Herencia

Fuente: hektorprofe.net/analyticslane.com/wikipedia.org

La **herencia** es la capacidad que tiene una clase de heredar los atributos y métodos de otra, algo que nos permite **reutilizar código**.

Partiremos de una clase sin herencia con muchos atributos y la iremos descomponiendo en otras clases más simples que nos permitan trabajar mejor con sus datos.

Ejemplo sin herencia

Si partimos de una clase que contenga todos los atributos, quedaría más o menos así:

Código

```
class Producto:
    def __init__(self, referencia, tipo, nombre,
                 descripcion,precio,productor="",
                 distribuidor="", isbn="", autor=""):
        self.referencia = referencia
        self.tipo = tipo
        self.nombre = nombre
        self.descripcion = descripcion
        self.precio = precio
        self.productor = productor
        self.distribuidor = distribuidor
        self.isbn = isbn
        self.autor = autor
```

Resultado

Obviamente esto es un despropósito, así que veamos cómo aprovecharnos de la herencia para mejorar el planteamiento.

Superclases

Así pues, la idea de la herencia es identificar una **clase base** (la **superclase**) con los atributos comunes y luego crear las demás clases heredando de ella (las **subclases**) **extendiendo** sus campos específicos. En nuestro caso esa clase sería el **Producto** en sí mismo.

```
class Producto:
    def __init__(self,referencia,nombre,descripcion,precio):
        self.referencia = referencia
        self.nombre = nombre
        self.descripcion = descripcion
        self.precio = precio

    def __str__(self):
        return "Producto: {} - {} - {} -
{}".format(self.referencia,self.nombre,self.descripcion,self.precio)

    def rebajar_producto(self,rebaja):
        self.precio = self.precio - rebaja
```

Subclases

Para heredar los atributos y métodos de una clase en otra sólo tenemos que pasarla entre paréntesis en la definición:

Importar contenido de otras clases

Se importará el contenido de otro archivo .py agregando en la cabecera del archivo **from** (nombre del archivo sin la extensión .py) **import** (Clase a importar).

Como tener que determinar constantemente la superclase puede ser fastidioso, Python nos permite utilizar un acceso directo mucho más cómodo llamado ***super()***.

Hacerlo de esta forma además nos permite llamar cómodamente los métodos o atributos de la superclase sin necesidad de especificar el self, pero ojo, **sólo se aconseja utilizarlo cuando tenemos una única superclase**:

Código

```
from producto import Producto

class Adorno(Producto):
    def __init__(self,referencia,nombre,descripcion,precio,estilo):
        super().__init__(referencia, nombre, descripcion,precio)
        self.estilo = estilo

    def __str__(self):
        return "{} - {} - {} -
{}".format(self.referencia,self.nombre,self.descripcion,self.estilo)
```

Como se puede apreciar es posible utilizar el comportamiento de una superclase sin definir nada en la subclase.

Respecto a las demás subclases como se añaden algunos atributos, podríamos definirlas de esta forma:

```
from producto import Producto

class Libro(Producto):

    def __init__(self,referencia,nombre,descripcion,precio,isbn,autor):
        super().__init__(referencia, nombre, descripcion,precio)
        self.isbn = isbn
        self.autor = autor

    def __str__(self):
        return "{} - {} - {} - {} - {} -
{}".format(self.referencia,self.nombre,self.descripcion,self.precio,self.isbn,self.autor)
```

Ahora para utilizarlas simplemente tendríamos que establecer los atributos después de crear los objetos:

Definición de la función main()

Esto está íntimamente ligado al modo de funcionamiento del intérprete Python:

- Cuando el intérprete lee un archivo de código, **ejecuta todo el código global que se encuentra en él**. Esto implica crear objetos para toda función o clase definida y variables globales.
- Todo módulo (archivo de código) en Python tiene un atributo especial llamado __name__ que define el espacio de nombres en el que se está ejecutando. Es usado para identificar de forma única un módulo en el sistema de importaciones.
- Por su parte "__main__" es el nombre del ámbito en el que se ejecuta el código de nivel superior (tu programa principal).
- El intérprete pasa el valor del atributo __name__ a la cadena '__main__' si el módulo se está ejecutando como programa principal (cuando lo ejecutas llamando al intérprete en la terminal con python my_modulo.py, haciendo doble click en él, ejecutándolo en el intérprete interactivo, etc).
- Si el módulo no es llamado como programa principal, sino que es importado desde otro módulo, el atributo __name__ pasa a contener el nombre del archivo en si.

