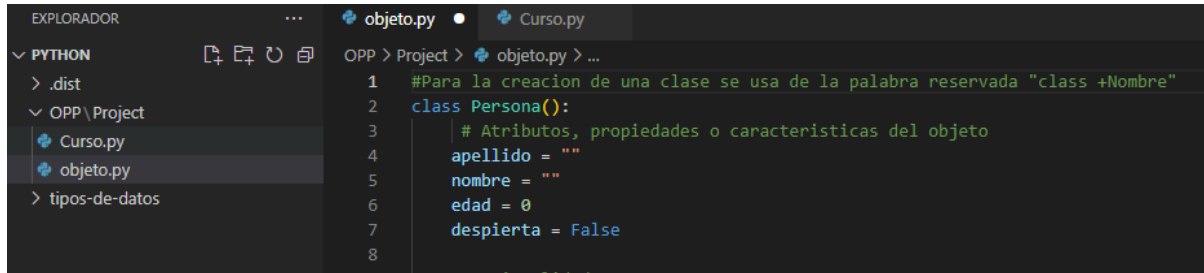


Programming language(Python)

Class: the word is used “class + name”

Atributos: Son las propiedades que recibe el objeto

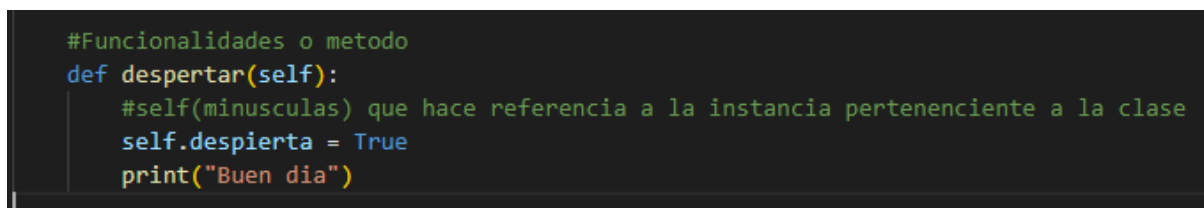


```
EXPLORADOR
...
objeto.py
Curso.py

PYTHON
> .dist
> OPP \ Project
  > Curso.py
  > objeto.py
  > tipos-de-datos

OPP > Project > objeto.py > ...
1  #Para la creacion de una clase se usa de la palabra reservada "class +Nombre"
2  class Persona():
3      # Atributos, propiedades o características del objeto
4      apellido = ""
5      nombre = ""
6      edad = 0
7      despierta = False
8
```

Methods: its functions that receive the variables are assigned with the reserved word “def” to define that function and y “self” refers to the class itself



```
#Funcionalidades o metodo
def despertar(self):
    #self(minusculas) que hace referencia a la instancia perteneciente a la clase
    self.despierta = True
    print("Buen dia")
```

Polymorphism: Python being a weakly typed language, that is to say that when passing a parameter it is not necessary to indicate the data type, since it implicitly interprets

