the existing value with the new list.



```python
character_names = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Tyrion Lannister", "Cersei Lannister"]

    # Inserting an element in a particular location
character_names.insert(2, "Daenerys Targaryen")
print(character_names)

    # Inserting more than one element in a particular location
character_names[2:2] = ["Daemon Targaryen", "Viserys Targaryen"]
print(character_names)

    # Inserting more than one element in a particular location but overwriting the existing element
character_names[4] = ["Daemon Targaryen", "Viserys Targaryen"]
print(character_names)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
['Jon Snow', 'Arya Stark', 'Daenerys Targaryen', 'Robert Baratheon', 'Tyrion Lannister', 'Cersei Lannister']
['Jon Snow', 'Arya Stark', 'Daemon Targaryen', 'Viserys Targaryen', 'Daenerys Targaryen', 'Robert Baratheon', 'Tyrion Lannister', 'Cersei Lannister']
['Jon Snow', 'Arya Stark', 'Daemon Targaryen', 'Viserys Targaryen', ['Daemon Targaryen', 'Viserys Targaryen'], 'Robert Baratheon', 'Tyrion Lannister', 'Cersei Lannister']

Process finished with exit code 0
```

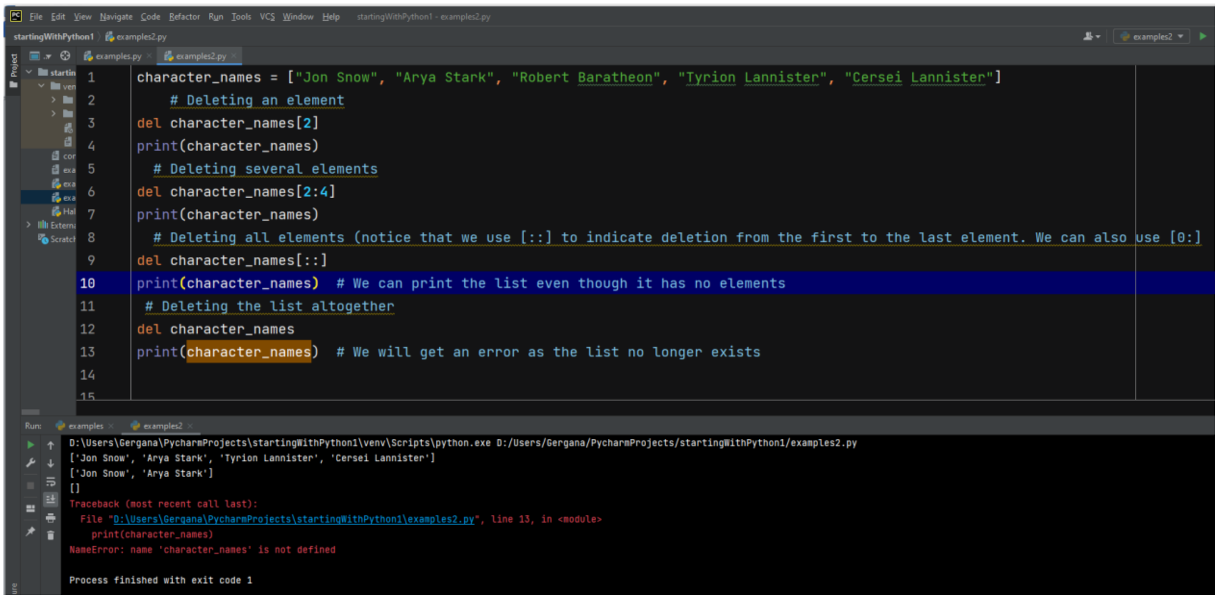
It is also possible to swap the positions (indices) of the elements. This is done by simply assigning the elements to different indexes, as shown in the example below.



```
character_names = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Tyrion Lannister", "Jon Snow",
                   "Cersei Lannister"]
print(character_names)

character_names[0], character_names[1], character_names[2] = character_names[2], character_names[0], character_names[1]
print(character_names)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
['Jon Snow', 'Arya Stark', 'Robert Baratheon', 'Tyrion Lannister', 'Jon Snow', 'Cersei Lannister']
['Robert Baratheon', 'Jon Snow', 'Arya Stark', 'Tyrion Lannister', 'Jon Snow', 'Cersei Lannister']

Process finished with exit code 0
```

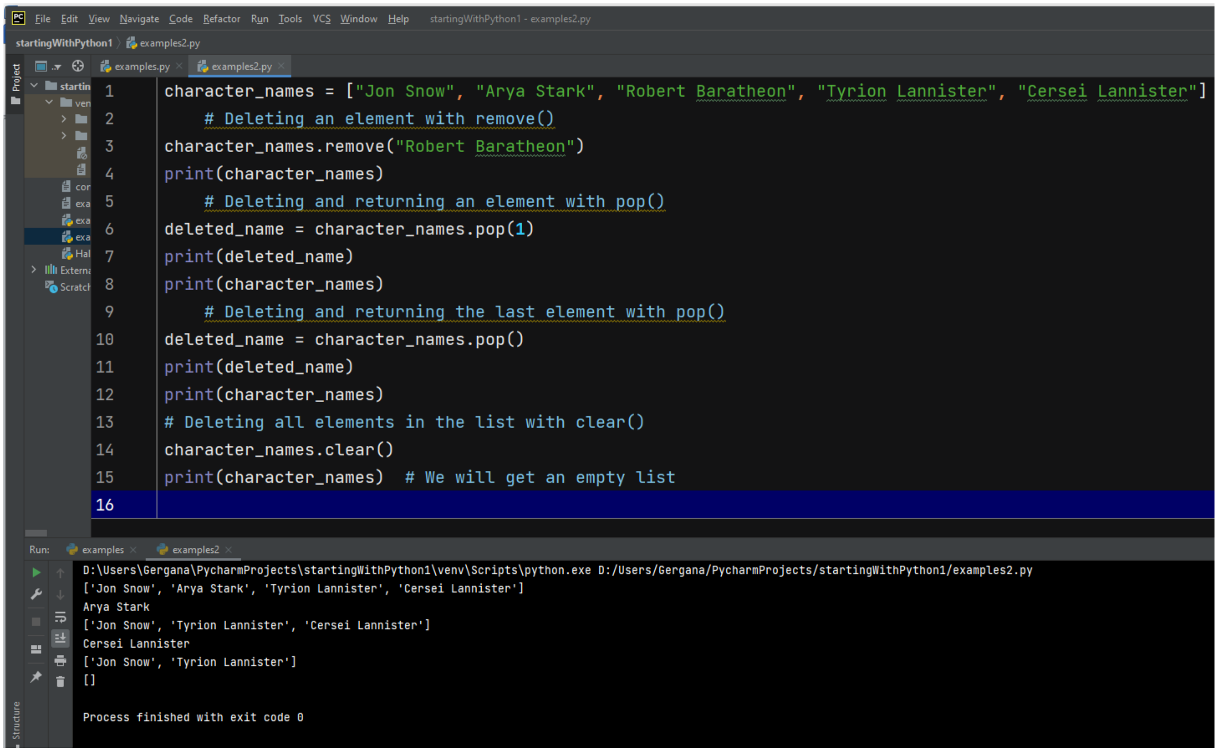
Deleting an element is done by accessing the element's index and the del keyword. We can delete one or more elements by indicating an index or a range of indices. We can also delete the list itself. Examples are shown below.



```
character_names = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Tyrion Lannister", "Cersei Lannister"]
    # Deleting an element
del character_names[2]
print(character_names)
  # Deleting several elements
del character_names[2:4]
print(character_names)
  # Deleting all elements (notice that we use [::] to indicate deletion from the first to the last element. We can also use [0:]
del character_names[::]
print(character_names)  # We can print the list even though it has no elements
 # Deleting the list altogether
del character_names
print(character_names)  # We will get an error as the list no longer exists
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
['Jon Snow', 'Arya Stark', 'Tyrion Lannister', 'Cersei Lannister']
['Jon Snow', 'Arya Stark']
[]
Traceback (most recent call last):
  File "D:\Users\Gergana\PycharmProjects\startingWithPython1\examples2.py", line 13, in <module>
    print(character_names)
NameError: name 'character_names' is not defined

Process finished with exit code 1
```

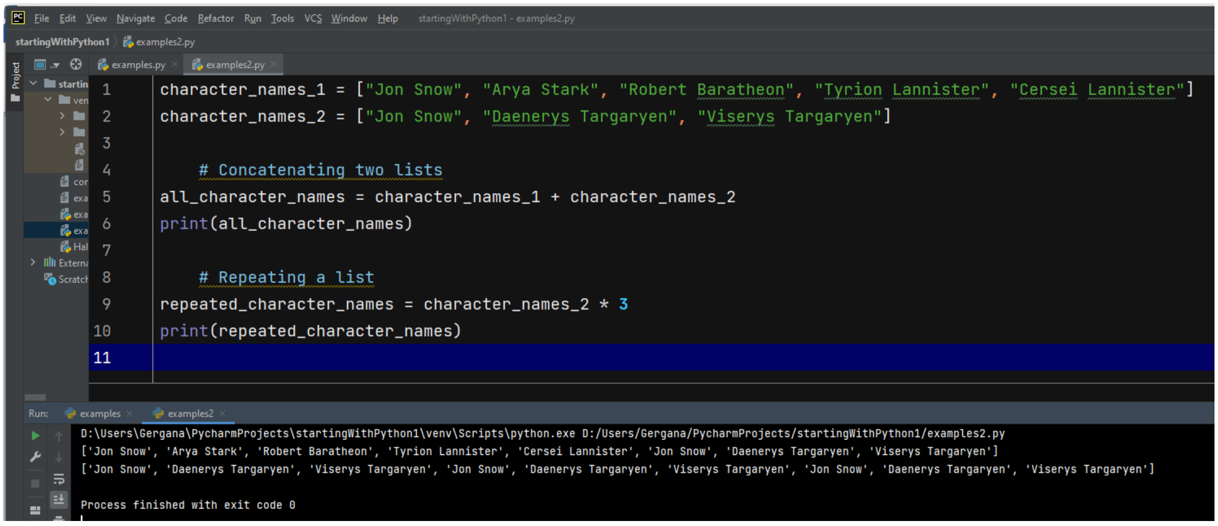
Apart from the del keyword, we can use the *list_name*.remove(*element*) method by passing in it as an argument the element that we need to remove. The *list_name*.pop(index) method works in the same way but we have to pass in as an argument the index of the element that we want to remove. This method also returns the element that is being removed, so we can use it in the program. If we do not pass in the index as an argument, pop() will remove the last element, which could be useful if we need to use the list as a *stack*. We can delete all the elements in the list by using the *list_name*.clear() method.



```
character_names = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Tyrion Lannister", "Cersei Lannister"]
    # Deleting an element with remove()
character_names.remove("Robert Baratheon")
print(character_names)
    # Deleting and returning an element with pop()
deleted_name = character_names.pop(1)
print(deleted_name)
print(character_names)
    # Deleting and returning the last element with pop()
deleted_name = character_names.pop()
print(deleted_name)
print(character_names)
# Deleting all elements in the list with clear()
character_names.clear()
print(character_names)  # We will get an empty list
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
['Jon Snow', 'Arya Stark', 'Tyrion Lannister', 'Cersei Lannister']
Arya Stark
['Jon Snow', 'Tyrion Lannister', 'Cersei Lannister']
Cersei Lannister
['Jon Snow', 'Tyrion Lannister']
[]

Process finished with exit code 0
```

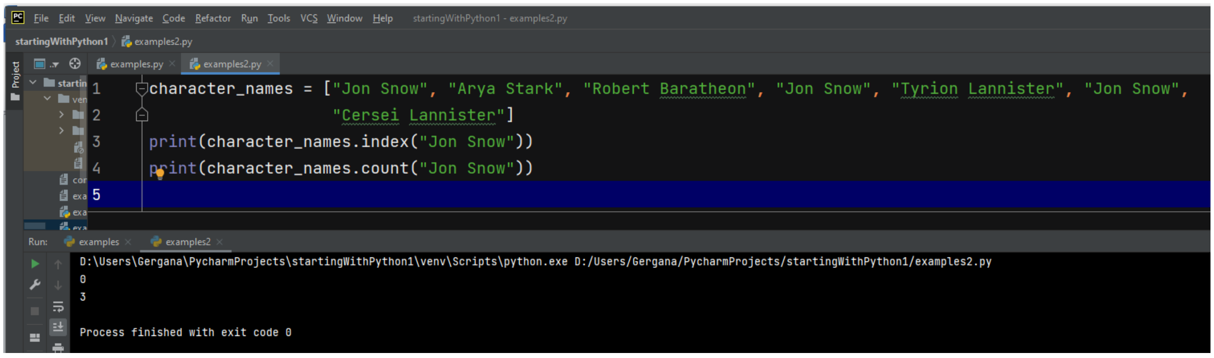
Two lists can be combined (concatenated) with the + operator. A list can be repeated several times using the * operator, followed by the number of repetitions. The result is a new list.



```
character_names_1 = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Tyrion Lannister", "Cersei Lannister"]
character_names_2 = ["Jon Snow", "Daenerys Targaryen", "Viserys Targaryen"]

    # Concatenating two lists
all_character_names = character_names_1 + character_names_2
print(all_character_names)

    # Repeating a list
repeated_character_names = character_names_2 * 3
print(repeated_character_names)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
['Jon Snow', 'Arya Stark', 'Robert Baratheon', 'Tyrion Lannister', 'Cersei Lannister', 'Jon Snow', 'Daenerys Targaryen', 'Viserys Targaryen']
['Jon Snow', 'Daenerys Targaryen', 'Viserys Targaryen', 'Jon Snow', 'Daenerys Targaryen', 'Viserys Targaryen', 'Jon Snow', 'Daenerys Targaryen', 'Viserys Targaryen']

Process finished with exit code 0
```

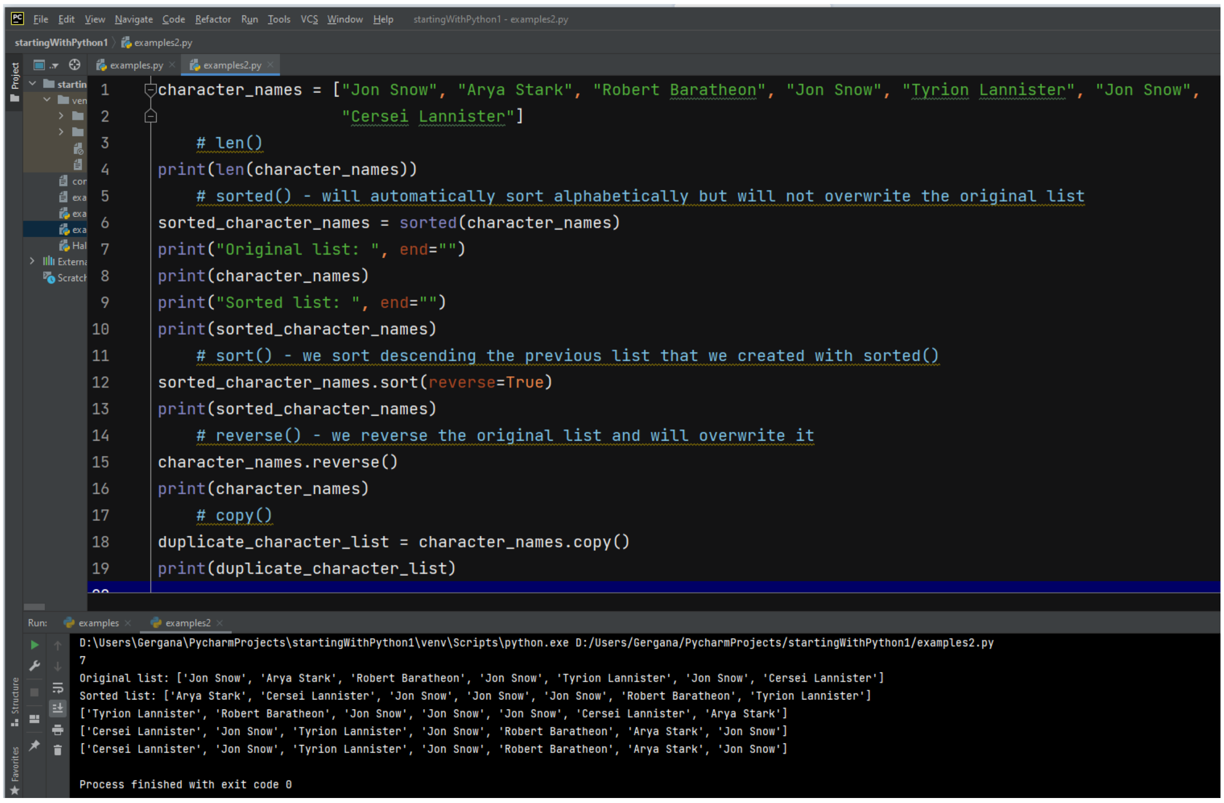
With the *list_name*.index(*element*) method we can get the index of an element that we know exists in the list. However, if the element is present in the list several times, we will only get the index of the first occurrence of this element. On the other hand, we can use the *list_name*.count(*element*) method to return the number of times a specified element appears in the list. In the latter case, the element is passed in as an argument.



```python
character_names = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Jon Snow", "Tyrion Lannister", "Jon Snow",
                   "Cersei Lannister"]
print(character_names.index("Jon Snow"))
print(character_names.count("Jon Snow"))
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
0
3

Process finished with exit code 0
```

Other useful in-built methods that can help us work with lists are for example len(*list_name*) (returns the number of elements in a list), *list_name*.sorted() (sorts the list automatically and returns the result as a new list; does not modify the original list), *list_name*.sort() (sorts the elements in the list automatically or according to a parameter we pass in as an argument, and overwrites the initial list), *list_name*.reverse() (reverses the order of the elements in the list and overwrites the initial list) and *list_name*.copy() (returns a new list that is a copy of the original; does not modify the original list). See some examples below. Try these methods yourself in the IDE.

