



PYTHON

Programación orientada a objetos

1

PROGRAMACIÓN ORIENTADA A OBJETOS

- Declaración:

```
class NombreDeLaClase(clase_base):
```

- Atributos de clase:

```
    nombre_atributo = "Valor de inicialización"
```

- Constructor:

```
    def __init__(self, parametro1, parametro2) -> None:  
        #Atributos de de instancia  
        self.atributo1 = parametro1  
        self.atributo2 = parametro2
```

PROGRAMACIÓN ORIENTADA A OBJETOS

<https://docs.python.org/3/tutorial/classes.html>

PROGRAMACIÓN ORIENTADA A OBJETOS

- **Métodos:**

```
def metodo1(self):  
    pass
```

```
def metodo2(self, parametro1):  
    pass
```

- **Destructor:**

```
def __del__(self):  
    pass
```

PROGRAMACIÓN ORIENTADA A OBJETOS

■ Clase:

```
class NombreDeLaClase(clase_base):  
    #Atributos de clase  
    nombre_atributo = "Valor de inicialización"  
  
    #Constructor  
    def __init__(self, parametro1, parametro2) -> None:  
        #Atributos de instancia  
        self.atributo1 = parametro1  
        self.atributo2 = parametro2  
  
    #Métodos  
    def metodo1(self):  
        pass  
  
    def metodo2(self, parametro1):  
        pass  
  
    #Destructor  
    def __del__(self):  
        pass
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- **Instanciación y uso:**

```
nombre_objeto = NombreDeLaClase("valor p1", "valor p2")  
nombre_objeto.metodo2(valor_parámetro)
```

- **Paso de objetos como parámetros:**

Siempre se pasan por referencia.

PROGRAMACIÓN ORIENTADA A OBJETOS

- **Atributos estático (de clase):**

```
#Atributos de clase  
nombre_atributo = "Valor de inicialización"
```

- **Métodos estáticos (de clase):**

```
@classmethod  
def soy_static(cls):  
    print(cls.atributo_clase)
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- Visibilidad atributos y métodos:
 - La visibilidad hace referencia a la capacidad de acceso a un atributo o método de un objeto desde otro.
 - Tipos de visibilidad:
 - Público: no hay restricciones de acceso.
 - `self.atributo_publico`
 - Privado: sólo pueden acceder objetos de la misma clase.
 - `self.__atributo_privado`
 - Se puede acceder mediante `_Clase__atributo_privado`
 - Protected: sólo pueden acceder objetos de la misma clase o clases derivadas.
 - `self._atributo_protegido`
 - No bloque el acceso. Es responsabilidad del programador.

PROGRAMACIÓN ORIENTADA A OBJETOS

- Atributos ocultos (“privados”):

```
self.__codigo = 3
```

- Métodos ocultos (“privados”):

```
def __metodo_privado(self):  
    pass
```

- El acceso de lectura queda bloqueado (el atributo y el método no son visibles):
 - `print(objeto.__codigo)#ERROR`
 - `objeto.__metodo_privado()#ERROR`
- En realidad el acceso está permitido, pero el atributo y el nombre del método están redefinidos como `__NombreClase__atributo` y `__NombreClase__nombre_método`.
 - `print(objeto._NombreClase__codigo)#ERROR`
 - `objeto._NombreClase__metodo_privado()#ERROR`

PROGRAMACIÓN ORIENTADA A OBJETOS

- Los atributos “privados” son visibles entre instancias de la misma clase.

```
class Subordinado:
    def __init__(self, nombre) -> None:
        self.__nombre = nombre
    def get_nombre(self):
        return self.__nombre
    def calcular_diferencia(self, otro):
        print( (len(self.__nombre)*1000) - (len(otro.__nombre)*1000) )
```

```
empleado1=Subordinado("Blas")
empleado2=Subordinado("Nabucodonosor")
empleado1.calcular_diferencia(empleado2)
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- Implementación de “getters” y “setters” mediante decoradores.
 - Permiten encapsular el acceso a atributos privados o protegidos a través de métodos invocados como atributos.

