PRO-UT2-A5. Interfaces y Clases abstractas

Interfaces

Definición y características

En la **programación orientada a objetos**, un interfaz define al **conjunto de métodos** que tiene que tener un objeto para que pueda cumplir una determinada función en nuestro sistema.

Dicho de otra manera, un interfaz define como se comporta un objeto y lo que se puede hacer con el, pero no implementa el código los métodos. La **interfaz** se centra en el **qué**, no en el **como**

Por ejemplo, para el caso de un mando a distancia de un televisor, todos los mandos nos ofrecen el mismo **interfaz** con las mismas funcionalidades (métodos). De forma resumida, lo **qué** hace un mando a distancia se podría resumir en:

- siguiente_canal
- canal_anterior
- subir_volumen
- bajar_volumen

Lo anterior es un listado de la funcionalidad `Mando`

Se dice que una determinada clase **implementa una interfaz** cuando añade código a los métodos que no lo tenían. Por tanto, **implementar una interfaz** consiste en pasar del **qué** al **cómo** se hace.

Siguiendo con el ejemplo anterior un `MandoLG` o un `MandoSamsung` implementan la interfaz `Mando` y cada uno lo hace internamente de formas diferentes.

Interfaces en Python

A diferencia de otros lenguajes de programación, Python no incluye la instrucción `Interfaz` para definir interfaces. A pesar de esto, existen dos formas de definir interfaces en Python:

- De manera informal
- Usando clases abstractas para crear interfaces formales

Interfaces informales en Python

Una interfaz informal puede ser definida con una simple clase que no implementa los métodos. El la forma en que se ha hecho para alguno de los ejercicios que ya hemos hecho. Por ejemplo:

```
class Mando:
    def siguiente_canal(self):
        pass
    def canal_anterior(self):
        pass
    def subir_volumen(self):
        pass
    def bajar_volumen(self):
        pass
```

Una vez definida la interfaz la podemos usar mediante herencia:

```
class MandoSamsung(Mando):
    def siguiente_canal(self):
        print("Samsung->Siguiente")
    def canal_anterior(self):
        print("Samsung->Anterior")
    def subir_volumen(self):
        print("Samsung->Subir")
    def bajar_volumen(self):
        print("Samsung->Bajar")

class MandoLG(Mando):
    def siguiente_canal(self):
        print("LG->Siguiente")
    def canal_anterior(self):
        print("LG->Anterior")
    def subir_volumen(self):
        print("LG->Subir")
    def bajar_volumen(self):
        print("LG->Bajar")
```

Acabamos de pasar del **qué** al **cómo**.

Esta solución es válida para la mayoría de casos, pero tiene el inconveniente de que las clases que heredan podrían dejar sin implementar alguno de los métodos de la interfaz informal. Si un método queda sin implementar, podríamos tener problemas en el futuro, ya que al llamar a dicho método no tendríamos código que ejecutar.

Interfaces formales usando clases abstractas

Los interfaces formales pueden ser definidos en Python utilizando el **módulo** por defecto llamado ABC (Abstract Base Classes). Este módulo nos permite crear un tipo de clase base especial: las clases **abstracta**.

Las **clases abstractas** permite definir una clase que puede heredarse en otras clases de forma que estas clases **hijas** tengan los métodos de la clase base.

Cuando definimos una clase como abstracta estamos definiendo la **interfaz** que **deben cumplir** las clases que hereden de la misma.

En general en la POO una clase abstracta cumple las siguientes condiciones:

- Puede declarar la **existencia** de métodos, pero **no es necesario que incluya el código** que implementa dichos métodos.
- Para que una clase sea abstracta debe poseer, **al menos, un método abstracto**.
- No es posible **instanciar** un objeto a partir de una clase abstracta.
- Las clases derivadas de las clases abstractas **deben implementar necesariamente** todos los métodos abstractos para poder crear una clase que se ajuste a la **interfaz** definida. En el caso de que no se defina alguno de los métodos obtendremos error al instanciar un objeto de dicha clase.

Python no provee por defecto clases abstractas, para poder usarlas Python provee el módulo **ABC** Abstract Base.

