

# PYTHON



Innovación  
en Formación  
Profesional

## 2. PROGRAMACIÓN ORIENTADA A OBJETOS

UT 2

Programación Orientada a Objetos



## ÍNDICE

- 1. PROGRAMACIÓN ORIENTADA A OBJETOS
  - 1.1. Clases y objetos
  - 1.2. Constructor de una clase
  - 1.3. Atributos, atributos de datos y métodos
  - 1.4. Atributos de datos
  - 1.5. Métodos
  - 1.6. Atributos de clase y atributos de instancia
- 2. HERENCIA
  - 1.1. `isInstance()`
  - 1.2. `issubclass()`
  - 1.3. Herencia múltiple
- 3. ENCAPSULACIÓN: ATRIBUTOS PRIVADOS
- 4. POLIMORFISMO

## 1. PROGRAMACIÓN ORIENTADA A OBJETOS

Algunas particularidades de POO en Python son las siguientes:

- Todo es un objeto, incluyendo los tipos y clases.
- Permite herencia múltiple.
- No existen métodos ni atributos privados.
- Los atributos pueden ser modificados directamente.
- Permite “monkey patching”\*
- Permite “duck typing”\*\*
- Permite la sobrecarga de operadores.
- Permite la creación de nuevos tipos de datos.

*\*El Monkey Patching es una técnica de programación de los lenguajes dinámicos que consiste en modificar el código en tiempo de ejecución.*

*\*\*En duck typing, el programador solo se ocupa de los aspectos del objeto que van a usarse, y no del tipo de objeto que se trata.*

