



**Python 3**

## 6. Programación orientada a objetos (POO)

Carolina Mañoso, Ángel P. de Madrid y Miguel Romero

# Índice

## ◆ Definición POO y conceptos

## ◆ Sintaxis

- Ejemplo 1
- Ejemplo 2

## ◆ Herencia

## ◆ Métodos especiales




# POO y conceptos

- ◆ *POO* es una paradigma de la programación en el que los conceptos del mundo real se modelan a través de *clases* y *objetos* y el programa consiste en interacciones entre los objetos.
- ◆ Un *objeto* es una entidad que agrupa un *estado* y una *funcionalidad* relacionados.
  - El estado se define a través de variables llamadas *atributos*, mientras que la funcionalidad se modela a través de funciones llamadas *métodos* del objeto.
  - Ejemplo objeto `coche`: atributos (`marca`, `modelo`, `tipo carburante`, ...) y métodos (`arrancar`, `frenar`, ...).
- ◆ Una *clase* es la plantilla genérica a partir de la cuál instanciar los objetos, en la que se define qué atributos y métodos tendrán los objetos de la clase.

# Sintáxis (1/3)

- ◆ Las clases se definen con la palabra clave `class` seguida del nombre de la clase (la primera en mayúsculas), dos puntos (`:`) y, a continuación, sangrado, el cuerpo de la clase.
- ◆ En el cuerpo se define el método `__init__` y el resto de métodos:
  - El método `__init__` (doble barra al principio y al final), llamado *constructor*, se ejecuta justo después de crear un nuevo objeto a partir de la clase (proceso que denominamos *instanciación*). Sirve para realizar cualquier proceso de inicialización que sea necesario.
  - El primer parámetro de `__init__` y del resto de métodos es `self`. Sirve para referirse al objeto actual.

```
class Coche:
```

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

## Sintáxis (2/3)

- El resto de métodos se definen a continuación:

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

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

# Sintáxis (3/3)

- ◆ Para crear el objeto basta con escribir el nombre de la clase seguido de los parámetros que sean necesarios entre paréntesis (son los que se pasarán a `__init__`):

```
mi_coche = Coche(3)
```

- ◆ Una vez creado el objeto, accedemos a los atributos y métodos mediante `objeto.atributo` y `objeto.metodo()`:

```
print(mi_coche.gasolina)
```

```
mi_coche.arrancar()
```

```
mi_coche.conducir()
```

# Ejemplo 1

◆ Crear la clase `Persona` con los atributos `edad` y `nombre`.

- Crear el método `cumpleaños`, que incrementa la edad en 1.

```
class Persona():  
  
    # método constructor de objeto.  
    def __init__(self, edad, nombre):  
        self.edad = edad # pone la edad  
        self.nombre = nombre # pone el nombre  
  
    # método cumpleaños, incrementa la edad en 1  
    def cumpleaños(self):  
        self.edad += 1  
  
luis = Persona(31, "Luis")  
print(luis.edad, luis.nombre)  
luis.cumpleaños()  
print("Despues del cumpleaños:", luis.edad)
```



## Ejemplo 2 (1/3)

◆ Crear una clase, `Volumen`, que sea el control de volumen de un reproductor de música: con el atributo del nivel de volumen, `nivel`, y con los métodos que definen lo que podemos hacer: `subir`, `bajar` o `silenciar` el volumen.

- Al poner en marcha el reproductor de música el volumen se situará en el nivel 3.
- Con los botones de subir o bajar el nivel aumentará o disminuirá de 1 en 1.
- El botón silenciar situará el nivel en el 0.
- El volumen no podrá superar el nivel 10, si se intenta subir permanecerá igual.
- El volumen no podrá bajar del nivel 0, si se intenta bajar permanecerá igual.
- Cada vez que se accione un botón se mostrará en pantalla el nivel de volumen.



