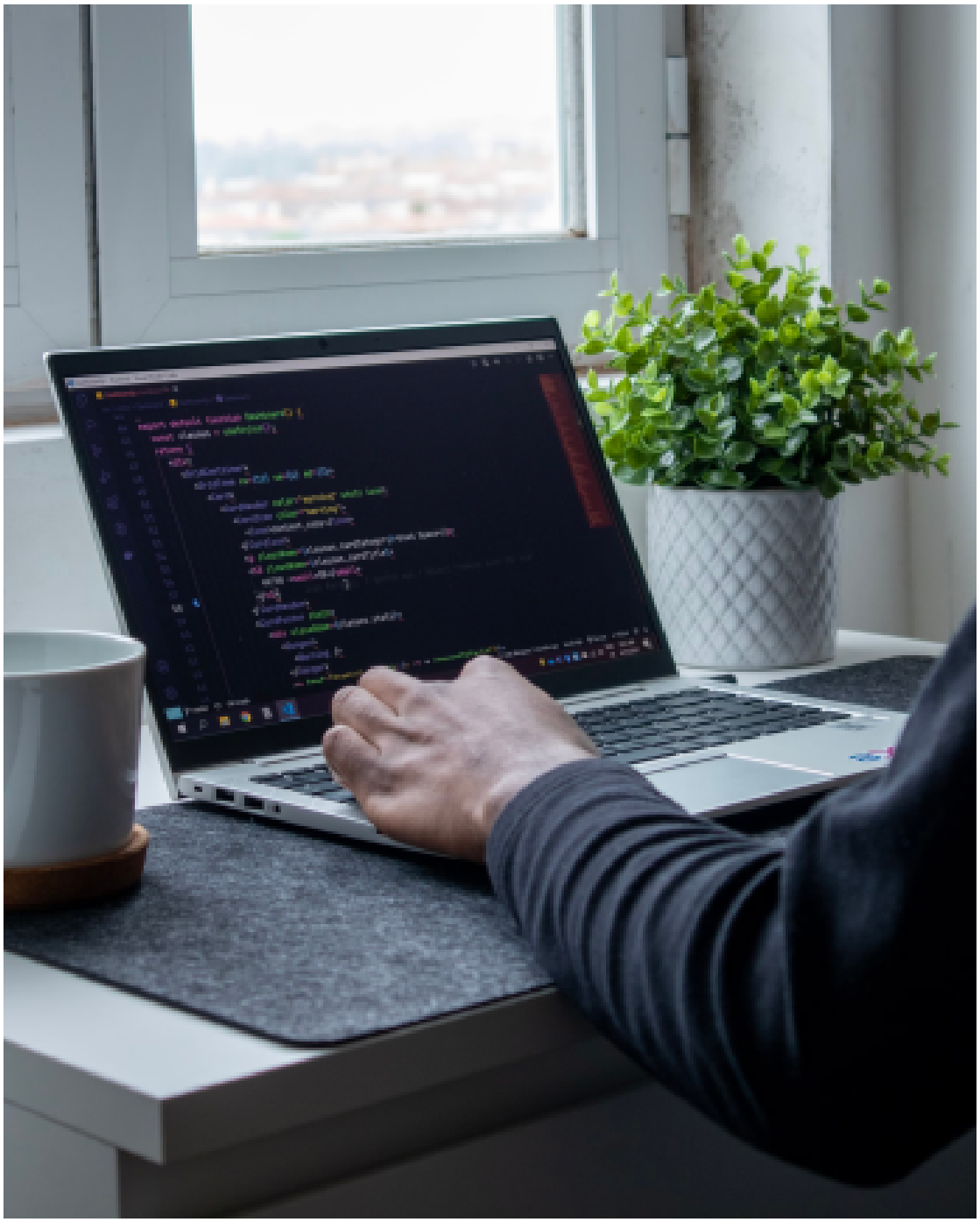
TIC

# Lección 2: Clases, Objetos y Herencia



**Tiempo de ejecución: 4 horas**

**Planteamiento de la sesión**

## Materiales

- Programación Orientada a Objetos | Aprende con Alf
- La función super en python bien explicada y con ejemplos! – OOP

Una tarjeta de crédito puede representarse como un objeto:

- Atributos: Número de la tarjeta, titular, balance, fecha de caducidad, pin, entidad emisora, estado (activa o no), etc.
- Métodos: Activar, pagar, renovar, anular.