Ventajas de usar def main()

- Otros lenguajes (como C y Java) tienen una función main() que se llama cuando se ejecuta el programa. Utilizando este if, podemos hacer que Python se comporte como ellos, lo cual es más familiar para muchas personas.
- El código será más fácil de leer y estará mejor organizado.
- Será posible ejecutar tests en el código.
- Podemos importar ese código en un shell de python y probarlo/depurarlo/ejecutarlo.
- variables dentro def main() son **locales**, mientras que las que están afuera son **globales**. Esto puede introducir algunos errores y comportamientos inesperados.
- Permite ejecutar la función si se importa el archivo como un módulo

Código (main.py)

```
from producto import Producto
from alimento import Alimento
from adorno import Adorno
from libro import Libro

def main():
    producto = Producto(2033,"Producto Genérico","1 kg",50)
    alimento = Alimento(2035, "Botella de Aceite de Oliva","250
ML",50,"Marca","Distribuidor")
    adorno = Adorno(2034, "Vaso adornado", "Vaso de porcelana",34,"De Mesa")
    libro = Libro(2036, "Cocina Mediterránea","Recetas buenas",75,"0-123456-78-
9","Autor")

if __name__ == "__main__":
    main()
```

Luego en los ejercicios os mostraré como podemos sobreescribir el constructor de una forma eficiente para inicializar campos extra, por ahora veamos como trabajar con estos objetos de distintas clases de forma común.

Trabajando en conjunto

Gracias a la flexibilidad de Python podemos manejar objetos de distintas clases masivamente de una forma muy simple.

Vamos a empezar creando una lista con nuestros tres productos de subclases distintas:

Código

```
productos = [adorno, alimento]
productos.append(libro)
```

Ahora si queremos recorrer todos los productos de la lista podemos usar un bucle *for*:

Código

```
for producto in productos:
    print(producto)
```

También podemos acceder a los atributos, siempre que sean compartidos entre todos los objetos:

Código

```
for producto in productos:
    print(producto.referencia, producto.nombre)
```

Si un objeto no tiene el atributo al que queremos acceder nos dará error:

Código

```
for producto in productos:
    print(producto.autor)
```

Por suerte podemos hacer una comprobación con la función *isinstance()* para determinar si una instancia es de una determinado clase y así mostrar unos atributos u otros:

Código

```
for producto in productos:
    if(isinstance(producto, Adorno)):
        print(producto.referencia, producto.nombre)
    elif(isinstance(producto, Alimento)):
        print(producto.referencia, producto.nombre, producto.productor)
    elif(isinstance(producto, Libro)):
        print(producto.referencia, producto.nombre, producto.isbn)
```

Polimorfismo

El polimorfismo es una propiedad de la herencia por la que objetos de distintas subclases pueden responder a una misma acción.

La polimorfia es implícita en Python, ya que todas las clases son subclases de una superclase común llamada **Object**.

Por ejemplo la siguiente función aplica una rebaja al precio de un producto, ubicar en **producto.py**:

```
def rebajar_producto(self,rebaja):
        self.precio = self.precio - rebaja
```

Gracias al polimorfismo no tenemos que comprobar si un objeto tiene o no el atributo *precio*, simplemente intentamos acceder y si existe premio:

Código

```
for producto in productos:
    producto.rebajar_producto(10)
    print(producto)
```

Por cierto, como podéis ver en el ejemplo, cuando modificamos un atributo de un objeto dentro de una función éste cambia en la instancia. Esto es por aquello que os comenté del paso por valor y referencia.

Herencia múltiple

Finalmente hablemos de la herencia múltiple: la capacidad de una subclase de heredar de múltiples superclases.

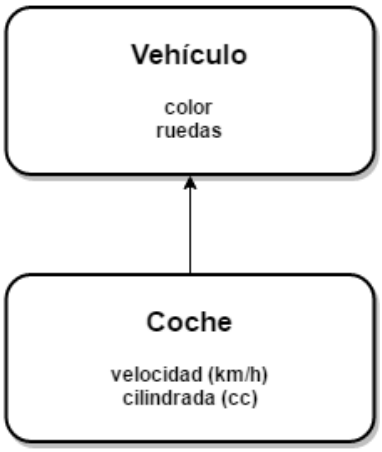
Esto conlleva un problema, y es que si varias superclases tienen los mismos atributos o métodos, la subclase sólo podrá heredar de una de ellas.