## Ejemplo 2 (2/3)

```
class Volumen:
    def __init__(self): # método constructor de objeto. Activa volumen
        self.nivel = 3 # sitúa el nivel de volumen en 3
        print('nivel', self.nivel)

    def subir(self): # método para subir el nivel de 1 en 1
        self.nivel += 1
        if self.nivel > 10: # al intentar sobrepasar el nivel 10...
            self.nivel = 10 # el nivel permanece en 10
        print('nivel', self.nivel)

    def bajar(self): # método para bajar el nivel de 1 en 1
        self.nivel -= 1
        if self.nivel < 0: # al intentar bajar por debajo del nivel 0...
            self.nivel = 0 # el nivel permanece en 0
        print('nivel', self.nivel)

    def silenciar(self): # método para silenciar
        self.nivel = 0 # el nivel se sitúa en el 0
        print('nivel', self.nivel)
```



## Ejemplo 2 (3/3)

```
controlVolumen = Volumen() # crea el objeto y volumen en el 3
controlVolumen.subir()     # sube el volumen del nivel 3 al 4
controlVolumen.subir()     # sube el volumen del nivel 4 al 5
controlVolumen.bajar()     # baja el volumen del nivel 5 al 4
controlVolumen.silenciar()  # baja del nivel 4 al 0
controlVolumen.bajar()     # como el volumen está en 0 se mantiene igual
```



# Herencia (1/5)

- ◆ Para crear una nueva clase (*subclase*) a partir de una existente (*superclase*), cuando se asigna el nuevo nombre se indica entre paréntesis el nombre de la clase de la cual queremos que herede sus propiedades y métodos.

```
class Superclase():  
    #código de la clase  
class Subclase(Superclase):  
    #código de la clase
```

- ◆ Si la nueva clase es exactamente igual, sin contenido, escribimos la sentencia `pass` que no hace nada.

- Se usa cuando una sentencia se requiere por sintaxis pero el programa no requiere ninguna acción.

```
class Superclase():  
    #código de la clase  
class Subclase(Superclase):  
    pass
```

## Herencia (2/5)

- Ejemplo: Creamos la clase `Graves` que funcionará exactamente igual que la clase `Volumen`.

```
class Graves(Volumen):  
    '''se crea la clase graves a partir de clase volumen'''  
    pass # como la función no tiene contenido: pass  
  
control_graves = Graves() # crea el objeto y nivel a 3  
control_graves.subir() # sube
```

- ◆ Si la nueva clase contiene otro parámetro se debe escribir un nuevo método `__init__` que se ejecutará en lugar del superior (sobreescribir métodos).
  - Ejemplo: se crea la nueva clase `Volumen_velocidad` que deriva también de la clase `Volumen`, modificando el método `__init__` constructor para añadir la propiedad `velocidad`.

# Herencia (3/5)

```
class Volumen_velocidad(Volumen):  
    def __init__(self, velocidad):  
        if velocidad > 120:  
            self.nivel = 5  
        elif velocidad > 100:  
            self.nivel = 4  
        else:  
            self.nivel = 3  
        print("nivel",self.nivel)
```

```
control_volumen_velocidad = Volumen_velocidad(110)  
Control_volumen_velocidad.subir()
```



# Herencia (4/5)

- ◆ Si el método `__init__` es igual pero añade algún parámetro nuevo, se referencia, al de la superclase:

```
class Subclase(Superclase):
    def __init__(self, old1, new1):
        Superclase.__init__(self, old1)
        self.new1 = new1

#-----
class Padre(Persona):
    def __init__(self, edad, nombre):
        Persona.__init__(self, edad, nombre)
        self.hijos = []
    def añade_hijo(self, hijo):
        self.hijos.append(hijo)
    def print_hijos(self):
        print("los hijos de", self.nombre, "son")
        for hijo in self.hijos:
            print(hijo.nombre)

luis = Padre(50, "Luis")
luisito = Persona(15, "Luisito")
luisita = Persona(12, "Luisita")
luis.añadir_hijo(luisito)
luis.añadir_hijo(luisita)
luis.print_hijos()
```



# Herencia (5/5)

- ◆ Herencia múltiple: Una clase puede heredar de varias clases a la vez. Basta con enumerar las clases de las que hereda separándolas por comas.

```
class Subclase(Superclase1, Superclase2):  
    #código de la clase
```

- ◆ En el caso de que alguna de las clases padre tuvieran métodos con el mismo nombre prevalecería el de la clase mas a la izquierda en la definición.

# Métodos especiales (1/6)

◆ En Python realmente *todo son clases*: números, booleanos, cadenas, diccionarios, listas y tuplas...

