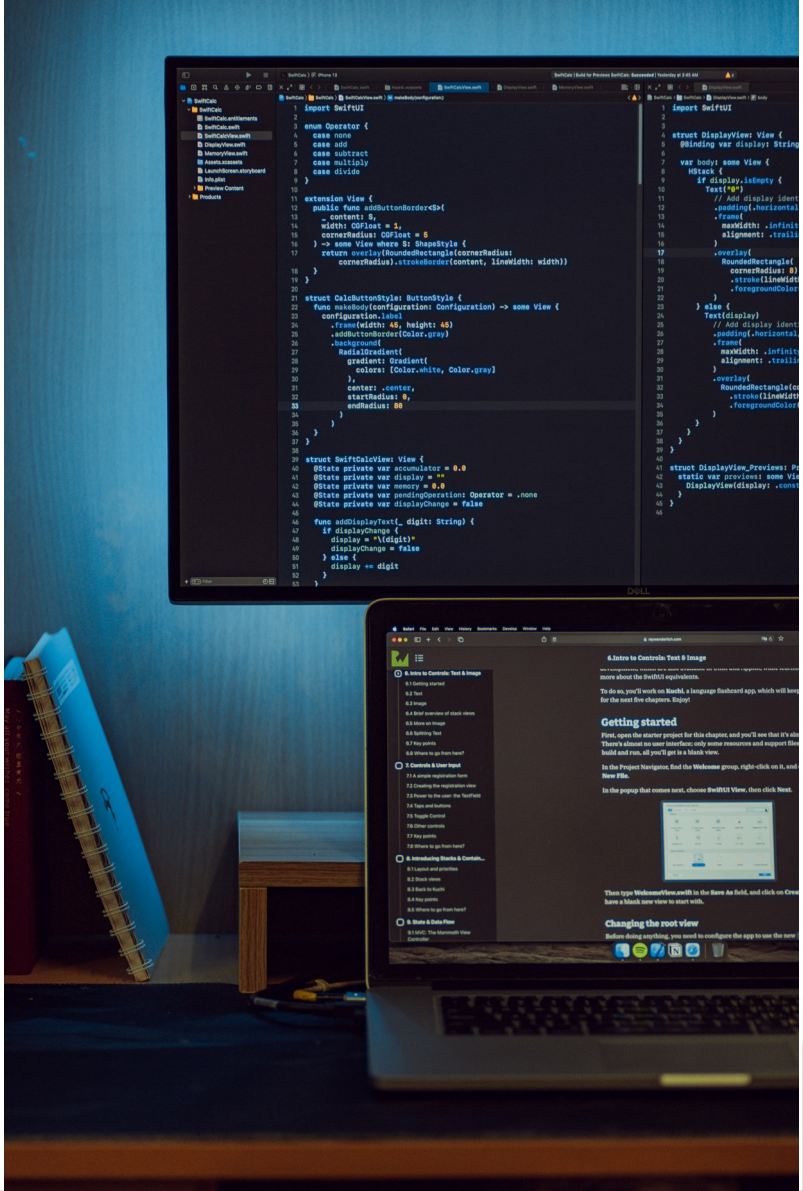
Curso Python

Escuela Politécnica Superior

Introducción a la Programación Orientada a Objetos



UNIVERSIDAD NEBRIJA

Profesor:
José Ignacio Naranjo
jnaranjo@nebrija.es

Paradigma

Es un paradigma que se basa en dos nuevos conceptos: **clase** y **objeto**

Con este nuevo paradigma -bueno, y con todos- esperamos producir código más fácil de:

- Mantener (¿código más comprensible?)
- Extender. Añadir nuevas características sin provocar problemas al software ya en producción

Paradigma II

Además:

La modelización de un problema usando clases nos permite crear una abstracción de la realidad más cercana a esta que un paradigma estructurado.

¿Qué es una clase?

Es una abstracción que representa una entidad o concepto y que tiene un estado y un comportamiento.

¿A qué nos referimos con el estado?

El estado se consigue mediante una serie de variables que pertenecen a la clase y que se conocen como **atributos**

¿A qué nos referimos con el comportamiento?

Una clase incorpora unos **métodos** para realizar las acciones para las que ha sido concebida

¿Qué es un objeto?

Un objeto es una instancia de una clase. Una clase no es más que la declaración, como el esqueleto, pero cuando estemos programando tendremos que ir instanciando objetos para ser usados.

