1000 PROGRAMADORES

Introducción a la
Programación en Python

MÓDULO 6

Programación estructurada vs POO.
Clases, objetos y atributos.
Métodos de clases y métodos especiales.
Objetos dentro de objetos.
Encapsular atributos y métodos.

Ministerio de Educación
Cultura, Ciencia y Tecnología
Gobierno de Salta

Ministerio de Economía
y Servicios Públicos
Gobierno de Salta

Universidad
Nacional de Salta

INTRODUCCIÓN

Entre los paradigmas de programación soportados por Python destacan el procedural, el funcional y la orientación a objetos. De este último y de su aplicación en el lenguaje nos ocuparemos en este módulo.

En líneas generales, la orientación a objetos se basa en la definición e interacción de unas unidades mínimas de código, llamadas objetos, para el diseño y desarrollo de aplicaciones.

Este paradigma comenzó a popularizarse a mediados de los años 90 y, en la actualidad, es uno de los más utilizados. Algunos lenguajes, como Java, están basados completamente en el uso del mismo. Otros como Ruby, C++ y PHP5 también ofrecen un amplio y completo soporte de la orientación a objetos. Se le conoce también por sus siglas en inglés: OOP (Object Oriented Programming) ó POO traducido al español.

En la actualidad los frameworks web, hacen un uso intensivo de este paradigma. Entre las claras ventajas de la orientación a objetos podemos destacar la reusabilidad de código, la modularidad y la facilidad para modelar componentes del mundo real.

PROGRAMACIÓN ESTRUCTURADA vs POO

Usando programación estructurada, de cierto modo, los programas son más fáciles de entender. Un programa estructurado puede ser leído en secuencia, de arriba hacia abajo, sin necesidad de estar saltando de un sitio a otro en la lógica, lo cual es típico de otros estilos de programación. La estructura del programa es más clara puesto que las instrucciones están más relacionadas entre sí, por lo que es más fácil comprender lo que hace cada función.

Algunas de sus ventajas son:

- Reducción del esfuerzo en las pruebas. El programa se puede tener listo para producción normal en un tiempo menor del tradicional; por otro lado, el seguimiento de las fallas se facilita debido a la lógica más visible, de tal forma que los errores se pueden detectar y corregir más fácilmente. Programas más sencillos y más rápidos.

- Aumento de la productividad del programador. Los programas quedan mejor documentados internamente. La programación estructurada permite la escritura de programas fáciles de leer y modificar. En un programa estructurado el flujo lógico se gobierna por tres estructuras de control básico: secuenciales, repetitivas y selectivas.

La Programación Orientada a Objetos (POO) es una forma especial de programar, más cercana a la forma de expresar las cosas en la vida real que otros tipos de programación. Con la POO tenemos que aprender a pensar las cosas de una manera distinta, para escribir nuestros programas en términos de objetos, propiedades, métodos y otros conceptos nuevos.

Algunas de sus ventajas son:

- Reusabilidad: Cuando hemos diseñado adecuadamente las clases, se pueden usar en distintas partes del programa y en numerosos proyectos.

- Mantenibilidad: Debido a la sencillez para abstraer el problema, los programas orientados a objetos son más sencillos de leer y comprender, pues nos permiten ocultar detalles de implementación dejando visibles sólo aquellos detalles más relevantes.

- Modificabilidad: La facilidad de añadir, suprimir o modificar nuevos objetos nos permite hacer modificaciones de una forma muy sencilla.

CLASES, OBJETOS Y ATRIBUTOS

Dentro del contexto de la POO, un objeto es un componente que tiene un rol específico y que puede interactuar con otros. En realidad, se trata de establecer una equivalencia entre un objeto del mundo real con un componente software. De esta forma, el modelado para la representación y resolución de problemas del mundo real a través de la programación, es más sencillo e intuitivo.

Si miramos a nuestro alrededor, todos los objetos presentan dos componentes principales: un conjunto de características y propiedades y un comportamiento determinado. Por ejemplo, pensemos en una moto. Esta tiene características como color, marca y modelo. Asimismo, su comportamiento puede ser descrito en función de las operaciones que puede realizar, por ejemplo, frenar, acelerar o girar. De la misma forma, en programación, un objeto puede tener atributos y se pueden definir operaciones que puede realizar. Los atributos equivaldrían a las características de los objetos del mundo real y las operaciones se relacionan con su comportamiento.