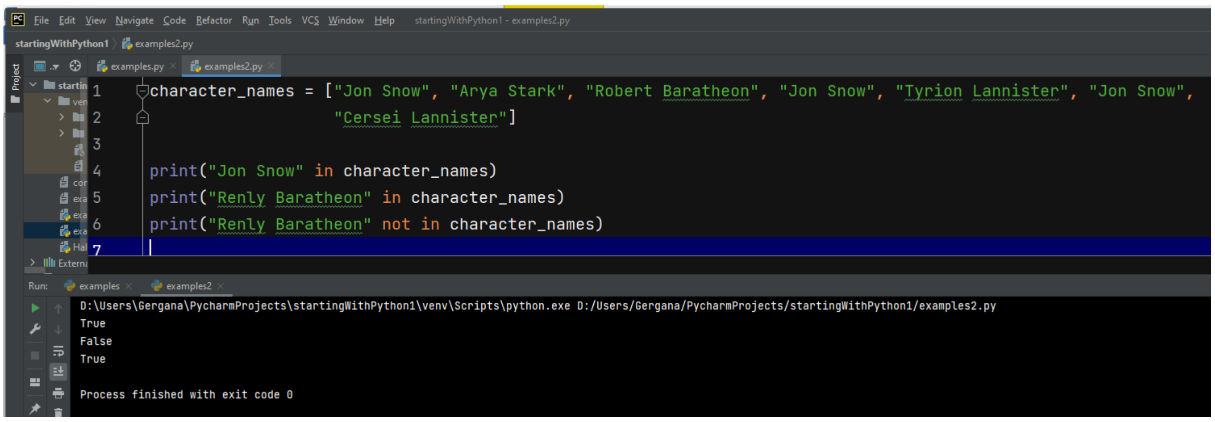


```python
character_names = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Jon Snow", "Tyrion Lannister", "Jon Snow",
                   "Cersei Lannister"]
    # len()
print(len(character_names))
    # sorted() - will automatically sort alphabetically but will not overwrite the original list
sorted_character_names = sorted(character_names)
print("Original list: ", end="")
print(character_names)
print("Sorted list: ", end="")
print(sorted_character_names)
    # sort() - we sort descending the previous list that we created with sorted()
sorted_character_names.sort(reverse=True)
print(sorted_character_names)
    # reverse() - we reverse the original list and will overwrite it
character_names.reverse()
print(character_names)
    # copy()
duplicate_character_list = character_names.copy()
print(duplicate_character_list)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
7
Original list: ['Jon Snow', 'Arya Stark', 'Robert Baratheon', 'Jon Snow', 'Tyrion Lannister', 'Jon Snow', 'Cersei Lannister']
Sorted list: ['Arya Stark', 'Cersei Lannister', 'Jon Snow', 'Jon Snow', 'Jon Snow', 'Robert Baratheon', 'Tyrion Lannister']
['Tyrion Lannister', 'Robert Baratheon', 'Jon Snow', 'Jon Snow', 'Jon Snow', 'Cersei Lannister', 'Arya Stark']
['Cersei Lannister', 'Jon Snow', 'Tyrion Lannister', 'Jon Snow', 'Robert Baratheon', 'Arya Stark', 'Jon Snow']
['Cersei Lannister', 'Jon Snow', 'Tyrion Lannister', 'Jon Snow', 'Robert Baratheon', 'Arya Stark', 'Jon Snow']

Process finished with exit code 0
```

It is often useful to check if an element is present in the list. This is done using the keyword in or the keywords not in. The program returns a Boolean – True or False.

Co-funded by the
Erasmus+ Programme
of the European Union

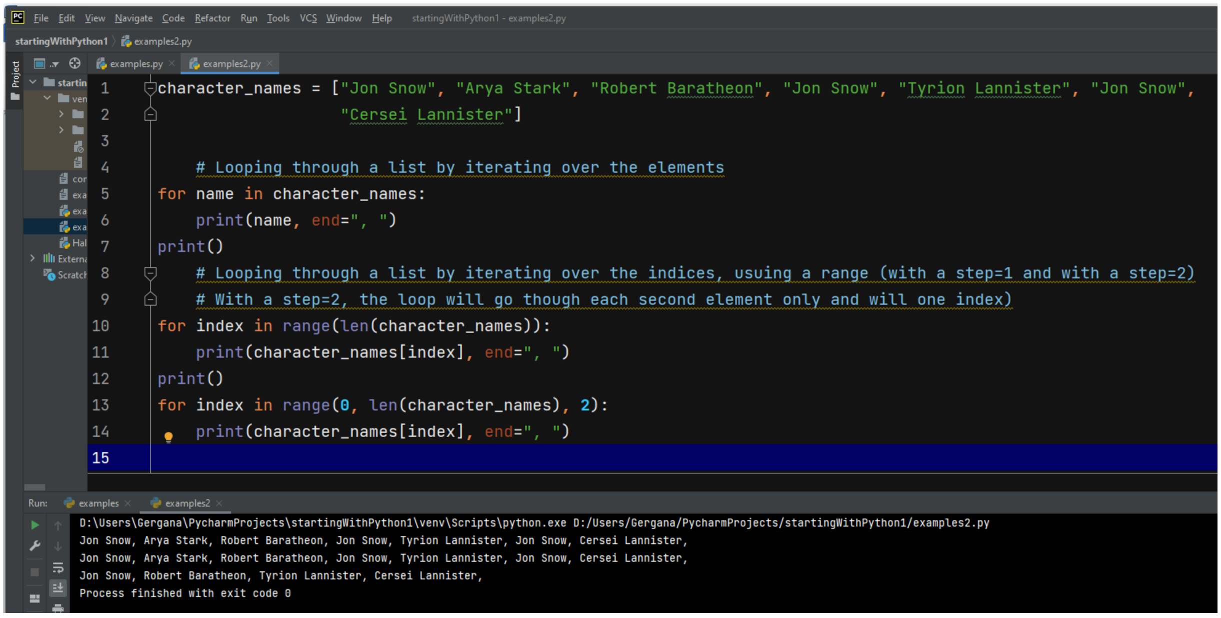


```python
character_names = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Jon Snow", "Tyrion Lannister", "Jon Snow",
                   "Cersei Lannister"]

print("Jon Snow" in character_names)
print("Renly Baratheon" in character_names)
print("Renly Baratheon" not in character_names)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
True
False
True

Process finished with exit code 0
```

Let us now learn how to loop through lists. We can use a *for* loop to iterate over the elements as shown in the first example below. The word "*name*" is a generic name for the variable that holds the value of the current element at each iteration. This variable is assigned a new value during each iteration as we go over the whole list. You can name this variable as appropriate, depending on what is included in your list (e.g. number, temperature, etc.).

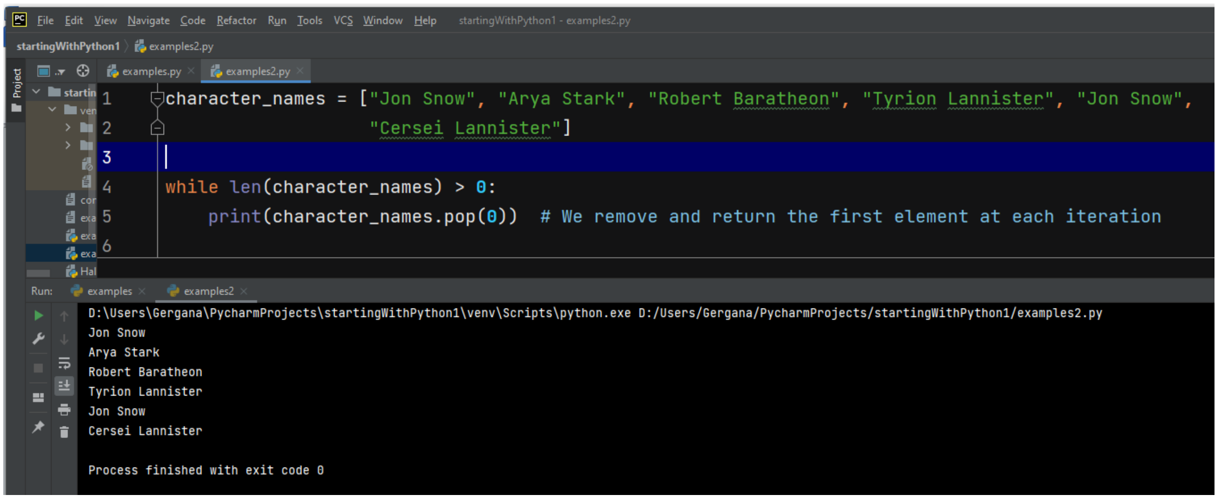
We can also iterate through the indices by defining a range. An additional advantage here is that we can specify a step for the iteration. For example, if we choose a step=2, the loop will skip one index when going to the next iteration and will only go through every second index. The syntax here is for *index* in range(start index, end index, step). If we only pass in one argument, this argument will be regarded as the end index and the loop will iterate from the beginning of the list to that index - 1. Remember that the end index is not included in the loop. If we only pass in two arguments, they will be regarded as start index and end index.



```python
character_names = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Jon Snow", "Tyrion Lannister", "Jon Snow",
                   "Cersei Lannister"]

    # Looping through a list by iterating over the elements
for name in character_names:
    print(name, end=", ")
print()
    # Looping through a list by iterating over the indices, usuing a range (with a step=1 and with a step=2)
    # With a step=2, the loop will go though each second element only and will one index)
for index in range(len(character_names)):
    print(character_names[index], end=", ")
print()
for index in range(0, len(character_names), 2):
    print(character_names[index], end=", ")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Jon Snow, Arya Stark, Robert Baratheon, Jon Snow, Tyrion Lannister, Jon Snow, Cersei Lannister,
Jon Snow, Arya Stark, Robert Baratheon, Jon Snow, Tyrion Lannister, Jon Snow, Cersei Lannister,
Jon Snow, Robert Baratheon, Tyrion Lannister, Cersei Lannister,
Process finished with exit code 0
```

As an alternative, we can use a *while* loop to iterate over a list. However, this would require that we use the pop() function to remove each element that we have already looped over. In the *while* loop, we can thus iterate as long as the list is not empty. When we iterate over the last element, we will remove it with pop() and the list will remain empty, which would end the loop. To check if the list is not empty, we use the len() function.

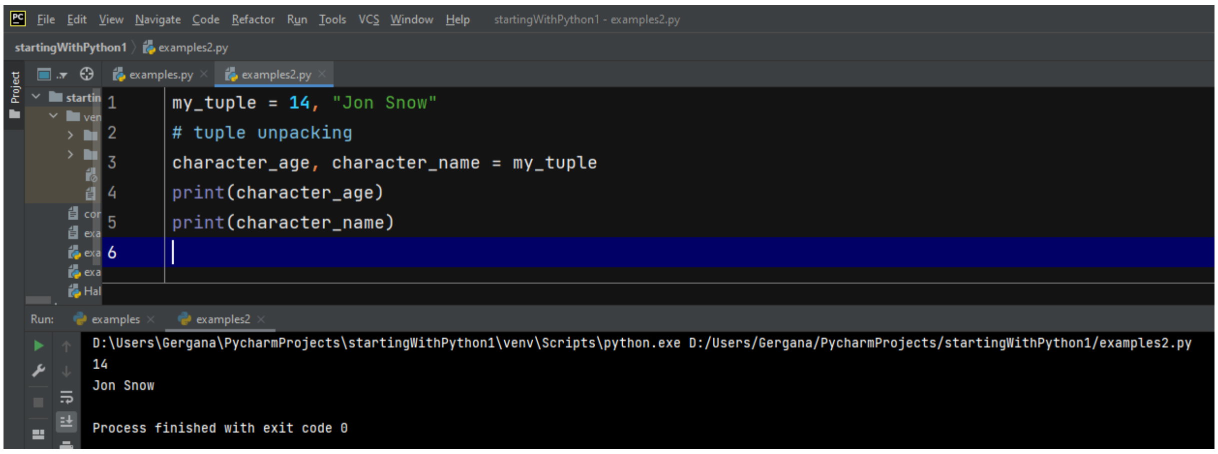


```
character_names = ["Jon Snow", "Arya Stark", "Robert Baratheon", "Tyrion Lannister", "Jon Snow",
                   "Cersei Lannister"]

while len(character_names) > 0:
    print(character_names.pop(0))  # We remove and return the first element at each iteration
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Jon Snow
Arya Stark
Robert Baratheon
Tyrion Lannister
Jon Snow
Cersei Lannister

Process finished with exit code 0
```

Tuples

In Python a tuple is a datatype - a collection of values that are *ordered* and *unchangeable* (*immutable*). It is very similar to a list, with the only difference being that the items are immutable once they have been placed inside the tuple. We also cannot add or remove elements from it. The items in the tuple are separated with comas and the tuple is written inside round brackets (). Duplicate members are allowed. Sometimes you can see tuples created without any brackets but this method is not recommended to use. The items inside a tuple can be of any datatype (e.g. integers, floats, strings and even collections such as other tuples or lists) and we can also create a tuple with different datatypes in it. It is possible to carry out tuple unpacking, by assigning the values that are contained within the tuple to variables. See an example of tuple unpacking.

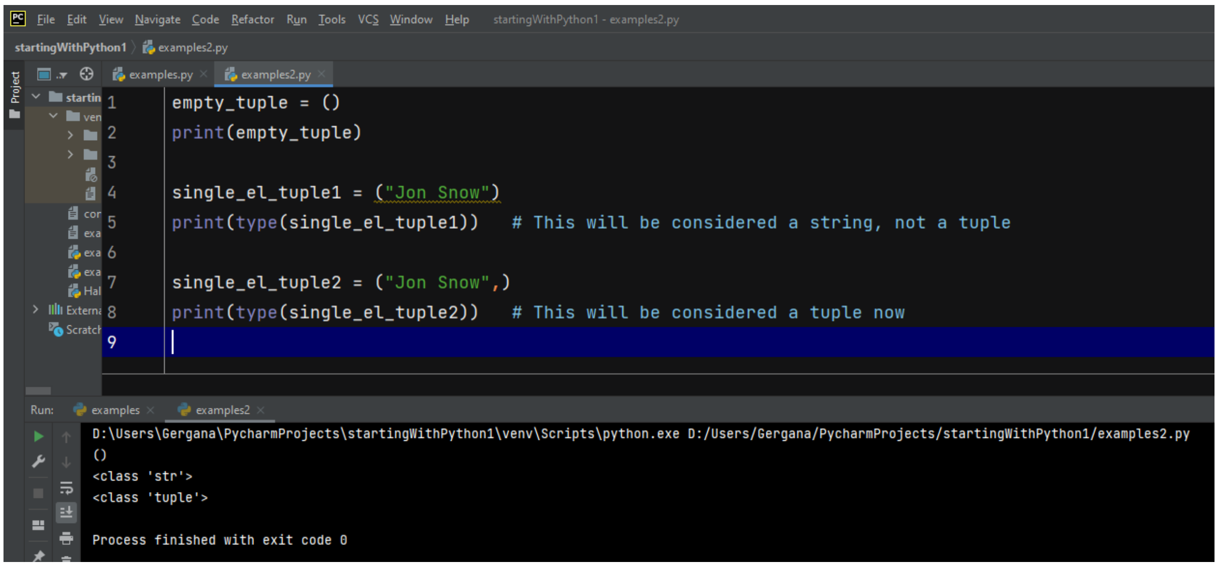


```
my_tuple = 14, "Jon Snow"
# tuple unpacking
character_age, character_name = my_tuple
print(character_age)
print(character_name)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
14
Jon Snow

Process finished with exit code 0
```

To initialize an empty tuple, we assign () to it. To initialize a tuple with one element, it is not enough to put that element in round brackets because Python will regard it datatype to be that of the element itself. We need to add a coma after that single element to indicate that we have a one-element tuple.