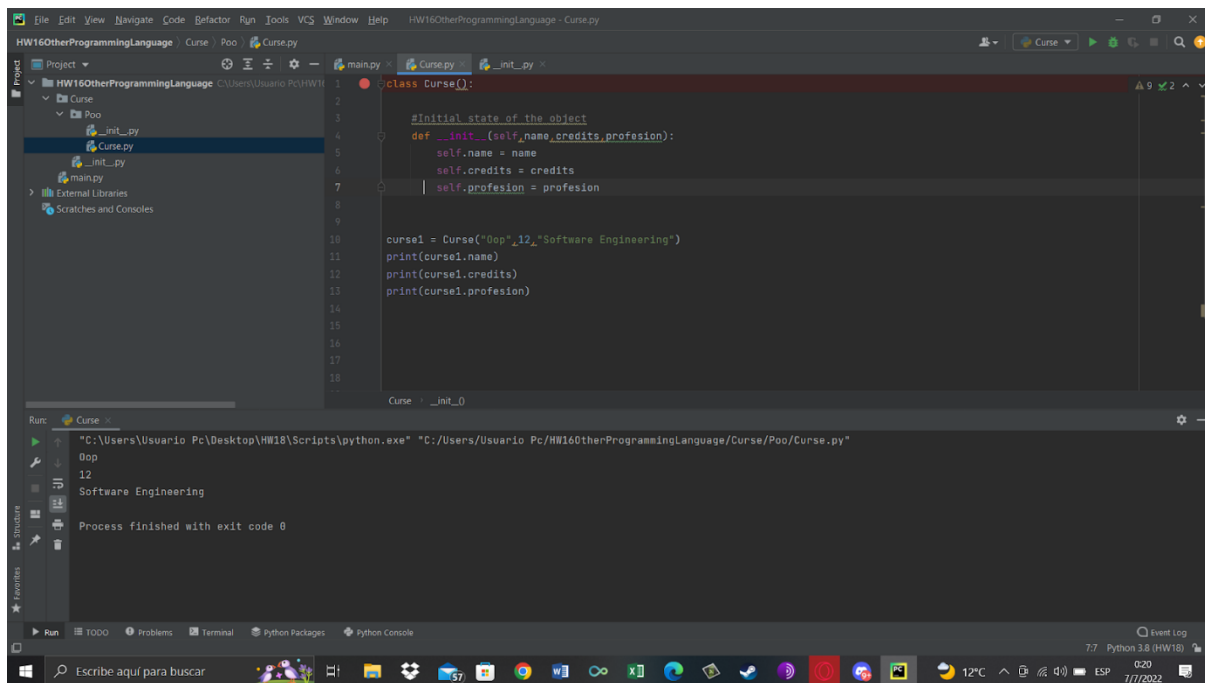


```
OPP > Project > Polimorfismo.py > describirPersona
1  class Estudiante():
2
3      def describir(self):
4          print("Soy un buen estudiante")
5
6  class Docente():
7
8      def describir(self):
9          print("Me dedico a enseñar a cursos")
10
11
12 class Trabajador():
13
14     def describir(self):
15         print("Trabajo dentro de una gran empresa")
16
17
18 def describirPersona(persona):
19     persona.describir()
20
21     doc1=Trabajador()
22     describirPersona(doc1)
```

Encapsulation:

```
OPP > Project > Curso.py > Curso > __init__
1  class Curso:
2      """
3      nombre = "Matematicas"
4      creditos = 5
5      carrera = "Ingenieria Civil"
6      """
7
8      #estado inicial del objeto
9      def __init__(self,nom,cre,pro):
10         self.nombre = nom
11         self.creditos = cre
12         self.carrera = pro
13         self.__imparticion = "Presencial" #propiedad encapsulada
14
15     def mostrarDatos(self):
16         dat="Nombre: {0} / Creditos: {1} / Modo de particion: {2}"
17         print(dat.format(self.nombre,self.creditos,self.__imparticion))
18
19 curso1 = Curso("matematicas",5,"ingenieria Civil")
20 print(curso1.nombre)
21 curso1.mostrarDatos()
22
```

Constructor: The constructor function is used to initialize all the attributes of the class. The name of this constructor function is the same as the name of the class. The concept of a constructor function is the same in Python, but the name of the constructor function is `__init__()` for all classes.



```
1 class Course():
2
3     #initial state of the object
4     def __init__(self, name, credits, profesion):
5         self.name = name
6         self.credits = credits
7         self.profesion = profesion
8
9
10    curse1 = Course("0op", 12, "Software Engineering")
11    print(curse1.name)
12    print(curse1.credits)
13    print(curse1.profesion)
```