En estos casos Python dará prioridad a las clases **más a la izquierda** en el momento de la declaración de la subclase:

Código

```
(a.py)
       class A:

    def a(self):
        print("Este método lo heredo de A")

    def b(self):
        print("Este método lo heredo de A")

(b.py)
class B:

    def b(self):
        print("Este método lo heredo de B")

(c.py)
from a import A
from b import B

class C(B,A):

    def __init__(self):
        print("Soy de la clase C")

    def c(self):
        print("Este método es de C")

(main.py)
from a import A
from b import B
from c import C

def main():
    c = C()
    c.a()
    c.b()
    c.c()

if __name__ == "__main__":
    main()
```

Hasta ahora sabemos que una clase heredada puede fácilmente extender algunas funcionalidades, simplemente añadiendo nuevos atributos y métodos, o sobreescribiendo los ya existentes. Como en el siguiente ejemplo:



Ejercicio

```
class Vehiculo():

    def __init__(self, color, ruedas):
        self.color = color
        self.ruedas = ruedas

    def __str__(self):
        return "Color {}, {} ruedas".format(self.color,self.ruedas)

class Coche(Vehiculo):

    def __init__(self, color, ruedas, velocidad, cilindrada):
        self.color = color
        self.ruedas = ruedas
        self.velocidad = velocidad
        self.cilindrada = cilindrada

    def __str__(self):
        return "color {}, {} km/h, {} ruedas, {} cc".format( self.color, self.velocidad,
self.ruedas, self.cilindrada )


coche = Coche("azul", 150, 4, 1200)
print(coche)
```

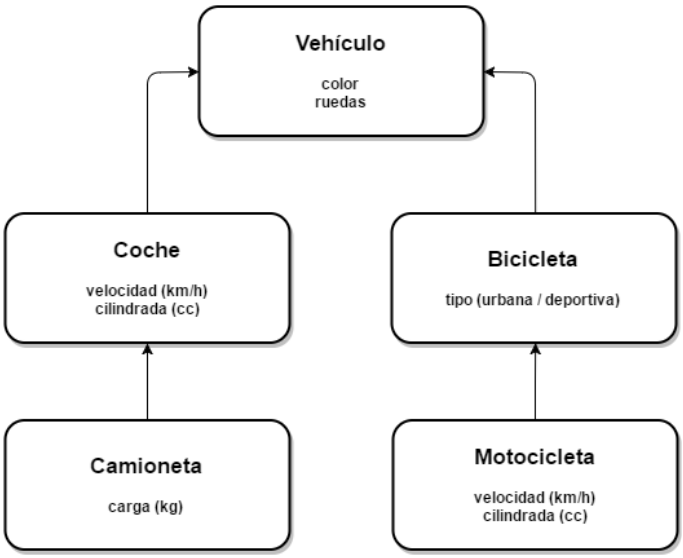
El inconveniente más evidente de ir sobreescribiendo es que tenemos que volver a escribir el código de la superclase y luego el específico de la subclase. Para evitarnos escribir código innecesario, podemos utilizar un truco que consiste en llamar el método de la superclase y luego simplemente escribir el código de la clase:

Ejercicio

```
(vehiculo.py)
class Vehiculo():

    def __init__(self, color, ruedas):
        self.color = color
        self.ruedas = ruedas

    def __str__(self):
        return "color {}, {} ruedas".format(self.color, self.ruedas)


from vehiculo import Vehiculo

(coche.py)
class Coche(Vehiculo):

    def __init__(self, color, ruedas, velocidad, cilindrada):
        Vehiculo.__init__(self, color, ruedas)
        self.velocidad = velocidad
        self.cilindrada = cilindrada

    def __str__(self):
        return Vehiculo.__str__(self) + ", {} km/h, {} cc".format(self.velocidad,
self.cilindrada)

(main.py)
from coche import Coche

def main():
    c = Coche("azul", 4, 150, 1200)
    print(c)

if __name__ == "__main__":
    main()
```

Enunciado

Utilizando esta nueva técnica extiende la clase Vehiculo y realiza la siguiente implementación:



- Crea al menos un objeto de cada subclase y añadelos a una lista llamada vehículos.
- Realiza una función llamada **catalogar()** que reciba la lista de vehículos y los recorra mostrando el nombre de su clase y sus atributos.
- Modifica la función **catalogar()** para que reciba un argumento optativo **ruedas**, haciendo que muestre únicamente los que su número de ruedas concuerde con el valor del argumento. También debe mostrar un mensaje **"Se han encontrado {} vehículos con {} ruedas:"** únicamente si se envía el argumento ruedas. Ponla a prueba con 0, 2 y 4 ruedas como valor.