Por ejemplo, podríamos diseñar una clase que se llamara **Persona**, pero en nuestro sistema es de esperar que necesitemos representar infinidad de instancias de personas que serían los objetos que usaríamos

```python
class Persona:
    def __init__(self, nombre, apellido):
        self.__nombre = nombre
        self.__apellido = apellido

    def saludo(self):
        print(f"Hola, soy {self.__nombre} {self.__apellido}")
```

```python
if __name__ == "__main__":
    persona1 = Persona("Alberto", "López")
    persona1.saludo()
    persona2 = Persona("Alba", "García")
    persona2.saludo()
```

```
Hola, soy Alberto López
Hola, soy Alba García

Process finished with exit code 0
```

Constructor

Cuando queremos obtener una instancia de una clase, tendremos que pedírselo a la propia clase. La instancia (**el objeto**) se devuelve gracias a que la clase define un **constructor**.

En python el "*constructor*" se define en el método **__init__**, que puede recibir un número indefinido de parámetros que se usarán para inicializar los **atributos** el nuevo objecto creado.

```
def __init__(self, nombre, apellido):
    self.__nombre = nombre
    self.__apellido = apellido
```

Esto significa que para instancia un objeto **Persona** tenemos que aportar un nombre y un apellido que serán almacenados únicamente en el nuevo objeto.

```
persona1 = Persona("Alberto", "López")
```

Métodos vs funciones

Hasta ahora conocíamos las funciones, que eran fragmentos de código que recibían unos parámetros y devolvían un resultado. Los métodos son similares, la diferencia es que una función es una entidad independiente y un método forma parte de la definición de una clase.

Los métodos están fuertemente acomplados a una clase. Son la definición de una de sus capacidades, por lo que para invocarlos tendremos que usar una instancia, a diferencia de las funciones que son independientes.

```
persona = Persona("Alberto", "López")
persona.saludo()
```

self

Quizás te has fijado en que los métodos definidos en una clase tienen un primer parámetro **self** en su declaración.

Todo método de una clase cuyo objetivo será ser ejecutado por un objeto necesita tener una referencia a sí mismo, para de esta forma tener acceso a sus **atributos**.

```python
class Persona:
    def __init__(self, nombre, apellido):
        self.__nombre = nombre
        self.__apellido = apellido

    def saludo(self):
        print(f"Hola, soy {self.__nombre} {self.__apellido}")
```

Los 4 pilares de la POO

- Encapsulamiento
- Abstracción
- Herencia
- Polimorfismo

Encapsulamiento

Es una técnica para definir las fronteras de uso y modificación de los datos (y los comportamientos) de una entidad.

No debería tener incidencia técnica, pero sí es una importante herramienta en el diseño de los programas en pos de aumentar la **legibilidad** y la **seguridad**.

Por ejemplo, una clase puede tener unos atributos que nos queremos que sean modificados ni accedidos desde otras entidades. Y, si queremos que esos datos puedan ser usados desde fuera, definiremos una interfaz que será el punto acordado para hacer uso de dichos datos

Encapsulamiento II

En el ejemplo que hemos visto hasta ahora, con la clase Persona, a pesar de que la clase tiene los atributos **nombre** y **apellido**, desde fuera del objeto no se puede acceder a sus valores y esto es por una decisión de diseño. HEMOS DECIDIDO QUE DESDE FUERA NO SE PUEDA MODIFICAR EL NOMBRE NI EL APELLIDO

```
if __name__ == "__main__":
    persona1 = Persona("Alberto", "López")
    persona1.saludo()
    print(persona1.__nombre)
```

```
Hola, soy Alberto López
Traceback (most recent call last):
  File "/home/josnar/PycharmProjects/Nebrija/curso_python_basico_
    print(persona1.__nombre)
AttributeError: 'Persona' object has no attribute '__nombre'
```

¿Cómo hemos logrado que desde fuera no se pueda acceder al nombre ni al apellido?

Encapsulamiento III

El encapsulamiento permite definir la visibilidad que se ofrece sobre los atributos y las métodos. Hay tres tipos de visibilidad:

- Pública. Significa que hay acceso libre al atributo/método
- Protegida. (lo veremos más adelante)
- Privada. Significa que NO hay acceso al atributo/método

Encapsulamiento IV

- La visiblidad pública en python se consigue al no poner ningún “underscore” al inicio de un atributo o método.

