Clases y objetos

En la POO se usan entidades que representan elementos del probelma a resolver y tienen atributos y comportamientos. Estas entidades se denomian objetos y Python proporciona soporte para este paradigma

La POO permite que el desarrollo de grandes proyectos de software sea más fácil e intuitivo, al representar en el software objetos del mundo real y sus relaciones

A diferencia de la programación estructurada, que está centrada en las funciones, la POO se basa en la definición de clases y objetos.

Definición de clases

Los nombres de las clases se escriben camelCase. Se definen con la palabra clave class seguida del nombre de la clase y dos puntos

```
class Persona:
  # Atributo
  #los objetos que pertenecen a la clase
  piernas= 2 #atributo

# instanciamos un objeto de la clase Persona
juan= Persona()

# imprimimos un atributo del objeto
print(juan.piernas) # 2
```

Los objetos pueden tener sus propios atributos, llamados atributos de instancia. Una manera de crearlos es usar directamente la notación punto

```
class Persona:
  piernas= 2

juan= Persona()
juan.edad= 34 #creamos un atributo para el objeto
print(juan.edad) #mostramos el atributo creado
```

Las variables dentro de la clase (atributos de clase) son compartidas por todos los objetos instanciados. Se definen dentro de la clase pero fuera de sus métodos

```
class Persona:
  piernas= 2

juan= Persona()
juan.edad= 34
print(juan.edad) #34
print(Persona.piernas) #2
```

Métodos

Los métodos permiten a los objetos de una clase realizar acciones. Se declaran con def, como las funciones pero dentro de la clase. Reciben parámetros y uno de ellos, el primero **self** es obligatorio

```
class Persona():
  piernas= 2
  def caminar(self):
    print("Está caminando")

juan= Persona() # instanciamos un objeto
juan.caminar() #invocamos el método caminar()
```

self hace referencia a la instancia perteneciente a la clase

Método constructor

Un constructor es un método que permite a la clase asignar valores a los atributos. Su primer parámetro es self, y los demás los requeridos para la inicialización. Luego de instanciar el objeto, establecemos los valores de los atributos mediante el constructor, y ya podemos utilizarlos normalmente.
Veamos un ejemplo:

```
class Persona():
  # Método constructor
  def constructor(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad
  def identificarse(self): # Método normal
    print(f"Hola. Soy {self.nombre} y tengo {self.edad} años.")

persona1 = Persona() # Instanciamos
persona1.constructor("Juan", 42)
persona1.identificarse()
persona1.edad = 43 # Modificamos la edad
persona1.identificarse()
```

Método _ str _ ()

Para mostrar objetos, Python provee otro método especial, llamado **str** , que debe devolver una cadena de caracteres con lo que queremos mostrar. Este método se invoca cada vez que se llama a la función str, por ejemplo, al imprimir el objeto. El método **str** tiene un solo parámetro, self.

```
#Creamos la clase "Alumno":
class Alumno:
  def __init__(self, nombre, nota):
  self.nombre = nombre
  self.nota = nota

  def __str__(self):
    return "La nota de {self.nombre} es {self.nota}"

alumno1 = Alumno("Pedro", 7)
print(alumno1) # La nota de Pedro es 7
alumno1.nota = 10
print(alumno1) # La nota de Pedro es 10
```

Método _ del _ ()

El método especial **del**() se invoca automáticamente cuando el objeto se elimina de la memoria. Se puede utilizar para realizar alguna acción especial cuando tiene lugar este evento. Su sintaxis es la que vemos en el ejemplo, y tiene como único parámetro self. Los objetos se borran con del, o se eliminan al finalizar el programa

Método __ init __ ()

El método `__init__()` es un método especial en Python que se utiliza para inicializar los atributos de un objeto cuando se crea una instancia de una clase. Se conoce como el "constructor" de la clase.

Cuando se crea una instancia de una clase, el método `__init__()` se llama automáticamente. Este método puede aceptar argumentos, que se utilizan para establecer los valores iniciales de los atributos de la instancia. Por lo general, el primer argumento del método `__init__()` es `self`, que se refiere a la instancia recién creada.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Colaboración entre clases y encapsulamiento

La colaboración entre clases se manifiesta como la posibilidad de usar en una clase objetos que son instancias de una clase diferente. Este mecanismo potencia el alcance que tiene la POO, facilitando aún más el desarrollo de soluciones a problemas complejos sin necesidad de escribir código extenso o difícil de leer

Composición

Cada clase que definimos es un nuevo tipo de daros, con sus atributos y métodos. Y al igual que los demás tipos de datos, los objetos instanciados a partir de ellas se pueden utilizar como parte de colecciones o incluso ser parte de otras clases.

La composición es uno de los conceptos fundamentales de la POO. Tiene lugar cuando una clase hace referencia a uno o más objetos de otras clases, como una variable de instancia.

