UD05 – POO Objetos y Clases

Introducción:

Hasta ahora hemos estado usando objetos de forma totalmente transparente, casi sin ser conscientes de ello. Pero, en realidad, todo en Python es un objeto, desde números a funciones. El lenguaje provee ciertos mecanismos para no tener que usar explícitamente técnicas de orientación a objetos.

Llegados a este punto, investigaremos en profundidad la creación y manipulación de clases y objetos, así como todas las técnicas y procedimientos que engloban este paradigma

Programación orientada a objetos

La programación orientada a objetos (POO) o en sus siglas inglesas OOP es una manera de programar (paradigma) que permite llevar al código mecanismos usados con entidades de la vida real.

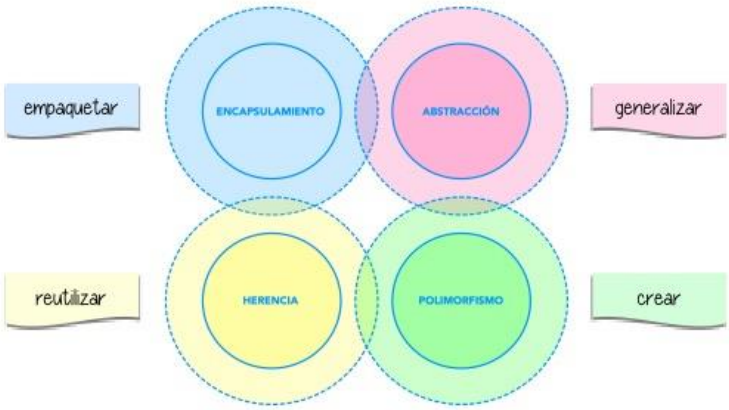
Sus beneficios son los siguientes:

Encapsulamiento: Permite empaquetar el código dentro de una unidad (objeto) donde se puede determinar el ámbito de actuación.

Abstracción: Permite generalizar los tipos de objetos a través de las clases y simplificar el programa.

Herencia: Permite reutilizar código al poder heredar atributos y comportamientos de una clase a otra.

Polimorfismo: Permite crear múltiples objetos a partir de una misma pieza flexible de código.

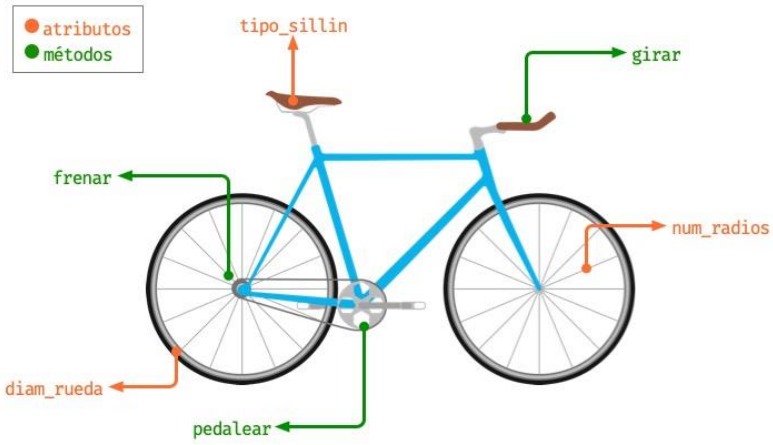


¿Qué es un objeto?

Un objeto es una estructura de datos personalizada que contiene datos y código:

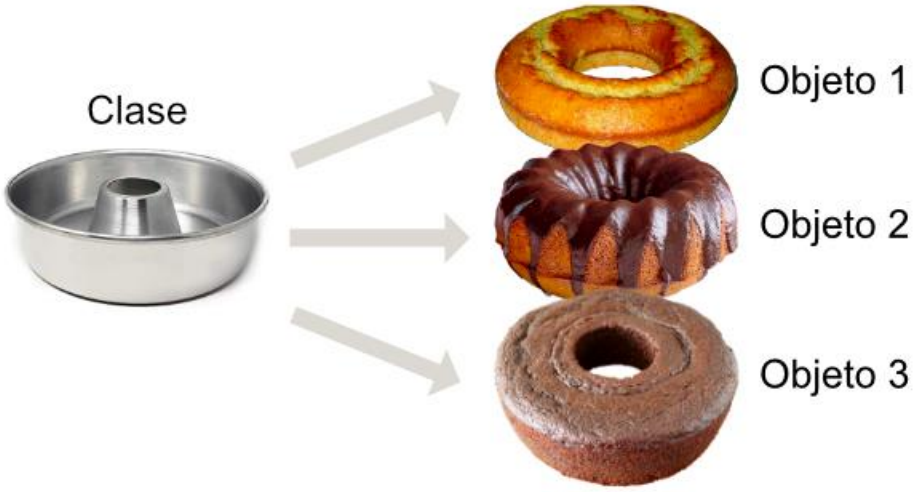
Elementos	¿Qué son?	¿Cómo se llaman?	¿Cómo se identifican?
Datos	Variables	Atributos	Mediante sustantivos
Código	Funciones	Métodos	Mediante verbos

Un objeto representa una instancia única de alguna entidad (a través de los valores de sus atributos) e interactúa con otros objetos (o consigo mismo) a través de sus métodos.



¿Qué es una clase?

Para crear un objeto primero debemos definir la clase que lo contiene. Podemos pensar en la clase como el molde con el que se crean nuevos objetos de ese tipo.



En el proceso de diseño de una clase hay que tener en cuenta – entre otros – el principio de responsabilidad única 7, intentando que los atributos y los métodos que contenga esa clase estén enfocados a un objetivo único y bien definido.

Creando objetos

Empecemos por crear nuestra primera clase. En este caso vamos a modelar algunos de los droides de la saga StarWars:

Para ello usaremos la palabra reservada class seguida del nombre de la clase:

```
class StarWarsDroid:
```

Consejo: Los nombres de clases se suelen escribir en formato CamelCase y en singular

Existen multitud de droides en el universo StarWars. Una vez que hemos definido la clase genérica podemos crear instancias/objetos (droides) concretos:

```
c3po = StarWarsDroid()
r2d2 = StarWarsDroid()
bb8 = StarWarsDroid()
```

Añadiendo métodos

Un método es una función que forma parte de una clase o de un objeto. En su ámbito tiene acceso a otros métodos y atributos de la clase o del objeto al que pertenece.

La definición de un método (de instancia) es análoga a la de una función ordinaria, pero incorporando un primer parámetro `self` que hace referencia a la instancia actual del objeto.

Una de las acciones más sencillas que se pueden hacer sobre un droide es encenderlo o apagarlo. Vamos a implementar estos dos métodos en nuestra clase:

```
class Droid:
    def switch_on(self):
        print("Hi! I'm a droid. Can I help you?")

    def switch_off(self):
        print("Bye! I'm going to sleep")


k2so = Droid()

k2so.switch_on()
Hi! I'm a droid. Can I help you?

k2so.switch_off()
Bye! I'm going to sleep
```

Añadiendo atributos

Un atributo no es más que una variable, un nombre al que asignamos un valor, con la particularidad de vivir dentro de una clase o de un objeto.

Supongamos que, siguiendo con el ejemplo anterior, queremos guardar en un atributo el estado del droide (encendido/apagado):

```
class Droid:
    power_on = False
    def switch_on(self):
        self.power_on = True
        print("Hi! I'm a droid. Can I help you?")

    def switch_off(self):
        self.power_on = False
        print("Bye! I'm going to sleep")


k2so = Droid()

k2so.switch_on()
k2so.power_on

k2so.switch_off()
Bye! I'm going to sleep
k2so.power_on
```

Inicialización

Existe un método especial que se ejecuta cuando creamos una instancia de un objeto. Este método es `__init__` y nos permite asignar atributos y realizar operaciones con el objeto en el momento de su creación. También es ampliamente conocido como el constructor.

