Lectura

Clases abstractas y decoradores de clase en Python

Python Intermedio

Introducción

La Programación Orientada a Objetos **(POO)** en Python es un paradigma, que permite modelar el comportamiento de sistemas complejos a través de clases, objetos y relaciones estructurales. Dentro de este enfoque, se introducen herramientas avanzadas que permiten crear jerarquías claras, componentes reutilizables y sistemas mantenibles. Este recurso se enfoca en dos pilares fundamentales: las **clases abstractas**, que definen comportamientos obligatorios en jerarquías de herencia, y los **decoradores de clase**, que amplían o modifican el comportamiento de métodos, sin alterar su implementación original. El dominio de estas herramientas permite crear soluciones robustas, flexibles y alineadas con principios de diseño, como la separación de responsabilidades, el principio de sustitución de Liskov o el uso de interfaces explícitas.

Clases abstractas

Una **clase abstracta** es una clase que **no puede ser instanciada directamente** y que **establece una interfaz base, para las clases derivadas**. Sirve como molde que define métodos, los cuales, deben ser implementados por cualquier clase “hija”. Su principal utilidad es **forzar un contrato de implementación**, asegurando que las clases que heredan, respeten una estructura funcional mínima.

En Python, la abstracción se implementa mediante el módulo estándar **abc (Abstract Base Classes)**, que proporciona herramientas para definir métodos abstractos usando el decorador @abstractmethod.

¿Por qué utilizar clases abstractas?

- **Diseño estructurado y mantenible:** Permite establecer expectativas claras respecto a cuáles métodos deben implementarse.
- **Reutilización parcial:** Una clase abstracta puede contener código funcional parcial, que se complementa en las subclases.
- **Control de jerarquías:** Facilita la validación de comportamiento común en jerarquías de herencia, sin depender solo de convenciones.
- **Simulación de interfaces:** Aunque Python no tiene un tipo *interface* como Java o C#, las clases abstractas permiten cubrir esta función.

La sintaxis general de una clase abstracta en Python es:

```python
from abc import ABC, abstractmethod

class ClaseBase(ABC):
    @abstractmethod
    def metodo_obligatorio(self):
        pass
```

En donde:

➔ ABC es una clase base especial, que indica cuál es una *ClaseBase* abstracta.

- ➔ *@abstractmethod* indica que este método **debe ser implementado** por cualquier subclase concreta.

A continuación se muestra un ejemplo completo de la **aplicación de clases abstractas:**

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, nombre):
        self.nombre = nombre

    @abstractmethod
    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        return "Guau"

class Gato(Animal):
    def hacer_sonido(self):
        return "Miau"
```

En el ejemplo anterior:

- ➔ *Animal* no puede instanciarse directamente: *a = Animal("algo")* lanzará un *TypeError*.
- ➔ Cualquier clase hija debe implementar *hacer_sonido()*.
- ➔ Las clases *Perro* y *Gato* cumplen el contrato y son válidas para instanciar.

Es importante mencionar que es obligatorio utilizar los métodos abstractos en clases que heredan a una clase abstracta, de lo contrario, se generará un error.

Métodos concretos y abstractos en una misma clase:

Las clases abstractas pueden incluir métodos implementados, que serán heredados por las subclases:

```python
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def area(self):
        pass

    def describir(self):
        return "Figura geométrica"
```

describir() está implementado y puede ser reutilizado sin redefinir.

Detección de instancias con *isinstance* e *issubclass:*

```python
issubclass(Perro, Animal)      # True
isinstance(Perro("Firulais"), Animal)  # True
```

Las clases hijas siguen siendo **subtipos válidos** del tipo base abstracto.

Buenas prácticas:

- ➔ Nombrar las clases abstractas con sufijos como *Base*, *Abstract* o *Interface*, para distinguirlas de las clases concretas.
- ➔ Evitar abusar de la herencia múltiple de clases abstractas no relacionadas.
- ➔ No mezclar métodos concretos y abstractos en la misma jerarquía, en caso de que no exista un diseño coherente.
- ➔ Usar docstrings en los métodos abstractos, para explicar el propósito esperado en su implementación.

Decoradores de clase

Un decorador es una función, que recibe otra función o método como argumento, y devuelve una versión modificada de esta misma, sin alterar su definición original. Son herramientas muy poderosas, que permiten aplicar comportamiento adicional a funciones o clases, de forma reutilizable y declarativa.

En programación orientada a objetos, los decoradores se utilizan sobre métodos para:

- Modificar su ejecución (por ejemplo, medir tiempo, validar parámetros).
- Convertir su tipo de comportamiento (método estático o de clase).
- Agregar lógica auxiliar sin cambiar su cuerpo.

Decoradores integrados en métodos de clase

Python proporciona decoradores especiales que se usan frecuentemente en clases:

@classmethod:

Este decorador **convierte un método, en método de clase**, es decir, lo asocia a la clase y no a la instancia. El primer parámetro que recibe es *cls*, una referencia a la propia clase. Se usa generalmente para:

- Cuando se necesita acceder o modificar atributos de clase.
- Para definir constructores alternativos (factory methods).

A continuación, se presenta un ejemplo de su uso:

