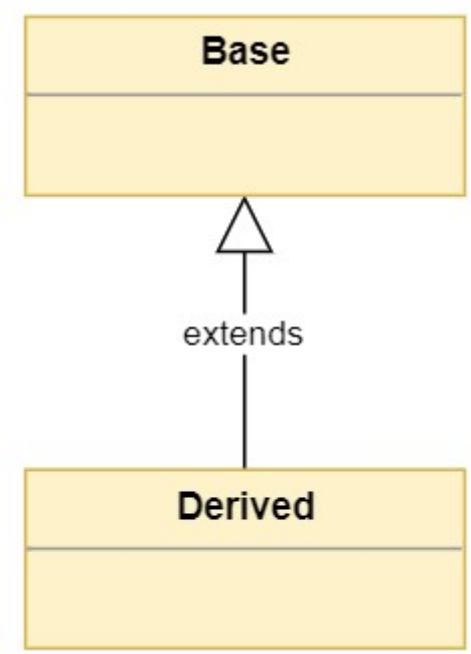
PRO-UT2-A3. Herencia Simple

Herencia en OOP

La **herencia** modela un tipo de relación llamada **es un** entre dos clases. Significa que cuando tenemos una clase **derivada** que hereda de una clase **base** se crea una relación en la que la clase `derivada` **es una** `especialización` de la clase `base`

En UML se representa de la siguiente forma:



Las cajas representan las clases y la flecha la relación; parte de la clase derivada hacia la clase base y se suele añadir la palabra **extends** a la flecha.

En una relación de herencia:

- Las clases que heredan de otra se llaman clases **derivadas**, **subclases** o **subtipos**
- Las clases desde las que otras derivan se llaman clases **base** o **superclases**
- De una clase derivada se dice que **deriva**, **hereda** o **extiende** una clase base.

Si por ejejmplo, tenemos una clase `Animal` y derivamos de la misma una clase `Gato`, la relación de herencia indica que el `Gato` **es un** `Animal`. Esto significa que el `Gato` **hereda** las **características** de un `Animal`.

Herencia simple en Python

Definir clase heredada

Si tenemos dos clases en Python. Por ejemplo:

```
class MotherClass:
    pass

class ChildClass:
    pass
```

Para establecer una relación de herencia entre ambas se hace especificando en la clase `hija` la clase `madre` como parámetro.

```
class MotherClass:
    pass

class ChildClass(MotherClass):
    pass
```

Miembros de clases heredadas

Su pongamos que tenemos una clase madre `Animal`:

```
class Animal:
    def __init__(self, legs):
        self.legs = legs

    def walk(self):
        return "Animal walking..."
```

Si queremos crear las clases perro (`Dog`) y pato(`Duck`) de forma que **hereden** las características de la clase animal los hacemos de la manera siguiente:

```
class Dog(Animal):
    def growl(self):
        return "A dog can growl but a duck can't. Grrr..."

class Duck(Animal):
    def quack(self):
        return "A duck can quack but a dog can't. Quack..."
```

Las clases hija cumplen con la característica de que un pato **es un** animal y un perro **es un** animal.

Al heredar, las clases hija tienen acceso a los miembros de la clase madre y por tanto incluyen:

- El atributo `legs`
- los métodos `__init__()` y `walk()` de la clase madre.

Por tanto, si creamos un objeto de una de las clases hija tiene las propiedades de la clase madre y las propiedades añadidas en la propia clase hija:

```
toby = Dog(4)
print(toby.legs)     # 4
print(toby.walk())   # "Animal walking..."
print(toby.growl()) # "A dog can growl but a duck can't. Grrr..."

lucas = Duck(2)
print(lucas.legs)     # 4
print(lucas.walk())   # "Animal walking..."
print(lucas.quack()) # "A duck can quack but a dog can't. Quack..."
```

Sobreescritura de métodos heredados

Si en una clase hija se crea un método con el mismo nombre que la clase madre el nuevo método **sobreescribe** el método de la clase madre.

```
class Animal:
    def __init__(self, legs):
        self.legs = legs

    def walk(self):
        return "Animal walking..."

class Dog(Animal):
    def __init__(self, legs, color):
        self.legs = legs
        self.color = color

    def walk(self):
        return "Dog walking..."
```

Polimorfismo

Aprovechemos que hemos visto algunas de las ideas de la herencia para introducir el concepto de polimorfismo.

