UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 2 de Prácticas

Python Orientado a Objetos

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2025-2026

Última modificación:
12 de septiembre de 2025

Sesión 2: Python Orientado a Objetos

Índice

2.1 Aspectos esenciales de Programación Orientada a Objetos en Python

La Programación Orientada a Objetos es una extensión de la Programación Estructurada y Modular (encargada de dividir programas complejos en módulos que se integran unos con otros). Los dos primeros conceptos más básicos son:

- Clase. Es una plantilla que encapsula atributos y métodos.
  - Atributo: es una variable que puede ser de cualquier tipo, incluidas otras clases.
  - Método: es el concepto de función en programación modular pero que se define dentro de una clase.
- Objeto. Un caso particular de la clase (o plantilla). También se llama instancia de la clase.
  Los atributos o variables con valores concretos representan las propiedades de un objeto (su estado) y los métodos definen su comportamiento (lo que es capaz de hacer u operaciones que realiza).

2.1.1. Componentes principales de una clase

Para definir una clase se necesitan definir los siguientes aspectos:

- Los atributos: son las variables que definirán los estados de los objetos. Pueden diferenciarse entre atributos de instancia y atributos de clase.
- Los métodos: representan el comportamiento de los objetos, asociados con acciones que puede realizar un objeto. Pueden tener diferentes objetivos: consultar el estado de los atributos de un objeto, modificar dichos atributos, o realizar operaciones haciendo uso de los atributos. Además, los métodos pueden ser de tres tipos: de instancia, de clase, o estáticos.
- El constructor: se puede entender como un método especial que permite establecer el estado inicial de los objetos cuando se construyen por primera vez.

En las siguientes secciones se profundiza sobre cada uno de estos componentes, ofreciendo ejemplos para comprender mejor su notación y funcionamiento.

2.1.2. Atributos

La mejor forma de ilustrar la definición de los dos tipos de atributos existentes en Python es mediante un ejemplo de código, que iremos incrementando en cada una de las secciones siguientes:

```
class Persona:
    # Definir los atributos permitidos mediante __slots__
    __slots__ = ['_nombre', '_edad', '_altura', '_activo', '_hobbies']

    especie: str = "Humano"            # Atributo de clase de tipo string

    def __init__(self, nombre: str, edad: int, altura: float, activo: bool, hobbies: list[str]):
        self._nombre: str = nombre     # Atributo de instancia de tipo string
        self._edad: int = edad         # Atributo de instancia de tipo entero
        self._altura: float = altura   # Atributo de instancia de tipo real
        self._activo: bool = activo    # Atributo de instancia de tipo booleano
        self._hobbies: list[str] = hobbies  # Atributo de instancia de tipo lista (compuesto)
```

El código anterior incluye una serie de atributos de instancia de diferentes tipos de datos, tanto primitivos como compuestos, así como el uso de su constructor __init__ capaz de darles un valor de inicialización. Como podemos ver, todos los nombres de atributos de instancia anteponen un guión bajo (_) para indicar que son privados (veremos más detalles en la siguiente sesión).

Por otro lado, se define un atributo de clase, especie, que define información de la clase en su conjunto. Esto es, todas las personas tienen en común que son humanas, independientemente de las características individuales que definan a cada persona (atributos de instancia), como son su nombre o su altura.

Es interesante el uso de __slots__ para especificar el listado de atributos de instancia que definen una clase dada. Así, será imposible definir nuevos atributos desde fuera de la clase, lo que también nos ayuda a evitar errores en caso de indicar nombres de atributos incorrectos.