Clases Abstractas

Un concepto importante en programación orientada a objetos es el de las clases abstractas. Son clases en las que se pueden definir tanto métodos como propiedades, pero que no pueden ser instanciadas directamente. Solamente se pueden usar para construir subclases. Permitiendo así tener una única implementación de los métodos compartidos, evitando la duplicación de código.

Propiedades de las clases abstractas

La primera propiedad de las clases abstractas es que no pueden ser instanciadas. Simplemente proporcionan una interfaz para las subclases derivadas evitando así la duplicación de código.

Otra característica de estas clases es que no es necesario que tengan una implementación de todos los métodos necesarios. Pudiendo ser estos abstractos. Los métodos abstractos son aquellos que solamente tienen una declaración, pero no una implementación detallada de las funcionalidades.

Las clases derivadas de las clases abstractas debe 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 no se podrá crear la clase.
Resumiendo, las clases abstractas definen una interfaz común para las subclases. Proporcionan atributos y métodos comunes para todas las subclases evitando así la necesidad de duplicar código. Imponiendo además los métodos que deber ser implementados para evitar inconsistencias entre las subclases

Creación de clases abstractas en Python

Para poder crear clases abstractas en Python es necesario importar la clase ABC y el decorador **abstractmethod** del módulo **abc** (Abstract Base Classes). Un módulo que se encuentra en la **librería estándar** del lenguaje, por lo que no es necesario instalarlo. Así para definir una clase abstracta solamente se tiene que crear una clase heredada de **ABC** con un método abstracto.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass
```

Ahora si se intenta crear una instancia de la clase animal, Python no lo permitirá indicando que no es posible. Es importante notar que, si la clase no hereda de ABC o contiene por lo menos un método abstracto, Python permitirá instancias de las clases.

```
class Animal(ABC):
    def mover(self):
        print("El animal se mueve")

animal = Animal()
```

Métodos en las subclases

Las subclases tienen que implementar todos los métodos abstractos, en el caso de que falte alguno de ellos Python no permitirá instancias tampoco la clase hija

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass

    @abstractmethod
    def comer(self):
        print('El animal come')
```

Por otro lado, desde los métodos de las subclases podemos llamar a las implementaciones de la clase abstracta con el comando super() seguido del nombre del método. La palabra **pass** permite no definir el contenido de un método.

```
(animal.py)
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass

    @abstractmethod
    def comer(self):
        print('Animal come')

(gato.py)
from animal import Animal

class Gato(Animal):

    def mover(self):
        print('Mover gato')

    def comer(self):
        super().comer()
        print('Gato come')

(main.py)
from gato import Gato

def main():
    g = Gato()
    g.mover()
    g.comer()

if __name__ == "__main__":
    main()
```

Copia de Objetos

Para realizar una copia a partir de sus valores podemos utilizar la función ***copy*** del módulo con el mismo nombre:

```
from copy import copy

class Test:
    pass

test1 = Test()
test2 = copy(test1)
```

Diagrama de Clases

Es un tipo de diagrama de estructura estática que describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos), y las relaciones entre los objetos.

Para especificar la visibilidad de un miembro de la clase (es decir, cualquier atributo o método), se coloca uno de los siguientes signos delante de ese miembro:

+	Público
-	Privado
#	Protegido

Ámbitos

UML especifica dos tipos de ámbitos para los miembros: instancias y clasificadores y estos últimos se representan con nombres subrayados.

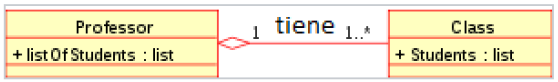
- Los miembros **clasificadores** se denotan comúnmente como "**estáticos**" en muchos lenguajes de programación. Su ámbito es la propia clase.
    - Los valores de los atributos son los mismos en todas las instancias
    - La invocación de métodos no afecta al estado de las instancias

- Los miembros **instancias** tienen como ámbito una instancia específica.
    - Los valores de los atributos pueden variar entre instancias
    - La invocación de métodos puede afectar al estado de las instancias(es decir, cambiar el valor de sus atributos)
    -

Para indicar que un miembro posee un ámbito de **clasificador**, hay que subrayar su nombre. De lo contrario, se asume por defecto que tendrá ámbito de **instancia**.

Relaciones a nivel de instancia

Agregación



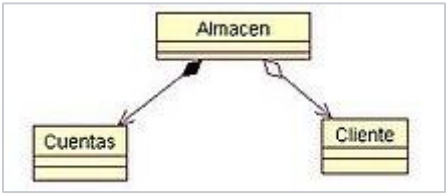
Como se puede ver en la imagen del ejemplo (en inglés), un Profesor **'tiene una o más clases'** clase a las que enseña.

