

## UD04. PYTHON (P3)

**Sistemas de Gestión Empresarial**

**2º Curso // CFGS DAM // Informática y Comunicaciones**

**Alfredo Oltra**

**Cicles  
Formatius**

## ÍNDIX

<b>1 CLASES</b>	<b>4</b>
1.1 Clases	4
1.2 Nomenclatura	6
1.3 Herencia	8
<b>2 MÓDULOS</b>	<b>9</b>
<b>3 GENERADORES</b>	<b>10</b>
3.1 Iterables e iteradores	10
3.2 Generadores	10
<b>4 BIBLIOGRAFIA</b>	<b>11</b>
<b>5 AUTORES</b>	<b>11</b>

Versión: 231113.0104

## Licencia



**Reconocimiento – NoComercial – CompartirIgual (by-nc-sa).** No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

## Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

☑ Atención. Importante prestar atención a esta información.

📖 Interesante. Ofrece información sobre algún detalle a tener en cuenta.

## 1 CLASES

### 1.1 Clases

Las clases en Python heredan inicialmente de la clase predefinida `object` (aunque no se especifique).



La clase `object` proporciona métodos que todas las clases tienen, como `__str__`.

Un ejemplo de definición de clase con atributos, constructor y métodos podría ser.

```
# Heredamos de object para obtener una clase.
class Humano(object):

    # Un atributo de clase es compartido por todas las instancias de esta
    # clase
    especie = "H. sapiens"

    # Constructor básico
    def __init__(self, nombre, edad):
        # Asigna el argumento al atributo nombre de la instancia
        self.nombre = nombre
        self.edad = edad

    # Un método de instancia. Todos los métodos toman self como primer
    # argumento
    def decir(self, msg):
        return "%s: %s" % (self.nombre, msg)

    # Un metodo de clase es compartido a través de todas las instancias
    # Son llamados con la clase como primer argumento
    @classmethod
    def get_especie(cls):
        return cls.especie

    # Un metodo estatico es llamado sin la clase o instancia como
    # referencia
    @staticmethod
    def roncar():
        return "*roncar*"
```

```
# Instancia una clase
i = Humano(nombre="Ian")
print i.decir("hi") # imprime "Ian: hi"

j = Humano("Joel")
print j.decir("hello") #imprime "Joel: hello"

# Llama nuestro método de clase
i.get_especie() # => "H. sapiens"

# Cambia los atributos compartidos
Humano.especie = "H. neanderthalensis"
i.get_especie() # => "H. neanderthalensis"
j.get_especie() # => "H. neanderthalensis"

# Llama al método estático
Humano.roncar() # => "*roncar*"
```

El constructor se define en la función `__init__`. Utiliza como primer parámetro *self*, que es una referencia al propio objeto (algo así como *this* en lenguajes como *java*), seguido de todos los parámetros que sean necesarios.



Es importante darse cuenta de que las variables inicializadas en el ámbito de la clase son atributos de la clase, no de la instancia (que es lo habitual en lenguajes con *Java* o *C++* o *C#*). La definición de variables de instancia se puede realizar desde cualquier método de la misma, aunque lo habitual es hacerlo desde el constructor para evitar problemas de inicialización.

Los métodos de instancia son los métodos normales, funciones que se ejecutan dentro del contexto del objeto. De manera similar al constructor, utilizan como primer parámetro *self*.

Los métodos de clase (*@classmethod*) son métodos que afectan a la propia clase, es decir, que sus acciones afectan a todas las instancias. Permiten modificar el estado de la clase, es decir, pueden modificar sus atributos de clase pero no los de la instancia. Requieren como parámetro *cls*, que hace referencia a la clase.



Básicamente la función del decorador *@classmethod* es hacer que la función que decora reciba como primer parámetro la clase y no el objeto.

Uno de los usos más habituales de este tipo de métodos es que sean utilizados como factorias, es decir, como generadores de objetos de esa clase:

```
@classmethod
def desdeFechaNacimiento(cls, nombre, anyo_nac):
    return cls(nombre, date.today().year - anyo_nac)

z = Humano.desdeNacimiento("Mateo", 2010)
```

Los métodos estáticos son muy parecidos a los métodos de clase, pero, a diferencia de ellos, no permiten acceder a la clase ni modificar su estado. Entonces ¿qué sentido tienen? Pues crear utilidades referidas a esa clase, utilidades que obtiene sus datos a través de parámetros.

El mismo método factoria mostrado en anteriormente pero desarrollado de manera estática

```
@staticmethod
def desdeFechaNacimiento(nombre, anyo_nac):
    return Humano(nombre, date.today().year - anyo_nac)

z = Humano.desdeNacimiento("Mateo", 2010)
```

En estos casos, si la factoria está pensada para generar objetos de la misma clase, solo que a partir de parámetros que no se encuentran en el constructor lo lógico es utilizar un método de clase. En caso de que lo que se busque generar son objetos de subclases, podemos optar por el método estático.

## 1.2 Nomenclatura

En muchas ocasiones, dentro de un clase, vemos un uso bastante grande de *underscores* (subrayados, `_`) a la hora de nombrar métodos y variables. Aunque en ocasiones simplemente se considera una aplicación de una guía de estilo que no afecta al funcionamiento, en otras si que su uso está vinculado con la funcionalidad:

### Un subrayado a modo de prefijo

Indica a otros programadores que la variable o el método está pensado para ser usado de manera interna a la clase. Es simplemente una manera de indicar que, por ejemplo, el método no debe ser importado para ser usado externamente.

```
_este_es_un_metodo_interno(self, nombre)
```

### Doble subrayado (o más) a modo de prefijo y no más de uno al final

El interprete sustituye el nombre del elemento por `__classname__nombre` de manera que además se asegura que no se producirá una colisión con otro nombre igual de otra clase. En muchas ocasiones se utiliza esta nomenclatura para simular elementos privados ya que al buscar por `__nombre` el interprete devolverá un error.

```
class Ejemplo(object):

    def __metodo(self, nombre):
        print("Hola!")
        return

i = Ejemplo()
i.__metodo()
AttributeError: 'Ejemplo' object has no attribute '__metodo'

i._Ejemplo__metodo()
"Hola"
```

### Doble subrayado a modo de prefijo y sufijo

Definen los llamados métodos mágicos o *dunders*. Son métodos especiales, que no suelen ser llamados de manera directa por el programador, sino que son llamados por el intérprete en determinadas circunstancias, por ejemplo `__init__`.