Para ver si un objeto tiene un determinado atributo o método se utiliza la siguiente función:

- hasattr(objeto, elemento): Devuelve True si elemento es un atributo o un método del objeto objeto y False en caso contrario.
- Para acceder a los atributos y métodos de un objeto se pone el nombre del objeto seguido del operador punto y el nombre del atributo o el método.
- objeto.atributo: Accede al atributo atributo del objeto objeto.
- objeto.método(parámetros): Ejecuta el método método del objeto objeto con los parámetros que se le pasen.

Similar al ejemplo de la sección anterior, podemos tener una clase mascota con algunos atributos como raza y nombre y métodos como asignar dichos valores

```
class Mascota:
def __init__(self, nombre, raza):
self.nombre = nombre
self.raza = raza

def get_nombre(self):
return self.nombre

def set_nombre(self, nombre):
self.nombre = nombre
```

```
def get_raza(self):
return self.raza

def set_raza(self, raza):
self.raza = raza

def __str__(self):
return f'Nombre: {self.nombre}, Raza: {self.raza}
```

y crearemos una instancia de la clase mascota de la siguiente manera:

```
from Mascota import Mascota

#create a Mascota object
mascota = Mascota('Firulais','Pastor Aleman')

#print the object
print(mascota)
```

Con lo que hemos aprendido hasta el momento nos podemos hacer una idea de cada uno de sus elementos, pero para comenzar a desarrollar utilizando POO, vamos a comenzar por algunos conceptos adicionales'

**Clase**

Los objetos con los mismos atributos y métodos se agrupan clases. Las clases definen los atributos y los métodos, y por tanto, la semántica o comportamiento que tienen los objetos que pertenecen a esa clase. Se puede pensar en una clase como en un molde a partir del cuál se pueden crear objetos.

## Características

Casi todos los programas y técnicas que has utilizado hasta ahora pertenecen al estilo de programación procedimental. Es cierto que has utilizado algunos objetos incorporados, pero cuando nos referimos a ellos, se mencionan lo mínimo posible.

La programación procedimental fue el enfoque dominante para el desarrollo de software durante décadas de TI, y todavía se usa en la actualidad. Además, no va a desaparecer en el futuro, ya que funciona muy bien para proyectos específicos (en general, no muy complejos y no grandes, pero existen muchas excepciones a esa regla).

El enfoque orientado a objetos es bastante joven (mucho más joven que el enfoque procedimental) y es particularmente útil cuando se aplica a proyectos grandes y complejos llevados a cabo por grandes equipos formados por muchos desarrolladores.

Este tipo de programación en un proyecto facilita muchas tareas importantes, por ejemplo, dividir el proyecto en partes pequeñas e independientes y el desarrollo independiente de diferentes elementos del proyecto.

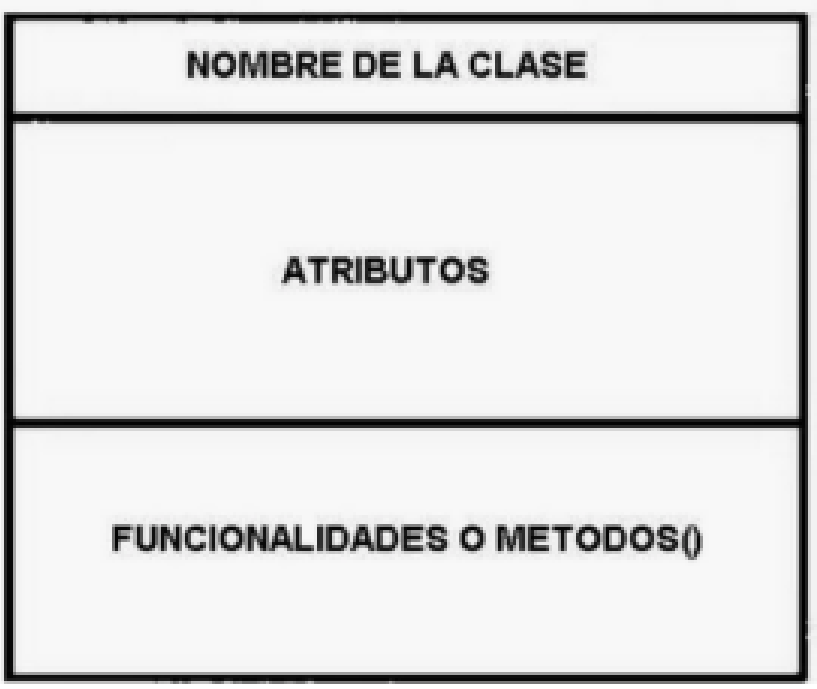
Python es una herramienta universal para la programación procedimental y orientada a objetos. Se puede utilizar con éxito en ambos enfoques.