Es importante distinguir entre clase y objeto. Para ello, introduciremos un nuevo concepto: la instancia. La definición de los atributos y operaciones de un objeto se lleva a cabo a través de una clase. La instanciación es un mecanismo que nos permite crear un objeto que pertenece a una determinada clase. De esta forma, podemos tener diferentes objetos que pertenezcan a la misma clase.

Cosas cotidianas como un perro o un auto pueden ser representadas con **clases**. Estas clases tienen diferentes características, que en el caso del perro podrían ser la edad, el nombre o la raza. Llamaremos a estas características, **atributos**.

Por otro lado, las clases tienen un conjunto de funcionalidades o cosas que pueden hacer. En el caso del perro podría ser andar o ladrar. Llamaremos a estas funcionalidades **métodos**.

Por último, pueden existir diferentes tipos de perro. Podemos tener uno que se llama Toby o el del vecino que se llama Laika. Llamaremos a estos diferentes tipos de perro objetos. Es decir, el concepto abstracto de perro es la clase, pero Toby o cualquier otro perro particular será el **objeto**.

Definir clases

Lo primero es crear una clase, para ello usaremos el ejemplo de perro.

```
# Creando una clase vacía
class Perro:
      pass
```

Se trata de una clase vacía y sin mucha utilidad práctica, pero es la mínima clase que podemos crear. En realidad el uso del `pass` no hace realmente nada, pero se generaría un error si después de los : no tenemos contenido.

Ahora que tenemos la **clase**, podemos crear un **objeto** de la misma. Podemos hacerlo como si se tratase de una variable normal. Nombre de la variable igual a la clase con los (). Dentro de los paréntesis irían los parámetros de entrada si los hubiera.

```
# Creamos un objeto de la clase perro
mi_perro = Perro()
```

Definir atributos

A continuación vamos a añadir atributos a nuestra clase, pero antes es importante distinguir que existen dos tipos de atributos:

- Atributos de **instancia**: Pertenecen a la instancia de la clase o al objeto. Son atributos particulares de cada instancia, en nuestro caso de cada perro.
- Atributos de **clase**: Se trata de atributos que pertenecen a la clase, por lo tanto serán comunes para todos los objetos.

Empecemos creando un par de **atributos de instancia** para nuestro perro, el nombre y la raza. Para ello creamos un método `__init__` que será llamado automáticamente cuando creemos un objeto. Se trata del **constructor**.

```
class Perro:
      # El método __init__ es llamado al crear el objeto
      def __init__(self, nombre, raza):
            print(f"Creando perro {nombre}, {raza}")

            # Atributos de instancia
            self.nombre = nombre
            self.raza = raza
```

Ahora que hemos definido el método *init* con dos parámetros de entrada, podemos crear el objeto pasando el valor de los atributos. Usando type() podemos ver cómo efectivamente el objeto es de la clase Perro.

```
mi_perro = Perro("Toby", "Bulldog")
print(type(mi_perro))
# Creando perro Toby, Bulldog
# <class '__main__.Perro'>
```

El `self` que se pasa como parámetro de entrada del método es una variable que representa la instancia de la clase, y deberá estar siempre ahí.

El uso de __init__ y el doble __ no es una coincidencia. Cuando veas un método con esa forma, significa que está reservado para un uso especial del lenguaje. En este caso sería lo que se conoce como **constructor**.

Por último, podemos acceder a los atributos usando el objeto y punto (.). Por lo tanto.

```
class Perro:
      def __init__(self, nombre, raza):
            print(f"Creando perro {nombre}, {raza}")
            self.nombre = nombre
            self.raza = raza

mi_perro = Perro("Toby", "Bulldog")
print(type(mi_perro))

print(mi_perro.nombre)
print(mi_perro.raza)
```

```
Creando perro Toby, Bulldog
<class '__main__.Perro'>
Toby
Bulldog
```

---

salida

Hasta aquí hemos definido atributos de instancia, ya que son atributos que pertenecen a cada perro concreto. Ahora vamos a definir un atributo de clase, que será común para todos los perros. Por ejemplo, la especie de los perros es algo común para todos los objetos Perro.

```
class Perro:
    # Atributo de clase
    especie = 'mamífero'

    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza
```