Al usar el nombre de la clase o al crear el objeto, se accede a los miembros de una clase dentro de otra clase. La composión permite crear tipos complejos mediante la combinación de objetos de diferentes clases

Este ejemplo es tener la clase Película. Se comporta como una clase que es parte de ya que ss instancias serán parte de la clase Catalogo

```
class Pelicula:
# Constructor de clase
def __init__(self, titulo, duracion, lanzamiento):
self.titulo = titulo
self.duracion = duracion
self.lanzamiento = lanzamiento
print(f'Se ha creado la película: {self.titulo}')
def __str__(self):
return f'{self.titulo} ({self.lanzamiento})'
```

La clase Catalogo se comporta como clase que tiene ya que uno de sus atributos es un objeto de la clase Pelicula

```
class Catalogo:
peliculas = [] # Esta de objetos de la clase Pelicula
# Constructor de clase
def __init__(self, peliculas=[]):
```

```
Catalogo.peliculas = peliculas
def agregar(self, p): # p es un objeto Pelicula
Catalogo.peliculas.append(p)
def mostrar(self):
for p in Catalogo.peliculas:
print(p) # Print toma por defecto str(p)
```

El programa principal sería

```
# Instanciamos una película
peli1 = Pelicula("El Padrino", 175, 1972)
# Instanciamos un catálogo que contiene una pelicula
c = Catalogo([peli1])
c.mostrar()
# Añadimos una nueva película al catálogo:
c.agregar(Pelicula("El Padrino: Parte 2", 202, 1974))
c.mostrar(
```

Encapsulamiento

Si el nombre de un atributo está precedido por un guión bajo, significa que es un **atributo protegido.** Solo puede ser accedido por esa clase y es una buena práctica para los atributos de uso interno, sobre todo en programas que recurren a muchas clases

```
class Vehiculo:
def __init__(self, marca, color, placa):
self._marca = marca
self._color = color
self._placa = placa
```

Encapculamiento se refiere al ocultamiento de los atributos o métodos de una clase al exterior, para que no se pueda acceder ni modificar desde fuera. Por defecto Python no oculta los atributos y métodos de una clase al exterior.

Setters

Un **setter** permite crear métodos que permiten modificar el valor de un atributo privado. En Python se declaran escribiendo antes del método un decorador con su nombre seguido de **.setter**

Junto con el decorador @property permiten leer y escribir atributos privados de manera segura

```
class Bebidas:
def __init__(self):
self.__bebida = 'Naranja'
@property
def favorita(self):
return f"La bebida preferida es:
{self.__bebida}"
@favorita.setter
def favorita(self, bebida):
self.__bebida = bebida
# Programa principal
obj1 = Bebidas()
obj1.favorita = "Pomeloc
print(obj1.favorita) # Pomelo
```

Los atributos de una clase pueden presentar tres niveles

- **Públicos:** creando un objeto de dicha clase y usando la sintaxis del punto, podemos acceder y modificar desde otra clase cualquier atributo
- **Protegidos:** solo puede ser accedido y modificado por la clase sí misma
- **Privados:** pueden ser accedidos únicamente desde la clase donde fueron definidos. Su nombre empieza con dobles guiones bajo

Getters

Un getter es un mecanismo que permite acceder a un método o atributo privado. En Python se declaran creando un método con el decorador @property

```
class Bebidas:
def __init__(self):
self.__bebida = 'Naranja'
@property
def favorita(self):
return f"La bebida preferida es:
{self.__bebida}"
obj1 = Bebidas()
print(obj1.favorita)
```

Herencia y polimorfismo

La herencia es un mecanismo de la POO que permite crear clases nuevas a partir de clases preexistentes. Usando este concepto, las clases nuevas pueden tomar atirbutos y métodos de clasea anteriores.

La clase que aporta los métodos y atributos para heredar se la denomina clase **padre** y las que se construyen a partir de una clase padre se denominan clases **hijas**

Herencia simple

Este ejemplo el objeto hijo1 heredó los métodos y atributos de la superclase Pare y podemos usarlos en hijo1

```
class Padre: # Superclase
  def __init__(self):
    self.apellido = "Volpin"
  def llevar(self):
    print("Papá me lleva al colegio.")

class Hijo(Padre): # Subclase
  def estudiar(self):
    print("Estoy en el colegio.")

hijo1 = Hijo() # Instanciamos hijo1
hijo1.llevar() # Papá me lleva al colegio.
hijo1.estudiar() # Estoy estudiando
print(hijo1.apellido) # Volpin (heredado)
```

De una superclase se puede construir muchas subclases derivadas o clases que heredan de ellas. Por ejemplo, de la superclase Persona podríamos derivar Docente, Empleado, Cliente, Proveedor, etc.

