UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 6 de Prácticas

Clases Abstractas

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2025-2026

Última modificación:
14 de octubre de 2025

Sesión 6: Clases Abstractas

Índice

6.1 Definición de una clase abstracta

En Python, las clases abstractas proporcionan una forma de definir una estructura o comportamiento común que otras clases pueden heredar. Una clase abstracta no se puede instanciar directamente; en su lugar, se utiliza como una plantilla que garantiza que las subclases implementen ciertos métodos necesarios. Las clases abstractas en Python se manejan a través del módulo `abc` (Abstract Base Classes).

Para definir una clase abstracta, dicha clase debe heredar de `ABC` y marcar los métodos que deben ser implementados por las subclases con el decorador `@abstractmethod`, de forma similar a como se gestionan los métodos estáticos y de clase. De esta manera, las subclases están obligadas a proporcionar una implementación de estos métodos abstractos. Esto garantiza que las subclases sigan el comportamiento esperado.

Para ilustrar este concepto, consideremos el caso de una clase abstracta llamada `Vehiculo`. Un `Vehiculo` tiene atributos comunes como `marca` y `modelo`, pero no tiene sentido instanciar un vehículo sin saber qué tipo de vehículo es, por ejemplo, un coche o una moto. Por lo tanto, podemos definir una clase abstracta `Vehiculo`, donde algunos métodos, como `arrancar()`, no tienen una implementación predeterminada y deben ser implementados por las subclases, como `Coche` y `Moto`.

Aquí tienes un ejemplo básico de la definición de la clase abstracta `Vehiculo` con métodos abstractos:

```
from abc import ABC, abstractmethod

# Definición de la clase abstracta Vehiculo
class Vehiculo(ABC):

    @abstractmethod
    def arrancar(self) -> None:
        pass

    @abstractmethod
    def detener(self) -> None:
        pass
```

En este ejemplo, hemos definido la clase `Vehiculo` con dos métodos abstractos: `arrancar()` y `detener()`. Estos métodos no tienen una implementación concreta, ya que cada subclase (como `Coche` o `Moto`) deberá proporcionar su propia implementación.

6.2 Constructor en una clase abstracta

Las clases abstractas pueden incluir constructores que permitan inicializar atributos comunes en las subclases. Esto es útil cuando hay propiedades compartidas entre todas las subclases que necesitan ser inicializadas de manera uniforme. En nuestro ejemplo, tanto los coches como las motos tienen una `marca` y un `modelo`, por lo que podemos inicializar estos atributos en la clase abstracta `Vehiculo`. Sin embargo, es esencial destacar que **usar el constructor de una clase abstracta no crea objetos de esa clase**. Se usa simplemente para la **inicialización de atributos comunes a sus subclases.**

Veamos cómo añadir un constructor a la clase `Vehiculo`:

```python
from abc import ABC, abstractmethod

class Vehiculo(ABC):

    def __init__(self, marca: str, modelo: str) -> None:
        self._marca: str = marca
        self._modelo: str = modelo

    # (...) Obviamos métodos get/set

    @abstractmethod
    def arrancar(self) -> None:
        pass

    @abstractmethod
    def detener(self) -> None:
        pass
```

Aquí hemos añadido un constructor a la clase abstracta `Vehiculo`, que inicializa los atributos `marca` y `modelo`. Aunque no podemos instanciar directamente un `Vehiculo`, las subclases que heredan de `Vehiculo` podrán llamar a este constructor para inicializar estos atributos comunes.

Ahora definamos las subclases `Coche` y `Moto`, que heredan de `Vehiculo` y proporcionan sus propias implementaciones de los métodos abstractos. Además, estas clases definen sus propios atributos que definen características propias a cada tipo de vehículo, como el número de puertas en el caso de un coche, o el tipo en el caso de una moto.

```python
class Coche(Vehiculo):

    def __init__(self, marca: str, modelo: str, numero_puertas: int) -> None:
        super().__init__(marca, modelo)
        self._numero_puertas: int = numero_puertas

    # (...) Obviamos métodos get/set

    def arrancar(self) -> None:
        print(f"El coche {self.get_marca()} {self.get_modelo()} con {self.get_numero_puertas()} puertas
            está arrancando.")

    def detener(self) -> None:
        print(f"El coche {self.get_marca()} {self.get_modelo()} con {self.get_numero_puertas()} puertas
            se ha detenido.")


class Moto(Vehiculo):

    def __init__(self, marca: str, modelo: str, tipo: str) -> None:
        super().__init__(marca, modelo)
        self.tipo: str = tipo

    # (...) Obviamos métodos get/set

    def arrancar(self) -> None:
        print(f"La moto {self.get_marca()} {self.get_modelo()} de tipo {self.get_tipo()} está arrancando.")

    def detener(self) -> None:
        print(f"La moto {self.get_marca()} {self.get_modelo()} de tipo {self.get_tipo()} se ha detenido.")
```