2.1.3. Métodos

Como se comentaba anteriormente, disponemos de tres tipos de métodos diferentes en Python: de instancia, de clase y estáticos. Además, tenemos otra categoría denominada métodos mágicos que veremos a continuación. Para comprender los métodos, ampliemos la clase Persona definida anteriormente para que tenga tres métodos:

```
class Persona:
    # Definir los atributos de instancia permitidos mediante __slots__
    __slots__ = ['_nombre', '_edad', '_altura', '_activo', '_hobbies']

    especie: str = "Humano"            # Atributo de clase de tipo string

    def __init__(self, nombre: str, edad: int, altura: float, activo: bool, hobbies: list):
        self._nombre: str = nombre     # Atributo de instancia de tipo string
        self._edad: int = edad         # Atributo de instancia de tipo entero
        self._altura: float = altura   # Atributo de instancia de tipo real
        self._activo: bool = activo    # Atributo de instancia de tipo booleano
        self._hobbies: list = hobbies  # Atributo de instancia de tipo lista (compuesto)

    # Método de instancia
    def obtener_info(self) -> str:
        return f"Nombre: {self._nombre}, Edad: {self._edad}, Altura: {self._altura}, Activo: {self._activo}"

    # Método de clase
    @classmethod
    def obtener_especie(cls) -> str:
        return f"La especie es {cls.especie}"

    # Método estático
    @staticmethod
    def es_mayor_edad(edad: int) -> bool:
        return edad >= 18
```

En primer lugar, el método de instancia obtener_info devuelve un string representando, de forma textual, una descripción del estado actual de la persona. Retorna información sobre todos los atributos de instancia salvo el listado de hobbies (por simplicidad en el ejemplo). Como podemos ver, este método recibe como único parámetro self, que es obligatorio para todo método de instancia, y representa al propio objeto que invocará a la función. Si el método requiriese más parámetros, se colocarían a la derecha de self.

En relación al método de clase, retornará una representación textual de la especie asociada a todas las personas. Es importante destacar que en los métodos de clase se debe indicar como primer parámetro cls, que representa a la propia clase sobre la que se invocará el método. Además, debemos colocar el decorador @classmethod sobre la definición de la función.

En cuanto al método estático, es capaz de determinar si, dado un valor de edad, se es mayor de edad o no. En este caso, se deberá usar el decorador @staticmethod. Estos métodos son independientes de cualquier variable de clase o de instancia

y, de hecho, no podrán hacer uso de ellos. Son de utilidad para crear métodos auxiliares que implementan funcionalidad de apoyo a la clase.

Finalmente, un método mágico en Python es un método especial que tiene una sintaxis definida por el lenguaje y que permite a los programadores personalizar el comportamiento de las clases de manera implícita. Estos métodos son útiles para integrar los objetos personalizados en el ecosistema del lenguaje, mejorando la legibilidad y la funcionalidad del código. Un método mágico en Python siempre comienza y termina con dos guiones bajos (__). Además, Python invoca estos métodos automáticamente en circunstancias particulares.

Veamos un listado de los métodos mágicos más comúnmente usados:

- **__init__(self, ...)**: Se ejecuta al crear una instancia de una clase; es el constructor que inicializa el estado del objeto.
- **__str__(self)**: Define la representación legible para humanos de un objeto, utilizada por `print()` y `str()`.
- **__len__(self)**: Define el comportamiento de `len()` aplicado a una instancia de la clase.
- **__add__(self, other)**, **__sub__(self, other)**, **__mul__(self, other)**, **__truediv__(self, other)**, **__floordiv__(self, other)**, **__mod__(self, other)**, **__pow__(self, other)**: Sobrecargan operadores aritméticos (+, -, *, /, //, %, **).
- **__eq__(self, other)**, **__ne__(self, other)**, **__lt__(self, other)**, **__le__(self, other)**, **__gt__(self, other)**, **__ge__(self, other)**: Sobrecargan los operadores de comparación (==, !=, <, <=, >, >=).

Para comprender mejor su funcionamiento, extenderemos el ejemplo para incorporar dos métodos mágicos: __str__ y __gt__.

```
class Persona:
    # Definir los atributos de instancia permitidos mediante __slots__
    __slots__ = ['_nombre', '_edad', '_altura', '_activo', '_hobbies']

    especie: str = "Humano"            # Atributo de clase de tipo string

    def __init__(self, nombre: str, edad: int, altura: float, activo: bool, hobbies: list):
        self._nombre: str = nombre     # Atributo de instancia de tipo string
        self._edad: int = edad         # Atributo de instancia de tipo entero
        self._altura: float = altura   # Atributo de instancia de tipo real
        self._activo: bool = activo    # Atributo de instancia de tipo booleano
        self._hobbies: list = hobbies  # Atributo de instancia de tipo lista (compuesto)

    # Método de instancia
    def obtener_info(self) -> str:
        return f"Nombre: {self._nombre}, Edad: {self._edad}, Altura: {self._altura}, Activo: {self._activo}"

    # Método de clase
    @classmethod
    def obtener_especie(cls) -> str:
        return f"La especie es {cls.especie}"

    # Método estático
    @staticmethod
    def es_mayor_edad(edad: int) -> bool:
        return edad >= 18

    # Método mágico __str__
    def __str__(self) -> str:
        return f"Persona(nombre={self._nombre}, edad={self._edad}, altura={self._altura}, activo={self._activo})"

    # Método mágico __gt__ para comparar la edad de dos personas
    def __gt__(self, otra_persona: 'Persona') -> bool:
        return self._edad > otra_persona._edad
```

