

# Programación Orientada a Objetos

Antonio Espín Herranz

# POO en Python

- Conceptos:
  - Clase.
  - Objeto, instancia.
  - Métodos, funcionalidades.
  - Atributo , Propiedad.
  - Interfaz de la clase (la parte pública de esta).
  - Herencia: Simple y Múltiple.
  - Polimorfismo.
  - Encapsulación.

# Conceptos POO

- La programación orientada a objetos es un paradigma de programación que busca representar entidades u objetos agrupando datos y métodos que puedan describir sus características y comportamiento.

# Conceptos

- Fomenta la reutilización y extensión del código.
- Permite crear sistemas más complejos.
- Relacionar el sistema al mundo real.
- Facilita la creación de programas visuales.
- Construcción de prototipos.
- Agiliza el desarrollo de software.
- Facilita el trabajo en equipo.
- Facilita el mantenimiento del software.

# Notas

- Cuando pasamos un objeto a una función, se pasa por referencia.
- Es decir, que si dentro de la función se modifica el objeto se verá afectado el original.
- Como ocurría con las listas y los diccionarios, eran objetos mutables.

# Objetos en python

- En Python todo es un objeto, hasta las funciones son objetos que tienen sus propias propiedades.
- Las clases se puede definir de 2 formas:
  - De forma imperativa la forma habitual.
  - Con el prototipo al estilo javascript.

# Objetos en python

- El objeto en curso se representa por **self**.
- Todos los métodos de la clase de instancia necesitan recibir con parámetro **self** (el objeto en curso).
- En los métodos de la clase: recibe **cls**.
- Python permite la sobrecarga de operadores, de tal forma que cuando llamemos a un operador con una instancia de la clase se lanzará el método correspondiente.
- De igual forma pasa con algunas funciones con **str**, **del**, **repr**, activarán métodos especiales de la clase.

# Objetos en python

- No se permite la sobrecarga de métodos, se puede realizar con los parámetros opcionales.
- En Python 3, **no es necesario heredar de object** esto ya ocurre por defecto.
- Podemos tener métodos de varios tipos:
  - De clase
  - De instancia
  - Estáticos.
- Atributos:
  - De instancia
  - De clase



# Clase declarativa

```
class declarativa:
    """clase declarativa"""

    att_de_clase = 42

    def __init__(self, name):
        self.name = name
        self.subs = []

    def __str__(self):
        return "{} ({}).format(self.name, ",".join(self.subs))

    def mostrar(self):
        print(self)
```

# Clase con prototipo

```
def proto__init__(self, name):  
    self.name = name  
    self.subs = []
```

```
def proto__str__(self):  
    return "{} ({}).format(self.name, ",".join(self.subs))
```

```
Prototipo = type("Prototipo", (object,), {  
    "__init__": proto__init__,  
    "__str__": proto__str__,  
    "att_de_clase": 42})
```

# Modificar la clase a posteriori

- En ambas opciones se puede modificar la clase:  
def mostrar(self):  
 print(self)

Prototipo.mostrar = mostrar

- También se pueden añadir atributos después de declarar la clase.
- Los objetos que estaban declarados **ANTES** de esta modificación ***también disponen de las nuevas características.***

# Tuplas con nombre

- Si no queremos escribir una clase se pueden utilizar tuplas con nombre.
- Se utiliza mas para almacenar información que para operar con ella.

```
from collections import namedtuple  
Punto = namedtuple('Punto', ['x','y'])  
p = Punto(8,9)  
print(p)  
print(p.x)  
print(p.y)
```

# Ejemplo de clase (forma imperativa)

class Coche:

```
    """Abstraccion de los objetos coche."""
```

```
    def __init__(self, gasolina):
        self.gasolina = gasolina
        print "Tenemos", gasolina, "litros"
```

```
    def arrancar(self):
        if self.gasolina > 0:
            print "Arranca"
        else:
            print "No arranca"
```

```
    def conducir(self):
        if self.gasolina > 0:
            self.gasolina -= 1
            print "Quedan", self.gasolina, "litros"
        else:
            print "No se mueve"
```

## **`__init__`**

Se ejecuta después de crea el objeto.

Después del método de clase: **`__new__`**

Sirve para inicializar, similar al constructor.

## **`self`**

Representa el objeto actual. Equivalente a `this` en otros lenguajes. Este parámetro lo deben tener todos los métodos de la clase.