Ahora las subclases `Coche` y `Moto` hacen uso del constructor de `Vehiculo` y proporcionan sus propias implementaciones de `arrancar()` y `detener()`.

6.3 Uso de métodos no abstractos

En algunas situaciones puede ser útil definir un método no abstracto en la clase abstracta. Esto es útil cuando todas las subclases deben tener una funcionalidad básica que puede ser reutilizada, pero que también puede ser sobrescrita si es necesario. Por tanto, podemos ver cómo no es obligatorio que todos los métodos de una clase abstracta sean abstractos. Así, una clase será abstracta si tiene al menos un método abstracto.

En el caso de `Vehiculo`, podríamos añadir un método `descripcion()` que proporcione una descripción básica del vehículo. Las subclases pueden heredar esta funcionalidad o sobrescribirla para proporcionar más detalles.

```python
class Vehiculo(ABC):
    # (...)

    def descripcion(self) -> str:
        return f"Vehículo marca {self.get_marca()}, modelo {self.get_modelo()}"

class Coche(Vehiculo):
    # (...)

    def descripcion(self) -> str:
```

```
        return super().descripcion() + f", con {self.get_numero_puertas()} puertas."

class Moto(Vehiculo):
    # (...)

    def descripcion(self) -> str:
        return super().descripcion() + f", tipo {self.get_tipo()}."
```

En este caso, hemos definido el método `descripcion()` con una implementación por defecto que proporciona una descripción básica del vehículo. Las subclases, como `Coche` o `Moto`, pueden heredar este método directamente, extenderlo o sobrescribirlo.

6.4 Ejemplo de clases abstractas: clases Persona y Libro

Esta sección continúa el proyecto de las sesiones de prácticas anteriores para ilustrar el uso de clases abstractas. En concreto, asumiremos que en nuestro proyecto concreto no tiene sentido crear objetos genéricos de la clase `Libro` y, por tanto, instanciaremos objetos de tipos de libros específicos (libros científicos, libros infantiles, etc). Además, el segundo motivo que justifica el definir esta clase como abstracta es la necesidad de definir métodos cuya funcionalidad sea determinada por cada tipo de libro de forma independiente. Veamos el código modificado de la clase `Libro`:

```
from abc import ABC, abstractmethod

class Libro(ABC):
    def __init__(self, titulo: str, autor: str, paginas: int, propietario: 'Persona' = None):
        self._titulo: str = titulo
        self._autor: str = autor
        self._paginas: int = paginas
        self._listado_lectores: list['Persona'] = []  # Lista de personas que han leído el libro
        self._propietario: 'Persona' = propietario  # Relación de agregación: un libro tiene un propietario

    # (...) Obviamos métodos get/set

    @abstractmethod
    def comprar_libro(self, propietario: 'Persona') -> bool:
        pass

    @abstractmethod
    def recomendar_libro(self, persona: 'Persona') -> bool:
        pass

    # (...) Método mágico __str__
```

Tras indicar que `Libro` hereda de `ABC`, es necesario identificar qué métodos van a ser abstractos. En este ejemplo, consideremos que el proceso de compra de un libro depende completamente de cada tipo de libro. Así, en el caso de `LibroCientifico`, había unos requisitos específicos para poder realizar su compra, en base al número de libros científicos leídos previamente. En el caso de `LibroInfantil`, simplemente consistía en cambiar el propietario del libro.

Además, definimos un método abstracto adicional denominado `recomendar_libro`, encargado de determinar si un tipo de libro es apto o no para recomendación a una persona concreta, en base a su perfil particular. De igual forma, consideramos que estas recomendaciones deben ser realizadas de forma específica por cada tipo de libro. En el caso del método **__str__**, mantenemos el mismo comportamiento de la sesión anterior: cada tipo de libro usa el método mágico de la clase padre con `super()`, extendiendo su funcionalidad para mostrar información específica de ese tipo de libro.