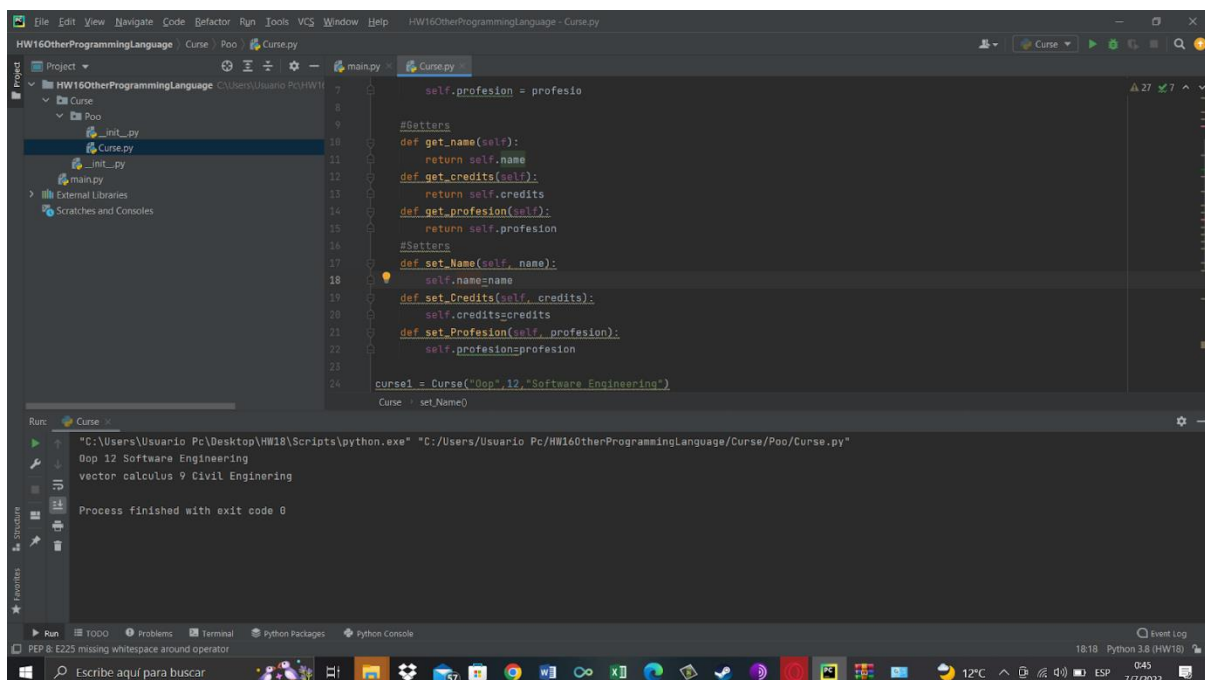
Run: Course

"C:\Users\Usuario Pc\Desktop\HW18\Scripts\python.exe" "C:/Users/Usuario Pc/HW160therProgrammingLanguage/Curse/Poo/Curse.py"

0op
12
Software Engineering

Process finished with exit code 0

Geetters and Setters: "Getters" and "Setters" are used in OOP to guarantee the principle of data encapsulation. These are typically used in Python: If we want to add validation logic to get and set a value.



```
7         self.profesion = profesio
8
9     #Getters
10    def get_name(self):
11        return self.name
12    def get_credits(self):
13        return self.credits
14    def get_profesion(self):
15        return self.profesion
16
17    #Setters
18    def set_Name(self, name):
19        self.name=name
20    def set_Credits(self, credits):
21        self.credits=credits
22    def set_Profesion(self, profesion):
23        self.profesion=profesion
24
25    curse1 = Course("0op", 12, "Software Engineering")
26    curse1.set_Name()
```

Run: Course

"C:\Users\Usuario Pc\Desktop\HW18\Scripts\python.exe" "C:/Users/Usuario Pc/HW160therProgrammingLanguage/Curse/Poo/Curse.py"

0op 12 Software Engineering
vector calculus 9 Civil Engineering

Process finished with exit code 0

The screenshot shows an IDE window titled 'HW16OtherProgrammingLanguage - Curse.py'. The code defines a `Course` class with attributes `name`, `credits`, and `profesion`, and methods `set_name`, `set_credits`, and `set_profesion`. An instance `course1` is created with the name 'Opap', 12 credits, and the profession 'Software Engineering'. The instance's attributes are then updated to 'vector calculus', 'Civil Engineering', and 9 credits. The output window shows the execution of these updates and the final state of the instance.

```
18 self.name=name
19 def set_credits(self, credits):
20     self.credits=credits
21 def set_profesion(self, profesion):
22     self.profesion=profesion
23
24 course1 = Course("Opap",12,"Software Engineering")
25 print(course1.get_name(),course1.get_credits(),course1.get_profesion())
26 course1.set_name("vector calculus")
27 course1.set_profesion("Civil Engineering")
28 course1.set_credits(9)
29 print(course1.get_name(),course1.get_credits(),course1.get_profesion())
30
31
32
33
34
35
```

Run: Curse

```
"C:\Users\Usuario Pc\Desktop\HW16\Scripts\python.exe" "C:/Users/Usuario Pc/HW16OtherProgrammingLanguage/Curse/Poo/Curse.py"
Opap 12 Software Engineering
vector calculus 9 Civil Engineering

Process finished with exit code 0
```

Exception: Python uses a special object called an exception to handle any errors that may occur during the execution of a program. When an error occurs during the execution of a program, Python creates an exception.