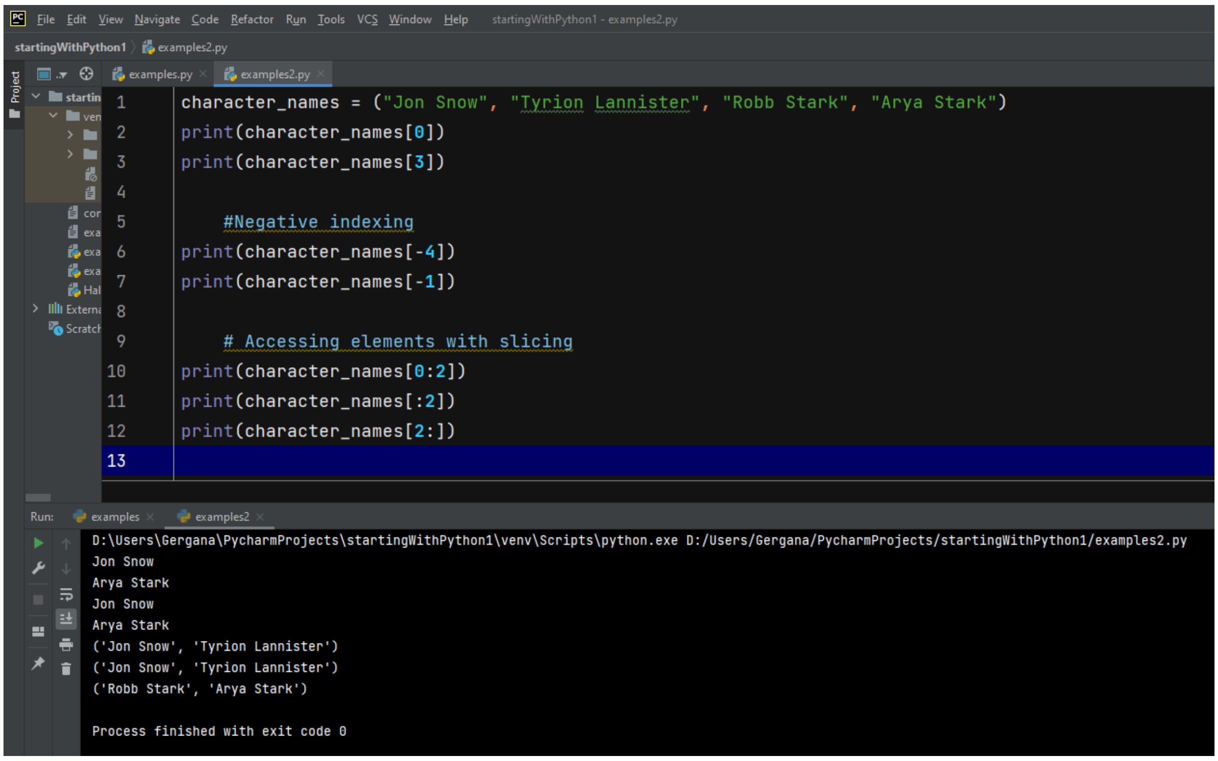
Co-funded by the Erasmus+ Programme of the European Union



```
empty_tuple = ()
print(empty_tuple)

single_el_tuple1 = ("Jon Snow")
print(type(single_el_tuple1))   # This will be considered a string, not a tuple

single_el_tuple2 = ("Jon Snow",)
print(type(single_el_tuple2))   # This will be considered a tuple now
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
()
<class 'str'>
<class 'tuple'>

Process finished with exit code 0
```

Tuple elements are ordered and have indices associated with them. Therefore, each element can be accessed by its index. Just like in strings and lists, indices start from 0, and the last index is the *tuple_length* – 1. Calling an index that is equal to the length of the tuple will result in "Index out of range error". Negative indexing is allowed: the last index is -1 and the first index is -(length of the tuple), i.e., for my_tuple with 6 elements, the first index can be called with my_tuple[-6] and the last index with my_tuple[-1]. More than one element can be accessed with *slicing*, whereby inside the square brackets we place the range of the slice, e.g. if we need from index 1 to index 5 we write [1:6]. Like in strings and lists, the start index is accessed, but end index is not, so in our case the last accessed index will be 5. If we only indicate inside the square brackets one index, followed by a colon, e.g. [2:], then we will access all indices from the indicated index (including it) until the end of the tuple. Similarly, if we indicate one index preceded by a colon, we will access all indices from the beginning of the tuple until the indicated index (excluding this index). See all of this demonstrated in the example.



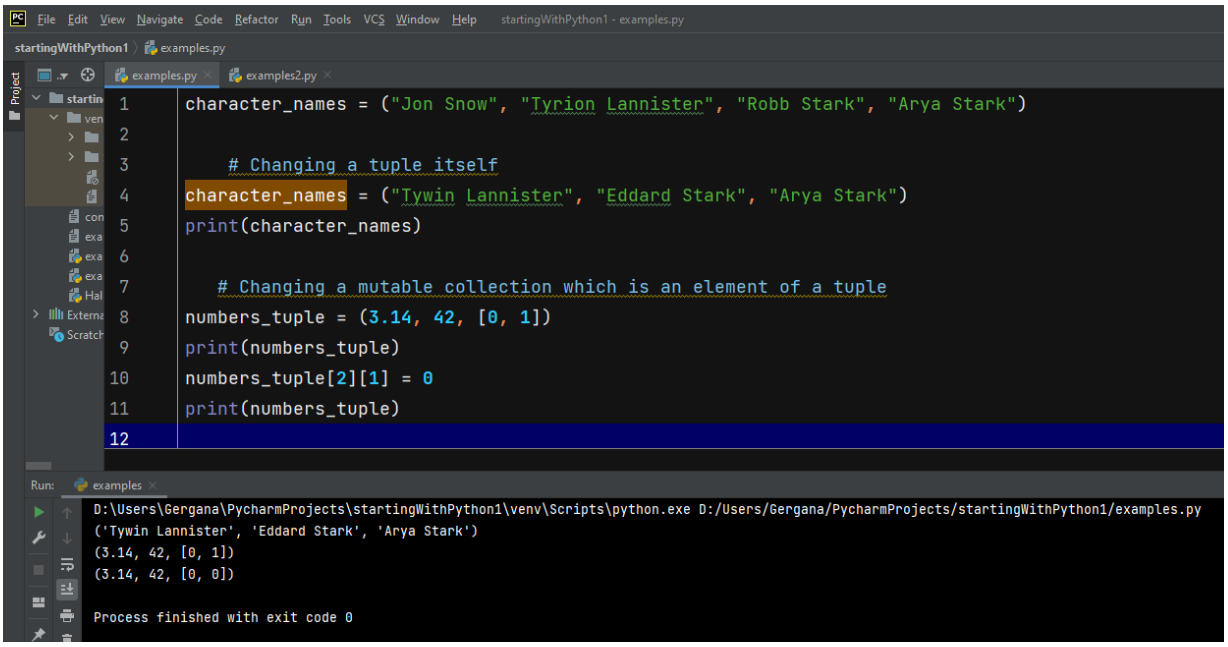
```
character_names = ("Jon Snow", "Tyrion Lannister", "Robb Stark", "Arya Stark")
print(character_names[0])
print(character_names[3])

    #Negative indexing
print(character_names[-4])
print(character_names[-1])

    # Accessing elements with slicing
print(character_names[0:2])
print(character_names[:2])
print(character_names[2:])
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Jon Snow
Arya Stark
Jon Snow
Arya Stark
('Jon Snow', 'Tyrion Lannister')
('Jon Snow', 'Tyrion Lannister')
('Robb Stark', 'Arya Stark')

Process finished with exit code 0
```

Co-funded by the Erasmus+ Programme of the European Union

Unlike the elements of a list, the elements of a tuple cannot be changed. The only exception to this rule is when the element is a mutable collection, such as a list. To do this, we access it with two square brackets [][] containing respectively the index of the mutable collection (e.g. a list) and the index of the element in that mutable collection that we want to change. We then assign this element a new value.

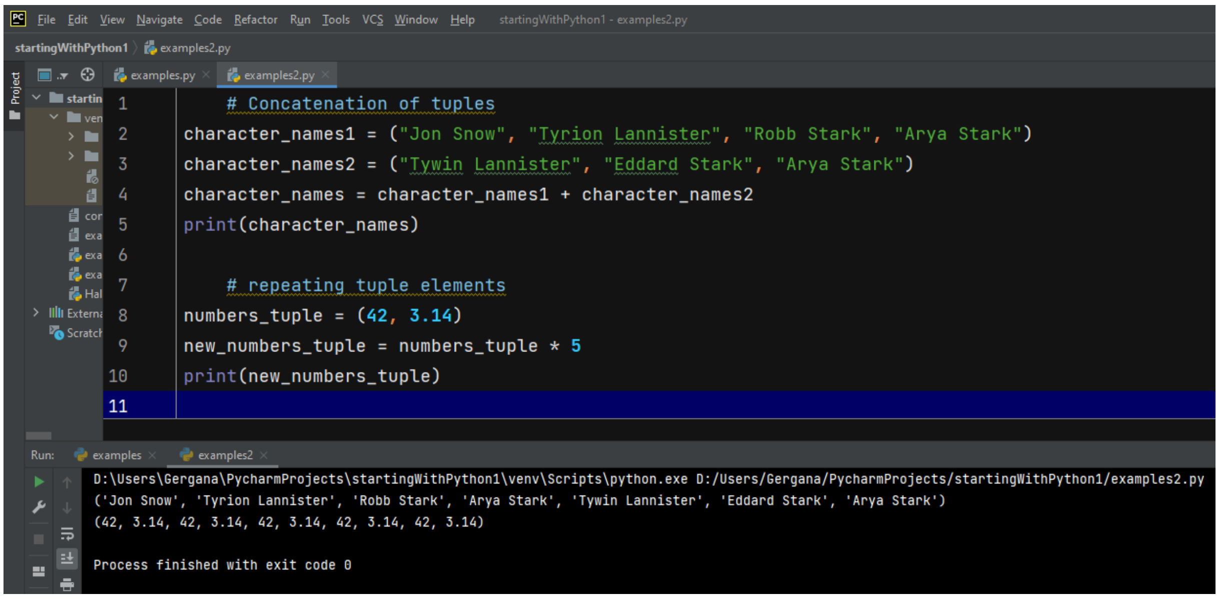
We can change the tuple itself if we assign it different values. Look at the examples below and try it yourself in PyCharm.



```
character_names = ("Jon Snow", "Tyrion Lannister", "Robb Stark", "Arya Stark")

    # Changing a tuple itself
character_names = ("Tywin Lannister", "Eddard Stark", "Arya Stark")
print(character_names)

   # Changing a mutable collection which is an element of a tuple
numbers_tuple = (3.14, 42, [0, 1])
print(numbers_tuple)
numbers_tuple[2][1] = 0
print(numbers_tuple)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
('Tywin Lannister', 'Eddard Stark', 'Arya Stark')
(3.14, 42, [0, 1])
(3.14, 42, [0, 0])

Process finished with exit code 0
```

We can combine multiple tuples together with the + operator, and the action is called *concatenation*. If the concatenated tuples contain some identical elements, these elements will be duplicated. We can also replicate all elements of a tuple using the * operator followed by a number indicating how many times the elements will be repeated. Both of these operations create a new tuple. See the examples below.



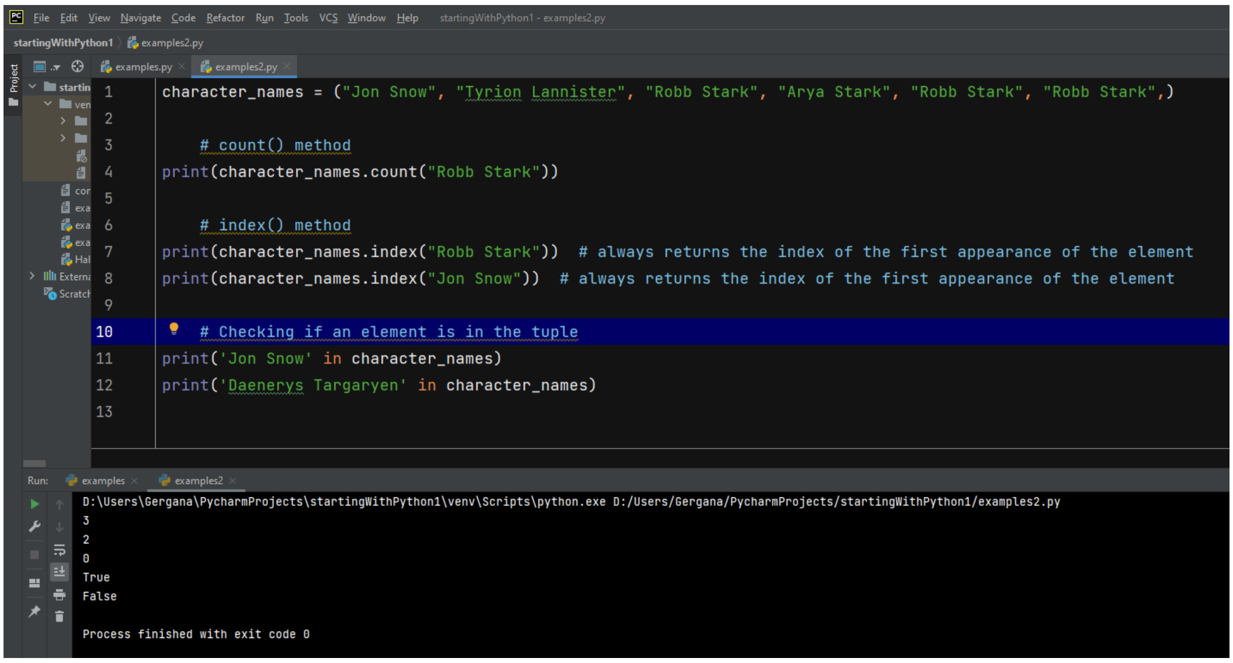
```
    # Concatenation of tuples
character_names1 = ("Jon Snow", "Tyrion Lannister", "Robb Stark", "Arya Stark")
character_names2 = ("Tywin Lannister", "Eddard Stark", "Arya Stark")
character_names = character_names1 + character_names2
print(character_names)

    # repeating tuple elements
numbers_tuple = (42, 3.14)
new_numbers_tuple = numbers_tuple * 5
print(new_numbers_tuple)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
('Jon Snow', 'Tyrion Lannister', 'Robb Stark', 'Arya Stark', 'Tywin Lannister', 'Eddard Stark', 'Arya Stark')
(42, 3.14, 42, 3.14, 42, 3.14, 42, 3.14, 42, 3.14)

Process finished with exit code 0
```

We cannot delete or add elements in the tuple. We can only delete the whole tuple with the del *my_tuple* command. Try it yourself in PyCharm.