Veamos un ejemplo de este método con nuestros droides en el que únicamente guardaremos el nombre del droide como un atributo del objeto:

```
class Droid:
    name = ""
    def __init__(self, name: str):
        self.name = name


droid = Droid('BB-8')

droid.name
```

Es importante tener en cuenta que si no usamos `self` estaremos creando una variable local en vez de un atributo del objeto:

```
class Droid:
    name = ""
    def __init__(self, name: str):
        name = name  # No lo hagas!


droid = Droid('BB-8')

droid.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Droid' object has no attribute 'name'
```

Ejercicio:

Escriba una clase MobilePhone que represente un teléfono móvil.

Atributos:

manufacturer (cadena de texto)

screen_size (flotante)

num_cores (entero)

apps (lista de cadenas de texto)

status (False: apagado, True: encendido)

Métodos:

__init__(self, manufacturer, screen_size, num_cores)

power_on(self)

power_off(self)

install_app(self, app) (no instalar la app si ya existe)

uninstall_app(self, app) (no borrar la app si no existe)

Propiedades

Como hemos visto previamente, los atributos definidos en un objeto son accesibles públicamente. Esto puede parecer extraño a personas que vengan de otros lenguajes de programación (véase Java). En Python existe un cierto «sentido de la responsabilidad» a la hora de programar y manejar este tipo de situaciones: Casi todo es posible a priori pero se debe controlar explícitamente.

Una primera solución «pitónica» para la privacidad de los atributos es el uso de propiedades. La forma más común de aplicar propiedades es mediante el uso de decoradores:

@property para leer el valor de un atributo («getter»).

@name.setter para escribir el valor de un atributo.

Veamos un ejemplo en el que estamos ofuscando el nombre del droide a través de propiedades:

```
class Droid:
    def __init__(self, name: str):
        self.hidden_name = name

    @property
```

```
    def name(self) -> str:
        print('inside the getter')
        return self.hidden_name

    @name.setter
    def name(self, name: str) -> None:
        print('inside the setter')
        self.hidden_name = name

droid = Droid('N1-G3L')

droid.name

droid.name = 'Nigel'

droid.name
```

En cualquier caso, seguimos pudiendo acceder directamente a .hidden_name:

```
droid.hidden_name
```

Incluso podemos cambiar su valor:

```
droid.hidden_name = 'waka-waka'
```

VALORES CALCULADOS

Una propiedad también se puede usar para devolver un valor calculado (o computado).

A modo de ejemplo, supongamos que la altura del periscopio de los droides astromecánicos se calcula siempre como un porcentaje de su altura. Veamos cómo implementarlo:

```
class AstromechDroid:
    def __init__(self, name: str, height: float):
        self.name = name
        self.height = height

    @property
    def periscope_height(self) -> float:
        return 0.3 * self.height


droid = AstromechDroid('R2-D2', 1.05)

droid.periscope_height  # podemos acceder como atributo


droid.periscope_height = 10  # no podemos modificarlo
```

Las propiedades no pueden recibir parámetros ya que no tiene sentido semánticamente:

```
class AstromechDroid:
    def __init__(self, name: str, height: float):
        self.name = name
        self.height = height

    @property
    def periscope_height(self, from_ground: bool = False) -> float:
```

```
        height_factor = 1.3 if from_ground else 0.3
        return height_factor * self.height


droid = AstromechDroid('R2-D2', 1.05)

droid.periscope_height


droid.periscope_height(from_ground=True)
```

En este caso tendríamos que implementar un método para resolver el escenario planteado.

Consejo: La ventaja de usar valores calculados sobre simples atributos es que el cambio de valor en un atributo no asegura que actualicemos otro atributo, y además siempre podremos modificar directamente el valor del atributo, con lo que podríamos obtener efectos colaterales indeseados.

CACHEANDO PROPIEDADES

En los ejemplos anteriores hemos creado una propiedad que calcula el alto del periscopio de un droide astromecánico a partir de su altura. El «coste» de este cálculo es bajo, pero imaginemos por un momento que fuera muy alto.