Además, puedes crear muchas aplicaciones útiles, incluso si no se sabe nada sobre clases y objetos, pero debes tener en cuenta que algunos de los problemas (por ejemplo, el manejo de la interfaz gráfica de usuario) puede requerir un enfoque estricto de objetos.

Para declarar una clase se utiliza la palabra clave class seguida del nombre
de la clase y dos puntos, de acuerdo a la siguiente sintaxis:

class <nombre-clase>:
<atributos>
<métodos>

Similar a como se representa en un diagrama de clases:




Las clases nos dan la posibilidad de crear estructuras de datos más complejas

## Clases primitivas

En Python existen clases predefinidas para los tipos de datos primitivos:

- int: Clase de los números enteros.
- float: Clase de los números reales.
- str: Clase de las cadenas de caracteres.
- list: Clase de las listas.
- tuple: Clase de las tuplas.
- dict: Clase de los diccionarios.

```
>>> type(1)
<class 'int'>
>>> type(1.5)
<class 'float'>
>>> type('Python')
<class 'str'>
>>> type([1,2,3])
<class 'list'>
>>> type((1,2,3))
<class 'tuple'>
>>> type({1:'A',2:'B'})
<class 'dict'>
```

Esto quiere decir que todos los elementos que creemos utilizando dichas clases van a ser instancias de dichas clases, nuevamente entonces, todo en Python es un objeto.

## Instanciación de clases

Para crear un objeto de una determinada clase se utiliza el nombre de la clase seguida de los parámetros necesarios para crear el objeto entre paréntesis.

- clase(parámetros): Crea un objeto de la clase clase inicializado con los parámetros dados.

Cuando se crea un objeto de una clase se dice que el objeto es una instancia de la clase.

Tal cual como se definió con la clase Mascota o Saludo.

mascota = Mascota('Firulais','Pastor Aleman')

## Definición de métodos

Los métodos de una clase son las funciones que definen el comportamiento de los objetos de esa clase.

Se definen como las funciones con la palabra reservada def. La única diferencia es que su primer parámetro es especial y se denomina self. Este parámetro hace siempre referencia al objeto desde donde se llama el método, de manera que para acceder a los atributos o métodos de una clase en su propia definición se puede utilizar la sintaxis self.atributo o
self.método.

```
def get_nombre(self):
return self.nombre

def set_nombre(self, nombre):
self.nombre = nombre
o
class Saludo:
mensaje ="Bienvenido " # Definición de un atributo
def saludar(self, nombre): # Definición de un método
print(self.mensaje + nombre)
return

s = Saludo()
s.saludar('Alf')

#Bienvenido Alf
```

Entre la clase Saludo y la clase Mascota hay un método diferente llamado
__init__

**El método __init__**

En la definición de una clase suele haber un método llamado __init__ que se conoce como inicializador. Este método es un método especial que se llama cada vez que se instancia una clase y sirve para inicializar el objeto que se crea. Este método crea los atributos que deben tener todos los objetos de la clase y por tanto contiene los parámetros necesarios para su creación, pero no devuelve nada. Se invoca cada vez que se instancia un objeto de esa clase.

```
class Tarjeta:
# Inicializador

def __init__(self, id, cantidad = 0):
# Creación del atributo id
self.id = id
# Creación del atributo saldo
self.saldo = cantidad
return

def mostrar_saldo(self):
print('El saldo es', self.saldo,'€')
return

# Creación de un objeto con argumentos
t = Tarjeta('1111111111', 1000)
t.muestra_saldo()
```

**Atributos de instancia VS atributos de clase**

Los atributos que se crean dentro del método __init__ se conocen como atributos del objeto, mientras que los que se crean fuera de él se conocen como atributos de la clase. Mientras que los primeros son propios de cada objeto y por tanto pueden tomar valores distintos, los valores de los atributos de la clase son los mismos para cualquier objeto de la clase.