Co-funded by the Erasmus+ Programme of the European Union

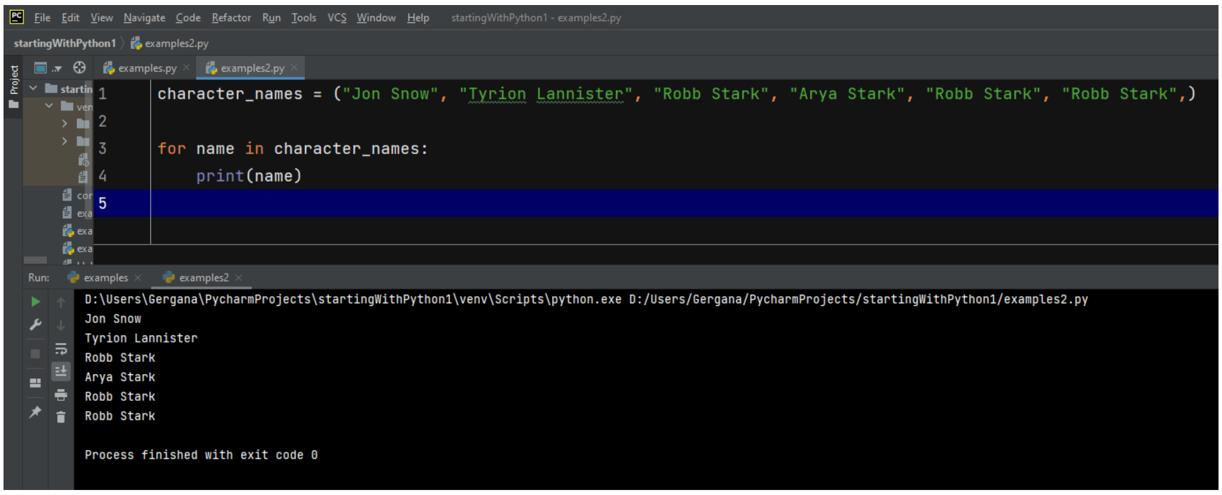
There are several methods available in Python that help us work with tuples. The first one is the count(*element*) method. We pass in it as an argument an element of the tuple and the method returns the number of times the specified element appears in the tuple. The index(*element*) method takes as an argument an element of the tuple and returns the index of the first appearance of this element. There is a method to also check if an element is found in the tuple or not. The method returns True or False. The syntax is *my_element* in *my_tuple* or *my_element* not in *my_tuple*. Essentially these methods are identical to those used in lists. See the examples below and try them yourself.



```python
character_names = ("Jon Snow", "Tyrion Lannister", "Robb Stark", "Arya Stark", "Robb Stark", "Robb Stark",)

    # count() method
print(character_names.count("Robb Stark"))

    # index() method
print(character_names.index("Robb Stark"))  # always returns the index of the first appearance of the element
print(character_names.index("Jon Snow"))  # always returns the index of the first appearance of the element

    # Checking if an element is in the tuple
print('Jon Snow' in character_names)
print('Daenerys Targaryen' in character_names)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
3
2
0
True
False

Process finished with exit code 0
```

Lastly, let us learn how to iterate over a tuple. This is also the same as for lists. Remember that the syntax for a *for* loop is: for keyword, followed by a name for the variable representing each current element, followed by the in keyword and followed by the name of the tuple variable (or the tuple itself written in round brackets). Now, inside the *for* loop, we can write code using the current element retrieved during the current iteration. See the example.



```python
character_names = ("Jon Snow", "Tyrion Lannister", "Robb Stark", "Arya Stark", "Robb Stark", "Robb Stark",)

for name in character_names:
    print(name)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Jon Snow
Tyrion Lannister
Robb Stark
Arya Stark
Robb Stark
Robb Stark

Process finished with exit code 0
```

Tip on why we could choose to use a tuple:

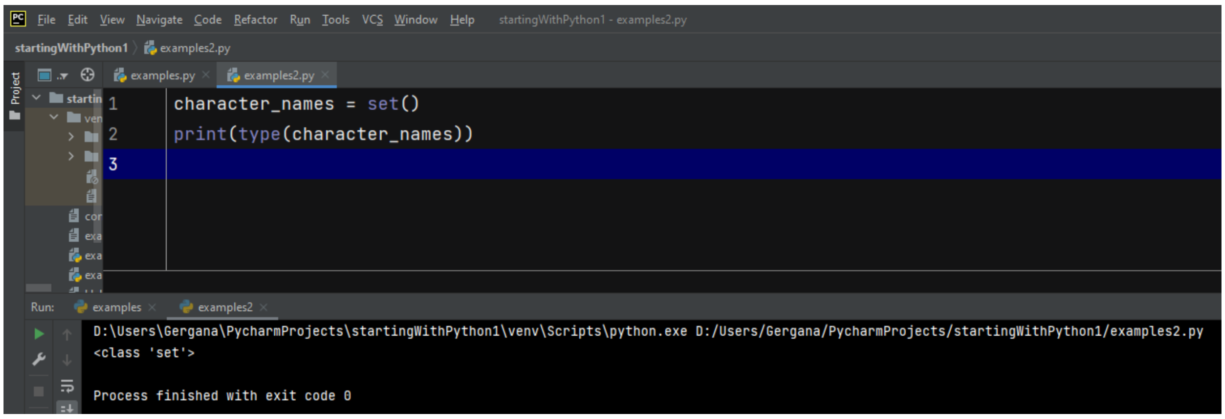
Tuples contain immutable elements and can be used to provide the keys for a dictionary. Lists cannot be used like this.

Co-funded by the Erasmus+ Programme of the European Union

Sets

In Python a set is a datatype - a collection of items that are *unordered* and *immutable*. However, the set itself is mutable, so we can add or remove items from it. Since the items are unordered, we cannot use the index of an item to retrieve it. The items in the set are separated with comas and the tuple is written inside curly brackets {}. The items inside a set can be of any datatype (e.g. integers, strings) and it is possible to also create a set with different datatypes in it. The elements in a set are unique and if we try to add duplicates, they will be disregarded.

If we need to initialize an empty set, we cannot do it by assigning empty curly brackets because this will create an empty dictionary. We can create an empty set with the set() function without passing any argument in it.


File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples2.py
startingWithPython1 examples2.py
examples.py
examples2.py
Run: examples examples2

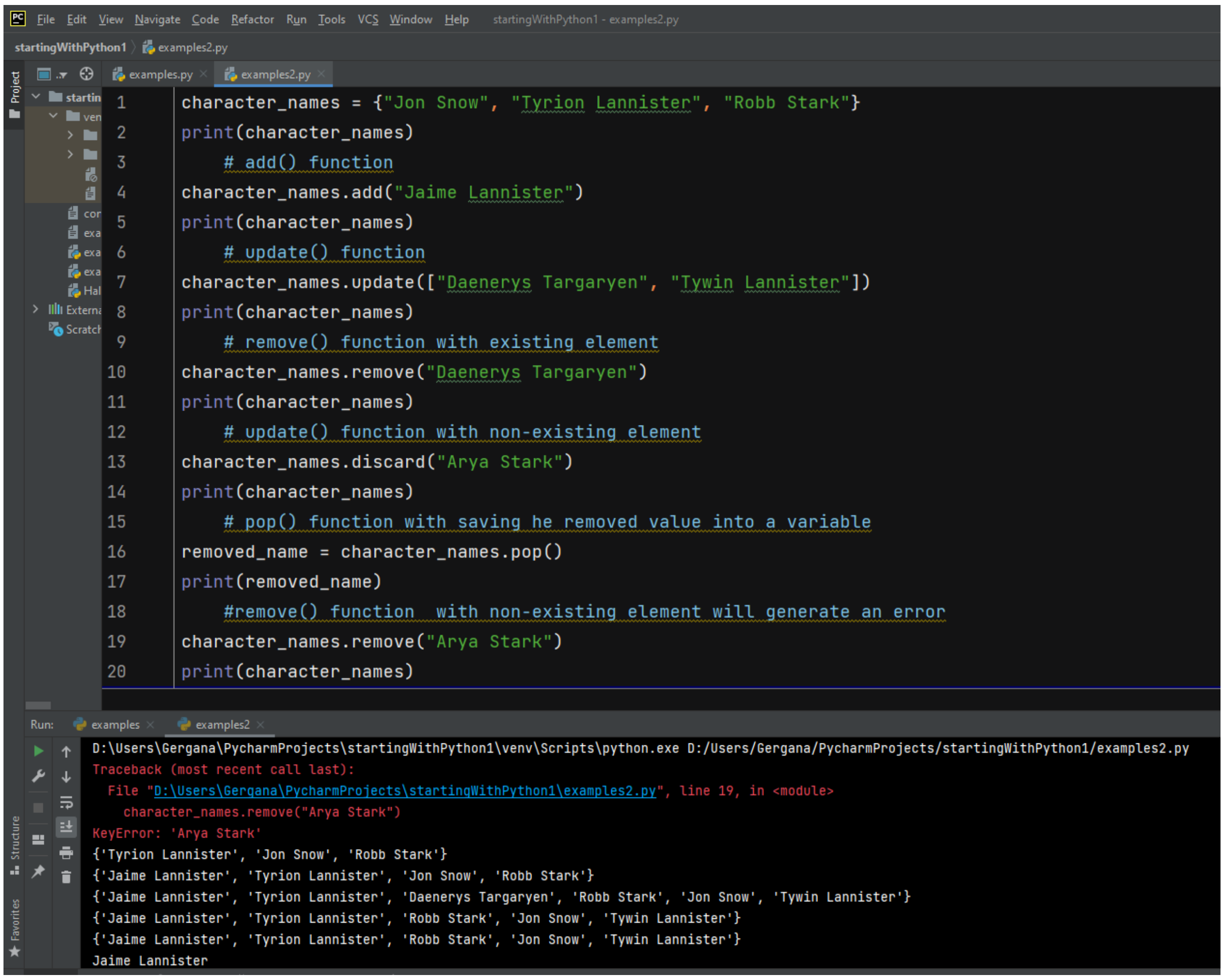
```
character_names = set()
print(type(character_names))
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
<class 'set'>

Process finished with exit code 0
```

We can add items to the set by using the add(*element*) method and passing in it the new item as an argument. If we need to add multiple items to the set, we use the update([*element_1, element_2*]) method and add these items as arguments, separated with comas and enclosed in square brackets. In the examples below, you will notice that the printed set has a different order than the order in which we added the elements. In fact, each new execution of the print() command will print the items in a different order.

We can remove elements from the set with the discard(*element*) and remove(*element*) methods. Both methods will remove from the set the element that we pass in as an argument. However, if the item we are trying to remove does not exist in the set, discard() will do nothing while remove() will generate an error to warn about attempting to remove an element that does not exist. The choice between the two options will depend on whether or not we want the program to crash and indicate an error, or to just try to remove the element and continue without interruption if the removal was unsuccessful. The pop() method removes an element from the set and returns it to the program, but it does so with a random element, because the set is unordered. Notice that the pop() method returns the value that it deletes, so you can use this value in your program. The examples below illustrate all the functions.

Co-funded by the Erasmus+ Programme of the European Union



```python
character_names = {"Jon Snow", "Tyrion Lannister", "Robb Stark"}
print(character_names)
    # add() function
character_names.add("Jaime Lannister")
print(character_names)
    # update() function
character_names.update(["Daenerys Targaryen", "Tywin Lannister"])
print(character_names)
    # remove() function with existing element
character_names.remove("Daenerys Targaryen")
print(character_names)
    # update() function with non-existing element
character_names.discard("Arya Stark")
print(character_names)
    # pop() function with saving he removed value into a variable
removed_name = character_names.pop()
print(removed_name)
    #remove() function  with non-existing element will generate an error
character_names.remove("Arya Stark")
print(character_names)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Traceback (most recent call last):
  File "D:\Users\Gergana\PycharmProjects\startingWithPython1\examples2.py", line 19, in <module>
    character_names.remove("Arya Stark")
KeyError: 'Arya Stark'
{'Tyrion Lannister', 'Jon Snow', 'Robb Stark'}
{'Jaime Lannister', 'Tyrion Lannister', 'Jon Snow', 'Robb Stark'}
{'Jaime Lannister', 'Tyrion Lannister', 'Daenerys Targaryen', 'Robb Stark', 'Jon Snow', 'Tywin Lannister'}
{'Jaime Lannister', 'Tyrion Lannister', 'Robb Stark', 'Jon Snow', 'Tywin Lannister'}
{'Jaime Lannister', 'Tyrion Lannister', 'Robb Stark', 'Jon Snow', 'Tywin Lannister'}
Jaime Lannister
```

We can also remove all the items from a set using the clear() method. A copy of the set can be created with the copy() method.

If we have multiple sets we can manipulate them through mathematical operations - *union*, *intersection*, *difference* and *symmetric difference*.

A *union* of two sets creates a new set containing all elements from both sets. However, duplicates will be removed. *Union* is performed by using the | operator or the union() method.

An *intersection* of two sets creates a new set containing only the elements that are common in the two initial sets. *Intersection* is performed by using the & operator or the intersection() method.

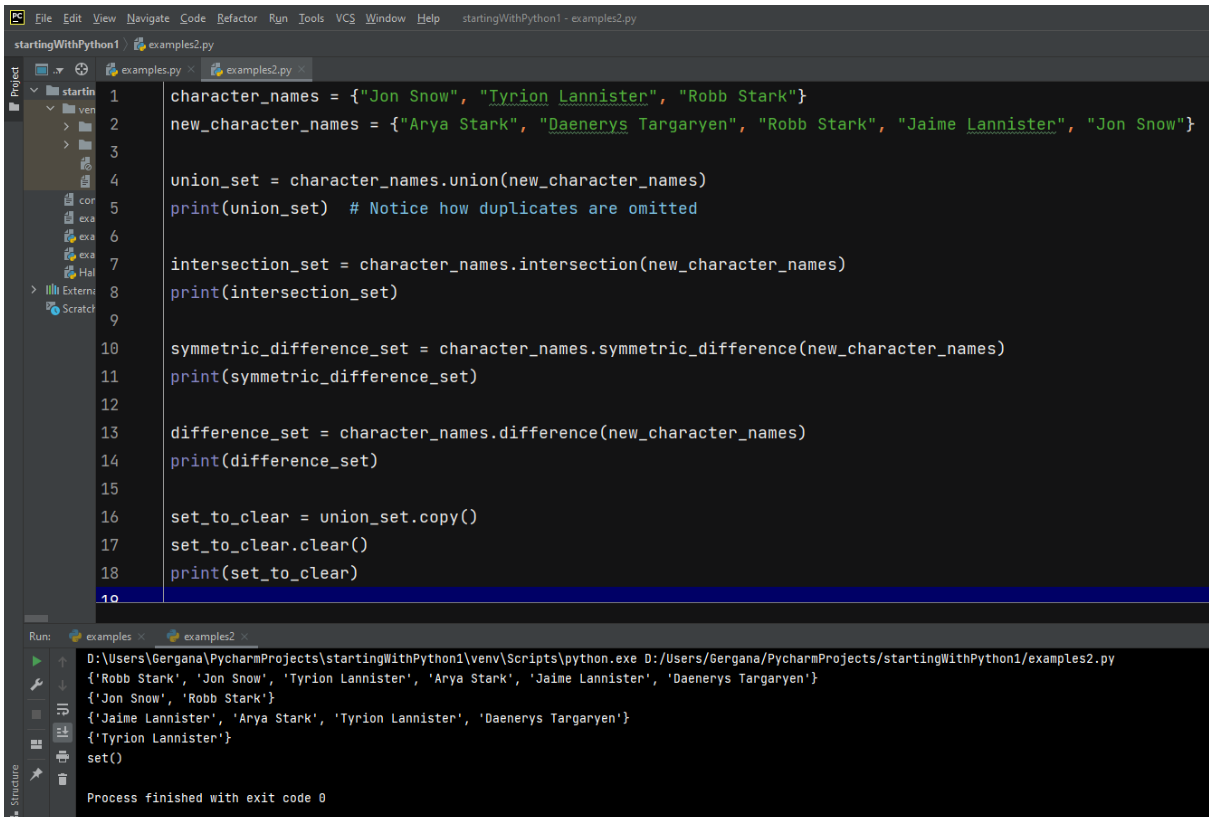
A *difference* of two sets creates a new set containing only the elements that are in the first set but are not present in the second set. *Difference* is performed by using the – operator or the difference() method. Unlike in the other two operations, here we need to be careful which set we are subtracting from and which one is subtracted – e.g. Set 1 – Set 2 will be very different from Set 2 – Set 1.

A *symmetric difference* of two sets creates a new set containing only the elements that are in the first or the second set, but not in both of them (i.e. common elements will be excluded from the new set, so this is the opposite of *intersection*). *Symmetric difference* is performed by using the ^ operator or the symmetric_difference() method.