```
class Banco:
    def __init__(self, nombre, propietario) -> None:
        self.nombre = nombre
        self.__propietario = propietario

    @property
    def propietario(self):
        print("get")
        return self.__propietario

    @propietario.setter
    def propietario(self, value):
        print("set")
        self.__propietario = value

b = Banco("Banco de Torrelavega", "D. Vicent Price ")
print(b.propietario)
b.propietario="Boris Karloff"
print(b.propietario)
```

PROGRAMACIÓN ORIENTADA A OBJETOS

■ HERENCIA:

- A continuación del nombre de la clase derivada, se indica entre paréntesis el nombre de la clase base.
- `super()` → referencia a la clase base

```
class MegaCiudad(Ciudad):  
    def __init__(self, nombre, provincia, ccaa) -> None:  
        super().__init__(nombre, provincia, ccaa)
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- **HERENCIA:**

- Métodos relacionados:

- `isinstance(obj, clase)` → Determina si un objeto es una instancia de una clase
 - `issubclass(clase, clase)` → Determina si una clase es un subclase de otra

PROGRAMACIÓN ORIENTADA A OBJETOS

- HERENCIA:

- Herencia múltiple:

- ClaseDerivada(ClaseBase1, ClaseBase2, ClaseBase3)
 - Búsquedas de atributos (y de métodos) de izquierda a derecha.

- Sobreescritura de métodos.

- Mismo nombre de método con distinta implementación en la clase derivada.
 - Ver: <https://docs.python.org/es/3/tutorial/classes.html#tut-private>

PROGRAMACIÓN ORIENTADA A OBJETOS

- Clases abstractas:
 - Una clase abstracta es aquella que tiene métodos sin implementar.
 - No se puede instanciar.
 - Paquete abc.
 - Decorador `@abstractmethod`

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

```
class Calculadora(ABC):  
    @abstractmethod  
    def sumar(self):  
        pass
```

```
class CalculadoraDecimal(Calculadora):  
    def reiniciar(self):  
        print("Reiniciando")  
    def sumar(self):  
        print("Sumando")
```

```
CalculadoraDecimal().reiniciar()
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- Una **interfaz** es una clase con todos los métodos abstractos.
- Se utilizan para definir distintas clases con comportamientos idénticos, permitiendo generalizar soluciones.
- Permiten garantizar la estructura de una clase que no se conoce a priori.
 - Hay varias formas de implementar interfaces.
 - Interfaces informales.
 - Interfaces basadas en metaclasses:
 - La metaclass permite la definición de clases.
 - Incluye métodos de validación.
 - Método `__instancecheck__`. Determina si un objeto es un subclase de otro. Utiliza el método `__subclasscheck__`.
 - Método `__subclasscheck__`. Evalúa si una clase es una subclase de la metaclass.
 - Basadas en métodos abstractos

PROGRAMACIÓN ORIENTADA A OBJETOS

- Interfaces informales: la clase base no tiene implementaciones de los métodos y en su lugar lanza excepciones. Las clases derivadas implementan los métodos no implementados.
- Ejemplo.

```
class ICalculadora:
    def sumar(self, s1, s2):
        raise NotImplementedError
    def restar(self, s1, s2):
        raise NotImplementedError

class CalculadoraDecimal(ICalculadora):
    def sumar(self, s1, s2):
        return s1+s2
    def restar(self, s1, s2):
        return s1-s2

class CalculadoraOctal(ICalculadora):
    def sumar(self, s1, s2):
        return oct(int(str(s1), 8) + int(str(s2), 8))
    def restar(self, s1, s2):
        return oct(int(str(s1), 8) - int(str(s2), 8))

def get_calculadora(numero):
    if (numero==10):
        return CalculadoraDecimal()
    elif (numero==8):
        return CalculadoraOctal()