En general, no deben usarse atributos de clase, excepto para almacenar valores constantes.

```
class Circulo:
# Atributo de clase
pi = 3.14159
def __init__(self, radio):
# Atributo de instancia
self.radio = radio
def area(self):
return Circulo.pi * self.radio ** 2

c1 = Circulo(2)
c2 = Circulo(3)
print(c1.area())
# 12.56636
print(c2.area())
# 28.27431
print(c1.pi)
# 3.14159
print(c2.pi)
# 3.14159
```

Ahora, si revisamos las clases creadas, nos encontramos un método diferente en la clase Mascota, una al final llamada __str__

**El método __str__**

Otro método especial es el método llamado __str__ que se invoca cada vez que se llama a las funciones print o str. Devuelve siempre una cadena que se suele utilizar para dar una descripción informal del objeto. Si no se define en la clase, cada vez que se llama a estas funciones con un objeto de la clase, se muestra por defecto la posición de memoria del objeto.

```
class Tarjeta:
def __init__(self, numero, cantidad = 0):
self.numero = numero
self.saldo = cantidad
return
def __str__(self):
return 'Tarjeta número {} con saldo {:.2f}€'.format(self.numero,
str(self.saldo))

t = Tarjeta('0123456789',1000)
print(t)
#Tarjeta número 0123456789 con saldo 1000.00€
```

**Objeto**

La programación orientada a objetos supone que **cada objeto existente puede estar equipado con tres grupos de atributos:**

- Un objeto tiene un **nombre** que lo identifica de forma exclusiva dentro de su namespace (aunque también puede haber algunos objetos anónimos).
- Un objeto tiene un **conjunto de propiedades individuales** que lo hacen original, único o sobresaliente (aunque es posible que algunos objetos no tengan propiedades).
- Un objeto tiene un **conjunto de habilidades para realizar actividades específicas,** capaz de cambiar el objeto en sí, o algunos de los otros objetos.

Existe una pista (aunque esto no siempre funciona) que te puede ayudar a identificar cualquiera de las tres esferas anteriores. Cada vez que se describe un objeto y se usa:

- Un sustantivo: probablemente se está definiendo el nombre del objeto.
- Un adjetivo: probablemente se está definiendo una propiedad del objeto.
- Un verbo: probablemente se está definiendo una actividad del objeto.

Dos ejemplos deberían servir como un buen ejemplo:

- Un Cadillac rosa pasó rápidamente.

Nombre del objeto = Cadillac
Clase = Vehículos con ruedas
Propiedad = Color (rosa)
Actividad = circular (rápidamente)

- Rodolfo es un gato grande que duerme todo el día.

Nombre del objeto = Rodolfo
Class = Gato
Propiedad = Tamaño (Grande)
Actividad = Dormir (Todo el día)


## Herencia

Una de las características más potentes de la programación orientada a objetos es la herencia, que permite definir una especialización de una clase añadiendo nuevos atributos o métodos. La nueva clase se conoce como clase hija y hereda los atributos y métodos de la clase original que se conoce como clase madre.

Para crear un clase a partir de otra existente se utiliza la misma sintaxis que para definir una clase, pero poniendo detrás del nombre de la clase entre paréntesis los nombres de las clases madre de las que hereda.
Ejemplo. A partir de la clase Tarjeta definida antes podemos crear mediante herencia otra clase Tarjeta_Descuento para representar las tarjetas de crédito que aplican un descuento sobre las compras.

```
tclass Tarjeta:
def __init__(self, numero, cantidad = 0):
self.numero = numero
self.saldo = cantidad
return
def get_id(self):
return self.numero
def __str__(self):
return 'Tarjeta número {} con saldo {}€'.format(self.numero, str(self.saldo))
```

```
class Tarjeta_Descuento(Tarjeta):
def __init__(self, numero, descuento):
self.numero = numero
self.descuento = descuento
return
def mostrar_descuento(self):
# Método exclusivo de la clase Tarjeta_descuento
print('Descuento de',self.descuento, % en los pagos.')
return
def __str__(self):
return 'Tarjeta número {} con descuento del {}%'.format(self.numero,

str(self.descuento))
```