- La visibilidad privada en python se consigue poniendo dos “underscore” como prefijo del nombre del atributo o método. Esto implica que el atributo o método será privado y solamente podrá ser accesible desde el objeto de turno y no desde fuera

Encapsulamiento V

- La visiblidad pública en python se consigue al no poner ningún “underscore” al inicio de un atributo o método.

- La visibilidad privada en python se consigue poniendo dos “underscore” como prefijo del nombre del atributo o método. Esto implica que el atributo o método será privado y solamente podrá ser accesible desde el objeto de turno y no desde fuera

Abstracción

Consiste en la extracción de métodos comunes (y datos) que residirán en la clase padre, evitando así repetición de código.

```
    def imprimir_resumen(self):
        puntos_contendiente1 = self.puntos_contendiente(self._contendiente1)
        puntos_contendiente2 = self.puntos_contendiente(self._contendiente2)
        print(f"""El partido disputado en {self.__lugar} con fecha {self.__fecha} tuvo como resultado un {self._resultado}\n
Como consecuencia, el contendiente {self._contendiente1} obtuvo {puntos_contendiente1} puntos y el contendiente {self._contendiente2} \
obtuvo {puntos_contendiente2} puntos
        """)
```

imprimir_resumen servirá para imprimir la información del partido, y no se encuentra implementada en cada clase hija

Herencia

La herencia es una característica propia de la orientación a objetos. Mientras que el encapsulamiento y la abstracción son características comunes a todos los paradigmas, la herencia es una característica única del paradigma orientado a objetos.

La herencia permite agrupar datos y funcionalidad de clases con características comunes, de forma que se genera una clase padre con comportamientos y datos comunes a las clases hijas.

La consecuencia es que permite encapsular partes comunes, facilitando el mantenimiento y la lectura del código.

Herencia II

Supongamos que queremos modelizar un sistema para calcular clasificaciones deportivas.

Damos por sentado que las clasificaciones son el resultado de una serie de enfrentamientos, que serán los partidos.

El resultado de un partido de baloncesto es estructuralmente diferente que el de uno de fútbol. Mientras que en el fútbol puede haber empate en el baloncesto no. Además, en el fútbol puede que la victoria valga 3 puntos o que valga 2, según las reglas de cada campeonato.

Herencia III

Primera aproximación de la clase **Partido**

```python
class Partido:
    def __init__(self, fecha, lugar, contendiente1, contendiente2):
        self.__fecha = fecha  # en UTC
        self.__lugar = lugar
        self._contendiente1 = contendiente1
        self._contendiente2 = contendiente2
        self.resultado = None

    def obtener_fecha(self, tz):
        # deberíamos tener en cuenta la tz pedida para devolver la fecha
        return self.__fecha

    def obtener_lugar(self):
        return self.__lugar

    def obtener_contendiente(self, local):
        if local:
            return self._contendiente1
        return self._contendiente2

    def puntos_contendiente(self, nombre_contendiente):
        ...
```

Herencia IV

Según el tipo de partido, tendremos que dar unos puntos diferentes.

Así que createmos clases hijas según tipo de partido. Habrá una clase para fútbol, otra para baloncesto, y así para todos los deportes.

¿Por qué?

- Ya hemos dicho que hay deportes que admiten el empate y otros que no
- Los resultados, que en este caso es un string, tienen representaciones distintas según el deporte, así que haremos que cada clase hija sepa interpretar su resultado. 3-1 en fútbol. 90-87 en baloncesto. 6-1 6-3 en tenis...

Herencia V

La sintaxis de python para el uso de la herencia requiere que la clase hija, en su declaración, indique que “hereda” de la clase padre.

```
class PartidoFutbol(Partido):
```

Además existe la palabra reservada super, que nos permite invocar métodos de la clase padre si existe un método con el mismo nombre en la hija.

```
super().__init__(fecha, lugar, contendiente1, contendiente2)
```

El diseño completo de nuestro ejemplo con partidos lo podéis encontrar en el código adjunto en la carpeta **herencia**

Poliformismo

Igual que la encapsulación está íntimamente ligada a la abstracción, el polimorfismo está igualmente ligado a la herencia.

En el anterior ejemplo de herencia usamos poliformismo al desarrollar un sistema en el que partido puede tener muchas formas. Además, nuestra solución respeta un principio fundamental del desarrollo de software, y es que nuestro código esté abierto a la extensión pero cerrado a la modificación.

(Si queremos hacer que nuestro sistema admita más tipos de deportes solamente tendremos que añadir un tipo distinto de partido sin tocar nada más)