Si cada vez que accedemos a dicha propiedad tenemos que realizar ese cálculo, estaríamos siendo muy ineficientes (en el caso de que la altura del droide no cambiara). Veamos una aproximación a este escenario usando el cacheado de propiedades:

```
class AstromechDroid:
    def __init__(self, name: str, height: float):
        self.name = name
        self.height = height  # llamada al setter

    @property
    def height(self) -> float:
        return self._height

    @height.setter
    def height(self, height: float) -> None:
        self._height = height
        self._periscope_height = None  # invalidar caché

    @property
    def periscope_height(self) -> float:
        if self._periscope_height is None:
            print('Calculating periscope height...')
            self._periscope_height = 0.3 * self.height
        return self._periscope_height
```

Probamos ahora la implementación diseñada, modificando la altura del droide:

```
droid = AstromechDroid('R2-D2', 1.05)

droid.periscope_height

droid.periscope_height  # Cacheado!
```

```
droid.height = 1.15

droid.periscope_height

droid.periscope_height  # Cacheado!
```

Ocultando atributos

Python tiene una convención sobre aquellos atributos que queremos hacer «privados» (u ocultos): comenzar el nombre con doble subguión __

```
class Droid:
    def __init__(self, name: str):
        self.__name = name


droid = Droid('BC-44')

droid.__name  # efectivamente no aparece como atributo
```

Lo que realmente ocurre tras el telón se conoce como «name mangling» y consiste en modificar el nombre del atributo incorporado la clase como un prefijo. Sabiendo esto podemos acceder al valor del atributo supuestamente privado:

```
droid._Droid__name
```

Nota: La filosofía de Python permite hacer casi cualquier cosa con los objetos que se manejan, eso sí, el sentido de la responsabilidad se traslada a la persona que desarrolla e incluso a la persona que hace uso del objeto.

Atributos de clase

Podemos asignar atributos a una clase y serán asumidos por todos los objetos instanciados de esa clase.

A modo de ejemplo, en un principio, todos los droides están diseñados para que obedezcan a su dueño. Esto lo conseguiremos a nivel de clase, salvo que ese comportamiento se sobreescriba:

```
class Droid:
    obeys_owner = True  # obedece a su dueño


good_droid = Droid()
good_droid.obeys_owner


t1000 = Droid()  # T-1000 (Terminator)
t1000.obeys_owner = False
t1000.obeys_owner


Droid.obeys_owner  # el cambio no afecta a nivel de clase
```

Truco: Los atributos de clase son accesibles tanto desde la clase como desde las instancias creadas.

Hay que tener en cuenta lo siguiente:

- Si modificamos un atributo de clase desde un objeto, sólo modificamos el valor en el objeto y no en la clase.
- Si modificamos un atributo de clase desde una clase, modificamos el valor en todos los objetos pasados y futuros.

Veamos un ejemplo de esto último:

```
class Droid:
    obeys_owner = True


droid1 = Droid()
droid1.obeys_owner


droid2 = Droid()
droid2.obeys_owner


Droid.obeys_owner = False  # cambia pasado y futuro

droid1.obeys_owner

droid2.obeys_owner


droid3 = Droid()
droid3.obeys_owner
```

Métodos de clase

Un método de clase es un método que modifica o accede al estado de la clase a la que hace referencia. Recibe cls como primer parámetro, el cual se convierte en la propia clase sobre la que estamos trabajando. Python envía este argumento de forma transparente. La identificación de estos métodos se completa aplicando el decorador @classmethod a la función.

Veamos un ejemplo en el que implementamos un método de clase que muestra el número de droides creados:

```
class Droid:
    count = 0

    def __init__(self):
        Droid.count += 1

    @classmethod
    def total_droids(cls) -> None:
        print(f'{cls.count} droids built so far!')
```

```
droid1 = Droid()
droid2 = Droid()
droid3 = Droid()

Droid.total_droids()
```

Consejo: El nombre cls es sólo una convención. Este parámetro puede llamarse de otra manera, pero seguir el estándar ayuda a la legibilidad.