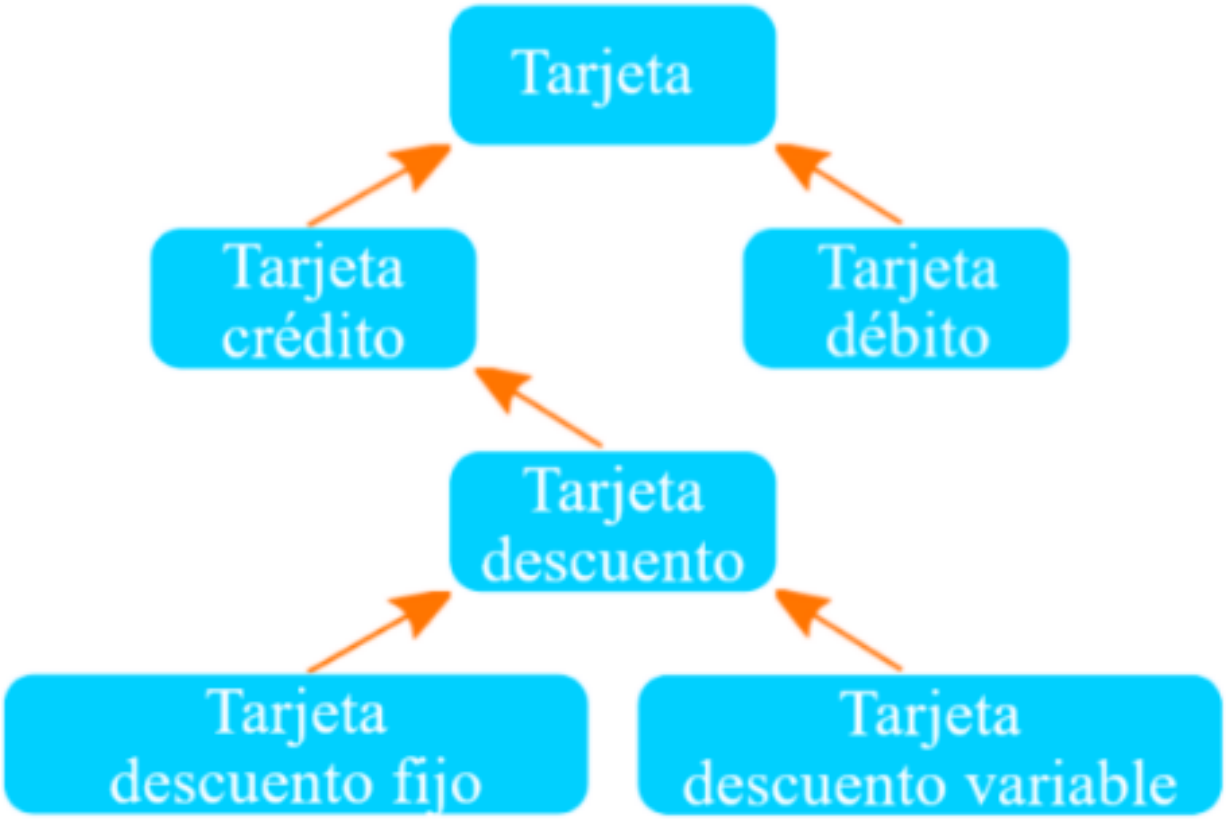
y crearemos un objeto así:

```
t_d = Tarjeta_Descuento('0123456789',2)print(t_d)
#Tarjeta número 0123456789 con descuento del 2%
print(t_d.get_id())
#0123456789
```