**Lo pasa automáticamente python.**

## **Crear un objeto:**

`miCoche = Coche(3)`

Se ejecuta **`init`**.

## **Ejecución de métodos:**

`miCoche.arrancar()`

Python agrega de forma automática `self`.

## **Acceso a propiedades (si no son privadas)**

`miCoche.gasolina`

# Métodos

- De instancia, se suelen invocar una instancia de la clase. Si se invoca desde la clase nos dará un error si no le pasamos un objeto de la clase. Viene a ser `self` que Python lo añade automáticamente.
- Estáticos y de clase se pueden invocar con instancias o con la clase, aunque es más típico hacerlo a través de la clase.

# Métodos

- En los métodos de instancia necesitamos self.
- En los de clase: cls
- En los métodos estáticos no existe self, y tampoco se pasa cls.
- Un método se puede asignar a una variable de la misma forma que una función y se ejecuta al añadir los paréntesis.  
F = instancia.método\_instancia  
F()

# Métodos

class A:

```
def metodo_instancia(self, *args, **kwargs):  
    return "Esto es un metodo de instancia"
```

**@classmethod**

```
def metodo_clase(cls, *args, **kwargs):  
    return "metodo de clase"
```

**@staticmethod**

```
def metodo_estatico(*args, **kwargs):  
    return "metodo estatico"
```

```
if __name__ == '__main__':  
    a = A()  
    print(a.metodo_instancia())  
    print(a.metodo_clase())  
    print(a.metodo_estatico())  
  
    print(A.metodo_instancia(a))  
    print(A.metodo_clase())  
    print(A.metodo_estatico())
```



# Métodos static

```
class ClaseA():

    accesos=[] # Variable static

    def __init__(self, a):
        self.a=a
        ClaseA.accesos.append(a)

    def __str__(self):
        return str(self.a)

    @staticmethod
    def listar_accesos(): # OJO, no hay self
        ClaseA.accesos.sort()
        for a in ClaseA.accesos:
            print("Acceso:",a)

uno=ClaseA(1)
dos=ClaseA(2)
ClaseA.listar_accesos()
```

# Variables de clase y de instancia

```
class Perro:
    tipo = 'canino' # variable de clase compartida por todas las
                    instancias

    def __init__(self, nombre):
        self.nombre = nombre # variable de instancia única para la instancia

>>> d = Perro('Fido')
>>> e = Perro('Buddy')
>>> d.tipo # compartido por todos los perros 'canino'
>>> e.tipo # compartido por todos los perros 'canino'
>>> d.nombre # único para d 'Fido'
>>> e.nombre # único para e 'Buddy'
```

# Notas

- Si utilizamos un objeto mutable como puede ser una lista como una variable de clase y lo modificamos a través de un método, este atributo cambiará en todos los objetos de la clase.

```
class Perro2:
```

```
    tipo = 'canino'
```

```
    trucos=[]
```

```
    def agregar_truco(self, truco):
```

```
        self.trucos.append(truco)
```

```
        # Así no funcionaría correctamente, tendría que ser una variable de  
        # objeto / o instancia.
```

```
        # Es decir, un atributo distinto para cada objeto.
```

# Sobrecarga de métodos

- A diferencia de C++ y java, **python no permite la sobrecarga de funciones por la lista de parámetros.**
- **Los métodos se definen sólo por su nombre y hay un único método por clase con un nombre dado.**
- **Para sobrecargar utilizar parámetros opcionales.**

# Métodos y Atributos especiales

- Los métodos y att. Especiales empiezan y terminan por `__`

- Por ejemplo,

`__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__ne__`

Se relacionan con `==`, `>=`, `>`, `<=`, `<`, `!=`

# Sobrecarga de operadores

`__add__( self, other)`

-> Oper. Suma

`__sub__( self, other)`

-> Oper. Resta

`__mul__( self, other)`

-> Oper. Multiplicacion

`__rmul__( self, other)`

-> Oper. Multi. Por Escalar

`__floordiv__( self, other)`

-> Oper. division Redondeo

`__mod__( self, other)`

-> Oper. modulo

`__divmod__( self, other)`

-> Oper. division

`__pow__( self, other[, modulo])`

-> Oper. Potencia

`__and__( self, other)`

-> Oper. and

`__xor__( self, other)`

-> Oper. xor

`__or__( self, other)`

-> Oper. or

# Más atributos

- Que no están vinculados a los operadores:
- `__class__`  
La clase a la que pertenece la instancia
- `__class__.__name__`  
El nombre de la clase
- `__doc__`  
El docstring con la documentación de la clase

```
class miclase
    """ La documentación de la clase """
```

# Más atributos

`__new__`

Crear una instancia, se llama antes de `init`

`__init__`

Inicializar una instancia

`__del__`

Se invoca cuando se llama a la función `del(mi_inst)`

`__str__`, `__repr__`

Cuando se llaman a la funciones `str()` y `repr()`

Ambas devuelven una representación del objeto en texto

`str` es mas informal.