Métodos estáticos

Un método estático es un método que no «debería» modificar el estado del objeto ni de la clase. No recibe ningún parámetro especial. La identificación de estos métodos se completa aplicando el decorador @staticmethod a la función.

Veamos un ejemplo en el que creamos un método estático para devolver las categorías de droides que existen en StarWars:

```
class Droid:
    def __init__(self):
        pass

    @staticmethod
    def get_droids_categories() -> tuple[str]:
        return ('Messeger', 'Astromech', 'Power', 'Protocol')


Droid.get_droids_categories()
```

MÉTODOS DECORADOS

Es posible que, según el escenario, queramos decorar ciertos métodos de nuestra clase. Esto lo conseguiremos siguiendo la misma estructura de decoradores que ya hemos visto, pero con ciertos matices.

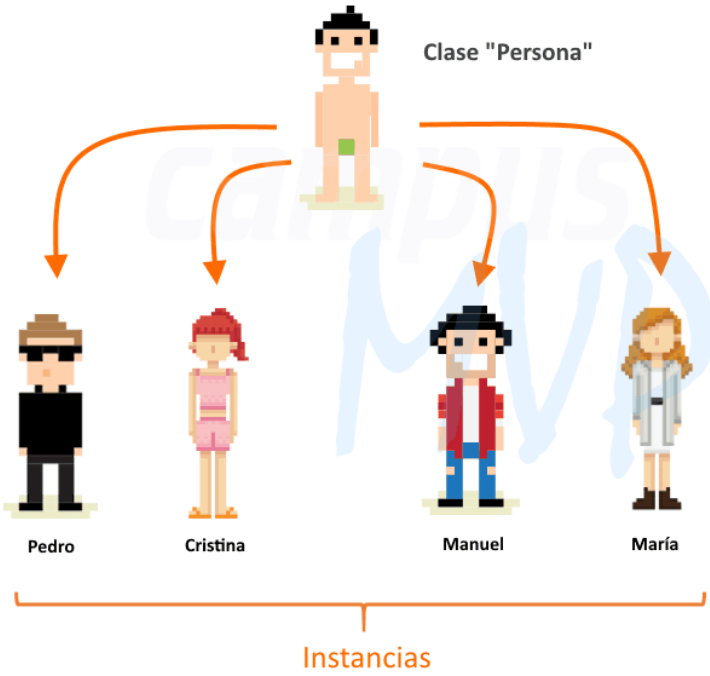
A continuación veremos un ejemplo en el que creamos un decorador para auditar las acciones de un droide y saber quién ha hecho qué:

```
class Droid:
    @staticmethod
    def audit(method):
        def wrapper(self, *args, **kwargs):
            print(f'Droid {self.name} running {method.__name__}')
            return method(self, *args, **kwargs)  # Ojo llamada!
        return wrapper

    def __init__(self, name: str):
        self.name = name
        self.pos = [0, 0]

    @audit
    def move(self, x: int, y: int):
        self.pos[0] += x
        self.pos[1] += y
```

```
    @audit
    def reset(self):
        self.pos = [0, 0]

droid = Droid('B1')

droid.move(1, 1)

droid.reset()
```

Herencia:

La herencia consiste en construir una nueva clase partiendo de una clase existente, pero que añade o modifica ciertos aspectos. La herencia se considera una buena práctica de programación tanto para reutilizar código como para realizar generalizaciones.



Nota: Cuando se utiliza herencia, la clase derivada, de forma automática, puede usar todo el código de la clase base sin necesidad de copiar nada explícitamente.

Heredar desde una clase base

Para que una clase «herede» de otra, basta con indicar la clase base entre paréntesis en la definición de la clase derivada.