Dado que es un atributo de clase, no es necesario crear un objeto para acceder al atributo. Se puede acceder también al atributo de clase desde el objeto. De esta manera, todos los objetos que se creen de la clase perro compartirán ese atributo de clase, ya que pertenecen a la misma. Veamos ambos casos:

```
class Perro:
      def __init__(self, nombre, raza):
            print(f"Creando perro {nombre}, {raza}")
            self.nombre = nombre
            self.raza = raza

print(Perro.especie)
# ---------------------------------
mi_perro = Perro("Toby", "Bulldog")
mi_perro.especie
```

```
mamífero
mamífero
```

---

salida

Definir métodos

En realidad cuando usamos __init__ anteriormente ya estábamos definiendo un método, solo que uno especial. Ahora vamos a ver cómo definir métodos que le den alguna funcionalidad interesante a nuestra clase, siguiendo con el ejemplo de perro.

Vamos a codificar dos métodos, ladrar y caminar. El primero no recibirá ningún parámetro y el segundo recibirá el número de pasos que queremos andar. Como hemos indicado anteriormente self hace referencia a la instancia de la clase. Se puede definir un método con def y el nombre, y entre () los parámetros de entrada que recibe, donde siempre tendrá que estar self el primero.

```
class Perro:
    # Atributo de clase
    especie = 'mamífero'

    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza

    def ladra(self):
        print("Guau")

    def camina(self, pasos):
        print(f"Caminando {pasos} pasos")
```

Por lo tanto si creamos un objeto mi_perro, podremos hacer uso de sus métodos llamándolos con **.** y el nombre del método. Como si se tratase de una función, pueden recibir y devolver argumentos.

```
mi_perro = Perro("Toby", "Bulldog")
mi_perro.ladra()
mi_perro.camina(10)
```

```
Guau
Caminando 10 pasos
```

---

salida

MÉTODOS DE CLASES Y MÉTODOS ESPECIALES

Ya hemos visto cómo se pueden crear métodos con **`def`** dentro de una clase, pudiendo recibir parámetros como entrada y modificar el estado (ej. atributos) de la instancia. Además usando decoradores pueden crearse diferentes tipos de métodos:

- Métodos de instancia "normales" que ya hemos visto como metodo()
- Métodos de clase usando el decorador @classmethod
- Métodos estáticos usando el decorador @staticmethod

En la siguiente clase tenemos un ejemplo donde definimos los tres tipos de métodos.

```
class Clase:
    def metodo(self):
        return 'Método normal', self

    @classmethod
    def metododeclase(cls):
        return 'Método de clase', cls

    @staticmethod
    def metodoestatico():
        return "Método estático"
```

Métodos de instancia

Los métodos de instancia reciben como parámetro de entrada tanto a `self` como a otros argumentos de entrada. El uso de "self" es totalmente arbitrario. Se trata de una convención acordada por los usuarios de Python.

```
class Clase:
    def metodo(self, arg1, arg2):
        return 'Método normal', self

#llamada al objeto
mi_clase = Clase()
mi_clase.metodo("a", "b")
```

```
('Método normal', <__main__.Clase at 0x7eff6cd48210>)
```

---

salida

Entonces, los métodos de instancia:

- Pueden acceder y modificar los atributos del objeto.
- Pueden acceder a otros métodos.
- Dado que desde el objeto self se puede acceder a la clase con "self.class", también pueden modificar el estado de la clase.

Métodos de clase (classmethod)

A diferencia de los métodos de instancia, los métodos de clase reciben como argumento **`cls`**, que hace referencia a la clase. Por lo tanto, pueden acceder a la clase pero no a la instancia.

```
class Clase:
    @classmethod
    def metododeclase(cls):
        return 'Método de clase', cls

#Se pueden llamar sobre la clase
Clase.metododeclase()
#Se pueden llamar sobre el objeto
mi_clase = Clase()
mi_clase.metododeclase()
```

```
('Método de clase', __main__.Clase)
#Misma salida para ambas llamadas
```

---

salida

Por lo tanto, los métodos de clase:

- No pueden acceder a los atributos de la instancia.
- Pero sí pueden modificar los atributos de la clase.