El primero mostrará una representación textual, similar al método obtener_info que ya teníamos definido. Así, definiendo el método mágico no sería necesario mantener la función antigua, aunque la dejamos como ejemplo de método de instancia. Por su parte, __gt__ comprobará si la persona actual (objeto que invoca al método) tiene una edad superior a otra persona que se le pase por parámetro.

Es importante destacar que a __gt__ se le proporciona un parámetro de la clase Persona. En este caso, el tipo del parámetro se debe especificar entre comillas. Esto se utiliza para indicar que el tipo es la misma clase que se está definiendo, pero que todavía no está disponible en ese momento (todavía no está completamente definida pues estamos en proceso de ello). Esto no es algo exclusivo de los métodos mágicos, si no de cualquier definición de parámetros en métodos con POO.

Al margen de los métodos mágicos más comunes, es importante destacar dos métodos mágicos que no deben usarse:

- **__new__(cls, ...)**: Es el método creador. No debe usarse nunca, ya que su uso lo gestiona internamente Python al crear variables de una determinada clase.
- **__del__(self)**: Se invoca cuando un objeto va a ser destruido (destructor). No es recomendable su uso, ya que Python cuenta con un mecanismo de recolección de basura que se encarga de eliminar automáticamente las variables en desuso. Puede ser útil en situaciones muy concretas, como liberar recursos externos (archivos, conexiones de red, etc.) cuando un objeto es destruido.

Por otro lado, Python proporciona muchos otros métodos mágicos adicionales, aunque no son relevantes en el contexto de la asignatura. Por ejemplo, proporciona métodos mágicos para la gestión de colecciones, definiendo cómo se debe retornar elementos de una colección (**__getitem__(self, key)**, **__setitem__(self, key, value)**, **__delitem__(self, key)**), cómo iterar sobre ellos (**__iter__(self)**, **__next__(self)**), o cómo consultar si contienen un elemento (**__contains__(self, item)**), entre otros. Pueden consultarse en los siguientes enlaces:

- A Guide to Python's Magic Methods: https://rszalski.github.io/magicmethods/
- Lista de métodos mágicos asociados a cada operación: https://docs.python.org/3/library/operator.html?highlight=operations