calculadora = get_calculadora(10)
if (isinstance(calculadora, ICalculadora)):
    calculadora.restar(10,2)
else:
    print("No es una calculadora")
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- Interfaces basadas en metaclasses. Ejemplo.

```
class CalculadoraMetaclass(type):
    def __instancecheck__(cls, instance):
        return cls.__subclasscheck__(type(instance))

    def __subclasscheck__(cls, subclass):
        return (hasattr(subclass, 'sumar') and
                callable(subclass.sumar) and
                hasattr(subclass, 'restar') and
                callable(subclass.restar))

class ICalculadora(metaclass=CalculadoraMetaclass):
    pass

class Calculadora(ICalculadora):
    def sumax(self, s1, s2):
        print("Sumando...")

    def restar(self, s1, s2):
        print("Restando...")

c = Calculadora()
if (issubclass(c, ICalculadora)):
    print("OK, es una implementación.")
else:
    print("KO")
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- Interfaces formales basadas en métodos abstractos:
 - Todos los métodos son abstractos.
 - En el uso del objeto que implementa la interfaz, se valida que el objeto es una instancia de la clase interfaz → Garantiza la disposición de métodos.

```
def sumar(calculadora):  
    if isinstance(calculadora, ICalculadora):  
        calculadora.sumar()  
    else:  
        print("Error")
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- Interfaces formales basadas en métodos abstractos:

```
from abc import ABC, abstractmethod
class ICalculadora(ABC):
    @abstractmethod
    def sumar(self):
        pass
    @abstractmethod
    def restar(self):
        pass

class CalculadoraDecimal(ICalculadora):
    def sumar(self):
        print("Sumando")
    def restar(self):
        pass

#Esta clase sería abstracta al no tener implementados todos los métodos abstractos
class CalculadoraErronea(ICalculadora):
    def sumar(self):
        print("Sumando...")

class CalculadoraFake():
    pass

def sumar(calculadora):
    if isinstance(calculadora, ICalculadora):
        calculadora.sumar()
    else:
        print("Error")

sumar(CalculadoraFake())#Error
sumar(CalculadoraDecimal())#OK
sumar(CalculadoraErronea())#Error
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- Clases internas (inner classes).
 - Son clases declaradas dentro de otras clases.
 - Sirven para encapsular clases.
 - Hay que referenciar a la clase externa (outer class) o a una instancia de la clase externa para utilizarlas.

PROGRAMACIÓN ORIENTADA A OBJETOS

- Clases internas (inner classes). Ejemplo (1/3).

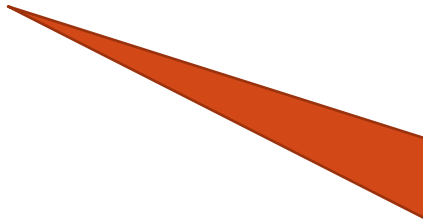
```
class CV(object):  
    def __init__(self, nombre, direccion, telefono, email) -> None:  
        self.nombre = nombre  
        self.direccion = direccion  
        self.experiencias = []  
        self.contacto = CV.Contacto(telefono,email)  
  
    def agregar_objeto_experiencia(self, experiencia):  
        self.experiencias.append(experiencia)
```

PROGRAMACIÓN ORIENTADA A OBJETOS

- Clases internas (inner classes). Ejemplo (2/3).

```
class ExperienciaProfesional():  
    def __init__(self, empresa, puesto, duracion=0) -> None:  
        self.empresa = empresa  
        self.puesto = puesto  
        self.duracion = duracion
```

```
class Contacto():  
    def __init__(self, telefono, email) -> None:  
        self.telefono = telefono  
        self.email = email
```



Las clases
ExperienciaProfesional y
Contacto (inner classes)
están declaradas dentro
de CV (outer classes)

PROGRAMACIÓN ORIENTADA A OBJETOS

- Clases internas (inner classes). Ejemplo (3/3).