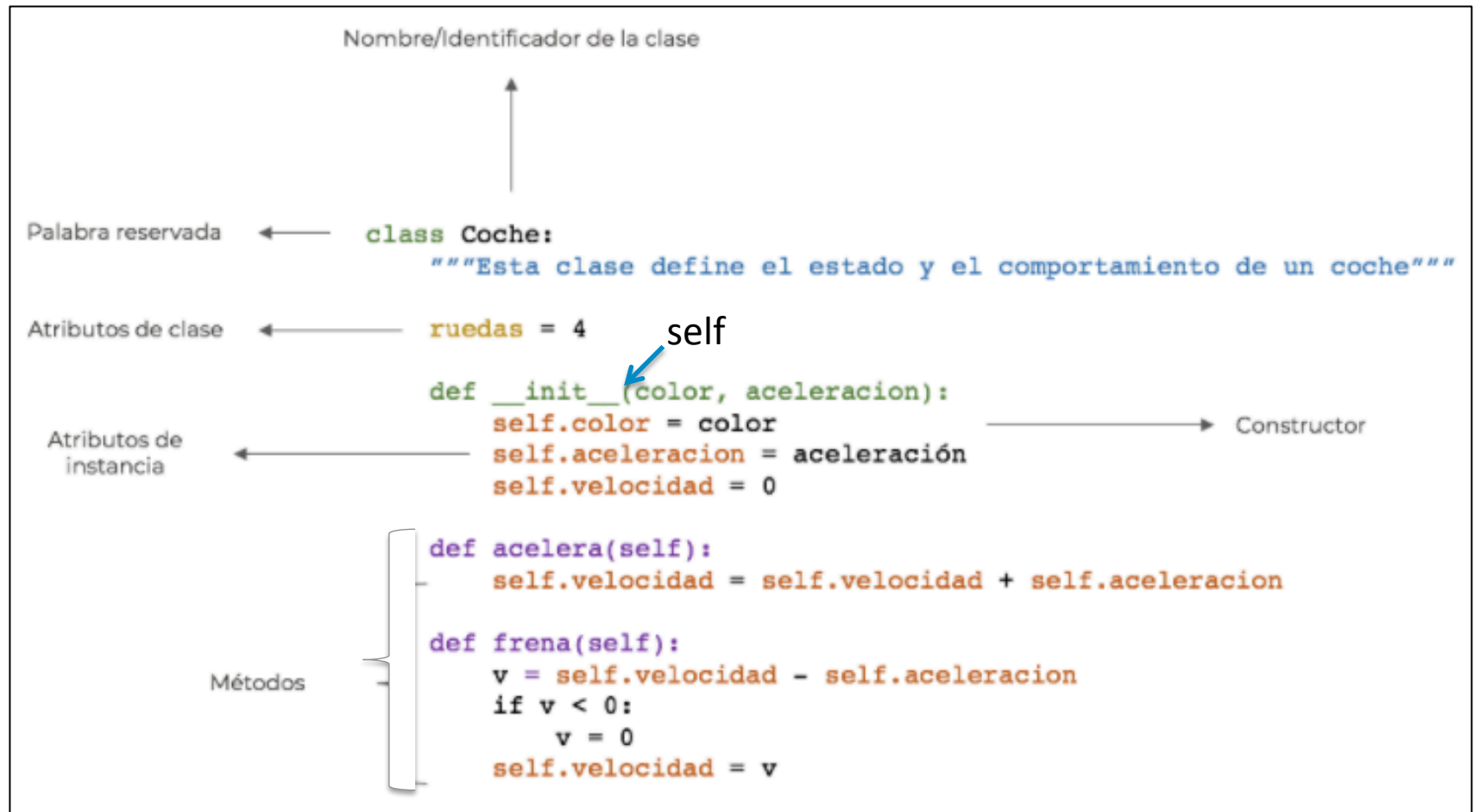
## 1.1. Clases y objetos

Una clase es una entidad que define una serie de elementos que determinan un estado (datos) y un comportamiento (operaciones sobre los datos que modifican su estado).

Por su parte, un objeto es una concreción o instancia de una clase.

Para definir una clase en Python se utiliza la palabra reservada ***class***. El siguiente esquema visualiza los elementos principales que componen una clase.

## 1.1. Clases y objetos



## 1.1. Clases y objetos

```
class Coche:
    ruedas = 4
    largoChasis=250
    anchoChasis=120
    enmarcha=False

    def __init__(self, color, aceleracion):
        self.color=color
        self.aceleracion=aceleracion
        self.velocidad=0

    def arrancar(self):
        self.enmarcha=True

    def estado(self):
        if(self.enmarcha):
            return "El coche esta en marcha"
        else:
            return "El coche esta parado"
```

## 1.1. Clases y objetos

```
def frena(self):  
    v = self.velocidad - self.aceleracion  
    if v<0:  
        v = 0  
    self.velocidad=v  
  
def __str__(self):  
    return "El color del coche es: " + str(self.color) + ", " \\  
        "su aceleración es de " + str(self.aceleracion) + " " \\  
        "km/h y su velocidad " \\  
        + str(self.velocidad)  
  
c1 = Coche('rojo',20)  
print(c1)  
print(c1.estado())
```



## 1.1. Clases y objetos

Cuando se crea una variable de tipo Coche, realmente se está instanciando un objeto de dicha clase. En el siguiente ejemplo se crean dos objetos de tipo Coche:

```
>>> c1 = Coche('rojo', 20)
>>> print(c1.color)
rojo
>>> print(c1.ruedas)
4
>>> c2 = Coche('azul', 30)
>>> print(c2.color)
azul
>>> print(c2.ruedas)
4
```

## 1.2. Constructor de una clase

El constructor de una clase se crea implementando el método especial `__init__()`.

El método `__init__()` establece un primer parámetro especial llamado **self** y puede especificar otros parámetros siguiendo las mismas reglas que cualquier otra función.

En nuestro caso, el constructor de la clase coche es el siguiente:

```
def __init__(self, color, aceleracion):  
    self.color = color  
    self.aceleracion = aceleracion  
    self.velocidad = 0
```



Los atributos deberían ser privados!!

## 1.2. Getter y setters

```
def __init__(self, color, aceleracion):  
    self.__color=color  
    self.__aceleracion=aceleracion  
    self.__velocidad=0
```

Constructor

```
@property  
def color(self):  
    return self.__color
```

Getter

```
@color.setter  
def color(self, valor):  
    self.__color = valor
```

Setter

## 1.3. Atributos, atributos de datos y métodos

Un objeto tiene dos tipos de atributos: atributos de datos y métodos.

Los atributos de datos definen el estado del objeto. En otros lenguajes son conocidos simplemente como atributos o miembros.

Los métodos son las funciones definidas dentro de la clase.

```
>>> c1 = Coche('rojo', 20)
>>> print(c1.color)
rojo
>>> print(c1.velocidad)
0
>>> c1.acelera()
>>> print(c1.velocidad)
20
```

Atributos

Método

## 1.4. Atributos de datos

A diferencia de otros lenguajes, los atributos de datos no necesitan ser declarados previamente. Un objeto los crea del mismo modo en que se crean las variables en Python, es decir, cuando les asigna un valor por primera vez.