2.1.4. Creación y uso de objetos

Esta sección describe cómo hacer uso de objetos en el __main__, donde se ilustra la definición e inicialización de objetos de la clase Persona y se invocan los métodos previamente definidos.

```
class Persona:
    # Definir los atributos de instancia permitidos mediante __slots__
    __slots__ = ['_nombre', '_edad', '_altura', '_activo', '_hobbies']

    especie: str = "Humano"            # Atributo de clase de tipo string

    def __init__(self, nombre: str, edad: int, altura: float, activo: bool, hobbies: list):
        self._nombre: str = nombre     # Atributo de instancia de tipo string
        self._edad: int = edad         # Atributo de instancia de tipo entero
        self._altura: float = altura   # Atributo de instancia de tipo real
        self._activo: bool = activo    # Atributo de instancia de tipo booleano
        self._hobbies: list = hobbies  # Atributo de instancia de tipo lista (compuesto)

    # Método de instancia
    def obtener_info(self) -> str:
        return f"Nombre: {self._nombre}, Edad: {self._edad}, Altura: {self._altura}, Activo: {self._activo}"

    # Método de clase
    @classmethod
    def obtener_especie(cls) -> str:
        return f"La especie es {cls.especie}"

    # Método estático
    @staticmethod
    def es_mayor_edad(edad: int) -> bool:
        return edad >= 18

    # Método mágico __str__
    def __str__(self) -> str:
        return f"Persona(nombre={self._nombre}, edad={self._edad}, altura={self._altura}, activo={self._activo})"

    # Método mágico __gt__ para comparar la edad de dos personas
    def __gt__(self, otra_persona: 'Persona') -> bool:
        return self._edad > otra_persona._edad

if __name__ == "__main__":
    # Crear dos instancias de Persona
    persona1: Persona = Persona("Ana", 25, 1.68, True, ['leer', 'escribir'])
    persona2: Persona = Persona("Juan", 30, 1.75, True, ['deporte', 'viajar'])

    # Usar obtener_info (método de instancia)
    print(persona1.obtener_info())  # Salida: Nombre: Ana, Edad: 25, Altura: 1.68, Activo: True
    print(persona2.obtener_info())  # Salida: Nombre: Juan, Edad: 30, Altura: 1.75, Activo: True

    # Usar __str__ para mostrar la información de las personas
    print(persona1)  # Salida: Persona(nombre=Ana, edad=25, altura=1.68, activo=True)
    print(persona2)  # Salida: Persona(nombre=Juan, edad=30, altura=1.75, activo=True)

    # Usar obtener_especie (método de clase)
    print(Persona.obtener_especie())  # Salida: La especie es Humano

    # Usar es_mayor_edad (método estático)
    print(Persona.es_mayor_edad(25))  # Salida: True (porque 25 >= 18)
    print(Persona.es_mayor_edad(17))  # Salida: False (porque 17 < 18)

    # Usar __gt__ para comparar las edades de persona1 y persona2
    if persona1 > persona2:
        print(f"{persona1._nombre} es mayor que {persona2._nombre}")
    else:
        print(f"{persona2._nombre} es mayor que {persona1._nombre}")  # Salida: Juan es mayor que Ana
```

Así, creamos dos objetos de la clase Persona para, posteriormente, obtener su información haciendo uso de los métodos obtener_info y __str__ (este último se invocará implícitamente al usar `print`(). Posteriormente, se usa la función estática para comprobar si dos edades diferentes corresponden o no a la mayoría de edad. Finalmente, usamos el operador $>$ entre objetos (implícitamente realiza una llamada a __gt__) para comparar la edad de dos personas.

Es importante destacar que en el caso de los métodos de clase y estático hemos invocado a las funciones sobre el nombre de la clase en lugar de sobre el objeto persona. En el caso del método de clase podríamos haberla invocado sobre el objeto persona sin problemas, mientras que para el método estático es obligatorio hacerlo sobre el nombre de la clase.

Para facilitar la revisión del código de ejemplo descrito en las sesiones de prácticas, tenéis a vuestra disposición en GitHub (`https://github.com/serloop/TP-Practicas-SesionesGuiadas`) el código final de cada una de las sesiones. El código se ha clasificado en diferentes *releases* (o versiones), cada una correspondiente a una sesión, y podéis descargar el código en formato *.zip* para su estudio y ejecución.

2.2 Relación de ejercicios

Aunque podríamos realizar todos los ejercicios de forma incremental en un mismo fichero, es recomendable tener un fichero .py por cada ejercicio realizado, de forma independiente (tendremos una solución completa por cada ejercicio). Así, cuando vayamos a comenzar con un nuevo ejercicio, copiaremos y pegaremos el contenido del ejercicio anterior en un nuevo fichero y comenzaremos a trabajar en el nuevo enunciado.