Sigamos con el ejemplo galáctico: Una de las grandes categorías de droides en StarWars es la de droides de protocolo. Vamos a crear una herencia sobre esta idea:

```
class Droid:
    """ Clase Base """
    pass


class ProtocolDroid(Droid):
```

```
    """ Clase Derivada """
    pass


issubclass(ProtocolDroid, Droid)  # comprobación de herencia


r2d2 = Droid()
c3po = ProtocolDroid()
```

Vamos a añadir un par de métodos a la clase base, y analizar su comportamiento:

```
class Droid:
    def switch_on(self):
        print("Hi! I'm a droid. Can I help you?")

    def switch_off(self):
        print("Bye! I'm going to sleep")


class ProtocolDroid(Droid):
    pass


r2d2 = Droid()
c3po = ProtocolDroid()

r2d2.switch_on()


c3po.switch_on()  # método heredado de Droid


r2d2.switch_off()
```

Sobreescribir un método

Como hemos visto, una clase derivada hereda todo lo que tiene su clase base. Pero en muchas ocasiones nos interesa modificar el comportamiento de esta herencia.

En el ejemplo anterior vamos a modificar el comportamiento del método switch_on() para la clase derivada:

```
class Droid:
    def switch_on(self):
        print("Hi! I'm a droid. Can I help you?")

    def switch_off(self):
        print("Bye! I'm going to sleep")


class ProtocolDroid(Droid):
    def switch_on(self):
        print("Hi! I'm a PROTOCOL droid. Can I help you?")


r2d2 = Droid()
c3po = ProtocolDroid()
```

```
r2d2.switch_on()


c3po.switch_on()  # método heredado pero sobreescrito
```

Añadir un método

La clase derivada puede, como cualquier otra clase «normal», añadir métodos que no estaban presentes en su clase base. En el siguiente ejemplo vamos a añadir un método translate() que permita a los droides de protocolo traducir cualquier mensaje:

```
class Droid:
    def switch_on(self):
        print("Hi! I'm a droid. Can I help you?")

    def switch_off(self):
        print("Bye! I'm going to sleep")


class ProtocolDroid(Droid):
    def switch_on(self):
        print("Hi! I'm a PROTOCOL droid. Can I help you?")

    def translate(self, msg: str, *, from_lang: str) -> str:
        """ Translate from language to Human understanding """
        return f'{msg} means "ZASCA" in {from_lang}'

r2d2 = Droid()
c3po = ProtocolDroid()

c3po.translate('kiitos', from_lang='Huttese')  # idioma de Watoo


r2d2.translate('kiitos', from_lang='Huttese')  # droide genérico no
puede traducir
```

Con esto ya hemos aportado una personalidad diferente a los droides de protocolo, a pesar de que heredan de la clase genérica de droides de StarWars.

Accediendo a la clase base

Cuando tenemos métodos (o atributos) definidos con el mismo nombre en la clase base y en la clase derivada (colisión) debe existir un mecanismo para diferenciarlos.

Para estas ocasiones Python nos ofrece super() como función para acceder a métodos (o atributos) de la clase base.

Este escenario es especialmente recurrente en el constructor de aquellas clases que heredan de otras y necesitan inicializar la clase base.

Veamos un ejemplo más elaborado con nuestros droides:

```
class Droid:
    def __init__(self, name: str):
        self.name = name


class ProtocolDroid(Droid):
```

```
    def __init__(self, name: str, languages: list[str]):
        super().__init__(name)  # llamada al constructor de la clase
base
        self.languages = languages


droid = ProtocolDroid('C-3PO', ['Ewokese', 'Huttese', 'Jawaese'])

droid.name  # fijado en el constructor de la clase base


droid.languages  # fijado en el constructor de la clase derivada
```

Estructura de una clase

Durante toda la sección hemos analizado con detalle los distintos componentes que forman una clase en Python. Pero cuando todo esto lo ponemos junto puede suponer un pequeño caos organizativo.

Aunque no existe ninguna indicación formal de la estructura de una clase, podríamos establecer el siguiente formato como guía de estilo:

```
class OrganizedClass:
    """Descripción de la clase"""

    # Constructor

    # Métodos de instancia

    # Propiedades

    # Métodos mágicos

    # Decoradores

    # Métodos de clase

    # Métodos estáticos

    ...
```