The screenshot shows an IDE window titled 'HW16OtherProgrammingLanguage - Exception2.py'. The code defines a class `Exception2` with a list of fruits and a method `chooseFruit`. The method prompts the user to choose a fruit by index. It handles `IndexError` (if the index is out of range) and `ValueError` (if the input is not an integer). The output window shows the execution of the program, including the list of fruits, the user's input, and the resulting output.

```
1 class Exception2():
2     fruits = ['0 -Banana', '1-Apple', '2-Grapefruit', '3-peach']
3     def chooseFruit(chooseFruit):
4         try:
5             print(chooseFruit)
6             index=int(input("Choose a fruit(put the number)"))
7             print(f"Tu fruta favorita es {chooseFruit[index]}")
8         except IndexError:
9             print(f"Wrong index, must be between 0 and {len(chooseFruit)-1}")
10        except ValueError:
11            print("You have to put an integer")
12
13        chooseFruit(fruits)
14        chooseFruit(fruits)
15        chooseFruit(fruits)
```

Run: Exception2

```
"C:\Users\Usuario Pc\Desktop\HW16\Scripts\python.exe" "C:/Users/Usuario Pc/HW16OtherProgrammingLanguage/Curse/Exception/Exception2.py"
['0 -Banana', '1-Apple', '2-Grapefruit', '3-peach']
Choose a fruit(put the number):
Tu fruta favorita es 1-Apple
['0 -Banana', '1-Apple', '2-Grapefruit', '3-peach']
Choose a fruit(put the number):
Tu fruta favorita es 3-peach
['0 -Banana', '1-Apple', '2-Grapefruit', '3-peach']
Choose a fruit(put the number):
Wrong index, must be between 0 and 3
```

The screenshot shows an IDE window titled 'HW160therProgrammingLanguage - Exception.py'. The code defines a custom exception class `Exception()` and a `division` function. The function attempts to divide `dividing` by `divider`. If a `ZeroDivisionError` occurs, it catches it and prints 'Cannot be divided by zero'. The script then calls `division(5, 5)`, `division(5, 0)`, `division(5, 0)`, `division(8, 0)`, and finally `print("Hello Word")`.

```
1 class Exception():
2     def division(dividing, divider):
3         try:
4             result = dividing / divider
5             print(result)
6         except ZeroDivisionError:
7             print("Cannot be divided by zero")
8
9     division(5, 5)
10    division(5, 0)
11    division(5, 0)
12    division(8, 0)
13
14    print("Hello Word")
```

The Run console shows the output of the script:

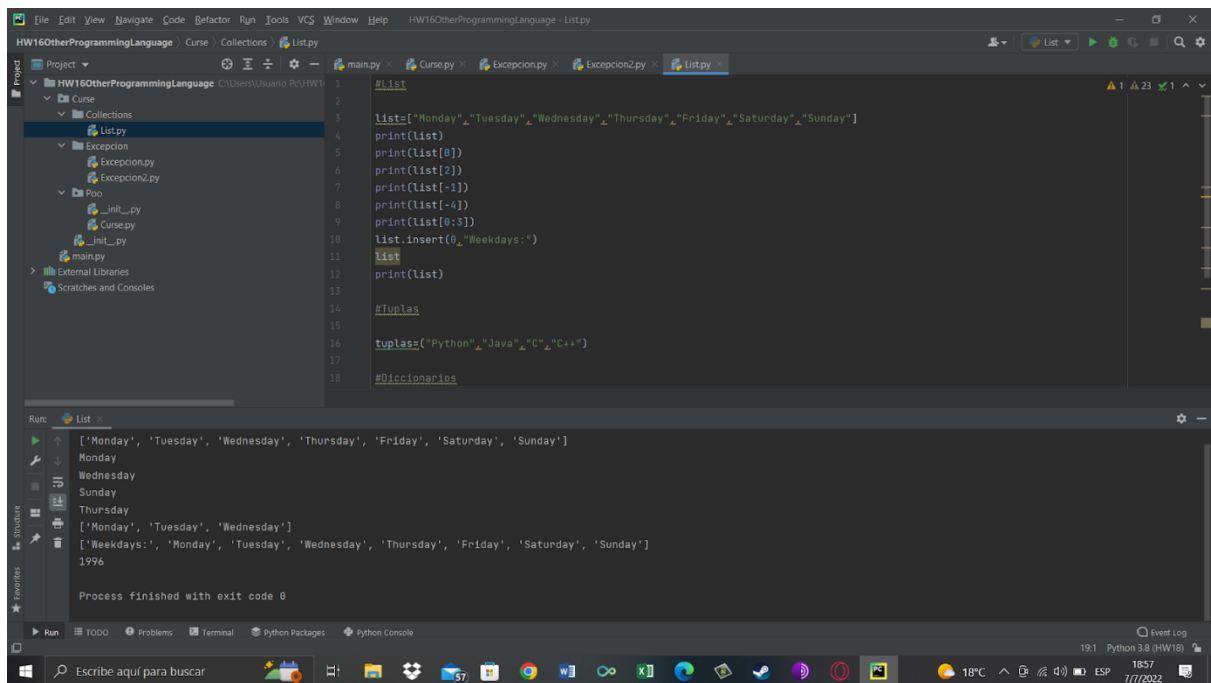
```
"C:\Users\Usuario Pc\Desktop\HW18\Scripts\python.exe" "C:/Users/Usuario Pc/HW160therProgrammingLanguage/Curse/Exception/Exception.py"
1.0
Cannot be divided by zero
0.75
Cannot be divided by zero
Hello Word
Process finished with exit code 0
```

Collection:

Lists: A list is an ordered set of objects. By objects we mean any of the data types already mentioned, including other lists.

Tuples: Tuples, like lists, are ordered collections. However, unlike these, they are immutable. That is, once the elements are assigned, they cannot be altered. In functional terms, it could be said that tuples are a subset of lists, since they support index operations to access their elements, but not assignment operations.

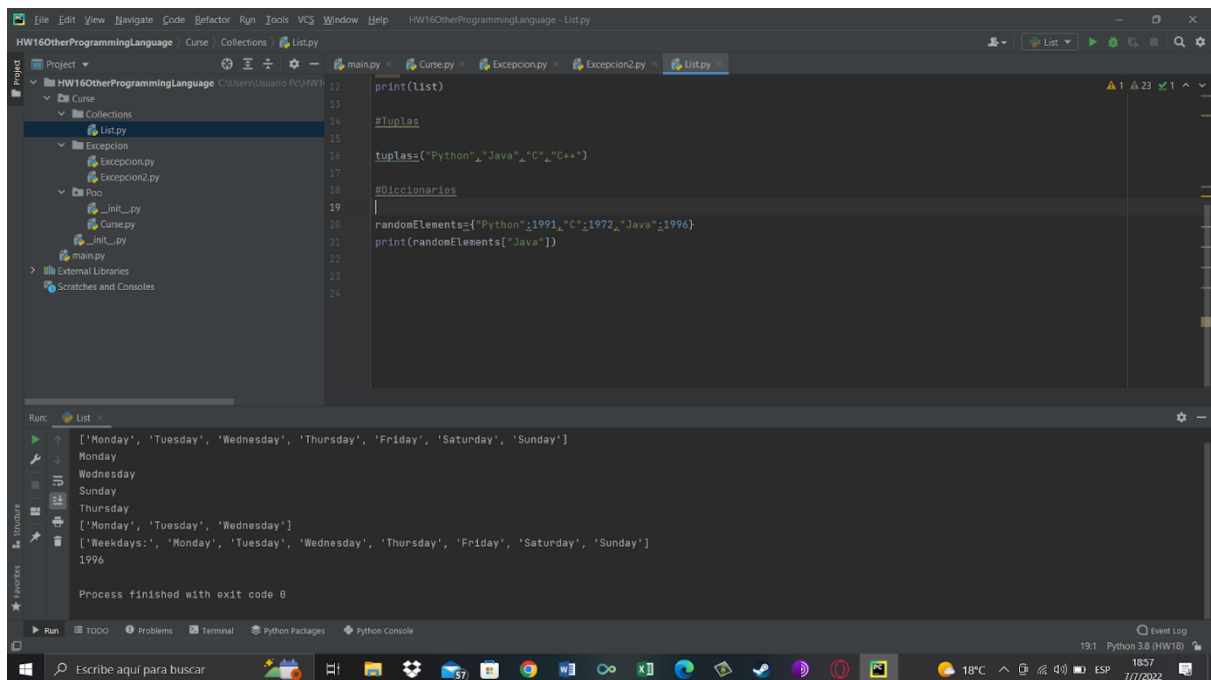
Dictionaries: Dictionaries, unlike lists and tuples, are unordered collections of objects. In addition, its elements have a peculiarity: they always make up a key-value pair. That is, when we add a value to a dictionary, it is assigned a unique key with which it can then be accessed (because the position is no longer a determinant).



```
1 list
2
3 list=["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"]
4 print(list)
5 print(list[0])
6 print(list[2])
7 print(list[-1])
8 print(list[-4])
9 print(list[0:3])
10 list.insert(0,"Weekdays:")
11 list
12 print(list)
13
14 #Tuplas
15
16 tuplas=("Python","Java","C","C++")
17
18 #Diccionarios
```