Métodos estáticos (staticmethod)

Los métodos estáticos se pueden definir con el decorador @staticmethod y no aceptan como parámetro ni la instancia ni la clase. Es por ello que **no pueden modificar el estado ni de la clase ni de la instancia**. Pero por supuesto pueden aceptar parámetros de entrada.

```
class Clase:
    @staticmethod
    def metodoestatico():
        return "Método estático"
mi_clase = Clase()
Clase.metodoestatico()
mi_clase.metodoestatico()
```

```
Método estático
```

salida

Por lo tanto el uso de los métodos estáticos puede resultar útil para indicar que un método no modificará el estado de la instancia ni de la clase. Es cierto que se podría hacer lo mismo con un método de instancia por ejemplo, pero a veces resulta importante indicar de alguna manera estas peculiaridades.

En otras palabras, los métodos estáticos se podrían ver como funciones normales, con la salvedad de que van ligadas a una clase concreta.

Métodos Mágicos

Son métodos especiales con nombres fijos, con sintaxis torpe, es decir, los guiones bajos dobles al principio y al final.

Es difícil nombrarlos, por ejemplo __init__ sería "*Subrayado subrayado inicial subrayado subrayado*", tal vez, "*Doble subrayado init doble subrayado*" es mucho mejor, pero la forma ideal es ***"dunder init dunder***". Es por eso que los métodos mágicos a veces se denominan métodos dunder.

Lo mágico de estos métodos es que no tienen que invocarse directamente. La invocación se realiza internamente. Cuando crea una instancia x de una clase A con la declaración "x = A ()", Python hará las llamadas necesarias a __new__ y __init__.

Varias veces hasta aquí, hemos usado el signo más (+) para agregar valores numéricos, concatenar cadenas o combinar listas. Es decir, que el operador + se comportó de acuerdo al tipo de dato involucrado, como:

- Una suma de escalares
- Una concatenación de cadenas de texto
- Una unión de listas

Recordemos algunos ejemplos:

```
print(3.8 + 9)
print("Peter" + " " + "Pan")
print([3,6,8] + [7,11,13])
```

```
12.8
Peter Pan
[3, 6, 8, 7, 11, 13]
```

salida

A esta implementación se la conoce como “sobrecarga” del operador, en inglés “overload”.

El mecanismo funciona así: si tenemos una expresión "x + y", con x siendo una instancia de la clase K, entonces Python verificará la definición de clase de K. Si K tiene un método __add__ se llamará con x .__ add __ (y), de lo contrario obtendremos un mensaje de error:

```
print("hola" + 9)
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-12-7f5d7548d713> in <module>()
----> 1 print("hola" + 9)
TypeError: can only concatenate str (not "int") to str
```

salida

A continuación veamos sobre cuáles operadores podemos trabajar implementado métodos mágicos para evitar la sobrecarga y así poder usar algunas operaciones particulares que necesitemos en nuestros programas.

Descripción general de los métodos mágicos

Operadores Binarios

Operador	Método
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Operadores Unitarios

Operador	Método
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

Operadores de Comparación

Operador	Método
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Analicemos la siguiente clase Longitud y cómo puede sobrecargar el operador "+" para su propia clase.
Para hacer esto, tenemos que sobrecargar el método __add__. Nuestra clase también contiene los métodos __str__ y __repr__. Las instancias de la clase Longitud contienen información de longitud o distancia. Los atributos de una instancia son self.value y self.unit.
Esta clase nos permite calcular expresiones con unidades mixtas como esta:

2,56 m + 3 km + 7,03 cm

```
class Length:

    __metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000}

    #constructor
    def __init__(self, value, unit = "m" ):
        self.value = value
        self.unit = unit

    #conversión a metros
    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]

    #suma de longitudes
    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit )
```

```
    #conversión a cadena
    def __str__(self):
        return str(self.Converse2Metres())

    #mostrar resultado
    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit + "')"

#podemos especificar la unidad de medida del argumento
#por defecto el argumento se considera como metros
z = Length(4.5, "km") + Length(1000)
print(z)

#objetivo inicial
L = Length
print(L(2.56,"m") + L(3,"km") + L(7.03,"cm"))
```

```
#los resultados se expresan en metros
5500.0
3002.6303
```

---

salida

OBJETOS DENTRO DE OBJETOS

Las clases son en realidad, un nuevo tipo de dato, y por todo lo que hemos estudiado de este lenguaje debería resultar natural que se puedan poner en colecciones e incluso utilizarlos dentro de otras clases.