# Ejemplo: `__new__` / `__init__`

```
class ClaseB():
```

```
    def __new__(cls, *args, **kwargs):  
        print('pasa por new')  
        new_instance = object.__new__(cls)  
        return new_instance
```

```
    def __init__(self, b):  
        self.b = b  
        print('pasa por init')
```

```
    def __str__(self):  
        return str(self.b)
```

```
b = ClaseB(5)  
print(b)
```

# Métodos especiales

- **`__len__(self)`**
  - Método llamado para comprobar la longitud del objeto.
  - Se utiliza cuando se llama a la función `len(obj)` sobre nuestro objeto.
  - Como es de suponer, el método debe devolver la longitud del objeto.
- **`__next__(self)`, `__iter__(self)`**
  - Se utilizan para definir iteradores dentro de la clase.

# Ejemplo

```
class Reversa:
    """Iterador para recorrer una secuencia marcha atrás."""
    def __init__(self, datos):
        self.datos = datos
        self.indice = len(datos)

    def __iter__(self):
        return self

    def __next__(self):
        if self.indice == 0:
            raise StopIteration
        self.indice = self.indice - 1
        return self.datos[self.indice]

rev = Reversa('spam')
print(iter(rev))
for char in rev:
    print(char)
```

# Gestión de atributos

- **getattr, setattr, delattr, hasattr**

```
if (hasattr(instancia, 'atributo')):
```

```
    getattr(a, atributo)
```

```
else:
```

```
    setattr(a, 'atributo', 'valor')
```

- Internamente una clase, almacena un **\_\_dict\_\_** con los atributos.

- *Los atributos se pueden añadir de forma dinámica:*

class Miclase:

```
def __init__(self, a,b,c):  
    self.a = a  
    self.b = b  
    self.c = c
```

```
if __name__ == '__main__':
```

```
    obj = Miclase(1,2,3)  
    print(obj.__dict__)
```

```
    d = {'d':4,'e':5}  
    print(d)  
    obj.__dict__.update(d)  
    print(obj.__dict__)
```

```
    print(obj.a)  
    print(obj.e)
```

# Ejemplo

# Borrar objetos

- En **python** no es necesario liberar la memoria de forma explícita de las instancias creadas.
- Se liberan de forma automática.
- En principio no hay pérdidas de memoria.

# Herencia

- En el caso de python, la **herencia** puede ser **simple** y **múltiple**. Se puede heredar de una clase o de varias.
- *Aplicar herencia cuando las clases tengan un conocimiento común o compartan algo.*
- **Simple:**
  - class Bateria(Instrumento)
  - class Guitarra(Instrumento)
  - Instrumento sería la clase base, padre o superclase.
  - Guitarra y Bateria la clase derivada, hija o subclase.
  - Las clases que escribimos heredan de **object**.

# Ejemplo

```
class Instrumento(object):
```

```
    def __init__(self, precio):  
        self.precio = precio
```

```
    def tocar(self):  
        print("Estamos tocando musica" )
```

```
    def romper(self):  
        print ("Eso lo pagas tu" )  
        print ("Son", self.precio, "$$$")
```

```
class Bateria(Instrumento):  
    pass
```

```
class Guitarra(Instrumento):  
    pass
```

Bateria y Guitarra **heredan** de Instrumento.

**pass**

Indica que la clase no define propiedades.

El cuerpo no puede estar vacío, equivale a { } de java.

Desde una clase hija podemos llamar a un método de la clase padre.