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 10)
>>> print(c1.color)
rojo
>>> print(c2.color)
azul
>>> c1.marchas = 6
>>> print(c1.marchas)
6
>>> print(c2.marchas)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Coche' object has no attribute 'marchas'
```

Los objetos c1 y c2 pueden referenciar al atributo color porque está definido en la clase Coche.

Solo el objeto c1 puede referenciar al atributo marchas, porque inicializa dicho atributo en esa línea.

Si el objeto c2 intenta referenciar al mismo atributo, como no está definido en la clase y tampoco lo ha inicializado, el intérprete lanzará un error.

## 1.5. Métodos

Los métodos son las funciones que se definen dentro de una clase y que, por consiguiente, pueden ser referenciadas por los objetos de dicha clase.

Observemos la función `acelera()`:



```
def acelera(self):  
    self.velocidad = self.velocidad + self.aceleracion
```

La función recibe un parámetro llamado `self`. No obstante, cuando se llama a esta función no se pasa ningún argumento. Esto se debe a que `acelera()` está siendo utilizada como un método por los objetos de la clase `Coche`, de tal manera que cuando un objeto referencia a dicha función, realmente pasa su propia referencia como primer parámetro de la función.

## 1.5. Métodos

El siguiente ejemplo muestra dos formas diferentes y equivalentes de llamar al método `acelera()`:

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 20)
>>> c1.acelera()
>>> Coche.acelera(c2)
>>> print(c1.velocidad)
20
>>> print(c2.velocidad)
20
```

Para la clase `Coche`, `acelera()` es una función. Sin embargo, para los objetos de la clase `Coche`, `acelera()` es un método.

## 1.5.1. Métodos especiales

Las clases en Python cuentan con múltiples métodos especiales, los cuales se encuentran entre dobles guiones bajos `__<metodo>__()`.

Los métodos especiales más utilizados son `__init__()`, `__str__()` y `__del__()`.

### `__str__()`

El método `__str__()` ejecuta una cadena representativa de la clase, la cual puede incluir formatos personalizados de presentación del objeto.

```
def __str__(self):  
    return "El color del coche es: " + __str__(self.color) + ", " \\  
           "su aceleración es de " + str(self.aceleracion) + " km/h y su velocidad " \\  
           + str(self.velocidad)  
  
c1 = Coche('rojo',20)  
print(c1)
```



## 1.6. Atributos de clase y atributos de instancia

Una clase puede definir dos tipos diferentes de atributos de datos: atributos de clase y atributos de instancia.

Los atributos de clase son atributos compartidos por todas las instancias de esa clase.

Los atributos de instancia, por el contrario, son únicos para cada uno de los objetos pertenecientes a dicha clase.

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 20)
>>> print(c1.color)
rojo
>>> print(c2.color)
azul
>>> print(c1.ruedas) # Atributo de clase
4
>>> print(c2.ruedas) # Atributo de clase
4
>>> Coche.ruedas = 6 # Atributo de clase
>>> print(c1.ruedas) # Atributo de clase
6
>>> print(c2.ruedas) # Atributo de clase
6
```

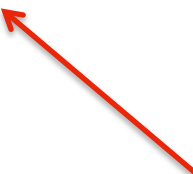
Para referenciar a un atributo de clase se utiliza, generalmente, el nombre de la clase. Al modificar un atributo de este tipo, los cambios se verán reflejados en todas y cada una de las instancias.

## 2. Herencia

En programación orientada a objetos, la herencia es la capacidad de reutilizar una clase extendiendo su funcionalidad. Una clase que hereda de otra puede añadir nuevos atributos, ocultarlos, añadir nuevos métodos o redefinirlos.

En Python, podemos indicar que una clase hereda de otra de la siguiente manera:

```
class CocheVolador(Coche):  
  
    ruedas = 6  
  
    def __init__(self, color, aceleracion, esta_volando=False):  
        super().__init__(color, aceleracion)  
        self.esta_volando = esta_volando  
  
    def vuela(self):  
        self.esta_volando = True  
  
    def aterriza(self):  
        self.esta_volando = False
```



Redefine el método `__init__()` de la clase hija usando la funcionalidad del método de la clase padre.

## 2. Herencia

Al utilizar la herencia, todos los atributos de la clase padre también pueden ser referenciados por objetos de las clases hijas. Al revés no ocurre lo mismo.

Veamos todo esto con un ejemplo:

```
>>> c = Coche('azul', 10)
>>> cv1 = CocheVolador('rojo', 60)
>>> print(cv1.color)
rojo
>>> print(cv1.esta_volando)
False
>>> cv1.acelera()
>>> print(cv1.velocidad)
60
>>> print(CocheVolador.ruedas)
6
>>> print(c.esta_volando)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Coche' object has no attribute 'esta_volando'
```



## 2. Herencia

- Vamos a crear un archivo que contenga dos clases “Persona” y “Empleado”.
- Toda persona tendrá nombre, edad y lugar de residencia.
- La clase Persona tendrá un método “getDATos” que mostrará todos los datos de la misma.
- La clase Empleado heredará de la clase Persona
- Todo empleado tendrá un salario y una antigüedad.
- Sobreescribe el método getDatos para que muestre todos los datos de un empleado.
- Crea un empleado que muestre sus datos

## 2. Herencia

```
class Persona():  
  
    def __init__(self, nombre, edad, lugar_residencia):  
        self.nombre=nombre  
        self.edad=edad  
        self.lugar_residencia=lugar_residencia  
  
    def getDatos(self):  
  
        print("Nombre: ", self.nombre, " Edad: ", self.edad, "Lugar de residencia: ", self.lugar_residencia)
```

```
class Empleado(Persona):  
  
    def __init__(self, salario, antigüedad, nombre_empleado, edad_empleado, residencia_empleado):  
  
        super().__init__(nombre_empleado, edad_empleado, residencia_empleado)  
        self.salario=salario  
        self.antigüedad=antigüedad  
  
    def getDatos(self):  
  
        super().getDatos()  
        print("Antigüedad: ", self.antigüedad, " Salario: ", self.salario)  
  
Pepito = Persona(1500, 5, "Pepito", 55, "Spain")  
Pepito.getDatos()
```

## 2.1. isinstance()

- isinstance devuelve True si el objeto sobre el que se aplica es una instancia de una clase concreta.

```
print(isinstance(Pepito, Empleado))
```

objeto

clase

**ACTIVIDAD PROPUESTA:** Crea un objeto de tipo Persona y comprueba si pertenece a la clase empleado.

## 2.2. isinstance()

`isinstance(clase, claseinfo)` comprueba la herencia de clases. Devuelve `True` en caso de que **clase** sea una subclase de **claseinfo**, `False` en caso contrario. **claseinfo** puede ser una clase o una tupla de clases.

```
>>> c = Coche('rojo', 20)
>>> cv = CocheVolador('azul', 60)
>>> isinstance(c, Coche)
True
isinstance(cv, Coche)
True
>>> isinstance(c, CocheVolador)
False
>>> isinstance(cv, CocheVolador)
True
>>> isinstance(CocheVolador, Coche)
True
>>> isinstance(Coche, CocheVolador)
False
```

## 2.3. Herencia múltiple

Python es un lenguaje de programación que permite herencia múltiple. Esto quiere decir que una clase puede heredar de más de una clase a la vez.

```
class A:
    def print_a(self):
        print('a')

class B:
    def print_b(self):
        print('b')

class C(A, B):
    def print_c(self):
        print('c')

c = C()
c.print_a()
c.print_b()
c.print_c()
```



### 3. ENCAPSULACIÓN: ATRIBUTOS PRIVADOS

La encapsulación hace referencia a la capacidad que tiene un objeto de ocultar su estado, de manera que sus datos solo se puedan modificar por medio de las operaciones (métodos) que ofrece.

Por defecto, en Python, todos los atributos de una clase (atributos de datos y métodos) son públicos.

En Python, para indicar que un método o atributo es privado, se hace usando el carácter guión bajo **\_atributo** antes del nombre del atributo que queramos ocultar.

### 3. ENCAPSULACIÓN: ATRIBUTOS PRIVADOS

```
class A:
    def __init__(self):
        self._contador = 0 # Este atributo es privado

    def incrementa(self):
        self._contador += 1

    def cuenta(self):
        return self._contador

class B(object):
    def __init__(self):
        self._contador = 0 # Este atributo es privado

    def incrementa(self):
        self._contador += 1

    def cuenta(self):
        return self._contador
```

```
>>> a = A()
>>> a.incrementa()
>>> a.incrementa()
>>> a.incrementa()
>>> print(a.cuenta())
3
>>> print(a._contador)
3
```

## ACTIVIDAD

- Crea una clase producto que tenga como atributos:
  - Código
  - Nombre
  - Precio
- Crea su constructor, los métodos getter y setter correspondientes, el método toString() y una función llamada calcular\_total que recibirá por argumento el número de unidades del producto.
  
- Añade una segunda clase Pedido que tenga como atributos:
  - Lista de productos
  - Lista de cantidades
  
- Añade las siguientes funciones:
  - Total\_pedido: muestra el precio final del pedido
  - Mostrar\_productos: muestra los productos del pedido

## ACTIVIDAD