Finalmente, es importante destacar que, en este ejemplo, podríamos perfectamente no haber definido la clase `Libro` como abstracta, permitiendo que hubiese libros genéricos sin una categoría clara. En muchos casos, la decisión de si una clase debe ser o no abstracta vendrá motivada por el problema a resolver.

Mostramos a continuación los cambios realizados en la clase `LibroCientifico`:

```
class LibroCientifico(Libro):
    def __init__(self, titulo: str, autor: str, paginas: int, campo_estudio: str,
                 nivel_dificultad: str, propietario: 'Persona' = None):
        super().__init__(titulo, autor, paginas, propietario)

        # Nuevos atributos de la clase hija
        self._campo_estudio: str = campo_estudio
        self._nivel_dificultad: str = nivel_dificultad
        self._referencias: list[Libro] = []

    # (...) Los métodos get/set y __str__ permanecen igual

    # Implementación del método abstracto
    def comprar_libro(self, propietario: 'Persona') -> bool:
        # Verificar si la persona ha leído libros científicos con anterioridad.
        # Uso de isinstance() para verificar el tipo de libro
        libros_cientificos_leidos = []
        for libro in propietario.get_libros_leidos():
            if isinstance(libro, LibroCientifico):
```

```
                libros_cientificos_leidos.append(libro)

        if len(libros_cientificos_leidos) >> 2: # Tiene experiencia previa leyendo libros científicos
            self._set_propietario(propietario)
            return True
        else:
            return False

    # Implementación del método abstracto
    def recomendar_libro(self, persona: 'Persona') -> bool:
        if self.get_campo_estudio() in persona.get_hobbies():
            return True
        return False
```

Como podemos ver, no se realizan cambios en el constructor respecto a la sesión anterior. Usaremos la clase padre abstracta como plantilla para definir atributos, haciendo uso de `super()`. En cuanto al método `comprar_libro()`, el código no requiere cambios respecto al que ya teníamos usando herencia. Es importante destacar que la clase `Libro` contiene métodos que sirven de soporte a los métodos abstractos implementados por las subclases, como el método `_set_propiterio()`. En el caso del método `recomendar_libro()`, consideramos que solo tiene sentido recomendar un libro científico siempre que el campo de estudio del libro se encuentre entre los hobbies de la persona.

Pasemos a ver el código modificado de la clase `LibroInfantil`:

```
class LibroInfantil(Libro):

    def __init__(self, titulo: str, autor: str, paginas: int, edad_recomendada: int, ilustraciones: bool,
                 interactividad: bool, propietario: 'Persona' = None):
        super().__init__(titulo, autor, paginas, propietario) # Llamada al constructor de la clase padre

        # Nuevos atributos de la clase hija
        self._edad_recomendada = edad_recomendada # Edad recomendada de lectura
        self._ilustraciones = ilustraciones # El libro tiene ilustraciones
        self._interactividad = interactividad # El libro es interactivo

    # (...) Los métodos get/set y __str__ permanecen igual

    # Implementación del método abstracto
    def comprar_libro(self, propietario: 'Persona') -> bool:
        self._set_propietario(propietario)
        return True

    # Implementación del método abstracto
    def recomendar_libro(self, persona: 'Persona') -> bool:
        if persona.get_edad() >= self.get_edad_recomendada():
            return True
        return False
```

En este caso, la clase debe dar implementación a los dos métodos abstractos heredados. En el caso de `comprar_libro()`, simplemente implica asignar al libro un propietario. Sin embargo, si en el futuro la funcionalidad de este método debiese cambiar, podría adaptarse a nuevos requisitos sin problemas. En el caso de la recomendación de libros, consideramos que se debe recomendar siempre que la edad de la persona sea mayor o igual a la edad recomendada del libro.

Finalmente, si aplicamos estos cambios veremos que el `main` nos da un error, indicándonos que el diario que tiene `Persona` como atributo no puede ser de tipo `Libro`, ya que es abstracta. Para ello, definimos un nuevo tipo de libro: LibroDiario:

```
from datetime import datetime

class LibroDiario(Libro):
    def __init__(self, titulo: str, autor: str, paginas: int, propietario: 'Persona' = None):
        super().__init__(titulo, autor, paginas, propietario)
        self.fecha_creacion = datetime.now()
        self.completado: bool = False
        self.numero_entradas: int = 0

    # (...) Métodos get/set obviados por simplicidad

    def __str__(self) -> str:
        descripcion_basica = super().__str__()
        detalles_diario = (f"Fecha de creación: {self.fecha_creacion}, "
                           f"Completado: {'Sí' if self.completado else 'No'}, "
                           f"Número de entradas: {self.numero_entradas}")
        return f"{descripcion_basica} | {detalles_diario}"

    # Implementación del método abstracto
    def comprar_libro(self, propietario: 'Persona') -> bool:
        return True

    # Implementación del método abstracto
    def recomendar_libro(self, persona: 'Persona') -> bool:
        return True
```