- La función `isinstance`, nos indica si un objeto concreto es una instancia de una clase particular:

```
>>> isinstance("abc", str)
```

```
>>> isinstance(25, str)
```

- Python tiene una clase llamada `object`. Cualquier otra clase deriva de ella.

```
>>> isinstance(25, object)
```

```
>>> isinstance(str, object)
```

- Se dice que `object` es la superclase y, por lo tanto, toda instancia de cualquier clase es un `object`.

```
>>> help(object)
```

```
Help on class object in module builtins:
```

```
class object
```

```
| The most base type
```



# Métodos especiales (2/6)

```
>>> help(int)
```

```
Help on class int in module builtins:
```

```
class int(object)
```

```
|   int(x=0) -> integer
```

```
|   int(x, base=10) -> integer
```

- Por lo tanto, existen métodos especiales de cada tipo de objeto para poder manejarlos. Así, en el caso de `int`

```
bit_length(...)
```

```
|       int.bit_length() -> int
```

```
|       Number of bits necessary to represent self  
in binary.
```

```
|       >>> bin(37)
```

```
|       '0b100101'
```

```
|       >>> (37).bit_length()
```

```
|       6
```

- **Práctica:** `dir(int)`, `help(int.bit_length)`, `help(str)`.

# Métodos especiales (3/6)

## ◆ Métodos especiales para manejar cadenas:

- `S.count(sub[, start[, end]])` Devuelve el número de veces que se encuentra `sub` en la cadena. Los parámetros opcionales `start` y `end` definen una subcadena en la que buscar.
- `S.find(sub[, start[, end]])` Devuelve la posición en la que se encontró por primera vez `sub` en la cadena o -1 si no se encontró.
- `S.join(sequence)` Devuelve una cadena resultante de concatenar las cadenas de la secuencia `seq` separadas por la cadena sobre la que se llama el método.
- `S.partition(sep)` Busca el separador `sep` en la cadena y devuelve una tupla con la subcadena hasta dicho separador, el separador en sí, y la subcadena del separador hasta el final de la cadena. Si no se encuentra el separador, la tupla contendrá la cadena en sí y dos cadenas vacías.

# Métodos especiales (4/6)

- `S.replace(old, new[, count])` Devuelve una cadena en la que se han reemplazado todas las ocurrencias de la cadena `old` por la cadena `new`. Si se especifica el parámetro `count`, este indica el número máximo de ocurrencias a reemplazar.
- `S.split([sep [,maxsplit]])` Devuelve una lista conteniendo las subcadenas en las que se divide nuestra cadena al dividir las por el delimitador `sep`. En el caso de que no se especifique `sep`, se usan espacios. Si se especifica `maxsplit`, este indica el número máximo de particiones a realizar.

# Métodos especiales (5/6)

## ◆ Métodos especiales para manejar listas:

- `L.append(object)` Añade un objeto al final de la lista.
- `L.count(value)` Devuelve el número de veces que se encontró `value` en la lista.
- `L.extend(iterable)` Añade los elementos del iterable a la lista.
- `L.index(value[, start[, stop]])` Devuelve la posición en la que se encontró la primera ocurrencia de `value`. Si se especifican, `start` y `stop` definen las posiciones de inicio y fin de una sublista en la que buscar.
- `L.insert(index, object)` Inserta el objeto `object` en la posición `index`.
- `L.pop([index])` Devuelve el valor en la posición `index` y lo elimina de la lista. Si no se especifica la posición, se utiliza el último elemento de la lista.
- `L.remove(value)` Eliminar la primera ocurrencia de `value` en la lista.
- `L.reverse()` Invierte la lista.

# Métodos especiales (6/6)

## ◆ Métodos especiales para manejar diccionarios:

- `D.get(k[, d])` Busca el valor de la clave `k` en el diccionario. Es equivalente a utilizar `D[k]` pero al utilizar este método podemos indicar un valor a devolver por defecto si no se encuentra la clave, mientras que con la sintaxis `D[k]`, de no existir la clave se lanzaría una excepción.
- `D.has_key(k)` Comprueba si el diccionario tiene la clave `k`. Es equivalente a la sintaxis `k in D`.
- `D.items()` Devuelve una lista de tuplas con pares clave-valor.
- `D.keys()` Devuelve una lista de las claves del diccionario.
- `D.pop(k[, d])` Borra la clave `k` del diccionario y devuelve su valor. Si no se encuentra dicha clave se devuelve `d` si se especificó el parámetro o bien se lanza una excepción.
- `D.values()` Devuelve una lista de los valores del diccionario.

# Aviso



Python 3 by C. Mañoso, A. P. de Madrid, M. Romero is licensed under a [Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional License](https://creativecommons.org/licenses/by-nc-sa/4.0/).

Esta colección de transparencias se distribuye con fines meramente docentes.

Todas las marcas comerciales y nombres propios de sistemas operativos, programas, hardware, etc. que aparecen en el texto son marcas registradas propiedad de sus respectivas compañías u organizaciones.