**Instrumento.\_\_init\_\_(self,precio)**  
**En este caso hay que indicar el parámetro self.**

Llamar a un método de la clase padre: Superclass.metodo(self, args).  
También se puede llamar **con super().método()**, en este caso no se manda self



## **SUPER:**

- Llamar a la clase Padre de la clase hija, ya sea desde un constructor o desde un método de la clase hija. Podemos utilizar `super()` o el nombre de la clase Padre.
- En la clase las siguientes llamadas son sinónimos:

```
class Empleado(Persona):
```

```
    def __init__(self, nombre, sueldo):  
        #Persona.__init__(self, nombre)  
        super().__init__(nombre)  
        self.sueldo = sueldo
```

```
    def __str__(self):  
        #return Persona.__str__(self) + " " + str(self.sueldo)  
        return super().__str__() + " " + str(self.sueldo)
```

# Herencia

- Con respecto a la sobrecarga y la herencia.
- Si una clase hija define su propio método `__init__` este sustituye al método `__init__` de su clase padre, aunque tenga una lista de atributos distinta.
- **El método `__init__` de la clase derivada debe llamar al método `__init__` de la clase base.**

# Funciones integradas en herencia

- **isinstance()** para verificar el tipo de una instancia: `isinstance(obj, int)` devuelve `True` solo si **`obj.__class__`** es `int` o alguna clase derivada de `int`.
- **issubclass()** para comprobar herencia de clase: `issubclass(bool, int)` da `True` ya que `bool` es una subclase de `int`.
  - Sin embargo, `issubclass(float, int)` devuelve `False` porque `float` no es una subclase de `int`.
- **`print(Persona.__subclasses__())`**
  - Devuelve una lista con las subclases de una clase.

# Ejemplo

```
class Persona(object):
```

```
    def __init__(self, nombre):  
        self.nombre = nombre
```

```
class Empleado(Persona):
```

```
    def __init__(self, nombre, sueldo):  
        Persona.__init__(self, nombre)  
        self.sueldo = sueldo
```

```
p = Persona("Pedro")  
e = Empleado("Ana", 2000)
```

```
if isinstance(e, Persona):  
    print("Si es una instancia")
```

```
if issubclass(Empleado,Persona):  
    print("Si es una subclase")
```

# Herencia Múltiple


- Separadas por comas se indican todas las clases de las que se hereda.

```
class Terrestre:  
    def desplazar(self):  
        print "El animal anda"
```

```
class Acuatico:  
    def desplazar(self):  
        print "El animal nada"
```

```
class Cocodrilo(Terrestre, Acuatico):  
    pass
```

```
c = Cocodrilo()  
c.desplazar()
```



**pass** en la clase Cocodrilo  
Significa que **NO** añade contenido  
**El cuerpo de la clase NO puede**  
**Quedar vacío.**

# Polimorfismo

- Es polimorfismo es la capacidad que tienen los ***objetos de distinta clase de responder al mismo mensaje*** o evento en función de los parámetros utilizados durante su invocación.
- Un objeto polimórfico es una entidad que pueden contener valores de diferentes tipos durante la ejecución del programa.

```
from functools import singledispatch
```

```
@singledispatch  
def func(arg):  
    print('Comportamiento por defecto')
```

```
@func.register(int)  
@func.register(float)  
def _(arg):  
    print('Recibo el numero ',arg)
```

```
class Clase:  
    pass
```

```
@func.register(Clase)  
def _(arg):  
    print('un objeto de Clase')
```

```
if __name__ == '__main__':  
    func('hola')  
    func(888)  
    func(8.99)  
    a = Clase()  
    func(a)
```

# Polimorfismo paramétrico

Los decoradores se pueden  
Agrupar o poner uno solo  
Por cada tipo.  
Se parece más a otros lenguajes  
Como java y C++

# Encapsulación

- La clase equivale a una caja negra que no sabemos que tiene dentro.
- La podemos utilizar a través de los métodos públicos que nos proporciona.
- No tocamos directamente sus propiedades si no a través de métodos.



# Encapsulación

- Miembros (métodos y atributos):
  - **Públicos:** Se pueden acceder desde fuera de la clase.
  - **Privados:** Sólo se puede acceder desde la propia clase.
- En python no tenemos modificadores de acceso como en C++ y Java.
- Para hacer algo privado en python se precede la propiedad o método de dos guiones de subrayado. **\_\_privado, publico.**
- Si sólo lleva un `_` al inicio se consideran privados a la clase aunque se pueden modificar dentro del módulo que contiene la clase.  
**\_privadoAlModulo**
- ***Cuando se utiliza import, no se importan variables, métodos o clases que empiezan por un carácter de subrayado.***

# Ejemplo

```
class figura:
    def __init__(self, lados = 0, longitud_lado = 0.0, apotema = 0.0):
        self.lado = lados
        self.long = longitud_lado
        self.__apotema = apotema
        self.__perimetro = self.lado * self.long
    def __area(self):
        return ((self.__apotema * self.__perimetro) / 2)
    def imprimir(self):
        a = self.__area()
        print a
```

Las propiedades **lado** y **long** son públicas (accesibles desde fuera de la clase).  
El método **imprimir** es público.