Como podemos ver en el código, definimos algunos atributos adicionales a los definidos en la clase `Libro`, como son su fecha de creación, si se ha completado o no, y el número de entradas que contiene. Estas propiedades son ejemplos y seguramente se nos ocurran otras tantas diferentes para extender la clase. Sin embargo, es importante fijarse en cómo hemos implementado los dos métodos abstractos. En este caso de ejemplo, hemos determinado que ambos métodos devuelvan `True`. Esto es, que siempre podremos comprar un diario (realizando la asignación del propietario, de forma equivalente a lo visto en *LibroInfantil*) y siempre recomendaremos este tipo de libro. Así, simplemente basta con modificar la clase `Persona` para que su atributo diario pase a ser de este nuevo tipo de libro, sin tener que cambiar los parámetros con los que se llamaba al constructor.

6.5 Relación de ejercicios

Esta sesión de prácticas parte del código implementado en el último ejercicio de la Sesión 5. En esta versión deberíamos tener una jerarquía de clases similar a la mostrada en la Figura 6.1, así como atributos y métodos similares.

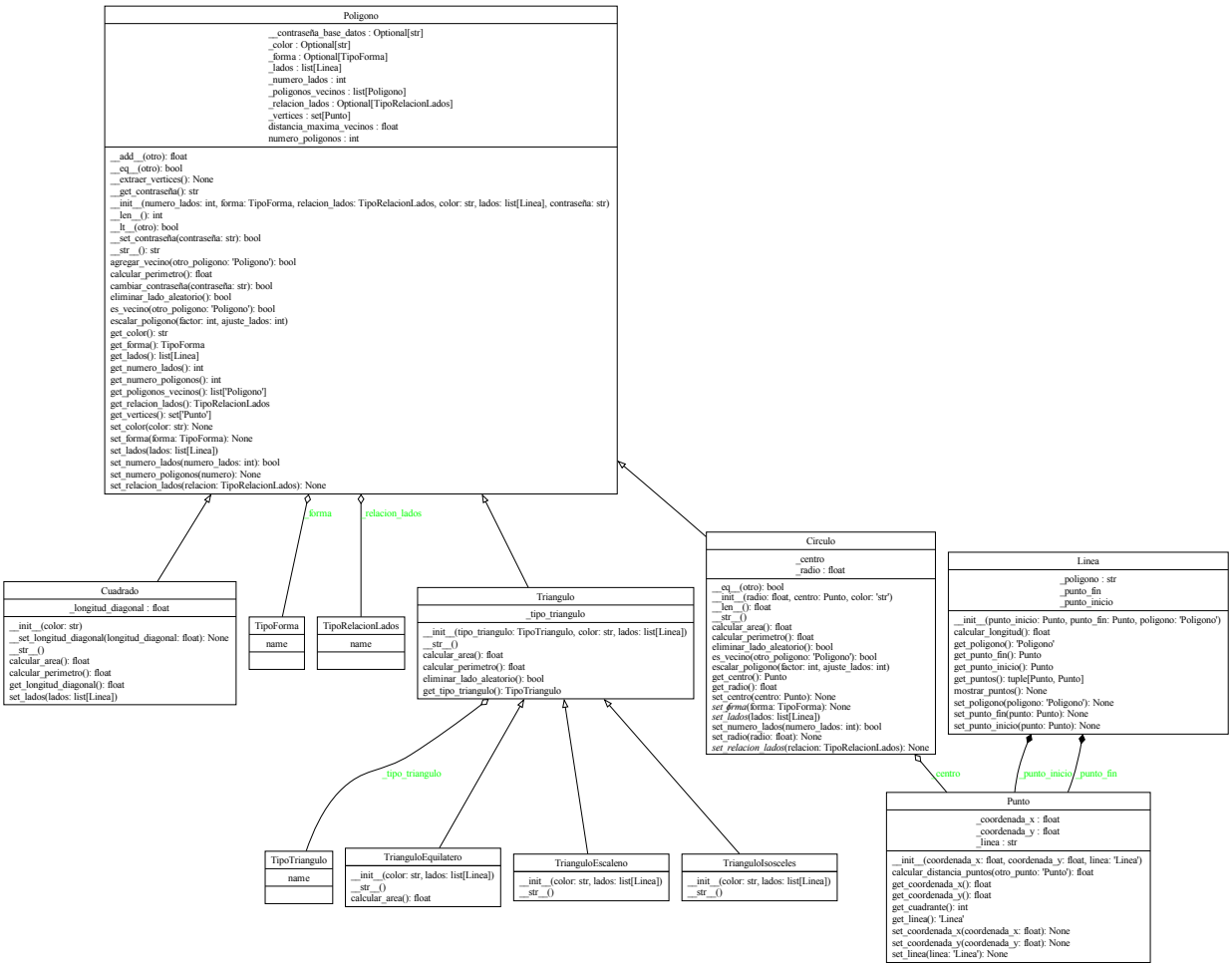

Figura 6.1: Diagrama UML representando el estado del proyecto tras la Sesión 5.

Como Círculo no tiene sentido que sea un polígono (lo definimos como un polígono con fines didácticos en la sesión anterior), eliminaremos el Círculo como clase hija. Así, modificaremos el método *es_vecino()* para que únicamente compruebe la distancia entre centros con otro círculo. Todos los métodos innecesarios que se heredaron de Polígono tendremos que borrarlos, para dejar un código limpio.

1. **Clase abstracta Poligono**. A partir de ahora asumimos que no es necesario crear objetos de tipo `Poligono`, pues tenemos cuadrados, triángulos y círculos con los que trabajaremos. Para ello, completa los siguientes apartados:
   - Convierte la clase `Poligono` a clase abstracta.
   - Convierte el método `calcular_area()` a método abstracto, de tal forma que cada tipo particular de polígono (triángulos y cuadrados) implementen su propia versión personalizada.
   - Haz las modificaciones oportunas en la función `main` para probar que los objetos creados de las subclases funcionan correctamente, y verifica que no podemos crear objetos de la clase `Poligono`.