Run: List

```
[ 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday' ]
Monday
Wednesday
Sunday
Thursday
[ 'Monday', 'Tuesday', 'Wednesday' ]
[ 'Weekdays:', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday' ]
1996
Process finished with exit code 0
```



```
12 print(list)
13
14 #Tuplas
15
16 tuplas=("Python","Java","C","C++")
17
18 #Diccionarios
19
20 randomElements={"Python":1991,"C":1972,"Java":1996}
21 print(randomElements["Java"])
22
23
24
```

Run: List

```
[ 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday' ]
Monday
Wednesday
Sunday
Thursday
[ 'Monday', 'Tuesday', 'Wednesday' ]
[ 'Weekdays:', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday' ]
1996
Process finished with exit code 0
```

Abstraction

Abstraction allows us to create abstract classes and methods. An abstract class must satisfy two basic conditions:

- 1.- An object will not be created directly from the class.
- 2.- It must contain at least one abstract method.

This serves as the base (main class) for the other subclasses. For example, an abstract class could be "Tool". They can inherit from her the "hammer", "screwdriver" or "tweezers". These are all tools and it makes sense to create such objects. However, it doesn't make sense to create a "tool" object because it's a very general concept.

```
from abc import ABC, abstractmethod

class Character(ABC):
    @abstractmethod
    def __init__(self, name):
        self.name = name
        self.level = 0
        self.inventory = []

    @abstractmethod
    def attack(self, target):
        pass
```

Interfaces

An interface is a common means for unrelated objects to communicate with each other. These are definitions of methods and values on which the objects agree to cooperate.

```
1 from app.repository.user_repository import UserRepository
2 from app.business.store_manager import StoreManager
3 from app.repository.database import Database
4 from app.repository.s3 import S3
5 from app.repository.file_system import FileSystem
6 from app.models.user import User
7
8
9 user = User("Ronny", "Ibarra", 21)
10
11 s3Repository = S3("321312321", "sdf32423", "MyBucket")
12 StoreManager.storeUser(user, s3Repository)
13
14 print("\n")
15
16 fileSystemRepository = FileSystem("/home/users")
```

PROBLEMAS 15 SALIDA CONSOLA DE DEPURACIÓN TERMINAL JUPYTER Python + v [] [] [] [] []

```
<root><name>Ronny</name><lastName>Ibarra</lastName><age>21</age></root>
Closing file

--->Storing user...

Opening database connection: localhost:admin@admin123
Storing user in the database Ronny

INSERT INTO USER VALUES('Ronny', 'Ibarra', 21)
Closing connection
PS C:\Users\ronny\OneDrive\Documents\Language_Poo>
```