```
cv_fp = CV(  
    "Fernando Paniagua",  
    "Madrid",  
    "915553311",  
    "fernando.Paniagua.formacion@gmail.com")  
cv_fp.agregar_objeto_experiencia(CV.ExperienciaProfesional("ITTI", "Docente"))
```

Clase interna (inner)

Clase externa (outer)

PROGRAMACIÓN ORIENTADA A OBJETOS

- **Conceptos OO:**
 - Sobreescritura (override) → Sustitución de un método de una clase base en una clase derivada.
 - Sobrecarga (overload) → Crear diversos métodos con mismo nombre y distinta parametrización. En Python no existe. Se consigue utilizando parámetros por defecto.
 - Sobrecarga de operadores → Sustituir el funcionamiento normal de un operador por otro comportamiento. En Python se realiza mediante la sobreescritura de los métodos

PROGRAMACIÓN ORIENTADA A OBJETOS

- Equivalencias
operador-método→

Operator	Method	Expression
+ Addition	<code>__add__(self, other)</code>	<code>a1 + a2</code>
- Subtraction	<code>__sub__(self, other)</code>	<code>a1 - a2</code>
* Multiplication	<code>__mul__(self, other)</code>	<code>a1 * a2</code>
@ Matrix Multiplication	<code>__matmul__(self, other)</code>	<code>a1 @ a2 (Python 3.5)</code>
/ Division	<code>__div__(self, other)</code>	<code>a1 / a2 (Python 2 only)</code>
/ Division	<code>__truediv__(self, other)</code>	<code>a1 / a2 (Python 3)</code>
// Floor Division	<code>__floordiv__(self, other)</code>	<code>a1 // a2</code>
% Modulo/Remainder	<code>__mod__(self, other)</code>	<code>a1 % a2</code>
** Power	<code>__pow__(self, other[, modulo])</code>	<code>a1 ** a2</code>
<< Bitwise Left Shift	<code>__lshift__(self, other)</code>	<code>a1 << a2</code>
>> Bitwise Right Shift	<code>__rshift__(self, other)</code>	<code>a1 >> a2</code>
& Bitwise AND	<code>__and__(self, other)</code>	<code>a1 & a2</code>
^ Bitwise XOR	<code>__xor__(self, other)</code>	<code>a1 ^ a2</code>
(Bitwise OR)	<code>__or__(self, other)</code>	<code>a1 a2</code>
- Negation (Arithmetic)	<code>__neg__(self)</code>	<code>-a1</code>
+ Positive	<code>__pos__(self)</code>	<code>+a1</code>
~ Bitwise NOT	<code>__invert__(self)</code>	<code>~a1</code>
< Less than	<code>__lt__(self, other)</code>	<code>a1 < a2</code>
<= Less than or Equal to	<code>__le__(self, other)</code>	<code>a1 <= a2</code>
= Equal to	<code>__eq__(self, other)</code>	<code>a1 == a2</code>
!= Not Equal to	<code>__ne__(self, other)</code>	<code>a1 != a2</code>
> Greater than	<code>__gt__(self, other)</code>	<code>a1 > a2</code>
>= Greater than or Equal to	<code>__ge__(self, other)</code>	<code>a1 >= a2</code>
[index] Index operator	<code>__getitem__(self, index)</code>	<code>a1[index]</code>
in In operator	<code>__contains__(self, other)</code>	<code>a2 in a1</code>
(*args, ...) Calling	<code>__call__(self, *args, **kwargs)</code>	<code>a1(*args, **kwargs)</code>

PROGRAMACIÓN ORIENTADA A OBJETOS

- Patrones
 - Factory.
 - Proporciona instancias de un objeto para evitar tener que instanciarlos directamente.
 - MVC. Model-View-Controller.
 - Permite separar la capa de presentación (vista), la capa intermediaria (controlador) y la capa de negocio (modelo).

PROGRAMACIÓN ORIENTADA A OBJETOS

■ Iteradores:

```
class Flota:
    def __init__(self, lista) -> None:
        self.lista = lista
        self.index = -1

    def __iter__(self):
        return self

    def __next__(self):
        self.index+=1
        if (self.index == len(self.lista)):
            raise StopIteration #Indica al iterador que debe parar
            #self.index=0#Lista circular
        return self.lista[self.index]

f = Flota(["Seat", "Audi", "Renault", "Kia"])

for vehiculo in f:
    print(vehiculo)
```