1. **Creación de clases**. Crea una clase en Python que represente un **Polígono**, dentro de un fichero llamado clases_geometria.py. En este primer ejercicio, el constructor (método __init__) **no recibirá ningún parámetro de inicialización** (salvo el self). La clase debe contener los siguientes atributos de instancia:
   - **numero_lados**: de tipo entero, y que representa el número de lados que tiene el polígono. Se establecerá a 3.
   - **color**: de tipo string, y que representa el color del polígono. Se establece a "BLANCO".
   - **forma**: este atributo será de tipo enumerado (de string) llamado TipoForma (recuerda que para ello se crea una clase *class TipoForma(Enum)*). El enumerado TipoForma tendrá tres opciones posibles (CONVEXO, CONCAVO o COMPLEJO) cuyos valore será 1, 2 o 3, respectivamente. El atributo se establecerá a CONVEXO en el momento de la construcción del objeto.
   - **relación_lados**: este atributo será de tipo enumerado (de string) llamado TipoRelacionLados (para ello, definir *class TipoRelacionLados(Enum)*). Sólo lo podría tener uno de los siguientes valores: REGULAR o IRREGULAR (como en el caso anterior, cada opción tendrá un entero asociado). El atributo se establecerá a REGULAR.

   Crea un objeto de dicha clase, llamado **mi_poligono**, y muestra por pantalla sus cuatro atributos desde el main, accediendo directamente a los atributos de la instancia (*RECUERDA: ¡esto último no es una buena práctica, por eso PyCharm lanza advertencias!*).

2. **Inicialización de la clase**. Modificar el constructor (método __init__) de la clase para que reciba como parámetros obligatorios los valores de **forma** y **relación de lados**. Añade el **número de lados** como parámetro opcional, con valor 3 por defecto. E incluye otro nuevo parámetro **color** que sea opcional, con valor "BLANCO" (string) por defecto.

   A consecuencia de estas modificaciones, ajusta la creación del objeto **mi_poligono** en el main adecuadamente para que la instancia se cree como un polígono CONVEXO y REGULAR, pero sin establecer ni el número de lados ni el color.

3. **Métodos adicionales**. Añadir los siguientes métodos de instancia a la clase Polígono:
   - Un método que permita obtener el número de lados del polígono.
   - Un método que permita obtener el color del polígono.
   - Un método que permita obtener la forma del polígono.
   - Un método que permita obtener la relación entre los lados del polígono.
   - Un método llamado **establecer_numero_lados** que permita **modificar el número de lados del polígono**. El método recibirá un parámetro con los lados, que debería ser mayor o igual que 3, devolviendo la cadena "*OK, se han establecido X lados en el polígono*", siendo *X* el parámetro introducido. Si es menor que 3, el método devolverá la cadena de texto "*ERROR: el número de lados de un polígono no puede ser menor de 3*".
   - Implementar el método mágico __str__ en la clase Polígono, que permita mostrar por pantalla el estado del objeto. Es decir, el valor de todos sus atributos en el momento de su invocación. Este método debe utilizar los cuatro métodos anteriores para evitar acceder directamente a los atributos de la clase.

   Tras la creación del objeto **mi_poligono**, establece ahora que el polígono definido tiene 4 lados a través del primer método creado. Evita ahora el acceso directo a los atributos haciendo uso de los métodos de consulta de atributos

implementados. (*Observa cómo ahora el programa principal accede a los atributos a través de métodos y no por la notación punto, así que PyCharm deja de lanzar avisos.*). Por último, muestra el objeto directamente haciendo uso de print(mi_poligono). ¿Qué sucede y por qué?

4. **Creación de atributos y métodos de clase**. Vamos a crear un atributo en la clase que permita obtener el número de objetos de tipo Polígono que existen actualmente creados en el programa. Para ello, añadir el **atributo de clase** llamado **numero_poligonos** que se inicialice a 0. Hay que realizar las modificaciones que consideres necesarias, para que cada vez que se cree un objeto Polígono, se incremente ese atributo de clase. *¿Cómo se haría?*
   Tras visualizar los datos del objeto **mi_poligono**, mostrar por pantalla el número de objetos Polígono actualmente creados. Para ello, puedes crear un **método de clase** encargado de devolver el número de polígonos creados hasta ahora.

5. **Creación de la clase Punto**. Crear una clase dentro del fichero que represente un **Punto**, definida por los siguientes atributos:

   - **coordenada_x**, con valor de tipo float.
   - **coordenada_y**, con valor de tipo float.