A través del siguiente ejemplo que describe un catálogo de películas podrán analizar cómo se diseña e implementa este anidamiento:

```
class Pelicula:

    # Constructor de clase
    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print('Se ha creado la película:', self.titulo)

    def __str__(self):
        return '{} ({})'.format(self.titulo, self.lanzamiento)


class Catalogo:

    peliculas = []  # Lista que contendrá objetos de clase Pelicula

    def __init__(self, peliculas=[]):
        self.peliculas = peliculas

    def agregar(self, p):  # p será un objeto Pelicula
        self.peliculas.append(p)

    def mostrar(self):
        for p in self.peliculas:
            print(p)  # Print toma por defecto str(p)


p = Pelicula("El Padrino", 175, 1972)
c = Catalogo([p])  # Se añade una lista con una película
c.mostrar()
c.agregar(Pelicula("El Padrino: Parte 2", 202, 1974))  #Otra peli
c.mostrar()
```

```
Se ha creado la película: El Padrino
El Padrino (1972)
Se ha creado la película: El Padrino: Parte 2
El Padrino (1972)
El Padrino: Parte 2 (1974)
```

---

salida

Se deja a los lectores el análisis de la implementación de las clases, invitándolos a pensar y desarrollar otros ejemplos.

ENCAPSULAR ATRIBUTOS Y MÉTODOS

El encapsulamiento o encapsulación en programación es un concepto relacionado con la programación orientada a objetos, y hace referencia al ocultamiento de los estados internos de una clase al exterior. Dicho de otra manera, encapsular consiste en hacer que los atributos o métodos internos a una clase no se puedan acceder ni modificar desde fuera, sino que tan solo el propio objeto pueda acceder a ellos.

Para quienes conozcan Java, le resultará un término muy familiar, pero en Python es algo distinto. Digamos que Python por defecto no oculta los atributos y métodos de una clase al exterior pero se puede simular precediendo atributos y métodos con dos barras bajas __ como indicando que son "especiales".

Ejemplos para el caso de los atributos:

```
class Ejemplo:
   atributo_comun = "Soy un atributo comun."
   __atributo_privado = "Soy un atributo inalcanzable desde fuera."

e = Ejemplo()
print(e.atributo_comun)
print(e.__atributo_privado)
```

```
Soy un atributo comun.                          # Todo OK
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-2f3c2f6dba24> in <module>()
----> 8 print(e.__atributo_privado)
AttributeError: 'Ejemplo' object has no attribute
'__atributo_privado'                            # ERROR
```

salida

Ejemplos para el caso de los métodos:

```
class Ejemplo:
    def metodo_comun(self):
        print("Soy un método común.")
    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera.")

e = Ejemplo()
e.metodo_comun()
e.__metodo_privado()
```

```
Soy un método común.                                    # Todo OK
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-16-587377989bbd> in <module>()
----> 9 e.__metodo_privado()
AttributeError: 'Ejemplo' object has no attribute
'__metodo_privado'                                      # ERROR
```

salida

En determinadas ocasiones es importante ocultar el estado interno de los objetos al exterior, para evitar que sean modificados de manera incorrecta. Para acceder a esos datos se deben crear métodos públicos que hagan de interfaz, lo que da lugar a las ***propiedades(@property)***, que no son más que atributos protegidos con interfaces de acceso:

```
class Clase:
    def __init__(self, mi_atributo):
        self.__mi_atributo = mi_atributo

    @property                                           # Usando decorador
    def mi_atributo(self):
        return self.__mi_atributo

e = Clase("valor de atributo")
print(e.mi_atributo)
```

```
valor de atributo
```

salida

```
class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera."

    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera.")

    def atributo_publico(self):     # Definiendo métodos públicos
        return self.__atributo_privado

    def metodo_publico(self):       # Definiendo métodos públicos
        return self.__metodo_privado()

e = Ejemplo()
print(e.atributo_publico())
e.metodo_publico()
```

```
Soy un atributo inalcanzable desde fuera.
Soy un método inalcanzable desde fuera.
```

salida

HERENCIA

La herencia permite a los programadores crear clases que se basan en clases existentes, y esto permite que una clase creada a través de la herencia herede los atributos y métodos de la clase principal. Esto significa que la herencia admite la reutilización del código. Los métodos o, en general, el software heredado por una subclase se considera reutilizado en la subclase. Las relaciones de objetos o clases a través de la herencia dan lugar a un grafo dirigido.