The same methods can be used with _update() added to them – intersection_update(), difference_update(), symmetric_difference_update (), union_update() – in which case the first set will be replaced by the respective result from the operation.

Co-funded by the Erasmus+ Programme of the European Union

Bellow we show the *union*, *intersection*, *difference* and *symmetric difference* methods, as well as the copy() and clear() methods. You can test all the rest yourself.



```python
character_names = {"Jon Snow", "Tyrion Lannister", "Robb Stark"}
new_character_names = {"Arya Stark", "Daenerys Targaryen", "Robb Stark", "Jaime Lannister", "Jon Snow"}

union_set = character_names.union(new_character_names)
print(union_set)  # Notice how duplicates are omitted

intersection_set = character_names.intersection(new_character_names)
print(intersection_set)

symmetric_difference_set = character_names.symmetric_difference(new_character_names)
print(symmetric_difference_set)

difference_set = character_names.difference(new_character_names)
print(difference_set)

set_to_clear = union_set.copy()
set_to_clear.clear()
print(set_to_clear)
```

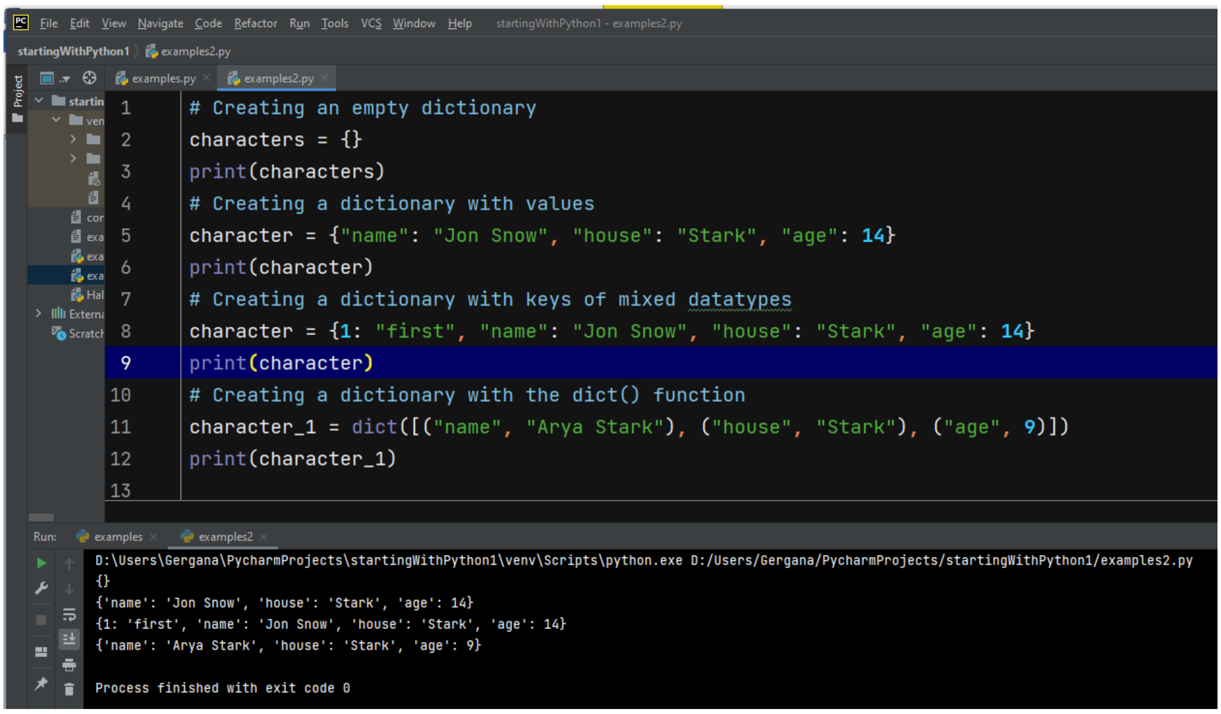
```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
{'Robb Stark', 'Jon Snow', 'Tyrion Lannister', 'Arya Stark', 'Jaime Lannister', 'Daenerys Targaryen'}
{'Jon Snow', 'Robb Stark'}
{'Jaime Lannister', 'Arya Stark', 'Tyrion Lannister', 'Daenerys Targaryen'}
{'Tyrion Lannister'}
set()

Process finished with exit code 0
```

Dictionaries

In Python a dictionary is a datatype - a collection of values that are *ordered* (as of Python 3.7) and *changeable* (*mutable*). It consists of pairs of a key and a value associated with it (*key: value*). The key-value pairs in the dictionary are separated with comas and the dictionary is written inside curly brackets {}. We can also create a dictionary with the dict() function, in which we pass as an argument a sequence of pairs. See the examples below.

Co-funded by the Erasmus+ Programme of the European Union

The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

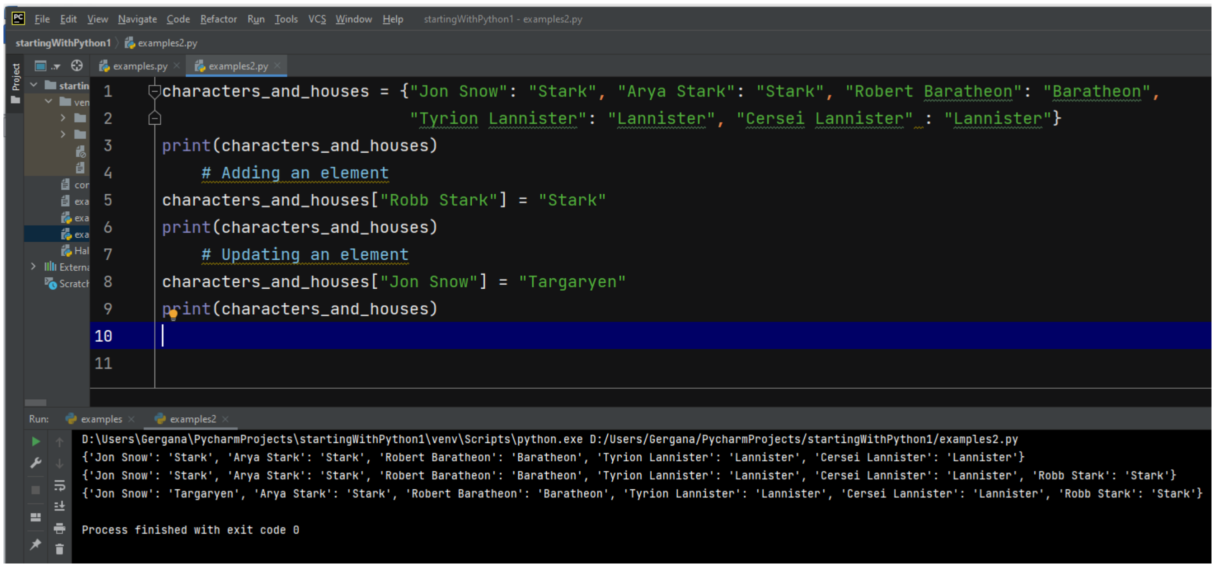


```python
# Creating an empty dictionary
characters = {}
print(characters)
# Creating a dictionary with values
character = {"name": "Jon Snow", "house": "Stark", "age": 14}
print(character)
# Creating a dictionary with keys of mixed datatypes
character = {1: "first", "name": "Jon Snow", "house": "Stark", "age": 14}
print(character)
# Creating a dictionary with the dict() function
character_1 = dict([("name", "Arya Stark"), ("house", "Stark"), ("age", 9)])
print(character_1)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
{}
{'name': 'Jon Snow', 'house': 'Stark', 'age': 14}
{1: 'first', 'name': 'Jon Snow', 'house': 'Stark', 'age': 14}
{'name': 'Arya Stark', 'house': 'Stark', 'age': 9}

Process finished with exit code 0
```

A key must be a datatype that is immutable (a string, a number or a tuple with immutable elements). The different keys in the same dictionary, however, can be of different datatypes. They must be unique; if we add to the dictionary a pair with a key that is identical to an already existing key, the existing key-value pair will be overwritten /updated (the value associated with the original key will be replaced with the newly provided value for this key. The values in the key-value pairs can be of any datatype and can be duplicates.

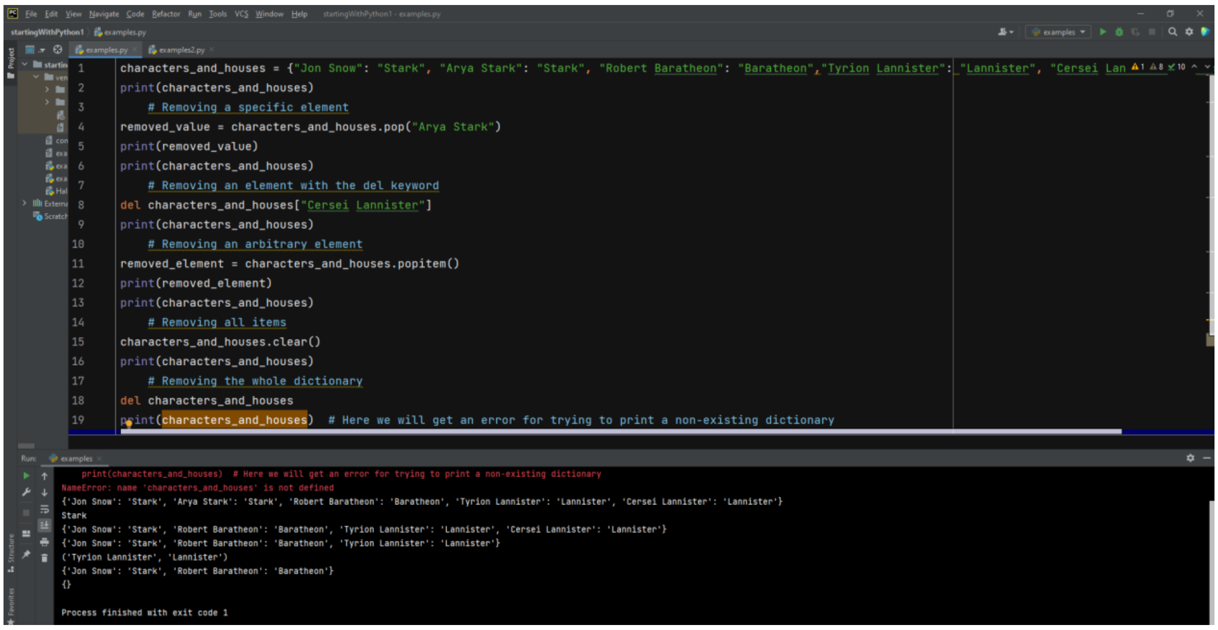
In dictionaries, we do not use indices to access elements; we use the keys. We can add new elements in a dictionary, or change the value of existing elements. The syntax for adding and for updating is essentially the same – *name_of_dictionary*[*key*] = *value*. You notice that instead of placing the index inside the square brackets, we place the key there. See the examples below.



```python
characters_and_houses = {"Jon Snow": "Stark", "Arya Stark": "Stark", "Robert Baratheon": "Baratheon",
                         "Tyrion Lannister": "Lannister", "Cersei Lannister" : "Lannister"}
print(characters_and_houses)
    # Adding an element
characters_and_houses["Robb Stark"] = "Stark"
print(characters_and_houses)
    # Updating an element
characters_and_houses["Jon Snow"] = "Targaryen"
print(characters_and_houses)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
{'Jon Snow': 'Stark', 'Arya Stark': 'Stark', 'Robert Baratheon': 'Baratheon', 'Tyrion Lannister': 'Lannister', 'Cersei Lannister': 'Lannister'}
{'Jon Snow': 'Stark', 'Arya Stark': 'Stark', 'Robert Baratheon': 'Baratheon', 'Tyrion Lannister': 'Lannister', 'Cersei Lannister': 'Lannister', 'Robb Stark': 'Stark'}
{'Jon Snow': 'Targaryen', 'Arya Stark': 'Stark', 'Robert Baratheon': 'Baratheon', 'Tyrion Lannister': 'Lannister', 'Cersei Lannister': 'Lannister', 'Robb Stark': 'Stark'}

Process finished with exit code 0
```

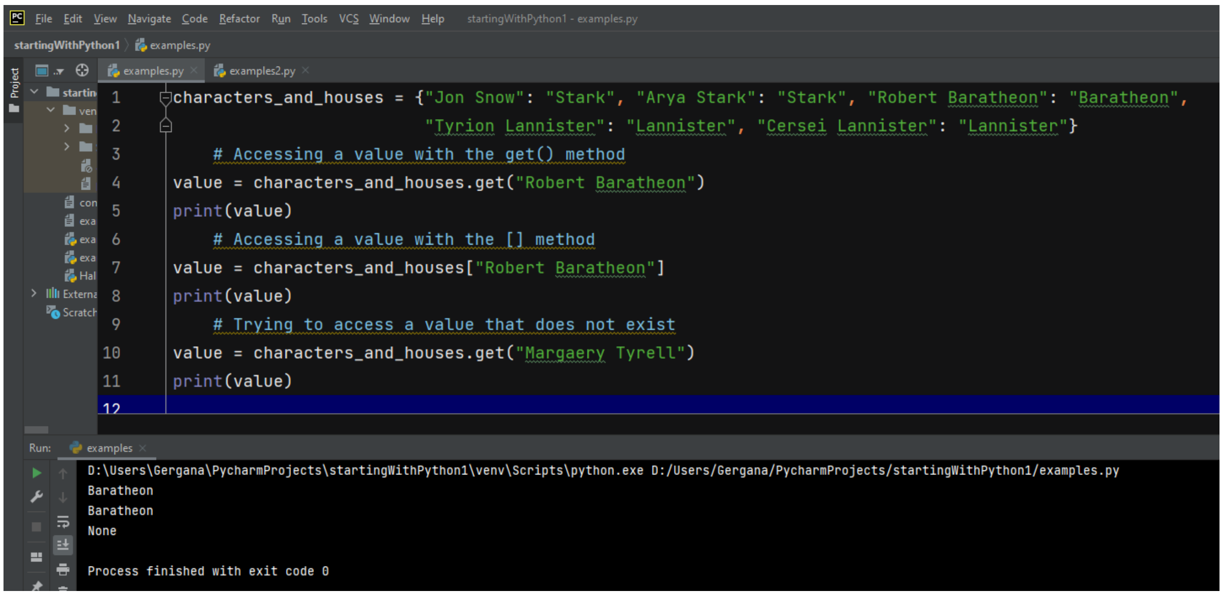
We can also remove items from the dictionary. There are different methods to achieve this - pop(), popitem() and del. Which method we use depends on what we want to achieve in our program. The pop(*key*) method removes a specified item (we specify the key that will be removed) and returns its *value*, which means that we can use it in the program. The popitem() method removes an arbitrary key-value pair and returns it. The del keyword can remove an individual item (del *dict_name*[*key*]) or the dictionary itself. All items from a dictionary can be removed at once with the clear() method, in which case the dictionary will remain empty. See the examples below.



```python
characters_and_houses = {"Jon Snow": "Stark", "Arya Stark": "Stark", "Robert Baratheon": "Baratheon","Tyrion Lannister": "Lannister", "Cersei Lan
print(characters_and_houses)
    # Removing a specific element
removed_value = characters_and_houses.pop("Arya Stark")
print(removed_value)
print(characters_and_houses)
    # Removing an element with the del keyword
del characters_and_houses["Cersei Lannister"]
print(characters_and_houses)
    # Removing an arbitrary element
removed_element = characters_and_houses.popitem()
print(removed_element)
print(characters_and_houses)
    # Removing all items
characters_and_houses.clear()
print(characters_and_houses)
    # Removing the whole dictionary
del characters_and_houses
print(characters_and_houses)  # Here we will get an error for trying to print a non-existing dictionary
```

```
    print(characters_and_houses)  # Here we will get an error for trying to print a non-existing dictionary
NameError: name 'characters_and_houses' is not defined
{'Jon Snow': 'Stark', 'Arya Stark': 'Stark', 'Robert Baratheon': 'Baratheon', 'Tyrion Lannister': 'Lannister', 'Cersei Lannister': 'Lannister'}
Stark
{'Jon Snow': 'Stark', 'Robert Baratheon': 'Baratheon', 'Tyrion Lannister': 'Lannister', 'Cersei Lannister': 'Lannister'}
{'Jon Snow': 'Stark', 'Robert Baratheon': 'Baratheon', 'Tyrion Lannister': 'Lannister'}
('Tyrion Lannister', 'Lannister')
{'Jon Snow': 'Stark', 'Robert Baratheon': 'Baratheon'}
{}

Process finished with exit code 1
```