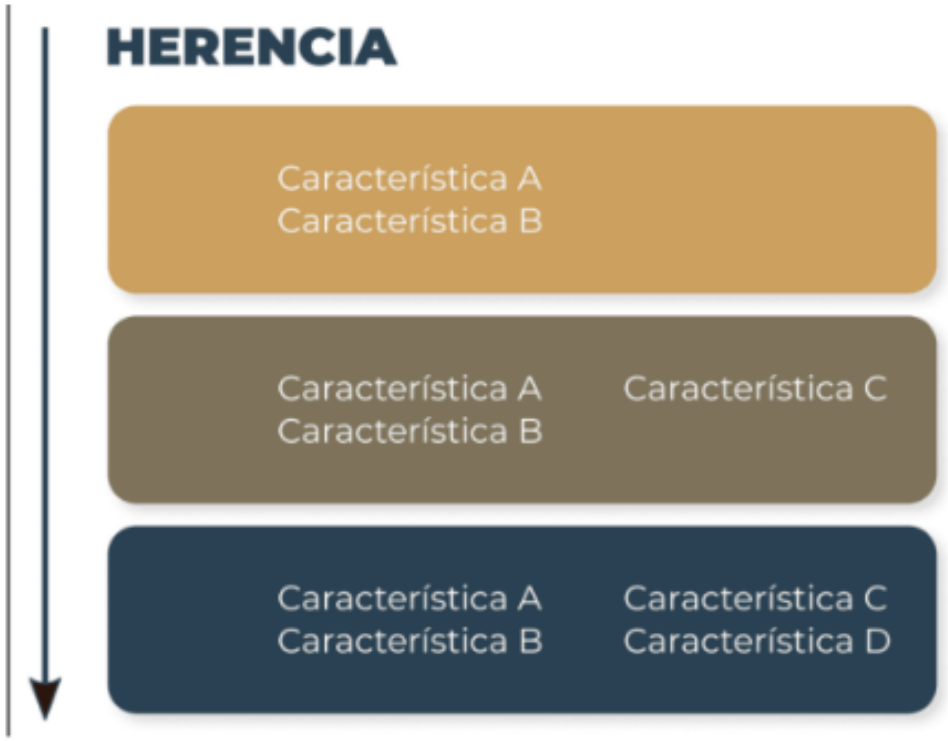
Si revisamos las dos clases creadas, el método get_id(), solo está definida en la clase padre, la clase Tarjeta, pero la estamos invocando desde la clase hija, esto es uno de los componentes de la herencia. De esta manera, cualquier cambio que se haga en el cuerpo del método en la clase madre, automáticamente se propaga a las clases hijas. Sin la herencia, este método tendría que replicarse en cada una de las clases hijas y cada vez que se hiciese un cambio en él, habría que replicarlo también en las clases hijas.

Cualquier objeto vinculado a un nivel específico de una jerarquía de clases hereda todos los rasgos (así como los requisitos y cualidades) definidos dentro de cualquiera de las superclases.

La clase de inicio del objeto puede definir nuevos rasgos (así como requisitos y cualidades) que serán heredados por cualquiera de sus superclases.



Tu perro (o tu gato) es un objeto incluido en la clase Mamíferos Domesticados, lo que significa explícitamente que también está incluido en la clase Animales.




## Ejercitación

Vamos a crear una nueva clase dentro de las tarjetas, que sería la tarjeta débito, tendrá un número y su saldo, y un método pagar que permite hacer pagos siempre y cuando el saldo sea suficiente, y verificamos si es instancia de la clase Tarjeta o Tarjeta_Debito

## Sobrecarga y polimorfismo

Los objetos de una clase hija hereda los atributos y métodos de la clase madre y, por tanto, a priori tienen el mismo comportamiento que los objetos de la clase madre. Pero la clase hija puede definir nuevos atributos o métodos o reescribir los métodos de la clase madre de manera que sus objetos presentan un comportamiento distinto. Esto último se conoce como **sobrecarga.**

De este modo, aunque un objeto de la clase hija y otro de la clase madre pueden tener un mismo método, al invocar ese método sobre el objeto de la clase hija, el comportamiento puede ser distinto a cuando se invoca ese mismo método sobre el objeto de la clase madre. Esto se conoce como **polimorfismo** y es otra de las características de la programación orientada a objetos.