La clase de la que hereda una clase se llama padre o superclase. Una clase que hereda de una superclase se llama subclase, también llamada clase heredera o clase hija. Las superclases a veces también se denominan ancestros.

Existe una relación jerárquica entre clases. Es similar a las relaciones o categorizaciones que conocemos de la vida real.

Pensemos en vehículos, por ejemplo. Las bicicletas, los automóviles, los autobuses y los camiones son vehículos. Las camionetas, camionetas, autos deportivos, descapotables y familiares son todos autos y por ser autos también son vehículos. Podríamos implementar una clase de vehículo en Python, que podría tener métodos como acelerar y frenar. Los automóviles, autobuses y camiones y bicicletas se pueden implementar como subclases que heredarán estos métodos del vehículo.

Desarrollaremos el ejemplo de una tienda que vende 3 tipos de productos: adornos, alimentos y libros. Todos los productos de la tienda tenían una serie de atributos comunes, como la referencia, el nombre, el pvp, pero algunos eran específicos de cada tipo.

Partiremos de una clase que contenga todos los atributos y luego usaremos la herencia para mejorar el planteo.

```
class Producto:
   def __init__(self, referencia, tipo, nombre,
                pvp,  descripcion, productor=None,
                distribuidor=None, isbn=None, autor=None):
        self.referencia = referencia
        self.tipo = tipo
        self.nombre = nombre
        self.pvp = pvp
        self.descripcion = descripcion
        self.productor = productor
        self.distribuidor = distribuidor
        self.isbn = isbn
        self.autor = autor

adorno = Producto('000A','ADORNO','Vaso Adornado',15,'Vaso de porcelana con dibujos')

print(adorno)
print(adorno.tipo)
```

```
<__main__.Producto object at 0x7fa9db7656d0>
ADORNO
```

salida

Superclases

La idea de la herencia es identificar una clase base (la superclase) con los atributos comunes y luego crear las demás clases heredando de ella (las subclases) y extendiendo sus campos específicos. En nuestro caso esa clase sería el Producto en sí mismo:

```
class Producto:
    def __init__(self,referencia,nombre,pvp,descripcion):
        self.referencia = referencia
        self.nombre = nombre
        self.pvp = pvp
        self.descripcion = descripcion

    def __str__(self):
        return f"REFERENCIA\t {self.referencia}\n" \
               f"NOMBRE\t\t {self.nombre}\n" \
               f"PVP\t\t {self.pvp}\n" \
               f"DESCRIPCIÓN\t {self.descripcion}\n"
```

Subclases

Para heredar los atributos y métodos de una clase en otra sólo tenemos que pasarla entre paréntesis durante la definición:

```
class Adorno(Producto):
  pass

adorno = Adorno(2034, "Vaso adornado", 15, "Vaso de porcelana")
print(adorno)
```

```
REFERENCIA   2034
NOMBRE       Vaso adornado
PVP          15
DESCRIPCIÓN  Vaso de porcelana
```

---

salida

Se puede apreciar que es posible utilizar el comportamiento de una superclase sin definir nada en la subclase. Respecto a las demás subclases como se añaden algunos atributos, podríamos definirlas de esta forma:

```
class Alimento(Producto):
    productor = ""
    distribuidor = ""

    def __str__(self):
        return f"REFERENCIA\t {self.referencia}\n" \
               f"NOMBRE\t\t {self.nombre}\n" \
               f"PVP\t\t {self.pvp}\n" \
               f"DESCRIPCIÓN\t {self.descripcion}\n" \
               f"PRODUCTOR\t\t {self.productor}\n" \
               f"DISTRIBUIDOR\t\t {self.distribuidor}\n"

class Libro(Producto):
    isbn = ""
    autor = ""

    def __str__(self):
        return f"REFERENCIA\t {self.referencia}\n" \
               f"NOMBRE\t\t {self.nombre}\n" \
               f"PVP\t\t {self.pvp}\n" \
               f"DESCRIPCIÓN\t {self.descripcion}\n" \
               f"ISBN\t\t {self.isbn}\n" \
               f"AUTOR\t\t {self.autor}\n"
```