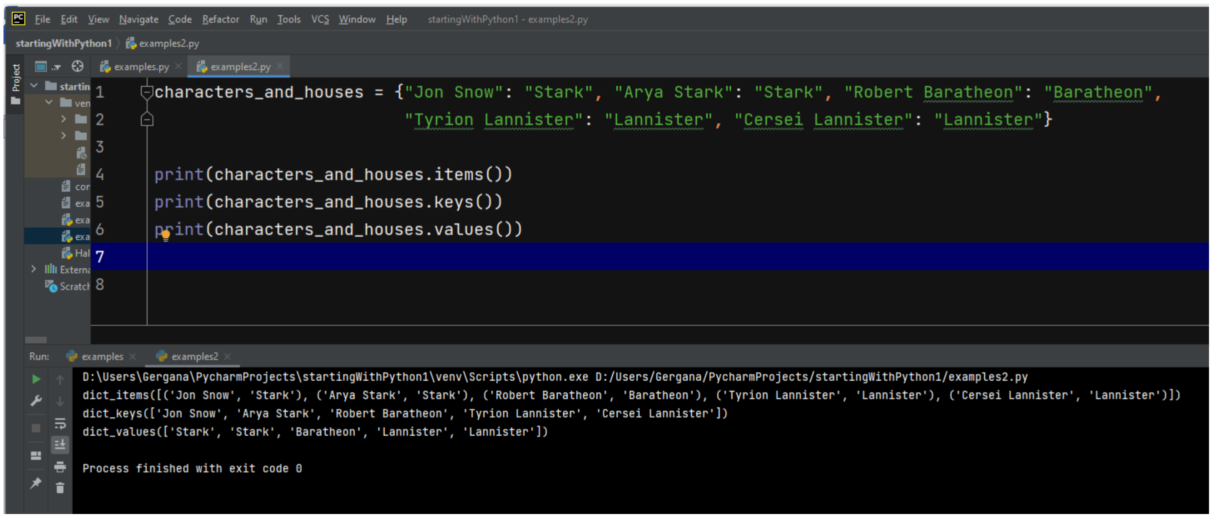
To get the value of a specific key, we use the get(key). If the key does not exist, None is returned by default. We can also access a value with *dict_name*[*key*] method, as already explained above. With this method, if the key is not found, we will get an error and the program will terminate.



```python
characters_and_houses = {"Jon Snow": "Stark", "Arya Stark": "Stark", "Robert Baratheon": "Baratheon",
                         "Tyrion Lannister": "Lannister", "Cersei Lannister": "Lannister"}
    # Accessing a value with the get() method
value = characters_and_houses.get("Robert Baratheon")
print(value)
    # Accessing a value with the [] method
value = characters_and_houses["Robert Baratheon"]
print(value)
    # Trying to access a value that does not exist
value = characters_and_houses.get("Margaery Tyrell")
print(value)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Baratheon
Baratheon
None

Process finished with exit code 0
```

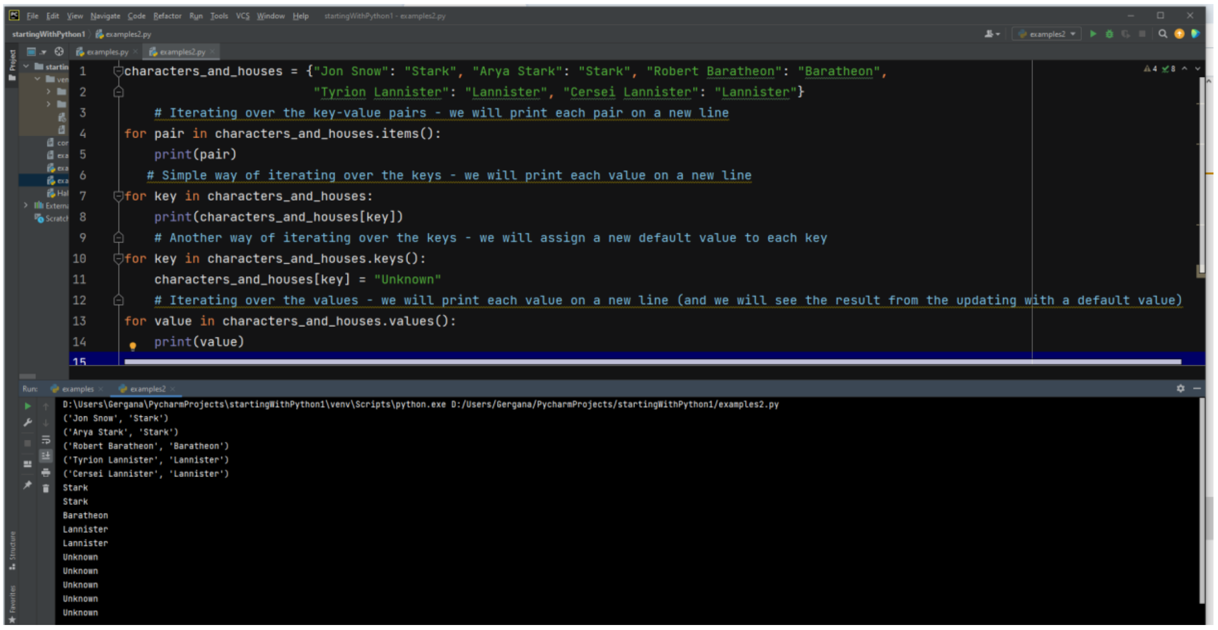
There are several methods that allow us to get all the key-value pairs (items()), all the values (values()) or all the keys (keys()) of a dictionary (e.g. in order to iterate over them). You will notice that these methods return lists containing the information we need. See the examples below. There are also other methods that you can learn gradually as you start working with dictionaries.



```python
characters_and_houses = {"Jon Snow": "Stark", "Arya Stark": "Stark", "Robert Baratheon": "Baratheon",
                         "Tyrion Lannister": "Lannister", "Cersei Lannister": "Lannister"}

print(characters_and_houses.items())
print(characters_and_houses.keys())
print(characters_and_houses.values())
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
dict_items([('Jon Snow', 'Stark'), ('Arya Stark', 'Stark'), ('Robert Baratheon', 'Baratheon'), ('Tyrion Lannister', 'Lannister'), ('Cersei Lannister', 'Lannister')])
dict_keys(['Jon Snow', 'Arya Stark', 'Robert Baratheon', 'Tyrion Lannister', 'Cersei Lannister'])
dict_values(['Stark', 'Stark', 'Baratheon', 'Lannister', 'Lannister'])

Process finished with exit code 0
```

Let us now see how we can iterate over a dictionary. We can iterate over the key-value pairs, over the values or over the keys, using the results from the methods shown above. There is also a simple way for iterating over the keys, namely for *key* in *dictionary_name*:.

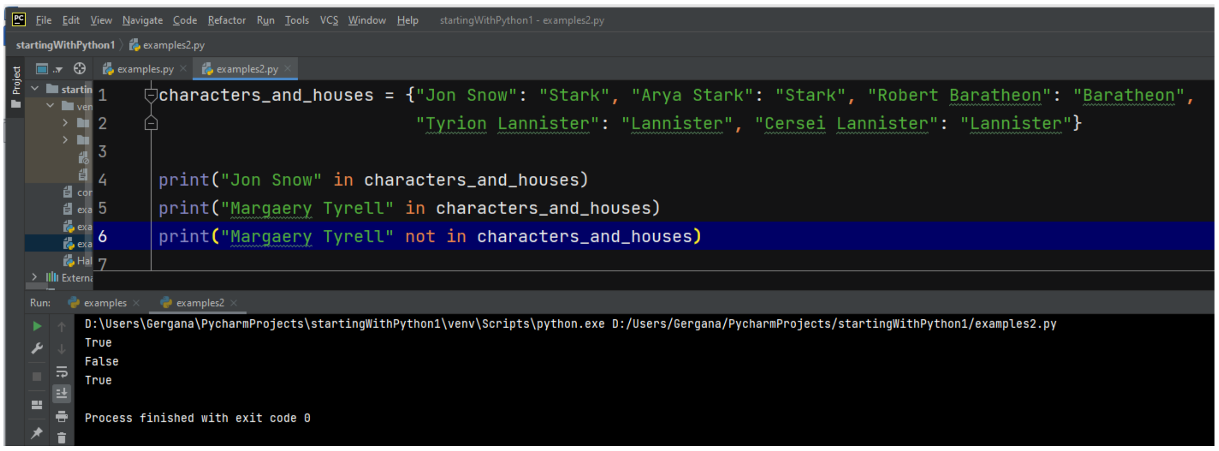


```python
characters_and_houses = {"Jon Snow": "Stark", "Arya Stark": "Stark", "Robert Baratheon": "Baratheon",
                         "Tyrion Lannister": "Lannister", "Cersei Lannister": "Lannister"}
    # Iterating over the key-value pairs - we will print each pair on a new line
for pair in characters_and_houses.items():
    print(pair)
   # Simple way of iterating over the keys - we will print each value on a new line
for key in characters_and_houses:
    print(characters_and_houses[key])
    # Another way of iterating over the keys - we will assign a new default value to each key
for key in characters_and_houses.keys():
    characters_and_houses[key] = "Unknown"
    # Iterating over the values - we will print each value on a new line (and we will see the result from the updating with a default value)
for value in characters_and_houses.values():
    print(value)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
('Jon Snow', 'Stark')
('Arya Stark', 'Stark')
('Robert Baratheon', 'Baratheon')
('Tyrion Lannister', 'Lannister')
('Cersei Lannister', 'Lannister')
Stark
Stark
Baratheon
Lannister
Lannister
Unknown
Unknown
Unknown
Unknown
Unknown
```

It is sometimes useful to check if a key is present in the dictionary. This is done using the keyword in or the keywords not in. The syntax is the same as for lists. The program will return a Boolean – True or False.

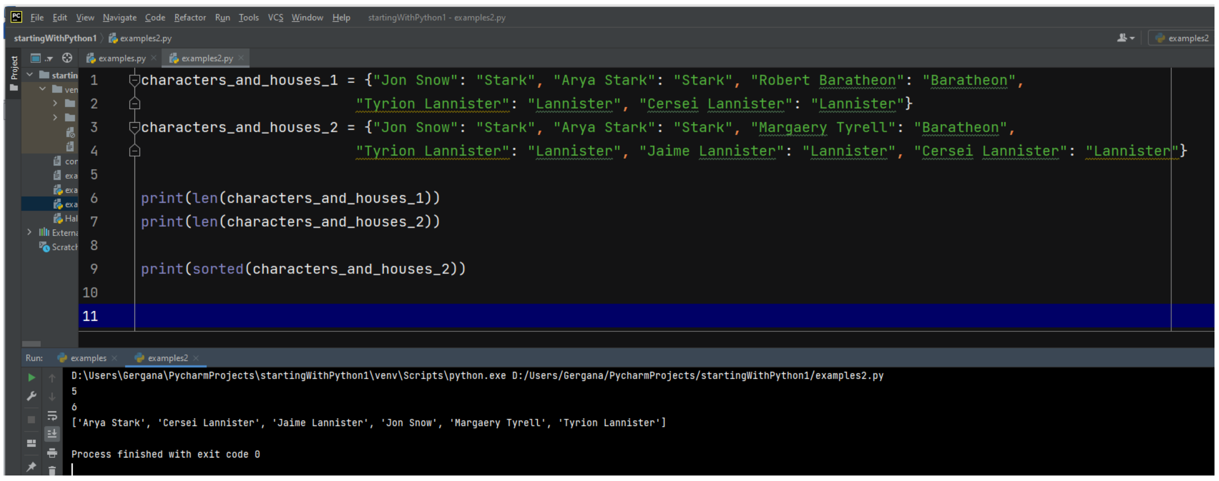
Co-funded by the Erasmus+ Programme of the European Union



```
characters_and_houses = {"Jon Snow": "Stark", "Arya Stark": "Stark", "Robert Baratheon": "Baratheon",
                         "Tyrion Lannister": "Lannister", "Cersei Lannister": "Lannister"}

print("Jon Snow" in characters_and_houses)
print("Margaery Tyrell" in characters_and_houses)
print("Margaery Tyrell" not in characters_and_houses)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
True
False
True

Process finished with exit code 0
```

Dictionaries in Python come with several useful in-built functions, such as len() (returns the number of elements in the dictionary) and sorted() (sorts the keys in a dictionary and returns them as a list).



```
characters_and_houses_1 = {"Jon Snow": "Stark", "Arya Stark": "Stark", "Robert Baratheon": "Baratheon",
                           "Tyrion Lannister": "Lannister", "Cersei Lannister": "Lannister"}
characters_and_houses_2 = {"Jon Snow": "Stark", "Arya Stark": "Stark", "Margaery Tyrell": "Baratheon",
                           "Tyrion Lannister": "Lannister", "Jaime Lannister": "Lannister", "Cersei Lannister": "Lannister"}

print(len(characters_and_houses_1))
print(len(characters_and_houses_2))

print(sorted(characters_and_houses_2))
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
5
6
['Arya Stark', 'Cersei Lannister', 'Jaime Lannister', 'Jon Snow', 'Margaery Tyrell', 'Tyrion Lannister']

Process finished with exit code 0
```

Functions

Functions in Pyhton are blocks of code that perform a specific task. They are given a name and can be called (*invoked*) by that name. They are useful because they break the program's code into smaller, easier to understand and reusable modules. Functions can take parameters and can return a result. There are, however, functions that take no parameters and/or return no result (if a function does not return a result, then it only executes a code, such as printing).

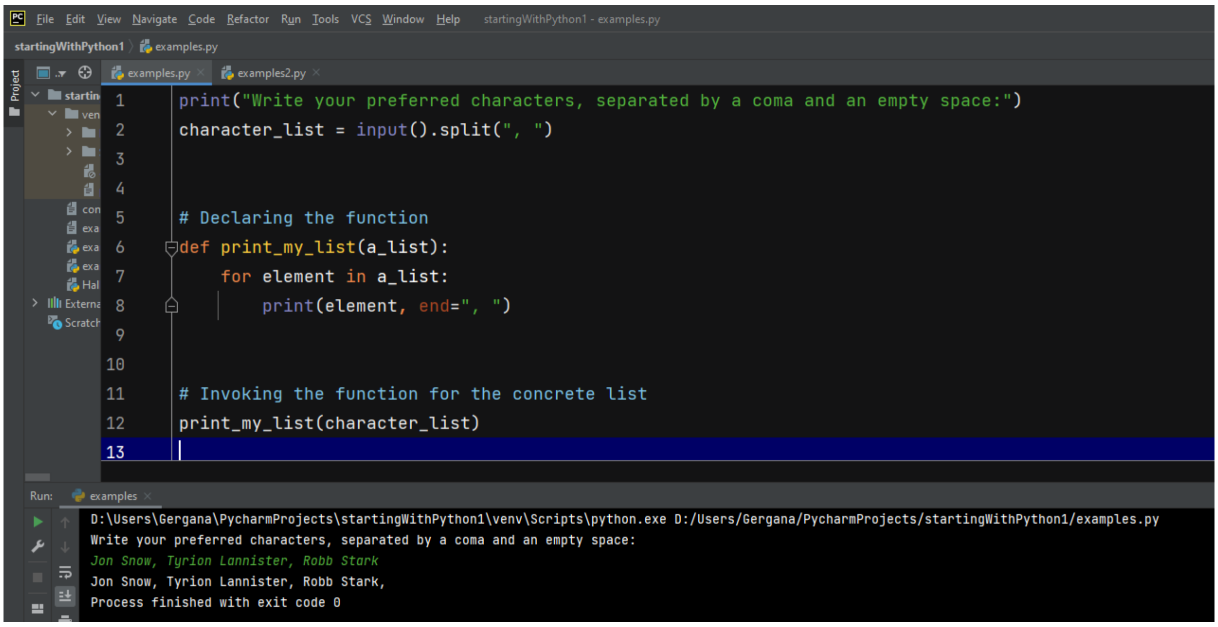
Functions are declared by the keyword def and their name, e.g. def *print_my_orders:*. If the function will accept *parameters*, they are added inside parentheses after the name. The executable bloc of code comes after the : (colon), on a new line, indented. The declared function is like a template for a small sub-program. It does not do anything until it is invoked in the main program and receives specific arguments if applicable.