Lo más habitual es usarlos para sobrecargarlos y que esas llamadas implícitas del intérprete hagan lo que queramos nosotros. Por ejemplo redefinir `__str__` permite implementar como se mostrará el objeto cuando sea convertido a una cadena o mostrado por pantalla con un *print*.

```
class Persona(object):

    def __init__(self, nom):
        self.nombre = nom
        return

    def __str__(self):
        str = self.nombre + "!!"
        return str
```

### 1.3 Herencia

Como se ha visto en los ejemplos anteriores, la herencia se puede conseguir pasando como parámetro el nombre de la clase padre a la hora de definir la clase.

Es posible llamar a los métodos de la clase padre utilizando la función *super()*

```
class Mujer(Persona):  
  
    def __init__(self, nombre, es_madre):  
        self.es_madre = es_madre  
        super().__init__(nombre)
```

Además Python soporta herencia multiple, es decir, que puede heredar no solo de una sino de varias clases. Eso puede producir colisiones en caso de que varias de las clases padre implementen un método con el mismo nombre. Para solucionarlo se crea un orden de búsqueda basado en el orden en el que se definen los padres.

```
class A1:  
    def met():  
        print("Hola A1")  
  
class A2:  
    def met():  
        print("Hola A2")  
  
class A3(A2,A1):  
    pass  
  
i = A3()  
A3.met()  # => "Hola A2"
```



## 2 MÓDULOS

Un módulo no es más que un fichero `.py` que tiene funciones, variables o clases que puede ser usadas en otro fichero, es decir, en otro módulo.



*Python* permite importar módulos, tanto creados por nosotros, como existentes en el sistema. Una herramienta para descargar módulos más populares es *pip*.

```
# Puedes importar módulos
import math
print(math.sqrt(16)) # => 4.0

# Puedes obtener funciones específicas desde un módulo
from math import ceil, floor
print(ceil(3.7)) # => 4.0
print(floor(3.7)) # => 3.0

# Puedes importar todas las funciones de un módulo. Es similar
# a import math pero sin que sea necesario utilizar math siempre como
# prefijo.
# Precaución: Esto no es recomendable
from math import *
print(sqrt(16)) # => 4.0

# Puedes acortar los nombres de los módulos
import math as m
math.sqrt(16) == m.sqrt(16) # => True

# Los módulos de Python son sólo archivos ordinarios de Python.
# Puedes escribir tus propios módulos e importarlos. El nombre del módulo
# es el mismo del nombre del archivo.

# Puedes encontrar qué funciones y atributos definen un módulo con dir
import math
dir(math)
```

### 3 GENERADORES

Ejemplo de creación de generadores:

```
# Los generadores te ayudan a hacer un código perezoso (lazy)
def duplicar_numeros(iterable):
    for i in iterable:
        yield i + i

# Un generador crea valores sobre la marcha.
# En vez de generar y retornar todos los valores de una vez, crea uno en
# cada iteración.

# Fíjate que 'range' es un generador. Crear una lista 1-9000000000 tomaría
# mucho tiempo en crearse.
_rango = range(1, 9000000000)
```

#### 3.1 Iterables e iteradores

Un iterable es un objeto que puede ser iterado, es decir, que puede ser accedidos con un índice, como una lista o una tupla (por ejemplo haciendo `tupla[2]`). Son los objetos a los que les puedo aplicar un `for`.

Por su parte, un iterador es un objeto que permite iterar, recorrer, un iterable con la función `next`. Por ejemplo:

```
# Creo una iterable, en este caso una lista
lista = [1,2,3,4]
# Con la función it creo que objeto de tipo iterable, en este caso para
# recorrer la lista
it = it(lista)
print(next(it)) # Muestra 1
```

#### 3.2 Generadores

Un generador es una función que puede ser iterada, es decir, sobre la que podemos crear un iterador y recorrer los valores que nos va dando. En cierta manera es una función que guarda el estado entre ejecuciones, de manera que cada vez que es llamada calcula el nuevo valor devolver o una excepción de fin de elementos (*StopIteration*).

Se pueden crear de dos maneras: utilizando `yield` o mediante una sintaxis similar a las *list comprehension*, pero usando paréntesis `()` en vez de corchetes `[]`

```
lista = [1,2,3,4]

# Generadpr con una función
def duplicar_numeros(iterable):
    for i in iterable:
        yield i + i

for n in duplicar_numeros(lista)
    print(n)

# Generador vía generator comprehension
gen = (i + i for i in lista)

for n in gen
    print(n)
```

## 4 BIBLIOGRAFIA

1. [Learn X in Y Minutes.](#)
2. [Aprende Python con Alf](#)
3. [Python para todos.](#)

## 5 AUTORES

A continuación ofrecemos en orden alfabético (por apellido) el listado de autores que han hecho aportaciones a este documento.

- Jose Castillo Aliaga
- Sergi García Barea
- Alfredo Oltra Orengo

Gran parte del contenido ha sido obtenido del material con licencia CC BY SA disponible en [LearnXinYminutes.](#)