Programacion

Alejandro Zubiri

AZE

Índice

AZE

CHAPTER 1

Python

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

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.

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.

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
```

4. Abstracción

En la POO, una interfaz es una herramienta que define qué funcionalidades debe terne 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

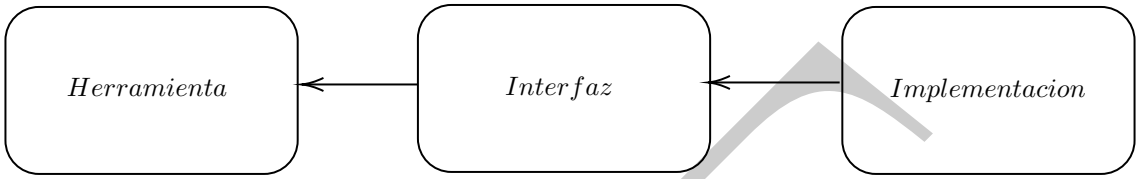

FIGURE 1. Interfaces

equivalentes a clases padre con funciones vacía:

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

Esto es similar a las **clases abstractas**, que son clases que no se puende 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
```

5. Archivos y operaciones

- CSV: permite hacer tablas, y es como una versión más ligera de Excel.
- JSON: JavaScript Object Notation

6. Apertura de archivos

Para llevar a cabo cualquier operación de archivos, utilizamos `open`

```
open(file='ruta', mode='modo de apertura')
```

Los modos son:

- `w`: escritura
- `r`: lectura
- `a`: append, escritura y lectura.

Similarmente, para cerrar archivos, utilizamos `archivo.close()`. Es necesario cerrar el archivo para que el sistema operativo recupere los recursos utilizados. En Python, existen los `context managers`, que nos permiten agilizar este proceso:

```
with open("archivo.txt", "r") as archivo:
        #codigo
```

De esta forma, el archivo se cerrará automáticamente.

7. Lectura y escritura

Una vez abierto el archivo, podemos leerlo con:

- `file.read()`: lee todo el archivo.
- Podemos iterar por cada línea así:

```
for linea in file:
    print(linea.strip())
```

Para escribir en el archivo:

```
texto = "Hola mundo"

with open("archivo.txt", "w") as file:
    file.write(texto)
```

8. Archivos CSV

Para utilizarla, utilizarémos la librería `csv`:

```
with open("archivo.csv", "r") as file:
    lectura = csv.reader(file)
    for linea in lectura:
        print(linea)
```

Para escribir en un archivo, utilizamos la clase `writer`:

```
with open("archivo.csv", "w") as file:
    writer = csv.writer(file)
    for fila in datos:
        writer.writerow(fila)
```

Para leer:

```
with ... as file:
    reader = csv.reader(file)
    for linea in reader:
        print(linea)
```

También podemos utilizar diccionarios:

```
    reader = csv.DictReader(file)
    for row in reader:
    print(row) #Formato diccionario
```

Y ahora para escribir

```
    writer = csv.DictWriter(file, fieldnames=["nombre", "edad", "ciudad"])
    writer.writeheader()
    for row in data:
        writer.writerow(row)
```

9. JSON

Para leer un archivo:

```
    data = json.load(file)
```

Y para escribir datos:

```
    json.dump(data, file)
```