En el diseño de jerarquías no siempre es fácil decidir cuando una clase debe extender a otra. La regla práctica para decidir si una clase S puede ser definida como heredera de otra T es que debe cumplirse que **S es un T.** Por ejemplo Perro es un Animal pero Vehiculo no es un Motor

La herencia múltiple ocurre cuando una subclase deriva de dos o más clases base. Al escribir el código de la subclase, las superclases de las que heredará métodos y atributos se indican de la misma forma,separando cada una con una coma

Vemos como el objeto hijo1 ha heredado los métodos de las superclase Padre y Madre. Podemos usar en hijo1 los métodos propios o los de las superclases. EN caso de que ambas superclases tengan un método con el mismo nombre, se hereda el que se escriba primero en la declaración

```
class Padre: # Superclase #
  def llevar(self):
    print("Papá me lleva al colegio")

class Madre: # Superclase #
  def programar(self):
    print("Mamá programa en Python")

class Hijo(Padre,Madre) # Subclase #
  def amar(self):
    print("Quiero a mis padres")

hijo1= Hijo()
hijo1.llevar()
hijo1.programar()
hijo1.amar()
```

Clases abstractas

Otro concepto importante de la POO es el de las clases abstractas. Son clases en las que se pueden definir tanto métodos como propiedades pero que no pueden ser instanciadas directamente. Solamente se puede usar para construir subclases (como si fueran moldes), permitiendo tener una única implementación de los métodos compartidos. Esto evita la duplicación de código.

Las clases abstractas definen una interfaz común para las subclases evitando así la necesidad de duplicar código, imponiendo además los métodos que deben ser implementados para evitar inconsistencias entre las subclases

Propiedades de las clases abstractas

- No pueden ser instanciadas, simplemente proporcionan una interfaz para las subclases derivadas evitando así la duplicación de código
- No es obligadorio que tengan una implementación de todos los métodos necesarios. Pudiendo ser estos abstractos. Los métodos abstractos son aquellos que solamente tienen una declaración pero no una implementación detallada de las funcionalidades
- Las clases derivadas de las clases abstractas deben implementar necesariamente todos los métodos abstractos para poder crear una clase que se ajuste a la interfaz definida. En el caso de que no se defina alguno de los métodos no se podrá crear la clase

Creación

Para poder crear clases abstractas es necesario importar la clase ABC y el decorador abstractmethod del modulo abc. Si se inenta crear una instancia de la clase Animal, Python no lo permite.

Si la clase no hereda de ABC o contiene por lo menos un método abstracto, Python permitirá crear instancias de la clase

```
from abc import ABC, abstractmethod

class Animal(ABC):
  @abstractmethod
  def mover(self):
    pass

perro= Animal()
```

Las subclases tienen que implementar todos los métodos abstractos definidos en la clase abstracta o Python no permitirá instanciar la clase hija

Desde los métodos de las subclases se pueden usar las implementaciones de la clase abstracta con el comando super() seguido del nombre del método. Por ejemplo

```
# Clase abstracta
class Animal(ABC):
  @abstractmethod
  def mover(self):
    pass

# Método de la subclase
def emitir_sonido(self):
  super().emitir_sonido()
```

Ejemplo: crearemos una clase abstracta Animal de la que heredarán métodos dos subclases (Gato y Perro)

- No podremos instanciar directamente objetos de la clase Aniamal, ya que se trata de una clase abstracta
- La clase Animal debe heredar de ABC e implementar solo métodos abstractos, para que funcione como clase abstracta
- Gato y Perro completan los métodos abstractos de Animal, cada uno con la característica propia de los objetos de esas clases

```
# Clase abstracta Animal
from ABC import ABC, abstractmethod

class Animal (ABC):
  @abstractmethod
  def mover(self):
    pass
@abstractmethod
def emitir_sonido(self):
  print("Animal dice: ")

# Subclase Gato
class Gato(Animal):
  def mover(self):
    print("El gato se mueve.")

def emitir_sonido(self):
```

```
    super().emitir_sonido()
    print("Miau")
```

Polimorfismo

Pilar básico de la POO, se refiere a que los objetos de diferentes clases pueden ser accedidos usando la misma interfaz, mostrando un comportamiento distinto según cómo sean accedidos

En la POO, el polimorfismo se logra a través de la herencia y la implementación de interfaces.

Ejemplo: definimos tres clases, Pato, Perro y Cerdo, todas con un método hablar(). Cada una de las clases implementa la versión que necesita del método, pero es importante ver que en todas tienen el mismo nombre

```
class Pato:
  def hablar(self):
    print("Cuac!")

class Perro:
  def hablar(self):
    print("Guau!")

class Cerdo:
  def hablar(self):
    print("Oink!")
```

En el programa principal define una función que recibe un objeto y usa su método hablar(). Esto es posible gracias al poliformismo: no es necesario escribir código para acceder a un mismo atributo o método de objetos de distinta clase, cuando estos tienen el mismo nombre

```
def hacer_hablar(x):
  x.hablar()

# Creamos un pato y lo hacemos hablar
mi_pato()= Pato()
hacer_hablar(mi_pato)

# Creamos un perro y lo hacemos hablar
mi_perro()= Perro()
hacer_hablar(mi_perro)
```