   El constructor (método __init__) de la clase **Punto** debe permitir ambas coordenadas como parámetro. Adicionalmente, habría que crear los siguientes métodos:

   - **obtener_coordenada_x**. Devuelve la coordenada x del punto.
   - **obtener_coordenada_y**. Devuelve la coordenada y del punto.
   - **obtener_cuadrante**. Devuelve un entero indicando el cuadrante en el que se encuentra el punto:
     - Devuelve 1 si está en el primer cuadrante (x e y positivos).
     - Devuelve 2 si está en el segundo cuadrante (x negativo e y positivo).
     - Devuelve 3 si está en el tercer cuadrante (x e y negativos).
     - Devuelve 4 si está en el cuarto cuadrante (x positivo e y negativo).
     - Devuelve 0 si el punto está en uno de los ejes X o Y.

   En el main, crea los cuatro siguientes objetos / instancias de la clase Punto, y muestra el cuadrante al que pertenece cada uno:

   - Un punto A con las coordenadas (-2,2)
   - Un punto B con las coordenadas (2,2)
   - Un punto C con las coordenadas (-2,-2)
   - Un punto D con las coordenadas (2,-2)

6. **Creación de la clase Línea**. Crear una clase dentro del fichero que represente una **Línea**. La clase debe contener dos atributos:

   - **punto_inicio**, que es de la clase Punto, y representa el punto de inicio de la línea.
   - **punto_fin**, que es de la clase Punto, y representa el punto de fin de la línea.

   El constructor (método __init__) de la clase **Linea** debe recibir los dos puntos que definen la línea. Adicionalmente, habría que crear los métodos siguientes:

   - **obtener_punto_inicio**. Obtiene el punto de inicio de la línea.
   - **obtener_punto_fin**. Obtiene el punto de fin de la línea.
   - **calcular_longitud**. Obtiene la longitud de la línea. Para calcularla, se puede utilizar la fórmula de distancia de Manhattan: d = $|x2 - x1| + |y2 - y1|$
   - **mostrar_puntos**. Este método muestra por pantalla las coordenadas del punto de inicio de la línea y del punto de fin.

   En el main, crea ahora 4 objetos de la clase línea, que conformarán en sí un cuadrado. Mostrar por pantalla las coordenadas de los puntos de inicio y fin y la longitud de cada una de ellas:

   - **linea_1**, con punto de inicio el punto A y punto de fin el punto B.
   - **linea_2**, con punto de inicio el punto B y punto de fin el punto D.
   - **linea_3**, con punto de inicio el punto D y punto de fin el punto C.
   - **linea_4**, con punto de inicio el punto C y punto de fin el punto A.

7. **Ampliación de la clase Polígono**. Vamos ahora a ampliar la definición y los métodos con los que cuenta nuestra clase Polígono. En primer lugar, la clase debería contar con los siguientes nuevos atributos:

   - **lados**, una lista de elementos de tipo Linea, y que contendrá cada una de las líneas que sirve de lado del polígono.

   También necesitaremos añadir los siguientes nuevos métodos:

   - **calcular_perimetro**, que obtiene el perímetro.
   - **establecer_lados**, que recibirá como parámetro una lista de Líneas que representan los nuevos lados del polígono. Debe comprobar que el número de líneas introducidos equivale al número de lados, retornando True en caso de que se pueda establecer la lista de líneas, y False en caso contrario.

   En el main, añade al objeto **mi_poligono** una **lista con las cuatro líneas** creadas en el ejercicio 6, y actualiza el método mágico __str__ para que también muestre el **perímetro** del polígono.

8. **Ampliación de la clase Polígono con métodos mágicos**. En este ejercicio, vamos a seguir trabajando con la clase **Polígono**, añadiendo algunos métodos mágicos adicionales para extender su funcionalidad:

   - **__eq__(self, otro)**. Comprueba si dos polígonos tienen el mismo número de lados.
   - **__add__(self, otro)**. Suma los perímetros de dos polígonos y devuelve el resultado.
   - **__lt__(self, otro)**. Compara si un polígono tiene un perímetro menor que otro.
   - **__len__(self)**. Devuelve el número de lados del polígono.

   Tras implementar los métodos mágicos, haz uso de ellos en la función __main__ para comprobar que se han implementado correctamente.
   Finalmente, muestra por pantalla el número de polígonos creados actualmente. Debería indicar que tenemos 2.