Once declared, a function can be invoked by its name, followed by parentheses, in which *arguments* are passed in (unless the function accepts no parameters). The *parameters* are thus the variables included in parentheses when we declare the function, and the *arguments* are the concrete values of these variables passed in the function when it is invoked.

In the example below, the function just prints and returns nothing. Notice that the parameter a_list is a variable to which we gave a generic name when we declare the function – this is not a specific list that we may use. When we invoke the function, we will pass in it as an *argument* a

Co-funded by the Erasmus+ Programme of the European Union

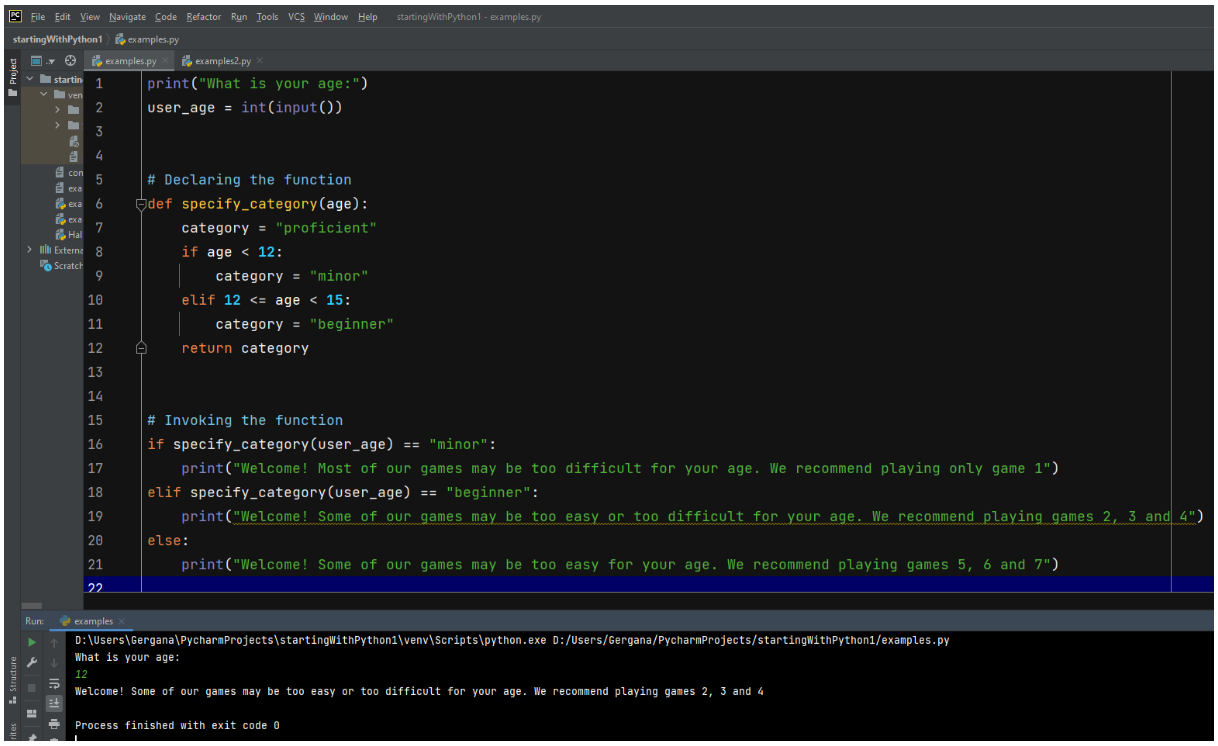
concrete list, and the function will perform the specified tasks with it. The function can thus be invoked multiple times for different lists, and will always perform the same tasks for the specific list that we pass in as an argument. This is why functions make code reusable. We do not need to write this code again and again every time we need this task completed, we just invoke the function and pass in it the specific list as an argument. The argument we pass in the function when invoking it has to be of the same datatype as the parameter we intended when declaring the function. While Python does not require us to specify the datatype, the function may not work if the datatype is incorrect.


File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples.py
startingWithPython1 examples.py
examples.py examples2.py
Run: examples

```python
print("Write your preferred characters, separated by a coma and an empty space:")
character_list = input().split(", ")


# Declaring the function
def print_my_list(a_list):
    for element in a_list:
        print(element, end=", ")


# Invoking the function for the concrete list
print_my_list(character_list)
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
Write your preferred characters, separated by a coma and an empty space:
Jon Snow, Tyrion Lannister, Robb Stark
Jon Snow, Tyrion Lannister, Robb Stark, 
Process finished with exit code 0
```

See also an example of a simple function that returns a result back to the program, which then uses this result as it needs to. After that, try inventing your own program which declares and invokes a function and test if it works correctly.

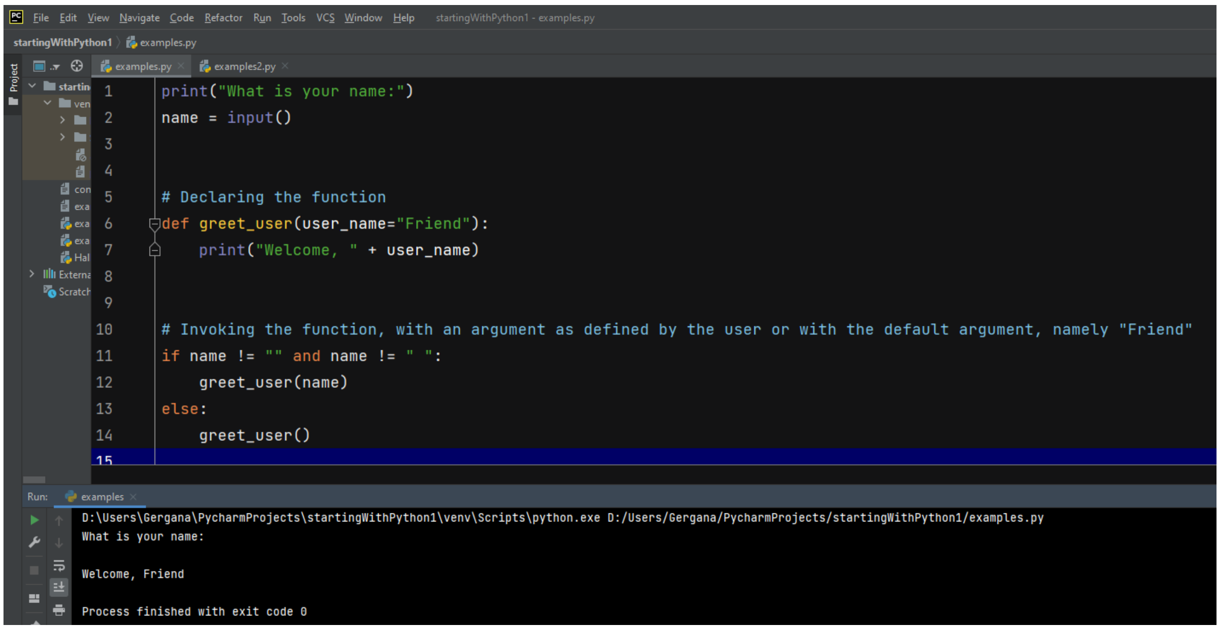

File Edit View Navigate Code Refactor Run Tools VCS Window Help startingWithPython1 - examples.py
startingWithPython1 examples.py
examples.py examples2.py
Run: examples

```python
print("What is your age:")
user_age = int(input())


# Declaring the function
def specify_category(age):
    category = "proficient"
    if age < 12:
        category = "minor"
    elif 12 <= age < 15:
        category = "beginner"
    return category


# Invoking the function
if specify_category(user_age) == "minor":
    print("Welcome! Most of our games may be too difficult for your age. We recommend playing only game 1")
elif specify_category(user_age) == "beginner":
    print("Welcome! Some of our games may be too easy or too difficult for your age. We recommend playing games 2, 3 and 4")
else:
    print("Welcome! Some of our games may be too easy for your age. We recommend playing games 5, 6 and 7")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
What is your age:
12
Welcome! Some of our games may be too easy or too difficult for your age. We recommend playing games 2, 3 and 4

Process finished with exit code 0
```

Co-funded by the
Erasmus+ Programme
of the European Union

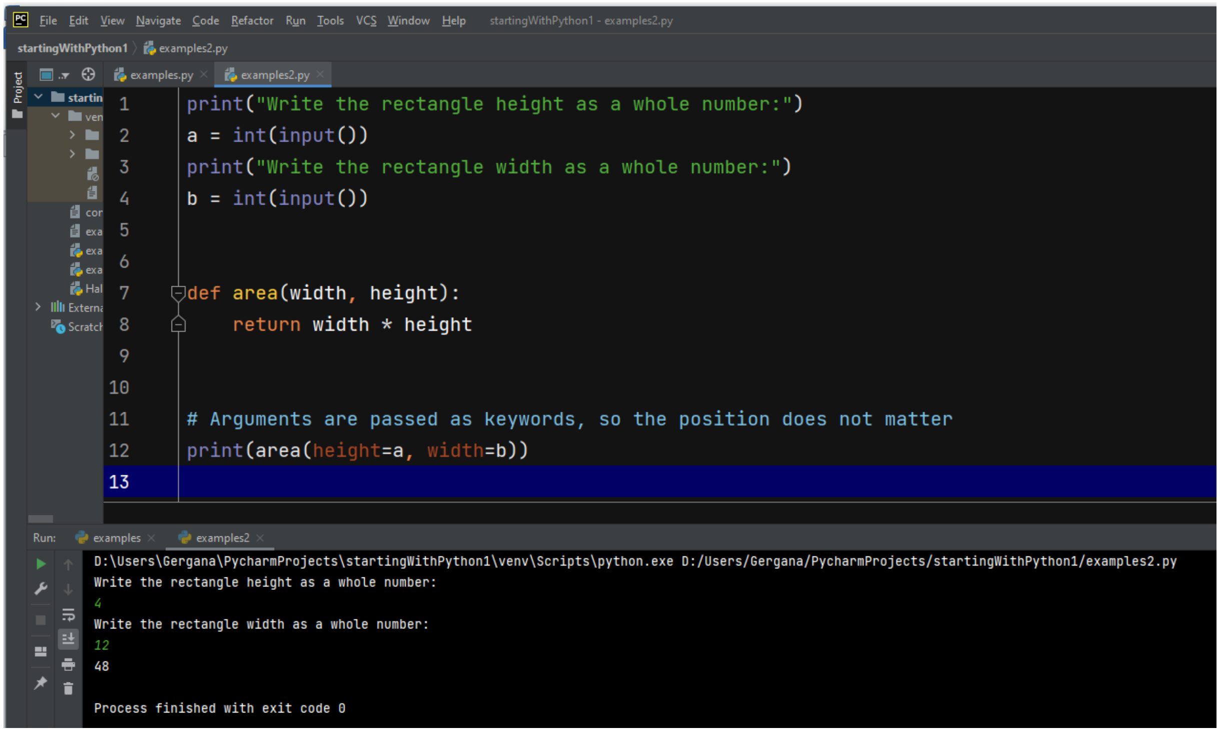
The parameter can have a default value, which will be taken as an argument if the function is invoked without an argument. In the example below, if the user does not write its name, we will greet her simply as “Friend”. This makes the program more suited to different user behavior, such as not typing anything on the console.



```
print("What is your name:")
name = input()


# Declaring the function
def greet_user(user_name="Friend"):
    print("Welcome, " + user_name)


# Invoking the function, with an argument as defined by the user or with the default argument, namely "Friend"
if name != "" and name != " ":
    greet_user(name)
else:
    greet_user()
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples.py
What is your name:

Welcome, Friend

Process finished with exit code 0
```

In Python, we can also functions with the so-called keyword arguments that are named just like the parameters. In that case, the order in which we pass in the arguments is not important because they will be identified by their name.



```
print("Write the rectangle height as a whole number:")
a = int(input())
print("Write the rectangle width as a whole number:")
b = int(input())


def area(width, height):
    return width * height


# Arguments are passed as keywords, so the position does not matter
print(area(height=a, width=b))
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Write the rectangle height as a whole number:
4
Write the rectangle width as a whole number:
12
48

Process finished with exit code 0
```

Objects and Classes

Objects are widely used in all programming languages in a paradigm of programming called Object Oriented Programming (OOP). When doing OOP, related properties (attributes) and

Co-funded by the Erasmus+ Programme of the European Union

behaviors are bundled together into Objects. The program maintains several Objects that can interact with each other. Objects solve a lot of issues in programming and, together with functions, allow for code to be reused instead of continuously retyped.

To do OOP in Python, we need to know several concepts and learn some syntax, which are in fact quite similar to what we already know from other programming languages.

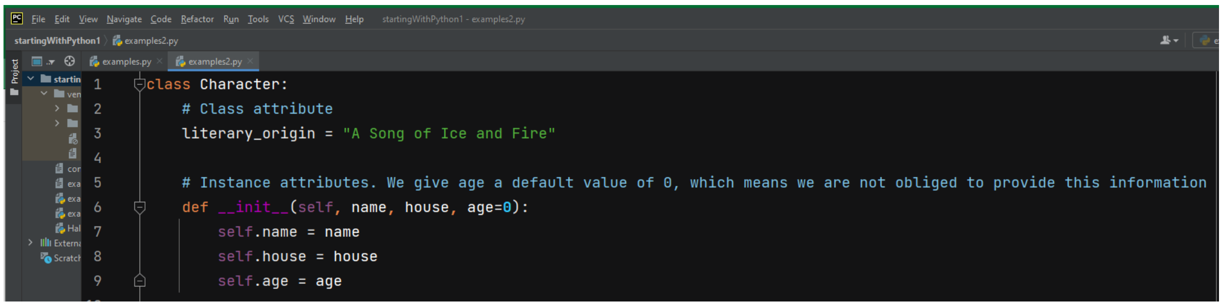
Classes

The Class is a *blueprint* for creating Objects. Creating a new class creates a new type of object, allowing new instances of that type to be made. The Class provides a description of the Object, such as what its basic attributes are and what its behaviors are. Each Object that we later create based on this blueprint is in fact a particular instance of its Class, with specific values for the properties. Classes contain class attributes (common to all instances) and instance attributes. Classes can include class methods for modifying the Class state and instance methods for modifying the instance.

The __init__() method initializes an object's initial attributes (it is written with a double leading and trailing underscore).

The parameter self is a reference to the current instance of the Class. We use it to access variables that belong to this class and it is also always the first parameter that is passed in instance methods and to the __init__ function.

If all this sounds too abstract, look at the example below. We create a Class Character. Each instance will have three attributes - a name, a House, age. These are the instance attributes of the Class, and they are also parameters in the __init__ function. When we create an instance, the __init__ function will expect 3 arguments – for the name, for the age and for the House. The Class has a Class attribute literary_origin, which is "A Song of Ice and Fire" for all instances of this Class. For attributes that are optional, we need to give a default value and they should be indicated as the last parameters in the __init__ function. In our case, we give a default age of 0.