```python
class Producto:
    iva = 0.16  # atributo de clase

    def __init__(self, precio):
        self.precio = precio

    @classmethod
    def cambiar_iva(cls, nuevo_valor):
        cls.iva = nuevo_valor

Producto.cambiar_iva(0.18)
print(Producto.iva)  # 0.18
```

@staticmethod:

Este decorador indica que el método **no accede** a la clase ni a la instancia. Se comporta como una función normal, pero vive dentro de una clase por razones organizativas.
Se usa cuando:

- ➔ El método no necesita ni *self* ni *cls*.
- ➔ Se desea mantener funciones auxiliares dentro del contexto lógico de una clase.

Un ejemplo de lo anterior es:

```python
class Matematica:
    @staticmethod
    def es_par(n):
        return n % 2 == 0

Matematica.es_par(4)  # True
```

Decoradores personalizados en clases

Además de los decoradores integrados, en Python puedes crear tus propios decoradores para modificar métodos de clase o de instancia.

Algunos ejemplos son:

```python
def registrar_llamada(func):
    def wrapper(*args, **kwargs):
        print(f"Llamando a: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

class Saludo:
    @registrar_llamada
    def decir_hola(self):
        print("¡Hola!")

class Saludo:
    @registrar_llamada
    def decir_hola(self):
        print("¡Hola!")
```

¿Cómo funciona internamente?

Cuando se escribe:

```
@decorador
def funcion():
    ...
```

Python lo transforma en:

```
funcion = decorador(funcion)
```

En el caso de métodos dentro de clases, el proceso es el mismo: **el decorador envuelve a la función, antes de ser asociada al objeto.**

Decoradores con parámetros

También es posible definir **decoradores parametrizados:**

```python
def repetir(n):
    def decorador(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorador

class Alerta:
    @repetir(3)
    def mensaje(self):
        print("Atención")

a = Alerta()
a.mensaje()
# Atención (impreso 3 veces)
```

Interacción con *@classmethod* y *@staticmethod*

Si se desea aplicar múltiples decoradores, el orden es importante. Primero se aplica el decorador más interno (el más cercano a la función), y luego los externos.

```
@staticmethod
@mi_decorador  # se aplica primero
def metodo():
    ...
```

Recomendación: evita mezclar decoradores, si no entiendes bien su orden de evaluación.

Buenas prácticas:

- ➔ Siempre usar *functools.wraps(func)* para preservar *__name__*, *__doc__*, *etc*.
- ➔ Documentar el propósito de los decoradores personalizados.
- ➔ No abusar de decoradores con demasiada lógica oculta.
- ➔ Utilizar decoradores cuando resuelvan un patrón repetido (ej. validación, logging, acceso).

Conclusión

Los decoradores de clase son una herramienta potente para modificar el comportamiento de los métodos, de forma flexible y reutilizable. Junto con *@classmethod* y *@staticmethod*, permiten adaptar el funcionamiento de los métodos al diseño lógico de la aplicación. Combinados con clases abstractas,

estos mecanismos ofrecen una base sólida para construir aplicaciones orientadas a objetos, bien estructuradas, mantenibles y extensibles.

Elaboró: Enrique Quezada Próspero
Contenido: Enrique Quezada Próspero
DI: Berenice Cedeño y Génesis Equihua

Referencias:

1. Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python* (2nd ed.). O'Reilly Media.

2. Hetland, M. L. (2017). *Python Algorithms: Mastering Basic Algorithms in the Python Language* (2nd ed.). Apress.

3. Python Software Foundation. (2023a). *abc — Abstract Base Classes*. Recuperado el 18 de agosto, de: https://docs.python.org/3/library/abc.html

4. Python Software Foundation. (2023b). *Built-in Functions: classmethod, staticmethod*. Recuperado el 18 de agosto, de: https://docs.python.org/3/library/functions.html

5. Python Software Foundation. (2023c). *Decorators in Python*. Recuperado el 18 de agosto, de: https://docs.python.org/3/glossary.html#term-decorator