El término **polimorfismo** tiene origen en las palabras *poly* (muchos) y *morfo* (formas). El **polimorfismo** es una de las propiedades básicas de la programación y en particular de la programación orientada a objetos. La **idea básica** es que tenemos una **función o método** con el **mismo nombre**, pero que al ser usada en diferentes tipos obtemos **resultados distintos**.

Podemos observarlo en diferentes casos.

Polimorfismo por herencia

Tal y como acabamos de ver, la herencia en POO permite que una clase hija herede los métodos de la clase madre. Si el método en la clase hija no se comporta de la misma forma que en la clase madre podemos **sobreescribirlo** .

Lo que conseguimos con esto es que si ejecutamos el mismo método en un objeto de la clase madre o en un objeto de la clase hija el resultado sea distinto y, por tanto, tenemos que un método **con el mismo nombre** adopta "forma" distinta en diferentes objetos.

En el ejemplo anterior, un mismo método, `walk()` genera diferentes salidas aplicado a un objeto de la clase madre `Animal` o a un objeto de la clase hija `Dog`

```
a = Animal()
b = Dog()
print(a.walk())
print(b.walk())
```

El mismo método aplicado a objetos con una relación de herencia hace que obtengamos resultados distintos si el método ha sido sobreescrito en la clase hija:

```
Animal walking...
Dog walking...
```

Polimorfismo con métodos de clase

No es necesario que haya una relación de herencia entre dors clases para que se utilice polimorfismo. Podemos crear dos clases distintas que contengan **métodos con el mismo nombre**:

```
class India():
    def capital(self):
        return "New Delhi is the capital of India."

    def language(self):
        return "Hindi is the most widely spoken language of India."


class USA():
    def capital(self):
        return "Washington, D.C. is the capital of USA."

    def language(self):
        return"English is the primary language of USA."
```

Y luego un **segmento de código**, por ejemplo un bucle, que llame a los métodos `capital()` y `language()` para un **objeto genérico**, sin tener en cuenta a partir de que clase se creo dicho objeto:

```
obj_ind = India()
obj_usa = USA()
countries = [obj_ind, obj_usa]
for country in countries:
    country.capital()
    country.language()
```

El resultado sería:

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
```

Obtenemos un resultado distinto ejecutando el mismo método.

Polimorfismo con función y objetos

También podemos conseguir polimorfismo si tenemos una función que tenga como parámetro uno o varios objetos. Si llamamos a la función pasándole objetos de distintas clases que incluyan métodos con el mismo nombre obtendremos resultados (**formas**) distintas.

Para entenderlo mejor veamos un ejemplo:

Tenemos las dos clases anteriores `India` y `USA` que contienen métodos con el mismo nombre. Si tenemos la siguiente funcion:

```
def func(obj):
    print(obj.capital())
    print(obj.language())
```

Si creamos un objeto de cada una de las clases y llamamos a la función pasándole cada uno de los objetos anteriores:

```
obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

El resultado es:

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
```

El **polimorfismo** permite que la función reciba como parámetros objetos instanciados a partir de cualquier clase y que el resultado sea distinto en función de la clase a partir de la cual se instancie el objeto.

Accediendo a los métodos de la clase madre con super()

El método **super()** permite reutilizar el código de la clase madre en las clases hija, lo que nos permite no tener que reescribir un método completamente si va a incluir código del método de la clase Madre.

En el siguiente ejemplo aprovechamos el código del constructor de la clase madre en la clase hija:

```
class Animal:
    def __init__(self, legs):
        self.legs = legs

    def walk(self):
        return "Animal. Walking..."

class Dog(Animal):
    def __init__(self, legs, color):
        super().__init__(legs)          # Reutilizamos constructor clase madre
        self.color = color

    def growl(self):
        return "A dog can growl but a duck can't. Grrr..."

class Duck(Animal):
    def quack(self):
        print("Duck child class. A duck can quack but a dog can't. Quack...")

fluffy = Dog(4, "grey")
print(fluffy.legs)      # 4
print(fluffy.color)     # "grey"
```

Recursos

- Herencia y Composición en Python - Realpython
- Herencia en OOP con Python - Jack Dong - medium.com
- OOP inheritance
- https://realpython.com/python-super/ - RealPython
- OOP in Python - RealPython
- OOP in Python vs Java - RealPython

tags: `pro` `ut2` `poo` `oop` `herencia`