Para crear una clase abstracta debemos importar en módulo **ABC** y la clase abstracta que queramos declarar heredará de `ABC`

```
from abc import ABC

class Person(ABC)
```

Para indicar que un método de una clase es abstracto usamos el decorador `@abstracmethod`. Este decorador, forzará a las clases que implementen dicha interfaz/clase abstracta a implementarlo:

Nuestro `Mando` usando clases abstractas para declarar formalmente su interfaz quedaría:

```
from abc import ABC, abstractmethod

class Mando(ABC):
    @abstractmethod
    def siguiente_canal(self):
        pass

    @abstractmethod
    def canal_anterior(self):
        pass

    @abstractmethod
    def subir_volumen(self):
        pass

    @abstractmethod
    def bajar_volumen(self):
        pass
```

Si intentamos instanciar un objeto de una clase abstracta que contiene un método abstracto se genera un error

```
mando = Mando()
```

Obtenemos

```
TypeError: Can't instantiate abstract class Animal with abstract methods ...
```

No sucede lo mismo si instanciamos un objeto de una clase heredada de una clase abstracta que implementa todos los métodos abastractos:

```
class MandoSamsung(Mando):
    def siguiente_canal(self):
        print("Samsung->Siguiente")
    def canal_anterior(self):
        print("Samsung->Anterior")
    def subir_volumen(self):
        print("Samsung->Subir")
    def bajar_volumen(self):
        print("Samsung->Bajar")

mando_samsung = MandoSamsung()
mando_samsung.bajar_volumen()
```

En este caso obtenemos:

```
Samsung->Bajar
```

Si alguno de los métodos abstractos no es implementado en la clase heredada obtenemos error al crear un objeto de la clase heredada:

```
class MandoLG(Mando):
    def siguiente_canal(self):
        print("LG->Siguiente")
    def canal_anterior(self):
        print("LG->Anterior")
mando_lg = MandoLG()
```

Obtenemos:

```
TypeError: Can't instantiate abstract class MandoLG with abstract methods
bajar_volumen, subir_volumen
```

Si en la clase abstracta se incluyen métodos no abstractos los podemos utilizar en la clase heredada sin estar obligados a reescribirlos:

```
from abc import ABC, abstractmethod

class Mando(ABC):
    def apagar(self):
        print("Apagando la TV")
    @abstractmethod
    def siguiente_canal(self):
        pass

    @abstractmethod
    def canal_anterior(self):
        pass

    @abstractmethod
    def subir_volumen(self):
        pass

    @abstractmethod
    def bajar_volumen(self):
        pass

class MandoSamsung(Mando):
    def siguiente_canal(self):
        print("Samsung->Siguiente")
    def canal_anterior(self):
        print("Samsung->Anterior")
    def subir_volumen(self):
        print("Samsung->Subir")
    def bajar_volumen(self):
        print("Samsung->Bajar")

mando_samsung = MandoSamsung()
mando_samsung.apagar()
```

Obtenemos:

```
Apagando la TV
```

Un método abstracto puede incluir código que podemos aprovechar en las clases heredadas:

```
from abc import ABC, abstractmethod

class Mando(ABC):
    @abstractmethod
    def siguiente_canal(self):
        print("Mando->Siguiente")      # Método abastracto que incluye código
        pass

    @abstractmethod
    def canal_anterior(self):
        pass

    @abstractmethod
    def subir_volumen(self):
        pass

    @abstractmethod
    def bajar_volumen(self):
        pass
```

Desde los métodos de las subclases podemos llamar a las implementaciones de la clase abstracta con el comando super() seguido del nombre del método.

```
class MandoLG(Mando):
    def siguiente_canal(self):
        super().siguiente_canal()
        print("LG->Siguiente")
    def canal_anterior(self):
        print("LG->Anterior")
    def subir_volumen(self):
        print("LG->Subir")
    def bajar_volumen(self):
        print("LG->Bajar")

mando_lg = MandoLG()
mando_lg.siguiente_canal()
```

El resultado sería:

```
Mando->Siguiente
LG->Siguiente
```

Por tanto, ahora mismo tenemos varios conceptos diferentes claramente identificados:

- Por un lado tenemos nuestro **interfaz** `Mando`. Se trata de una clase que, en principio, define el comportamiento de un mando genérico, pero sin centrarse en los detalles de cómo funciona. Se centra en el **qué**.

- Por otro lado tenemos dos clases `MandoSamsung` y `MandoLG` que implementan/heredan el interfaz anterior, añadiendo un código concreto y diferente para cada mando. Ambas clases representan el **cómo**.
- Si una clase abstracta incluye métodos no abstractos estos se heredan de forma normal. En este caso, en la clase abstracta estamos incluyendo el **qué** y parte del **cómo**
- Podemos implementar funcionalidad en los métodos abstractos que puede ser reutilizada en las clases que heredan de la clase abstracta utilizando el método `super()`. También, en este caso, en la clase abstracta estamos incluyendo el **qué** y parte del **cómo**

Recursos

- Interfaces y Abstract Base Class (ABC) - El libro de Python
- Clases abstractas en Python - Analytics Lane
- Abstract classes in python - geeks for geeks

tags: `pro` `ut2` `poo` `oop` `herencia` `abstracción` `clases` `abstractas`