En cambio las propiedades **\_\_apotema** y **\_\_perimetro** y el método **\_\_area** son privadas.

# Otro ejemplo

```
class Factura:
```

```
    __tasa = 19 # Variable de clase
```

```
    def __init__(self, unidad, precio):
```

```
        self.unidad = unidad
```

```
        self.precio = precio
```

```
    def a_pagar(self):
```

```
        total = self.unidad * self.precio
```

```
        impuesto = total * Factura.__tasa / 100
```

```
        return(total + impuesto)
```

```
compra1 = Factura(12, 110)
```

```
print(compra1.unidad)
```

```
print(compra1.precio)
```

```
print(compra1.a_pagar(), "euros")
```

```
print(Factura.__tasa) # Error:
```

```
# AttributeError: type object 'Factura' has no attribute '__tasa'
```

# Encapsulación

- Otra forma más sofisticada de implementarla es sobrescribiendo en nuestra clase los métodos:

**`__getattr__`**

**`__setattr__`**

**`__delattr__`**

# Ejemplo

class A:

```
    read_only = ['x','y']
```

```
    x,y,z = 'x', 'y', 'z'
```

```
    def __setattr__(self, name, value):
```

```
        if name in self.read_only:
```

```
            Raise Exception(...)
```

```
        else:
```

```
            Return object.__setattr__(self, name, value)
```

```
    # Algo similar se puede hacer con __delattr__
```

Permiten utilizar un método como si fuera un attr.

class Boletin:

```
def __init__(self, *notas):  
    self.notas = list(notas)
```

**@property**

```
def media(self):  
    if len(self.notas):  
        return sum(self.notas)/len(self.notas)  
    return 0
```

**@property**

```
def ultima_nota(self):  
    if len(self.notas):  
        return self.notas[-1]
```

**@ultima\_nota.setter**

```
def ultima_nota(self, nota):  
    self.notas.append(nota)
```

**@ultima\_nota.deleter**

```
def ultima_nota(self):  
    self.notas.pop()
```

```
if __name__ == '__main__':
```

```
    b = Boletin(1,5,8,7,5)  
    print(b.media)  
    b.ultima_nota = 10  
    print(b.media)  
    del b.ultima_nota  
    print(b.media)
```

# Definición de propiedades

# Evaluación Booleana

- Se puede sobrecargar **`__bool__`** para evaluaciones de las instancias de la clase.

```
class A:
```

```
    def __init__(self, value):
```

```
        self.value = value
```

```
    def __bool__(self):
```

```
        return self.name > 0
```

```
a = A()
```

```
bool(a)
```

# \_\_hash\_\_()

- Este método lo vamos a sobrescribir cuando tengamos un diccionario y claves sean objetos de una clase personalizada, es decir, que no sea str, int, etc.
- El diccionario cuando añade una nueva clave llama al método \_\_hash\_\_() del objeto.
- Tiene que devolver un número.



# Ejemplo

```
class Hora:
```

```
    """
```

```
    Clase hora almacena: h, m, s
```

```
    """
```

```
    def __init__(self, h=0, m=0, s=0):
```

```
        self.__h = h
```

```
        self.__m = m
```

```
        self.__s = s
```

```
        self.__ajuste()
```

```
    def __hash__(self):
```

```
        return hash(self.__h * 3600 + self.__m * 60 + self.__s)
```

```
    def __eq__(self, o):
```

```
        return self.__h == o.__h and self.__m == o.__m and self.__s == o.__s
```

# Instancias como funciones

- Si `f` es una función equivale a `f.__call__()`
- Si la clase implementa el método `__call__` y se utiliza la instancia como una función ejecutará dicho método.

```
class A:
```

```
    def __call__(self, param):  
        return 'mensaje ' + param
```

```
a = A()
```

```
a('hola')
```

# Clases Abstractas

- No está implementado en Python aunque se dispone del **módulo abc**.

- Se puede simular:

```
class A:  
    def método(self):  
        raise NotImplementedError  
  
a = A()  
a.método() # Lanza la exception
```

- Si tenemos una clase que hereda de A, debería sobrescribir método() si no lo hace sería también abstracta.