

# Programacion

Alejandro Zubiri

March 26, 2025

# Índice

<b>1</b>	<b>Python</b>	<b>2</b>
1.1	Programación Orientada a Objetos (POO) . . . . .	2
1.1.1	Encapsulamiento . . . . .	2
1.2	Declaración de clases . . . . .	2
1.2.1	Métodos . . . . .	3
1.3	Herencia y abstracción . . . . .	3
1.4	Abstracción . . . . .	4

# Chapter 1

# Python

## 1.1 Programación Orientada a Objetos (POO)

Permite programar realizando abstracciones. Consiste en definir plantillas que desarrollan como cada **objeto** (instancias de las clases) definirá sus atributos y los diferentes métodos que estos tienen.

- Clase:
  - Atributos
  - Métodos

### 1.1.1 Encapsulamiento

Es el concepto de aislar el funcionamiento de los métodos y variables de una clase.

## 1.2 Declaración de clases

```
class Clase:
    pass

objeto = Clase()
```

Cada clase tiene atributos predefinido, pero luego tenemos los atributos de **instancia**, que permite al usuario modificar los atributos en su creación:

```
class Clase:
    def __init__(self, at1, at2):
        self.var1 = at1
        self.var2 = at2

obj = Clase(1, 2)
obj.var1 #1
obj.var2 #2
```

Las diferencias entre ambos es que los atributos de clase se almacenan en los metadatos de la clase, mientras que los de instancia se guardan en los metadatos del objeto.

### 1.2.1 Métodos

Son funciones definidas dentro de una propia clase, y representan las acciones que cada instancia de dicha clase puede realizar. Estos métodos también pueden acceder a los atributos de una clase, por lo que es una forma de encapsular estos. Existen dos tipos de métodos:

- Métodos de instancia: relacionados con cada objeto.
- Métodos de clase: solo se pueden usar con la clase.

## 1.3 Herencia y abstracción

La herencia es un concepto fundamental de la OOP. Consiste en la creación de clases base, con una funcionalidad definida, y luego definir clases que **hereden** de esta, de forma que tengan toda la funcionalidad de la clase padre, además de nuevas funcionalidades que definamos en esta.

```
class Padre:
    def __init__(self):
        self.foo = bar

class Hija(Padre):
    def __init__(self):
        #Modificamos el valor de la clase padre
        self.bar = fizz

hija = Hija()
print(hija.foo) #bar
```

También podemos extender la funcionamiento de clases mediante **super**:

```
def presentarse(self):
    super().presentarse() #Llamamos a la funcion de la clase padre
    # codigo
```

Además, también podemos ahorrarnos código para utilizar la función de definición de clases padre:

```
class Hija(Padre):
    def __init__(self, nombre, apellido, edad):
        super().__init__(nombre, apellido)
        self.edad = edad
```

También es posible hereder de múltiples clases:

```
class Clase_1:
    pass
class Clase_2:
    pass
class Clase_3(Clase_1, Clase_2):
    pass
```

## 1.4 Abstracción

En la POO, una interfaz es una herramienta que define qué funcionalidades debe tener un cierto objeto. Una interfaz permite saber al desarrollador qué funcionalidad va a tener un objeto o clase, sin saber realmente cómo va a implementarla.

De esta misma forma, podemos utilizar las implementaciones que una interfaz nos garantiza que dicha herramienta va a tener. Sin embargo, las interfaces no existen en Python, por lo que las interfaces son



Figure 1.1: Interfaces

equivalentes a clases padre con funciones vacías:

```
class Interfaz:
    def implementacion_1():
        pass
```

Esto es similar a las **clases abstractas**, que son clases que no se puede instanciar, sino que solo se puede derivar de ellas.

Para definir funciones abstractas en Python, utilizamos el decorador `@abstractmethod`, y la clase debe empezar debe heredar de `ABC`:

```
from abc import ABC, abstractmethod
```

```
class Abstract(ABC):
    @abstractmethod
    def funcionAbstracta(self):
        # codigo
```