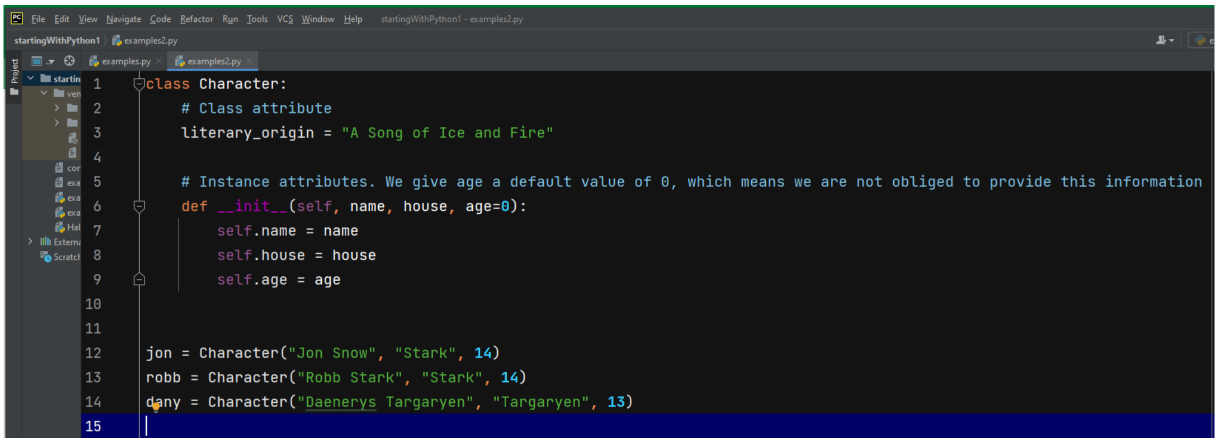


```python
class Character:
    # Class attribute
    literary_origin = "A Song of Ice and Fire"

    # Instance attributes. We give age a default value of 0, which means we are not obliged to provide this information
    def __init__(self, name, house, age=0):
        self.name = name
        self.house = house
        self.age = age
```

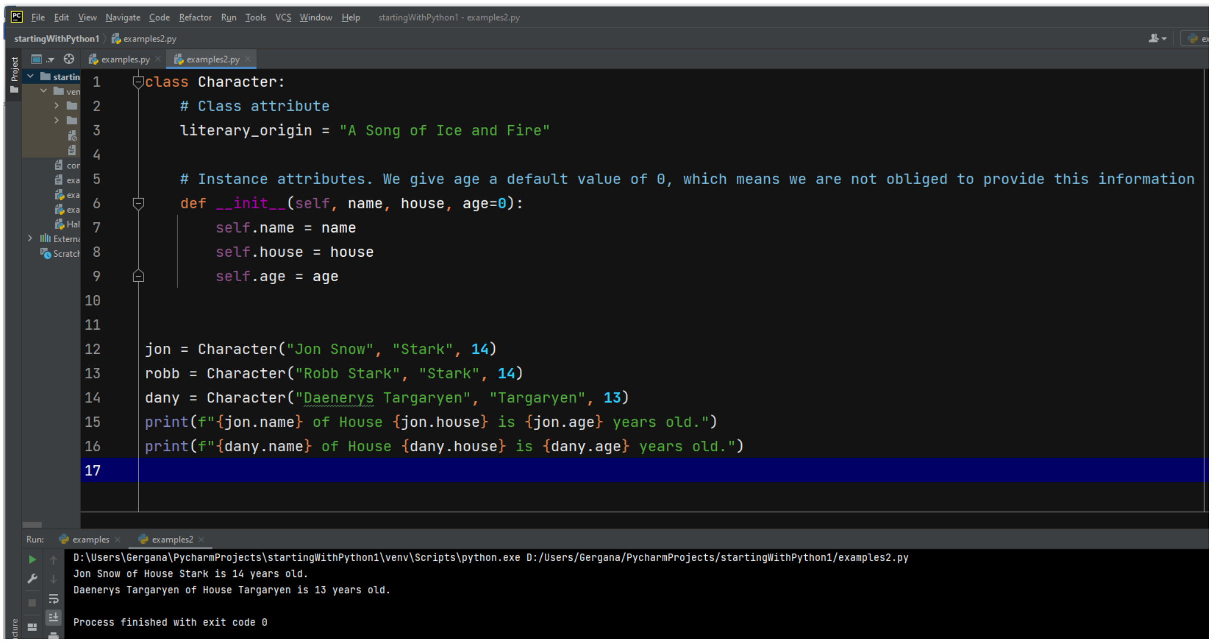
Instances of Classes

We learned until now that the class is like a blueprint while an instance is a copy of the class with actual values. Now we are ready to create as many instances of this Class as we need, by passing in it the concrete names, Houses and ages as arguments for each instance. An instance of the Class is an Object. Below we create 3 concrete characters based on the Class blueprint. This is done by initializing an object with *name_of_class*(attributes).

Co-funded by the Erasmus+ Programme of the European Union


```python
class Character:
    # Class attribute
    literary_origin = "A Song of Ice and Fire"

    # Instance attributes. We give age a default value of 0, which means we are not obliged to provide this information
    def __init__(self, name, house, age=0):
        self.name = name
        self.house = house
        self.age = age


jon = Character("Jon Snow", "Stark", 14)
robb = Character("Robb Stark", "Stark", 14)
dany = Character("Daenerys Targaryen", "Targaryen", 13)
```

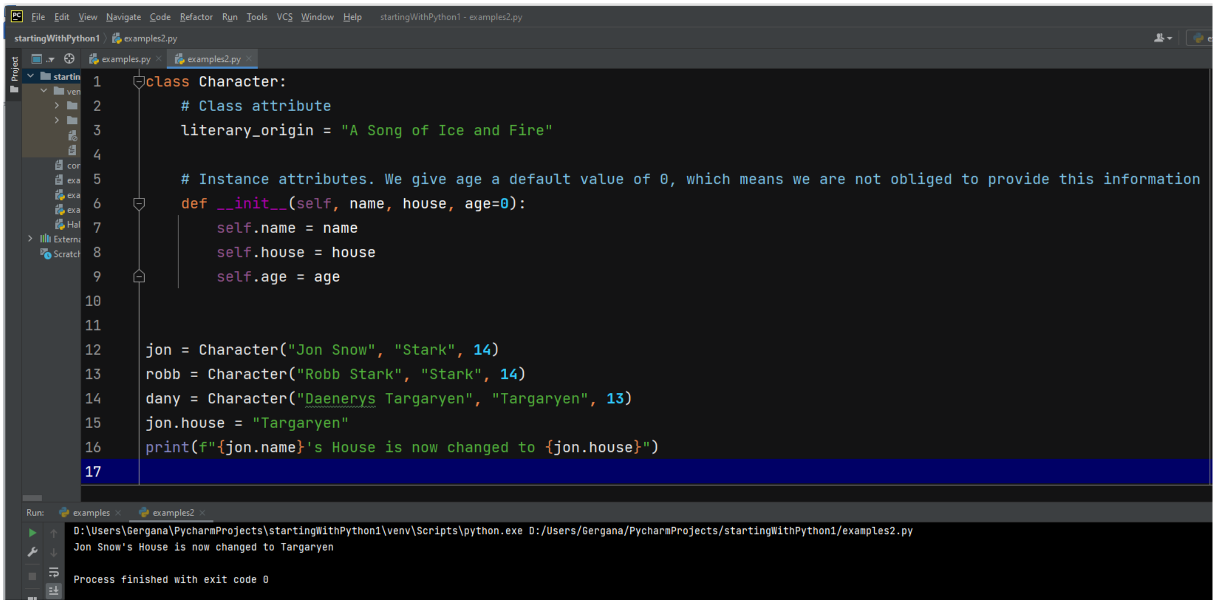
To access a concrete Object's attributes, we write the name of the object, followed by a dot and the name of the attribute. For example, we can access Robb's age by robb.age. See the example below and notice the output. The program has retrieved the attributes of our characters.


```python
class Character:
    # Class attribute
    literary_origin = "A Song of Ice and Fire"

    # Instance attributes. We give age a default value of 0, which means we are not obliged to provide this information
    def __init__(self, name, house, age=0):
        self.name = name
        self.house = house
        self.age = age


jon = Character("Jon Snow", "Stark", 14)
robb = Character("Robb Stark", "Stark", 14)
dany = Character("Daenerys Targaryen", "Targaryen", 13)
print(f"{jon.name} of House {jon.house} is {jon.age} years old.")
print(f"{dany.name} of House {dany.house} is {dany.age} years old.")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Jon Snow of House Stark is 14 years old.
Daenerys Targaryen of House Targaryen is 13 years old.

Process finished with exit code 0
```

Once we have accessed an attribute of the Object, we can modify it by assigning a new value to it. See the example.

Co-funded by the Erasmus+ Programme of the European Union



```python
class Character:
    # Class attribute
    literary_origin = "A Song of Ice and Fire"

    # Instance attributes. We give age a default value of 0, which means we are not obliged to provide this information
    def __init__(self, name, house, age=0):
        self.name = name
        self.house = house
        self.age = age


jon = Character("Jon Snow", "Stark", 14)
robb = Character("Robb Stark", "Stark", 14)
dany = Character("Daenerys Targaryen", "Targaryen", 13)
jon.house = "Targaryen"
print(f"{jon.name}'s House is now changed to {jon.house}")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Jon Snow's House is now changed to Targaryen

Process finished with exit code 0
```

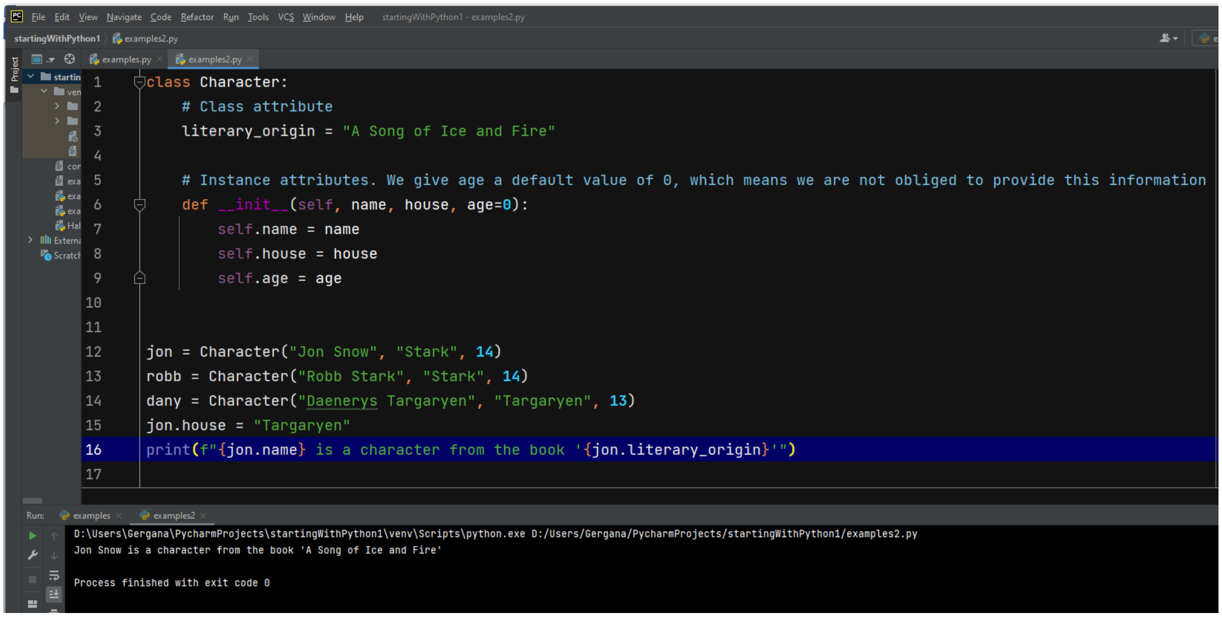
We can access the Class attributes in a similar way. See below.

Tip on formatting strings:

If you want to write quotation marks inside a formatted string, you have two options:

One is to use the quotation marks format that is different from the format you typically use for strings. Below, we write strings in double quotation marks, so we use single quotation marks inside the text.

Another option is to "escape" the character by typing \ before it. Thus, Python will know that it is not part of the code but should be printed as a special character. You can escape any special character in this way. Try it yourself.



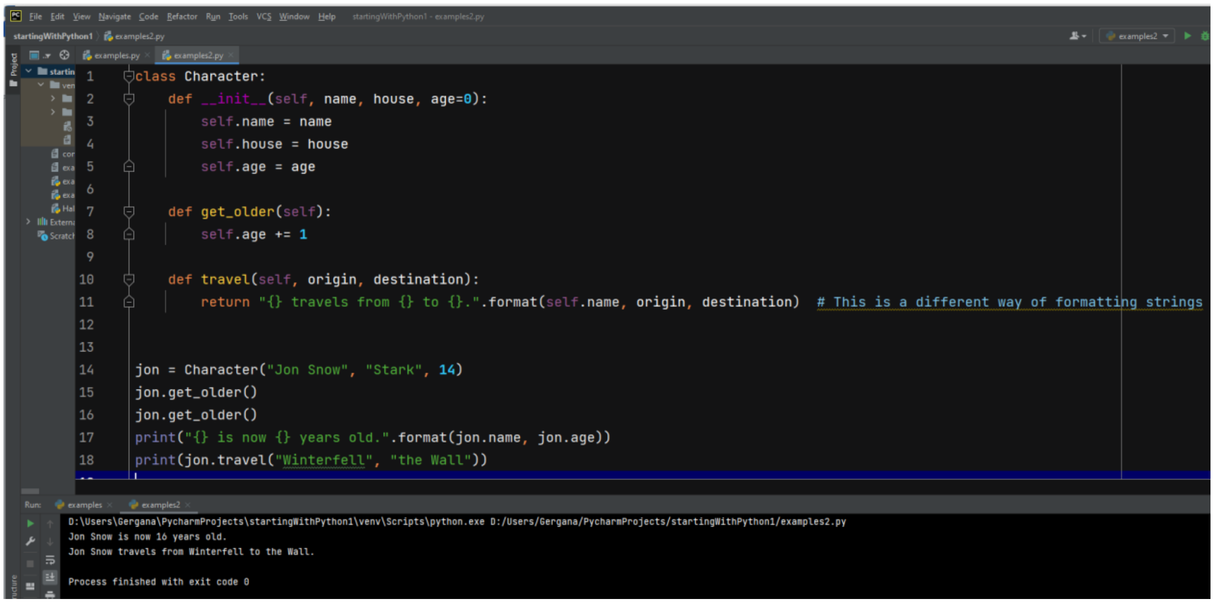
```python
class Character:
    # Class attribute
    literary_origin = "A Song of Ice and Fire"

    # Instance attributes. We give age a default value of 0, which means we are not obliged to provide this information
    def __init__(self, name, house, age=0):
        self.name = name
        self.house = house
        self.age = age


jon = Character("Jon Snow", "Stark", 14)
robb = Character("Robb Stark", "Stark", 14)
dany = Character("Daenerys Targaryen", "Targaryen", 13)
jon.house = "Targaryen"
print(f"{jon.name} is a character from the book '{jon.literary_origin}'")
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Jon Snow is a character from the book 'A Song of Ice and Fire'

Process finished with exit code 0
```

Let us now look at instance methods. They are defined inside the Class but are used to get the contents of an instance or to do something with the instance attributes. They usually define behaviors. Their first parameter should always be self, like in the __init__ function. In the example below, we create for our characters two methods – get_older() and travel(). The get_older() method ages the character with 1 year, while the travel() method returns a text

Co-funded by the Erasmus+ Programme of the European Union

specifying where s/he travels from and where s/he travels to. When we call these methods in the program, we call them in the same way as we access attributes – *Object's name*, followed by *.method_name* – e.g. jon.travel(). We also need to pass in any arguments that the method expects. The get_older() method does not require an argument because it always ages the character with 1 year. The travel() method requires as arguments the travel origin and the travel destination. You can modify the get_older() method to increase the character's age with a specified number of years. In this case, you need to include this number as a parameter when defining the method and then pass it in as an argument when you call the method. Try this yourself.



```python
class Character:
    def __init__(self, name, house, age=0):
        self.name = name
        self.house = house
        self.age = age

    def get_older(self):
        self.age += 1

    def travel(self, origin, destination):
        return "{} travels from {} to {}.".format(self.name, origin, destination)  # This is a different way of formatting strings


jon = Character("Jon Snow", "Stark", 14)
jon.get_older()
jon.get_older()
print("{} is now {} years old.".format(jon.name, jon.age))
print(jon.travel("Winterfell", "the Wall"))
```

```
D:\Users\Gergana\PycharmProjects\startingWithPython1\venv\Scripts\python.exe D:/Users/Gergana/PycharmProjects/startingWithPython1/examples2.py
Jon Snow is now 16 years old.
Jon Snow travels from Winterfell to the Wall.

Process finished with exit code 0
```

Concluding remarks

Congratulations, you have learned the basics of Python. If you took the time to re-create the code or code your own examples, you are ready to start working on more complex algorithms and to solve exercises. Go to HackerRank and start preparation, going gradually from Easy to Medium and Hard modes.

There is of course a lot more to learn in Python, but as soon you encounter a task that you cannot solve, you can search the internet for solutions. The information available on Python is ample. Sites such as https://stackoverflow.com can help you find solutions to even seemingly difficult problems. Another useful source is the official Python documentation at https://docs.python.org. Finally, in PyCharm itself, by holding CTRL and clicking on an in-built method, you will get all the relevant documentation, which is a great time-saver.

Happy coding!

Co-funded by the Erasmus+ Programme of the European Union