```
class Producto:
    def __init__(self, codigo, nombre, precio):
        self.__codigo=codigo
        self.__nombre=nombre
        self.__precio=precio

    @property
    def codigo(self):
        return self.__codigo
    @codigo.setter
    def codigo(self, valor):
        self.__codigo=valor

    @property
    def nombre(self):
        return self.__nombre
    @nombre.setter
    def nombre(self, valor):
        self.__nombre = valor
```

## ACTIVIDAD

```
@property
def precio(self):
    return self.__precio

@precio.setter
def precio(self, valor):
    self.__precio = valor

def __str__(self):
    return "Codigo " + str(self.__codigo) + " , " \
           "nombre: " + self.nombre + " , precio: " + str(self.precio)

p1 = Producto(1, "Ordendador", 1000)
p2 = Producto(2, "Impresora", 200)
p3 = Producto(3, "Teclado", 60)
print(p1)
print(p2)
print(p3)

def calcular_total(self, unidades):
    return self.__precio * unidades
```

## ACTIVIDAD

```
class Pedido:

    def __init__(self, productos, cantidades):
        self.__productos=productos
        self.__cantidades=cantidades

    def total_pedido(self):
        total = 0

        for(i,j) in zip(self.__productos, self.__cantidades):
            total = total + i.calcular_total(j)
        return total

    def mostrar_pedido(self):
        for (i, j) in zip(self.__productos, self.__cantidades):
            print("Producto: " + i.nombre + " , cantidad: " + str(j))
```



## ACTIVIDAD

```
p1 = Producto(1, "Ordendador", 1000)
p2 = Producto(2, "Impresora", 200)
p3 = Producto(3, "Teclado", 60)
print(p1)
print(p2)
print(p3)

productos = [p1, p2, p3]
cantidades = [5, 10, 15]
pedido = Pedido(productos, cantidades)

print("Total pedido: " + str(pedido.total_pedido()))
pedido.mostrar_pedido()
```



## ACTIVIDAD

Añade la siguiente funcionalidad a la clase Pedido:

### **Añadir\_pedido:**

- Pasamos un producto y una cantidad
- Añadir ese producto y esa cantidad a su respectiva lista
- Debemos validar que el dato que nos pasen es correcto, es decir, que sea un producto y que la cantidad sea válida. En caso de que no sea válida, devolver una excepción.

### **Eliminar producto:**

- Pasamos el producto a borrar, si existe, lo eliminamos.
- Si no existe el producto o no es un producto, devolver una excepción indicándolo



## ACTIVIDAD

```
class Pedido:

    def __init__(self):
        self.__productos=[]
        self.__cantidades=[]

    def anyadir_producto(self, producto, cantidad):
        if not isinstance(producto, Producto):
            raise Exception("anyadir_producto: producto debe ser de la clase Producto")

        if not isinstance(cantidad, int):
            raise Exception("anyadir_producto: cantidad debe ser un numero")

        if cantidad <= 0:
            raise Exception("anyadir_producto: cantidad debe ser mayor que cero")

        if producto in self.__productos:
            indice = self.__productos.index(producto)
            self.__cantidades[indice] = self.__cantidades[indice] + cantidad
        else:
            self.__productos.append(producto)
            self.__cantidades.append(cantidad)
```



## ACTIVIDAD

```
def eliminar_producto(self, producto):  
    if not isinstance(producto, Producto):  
        raise Exception("eliminar_producto: producto debe ser de la clase Producto")  
  
    if producto in self.__productos:  
        indice = self.__productos.index(producto)  
        del self.__productos[indice]  
        del self.__cantidades[indice]  
    else:  
        raise Exception("eliminar_producto: El producto no existe")
```



## ACTIVIDAD

```
pedido = Pedido()

try:
    pedido.anyadir_producto(p1, 5)
    pedido.anyadir_producto(p2, 5)
    pedido.anyadir_producto(p3, 5)

    print("Total pedido: " + str(pedido.total_pedido()))
    pedido.mostrar_pedido()

    pedido.eliminar_producto(p1)

    print("Total pedido: " + str(pedido.total_pedido()))
    pedido.mostrar_pedido()

except Exception as e:
    print(e)
```



*La mejor forma de aprender es “haciendo”.  
Educación y empresas forman un binomio inseparable*