Una agregación puede tener un nombre e indicaciones de cardinalidad en los extremos de la línea. Sin embargo, una agregación no puede incluir más de dos clases; debe ser una asociación binaria.

Una ***agregación*** se puede dar cuando una clase es una colección o un contenedor de otras clases, pero a su vez, el tiempo de vida de las clases contenidas **no tienen una *dependencia fuerte del tiempo de vida*** de la clase contenedora. Es decir, el contenido de la clase contenedora **no se destruye** automáticamente cuando desaparece dicha clase.

Se representa gráficamente con un ***rombo hueco*** junto a la clase contenedora con una línea que lo conecta a la clase contenida. Todo este conjunto es, semánticamente, un objeto extendido que es tratado como una única unidad en muchas operaciones, aunque físicamente está hecho de varios objetos más pequeños.

Composición



El rombo negro muestra una **relación de composición**: el almacén esta **compuesto** de cuentas, si se elimina el almacén las cuentas por si solas no tienen sentido como una entidad separada del almacén y **se eliminan también**. El rombo sin rellenar muestra una relación de agregación: el almacén tiene clientes, si el almacén cierra los clientes irán a otro, su razón de existir sigue teniendo sentido sin el almacén.

La representación de una relación de composición es mostrada con una figura de diamante rellenado del lado de la clase contenedora, es decir al final de la línea que conecta la clase contenido con la clase contenedor.

Diferencias entre Composición y Agregación

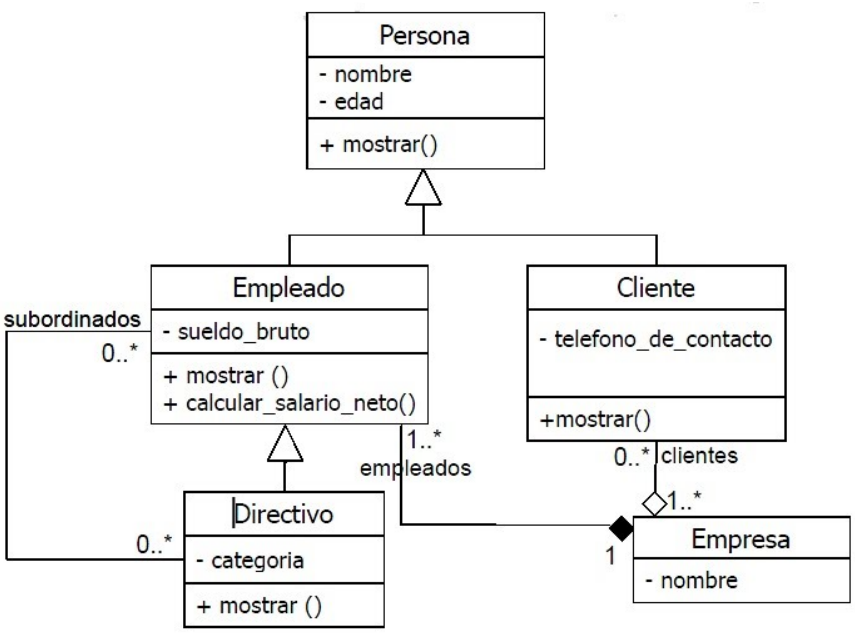
Relación de Composición

1. Cuando intentamos representar un todo y sus partes. Ejemplo, un motor es una parte de un coche.
2. Cuando se elimina el contenedor, el contenido también es eliminado. Ejemplo, si eliminamos una universidad eliminamos igualmente sus departamentos.

Relación de Agregación

1. Cuando representamos las relaciones en un software o base de datos. Ejemplo, el modelo de motor MTR01 es parte del coche MC01. Como tal, el motor MTR01 puede ser parte de cualquier otro modelo de coche, es decir si eliminamos el coche MC01 no es necesario eliminar el motor pues podemos usarlo en otro modelo.
2. Cuando el contenedor es eliminado, el contenido usualmente no es destruido. Ejemplo, un profesor tiene estudiantes, cuando el profesor muere los estudiantes no mueren con él o ella.

Así, una relación de agregación es a menudo "clasificar" o "catalogar" contenido para distinguirlo del todo "físico" del contenedor.


Persona
- nombre
- edad
+ mostrar()
Empleado
- sueldo_bruto
+ mostrar ()
+ calcular_salario_neto()
Cliente
- telefono_de_contacto
+mostrar()
subordinados
0..*
1..*
empleados
0..*
clientes
1..*
Directivo
- categoria
+ mostrar ()
0..*
1
Empresa
- nombre