Agreguemos un método pagar en la clase Tarjeta, donde no se valide nada, solamente hacemos el descuento del saldo, y vamos a crear una clase
Tarjeta Especial que aplica un porcentaje de descuento cuando la utilizamos, por ejemplo un valor fijo del 1% o uno definido por el sistema.

```
class Tarjeta:

def __init__(self, numero, cantidad = 0):
self.numero = numero
self.saldo = cantidad
return
def get_id(self):
return self.numero
def pagar(self, cantidad):
self.saldo -= cantidad
print('Pago de',cantidad,€ realizado con éxito.')
return
def __str__(self):
return 'Tarjeta número {} con saldo {}€'.format(self.numero,
str(self.saldo))
```

Ahora la clase con descuento, sin el método pagar inicialmente:

```
class Tarjeta_Oro(Tarjeta):
def __init__(self, numero, cantidad = 0, descuento = 1):
self.numero = numero
self.saldo = cantidad
self.descuento = descuento
return
def __str__(self):
return 'Tarjeta número {} con saldo {}€'.format(self.numero,
str(self.saldo))
```

## Principios de la programación orientada a objetos

La programación orientada a objetos se basa en los siguientes principios:

**Encapsulación:**Agrupar datos (atributos) y procedimientos (métodos) en unidades lógicas (objetos) y evitar manipular los atributos accediendo directamente a ellos, usando, en su lugar, métodos para acceder a ellos.

**Abstracción:** Ocultar al usuario de la clase los detalles de implementación de los métodos. Es decir, el usuario necesita saber qué hace un método y con qué parámetros tiene que invocarlo (interfaz), pero no necesita saber cómo lo hace.

**Herencia:**Evitar la duplicación de código en clases con comportamientos similares, definiendo los métodos comunes en una clase madre y los métodos particulares en clases hijas.

**Polimorfismo:**Redefinir los métodos de la clase madre en las clases hijas cuando se requiera un comportamiento distinto. Así, un mismo método puede realizar operaciones distintas dependiendo del objeto sobre el que se aplique.

## Herencia múltiple

En Python es posible realizar **herencia múltiple**. En los ejercicios anteriores hemos visto como se podía crear una clase padre que hereda a una clase hija, pudiendo hacer uso de sus métodos y atributos. La herencia múltiple es similar, pero una clase **hereda de varias clases** padre en vez de una sola.

## Principios de la programación orientada a objetos

La programación orientada a objetos se basa en los siguientes principios:

**Encapsulación:**Agrupar datos (atributos) y procedimientos (métodos) en unidades lógicas (objetos) y evitar manipular los atributos accediendo directamente a ellos, usando, en su lugar, métodos para acceder a ellos.

**Abstracción:** Ocultar al usuario de la clase los detalles de implementación de los métodos. Es decir, el usuario necesita saber qué hace un método y con qué parámetros tiene que invocarlo (interfaz), pero no necesita saber cómo lo hace.

**Herencia:**Evitar la duplicación de código en clases con comportamientos similares, definiendo los métodos comunes en una clase madre y los métodos particulares en clases hijas.

**Polimorfismo:**Redefinir los métodos de la clase madre en las clases hijas cuando se requiera un comportamiento distinto. Así, un mismo método puede realizar operaciones distintas dependiendo del objeto sobre el que se aplique.

## Herencia múltiple

En Python es posible realizar **herencia múltiple**. En los ejercicios anteriores hemos visto como se podía crear una clase padre que hereda a una clase hija, pudiendo hacer uso de sus métodos y atributos. La herencia múltiple es similar, pero una clase **hereda de varias clases** padre en vez de una sola.

Es posible también que una clase herede de otra clase y a su vez otra clase
herede de la anterior.

```
class Clase1:
pass
class Clase2(Clase1):
pass
class Clase3(Clase2):
pass
```

Llegados a este punto nos podemos plantear lo siguiente. Como sabemos las clases hijas heredan los métodos de las clases padre, pero también pueden reimplementarlos de manera distinta. Entonces, si llamo a un método que todas las clases tienen en común ¿a cuál se llama?. Pues bien, existe
una forma de saberlo.