Ahora para utilizarlas simplemente tendríamos que establecer los atributos después de crear los objetos:

```
alimento = Alimento(2035, "Botella Aceite de Oliva", 5, "250 ML")
alimento.productor = "La Aceitera"
alimento.distribuidor = "Distribuciones SA"
print(alimento)

libro = Libro(2036, "Cocina Mediterránea",9, "Recetas sanas")
libro.isbn = "0-123456-78-9"
libro.autor = "Doña Juana"
print(libro)
```

```
REFERENCIA   2035
NOMBRE            Botella Aceite de Oliva
PVP          5
DESCRIPCIÓN  250 ML
PRODUCTOR         La Aceitera
DISTRIBUIDOR           Distribuciones SA

REFERENCIA   2036
NOMBRE            Cocina Mediterránea
PVP          9
DESCRIPCIÓN  Recetas sanas
ISBN         0-123456-78-9
AUTOR        Doña Juana
```

salida

Gracias a la flexibilidad de Python podemos manejar objetos de distintas clases masivamente de una forma muy simple. Vamos a empezar creando una lista con nuestros tres productos de subclases distintas:

```
productos = [adorno, alimento]
productos.append(libro)
print(productos)
```

```
[<__main__.Adorno object at 0x7fa9d4066910>,
<__main__.Alimento object at 0x7fa9d1d4a410>,
<__main__.Libro object at 0x7fa9d8116fd0>]
```

salida

Ahora si queremos recorrer todos los productos de la lista podemos usar un bucle *for*:

```
for producto in productos:
   print(producto, "\n")
```

```
REFERENCIA      2034
NOMBRE          Vaso adornado
PVP             15
DESCRIPCIÓN     Vaso de porcelana

REFERENCIA      2035
NOMBRE          Botella de Aceite de Oliva
PVP             5
DESCRIPCIÓN     250 ML
PRODUCTOR       La Aceitera
DISTRIBUIDOR    Distribuciones SA
```

```
REFERENCIA      2036
NOMBRE          Cocina Mediterránea
PVP             9
DESCRIPCIÓN     Recetas sanas y buenas
ISBN            0-123456-78-9
AUTOR           Doña Juana
```

salida

También podemos acceder a los atributos, siempre que sean compartidos entre todos los objetos:

```
for producto in productos:
   print(producto.referencia, producto.nombre)
```

```
2034 Vaso adornado
2035 Botella Aceite de Oliva
2036 Cocina Mediterránea
```

salida

Si un objeto no tiene el atributo al que queremos acceder nos dará error:

```
for producto in productos:
   print(producto.autor)
```

```
AttributeError                            Traceback (most recent call last)

<ipython-input-32-9a7f6afd4a4a> in <module>()
     11
     12 for producto in productos:
---> 13     print(producto.autor)

AttributeError: 'Adorno' object has no attribute 'autor'
```

salida

Sin embargo, podemos hacer una comprobación con la función *isinstance()* para determinar si una instancia es de una determinada clase y así mostrar unos atributos u otros:

```
for producto in productos:
  if( isinstance(producto, Adorno) ):
    print(producto.referencia, producto.nombre)
  elif( isinstance(producto, Alimento) ):
    print(producto.referencia, producto.nombre, producto.productor)
  elif( isinstance(producto, Libro) ):
    print(producto.referencia, producto.nombre, producto.isbn)
```

```
2034 Vaso adornado
2035 Botella Aceite de Oliva La Aceitera
2036 Cocina Mediterránea 0-123456-78-9
```

---

salida

BIBLIOGRAFÍA

[1] Python para todos. R. Gonzalez Duque. Licencia Creative Commons Reconocimiento 2.5 España.
Descarga gratuita de versión más reciente en http://mundogeek.net/tutorial-python/

[2] Python 3 al descubierto 2da Edición. A. Fernández Montoro. Alfaomega. Año 2013

[3] Tutoriales Web varios.