   - ¿Sería conveniente convertir el método `calcular_perimetro()` a método abstracto?
2. **Clase abstracta Triangulo**. De igual forma a lo que pasaba anteriormente con `Poligono`, consideramos que no es conveniente en nuestro proyecto usar triángulos genéricos, pues los tres tipos de triángulos actuales (equiláteros, isósceles y escalenos) representan la totalidad de triángulos posibles.
   - Convierte la clase `Triangulo` a abstracta.
   - Implementa el método abstracto `calcular_altura()`, que es dependiente del tipo de triángulo.

- Triángulo equilátero: $h = \frac{\sqrt{3}}{2} \times L$, donde L es la longitud de uno de los lados.
- Triángulo isósceles: $h = \sqrt{a^2 - \left(\frac{b}{2}\right)^2}$, Donde a es la longitud de uno de los lados iguales y b es la longitud de la base.
- Triángulo escaleno: $h = \frac{2 \times A}{b}$, donde A es el área calculada en base a la fórmula de Herón.

- Haz los cambios pertinentes en el `main`.

3. **Clase FiguraCurva**. Este ejercicio define una nueva clase abstracta llamada `FiguraCurva`, que servirá como clase padre para las subclases `Circulo` (ya implementada en la sesión anterior) y `Elipse`, ambas figuras geométricas curvas. Deberás gestionar convenientemente a nivel de paquetes esta nueva familia de figuras geométricas.

   - Implementa la clase `FiguraCurva`, sabiendo que toda figura curva cuenta con centro de tipo `Punto` y un color. Define la visibilidad y los métodos get/set adecuados. Además, toda figura curva permite calcular su área y su perímetro. ¿El cálculo del área y del perímetro deben ser métodos abstractos?
   - Adapta la clase `Circulo` para que sea una subclase de FiguraCurva.
   - Implementa la clase `Elipse`, sabiendo que una elipse tiene dos atributos principales: el semieje mayor y el semieje menor. Los semiejes son segmentos que describen las dimensiones principales de la figura, dividiéndolas en dos mitades iguales. Por ejemplo, el semieje mayor es el segmento más largo de la elipse que atraviese su centro y conecta dos puntos opuestos en el borde de la elipse. Representaremos los semiejes como valores reales de distancia.