La forma de saber a qué método se llama es consultar el **MRO** o Method Order Resolution. Esta función nos devuelve una tupla con el orden de búsqueda de los métodos. Como era de esperar se empieza en la propia clase y se va subiendo hasta la clase padre, de izquierda a derecha.

```
class Clase1:
pass
class Clase2:
pass
class Clase3(Clase1, Clase2):
pass
print(Clase3.__mro__)
# (<class '__main__.Clase3'>, <class '__main__.Clase1'>, <class
'__main__.Clase2'>, <class 'object'>)
```

Hasta el momento hemos hablado y visto ejemplos de diferentes principios, veamos entonces como funciona el encapsulamiento dentro de Python, ya que es un poco diferente a como se utiliza en, por ejemplo, java.

**Encapsulamiento**

Hace referencia al ocultamiento de los estados internos de una clase al exterior. Dicho de otra manera, encapsular consiste en hacer que los atributos o métodos internos a una clase no se puedan acceder ni modificar desde fuera, sino que tan solo el propio objeto pueda acceder a ellos.

```
class Clase:
atributo_clase = "Hola"
def __init__(self, atributo_instancia):
self.atributo_instancia = atributo_instancia

mi_clase = Clase("Que tal")
mi_clase.atributo_clase
mi_clase.atributo_instancia

# 'Hola'
# 'Que tal'
```

Ambos atributos son perfectamente accesibles desde el exterior. Sin embargo esto es algo que tal vez no queramos. Hay ciertos métodos o atributos que queremos que pertenezcan sólo a la clase o al objeto, y que sólo puedan ser accedidos por los mismos. Para ello podemos usar la doble __ para nombrar a un atributo o método. Esto hará que Python los interprete como “privados” de manera que no podrán ser accedidos desde el exterior.

```
class Clase:
atributo_clase ="Hola" # Accesible desde el exterior
__atributo_clase ="Hola" # No accesible

# No accesible desde el exterior
def __mi_metodo(self):
print("Haz algo")
self.__variable = 0

# Accesible desde el exterior
def metodo_normal(self):
# El método si es accesible desde el interior
self.__mi_metodo()

mi_clase = Clase()
#mi_clase.__atributo_clase # Error! El atributo no es accesible
#mi_clase.__mi_metodo() # Error! El método no es accesible
mi_clase.atributo_clase # Ok!
mi_clase.metodo_normal() # Ok!
```

Cómo Python encuentra propiedades y métodos

```
class Super:
def __init__(self, name):
self.name = name
def __str__(self):
return "Mi nombre es " + self.name + "."
class Sub(Super):
def __init__(self, name):
Super.__init__(self, name)
obj = Sub("Andy")
print(obj)
```

- Existe una clase llamada Super, que define su propio constructor utilizado para asignar la propiedad del objeto, llamada name.
- La clase también define el método __str__(), lo que permite que la clase pueda presentar su identidad en forma de texto.
- La clase se usa luego como base para crear una subclase llamadaSub.
- La clase Sub define su propio constructor, que invoca el de la superclase. Toma nota de como lo hemos hecho: Super.__init__(self, name).
- Hemos nombrado explícitamente la superclase y hemos apuntado al método para invocar a __init__(), proporcionando todos los argumentos necesarios.
- Hemos instanciado un objeto de la clase Sub y lo hemos impreso.

La salida es:

porcionando todos los argumentos necesarios.
Hemos instanciado un objeto de la clase Sub y lo hemos impreso.

La salida es:

Mi nombre es Andy.

Mira el código en el editor. Lo hemos modificado para mostrarte otro método de acceso a cualquier entidad definida dentro de la superclase.

En el ejemplo anterior, nombramos explícitamente la superclase. En este ejemplo, hacemos uso de la función super(), la cual accede a la superclase sin necesidad de conocer su nombre:

```
class Super:
def __init__(self, name):
self.name = name

def __str__(self):
return "Mi nombre es " + self.name + "."

class Sub(Super):
def __init__(self, name):
super().__init__(name)

obj = Sub("Andy")

print(obj)
```

La función super() crea un contexto en el que no tiene que (además, no debe) pasar el argumento propio al método que se invoca; es por eso que es posible activar el constructor de la superclase utilizando solo un argumento.

Nota: puedes usar este mecanismo no solo para **invocar al constructor de la superclase, pero también para obtener acceso a cualquiera de los recursos disponibles dentro de la superclase.**