Programación Orientada a Objetos

Objetos

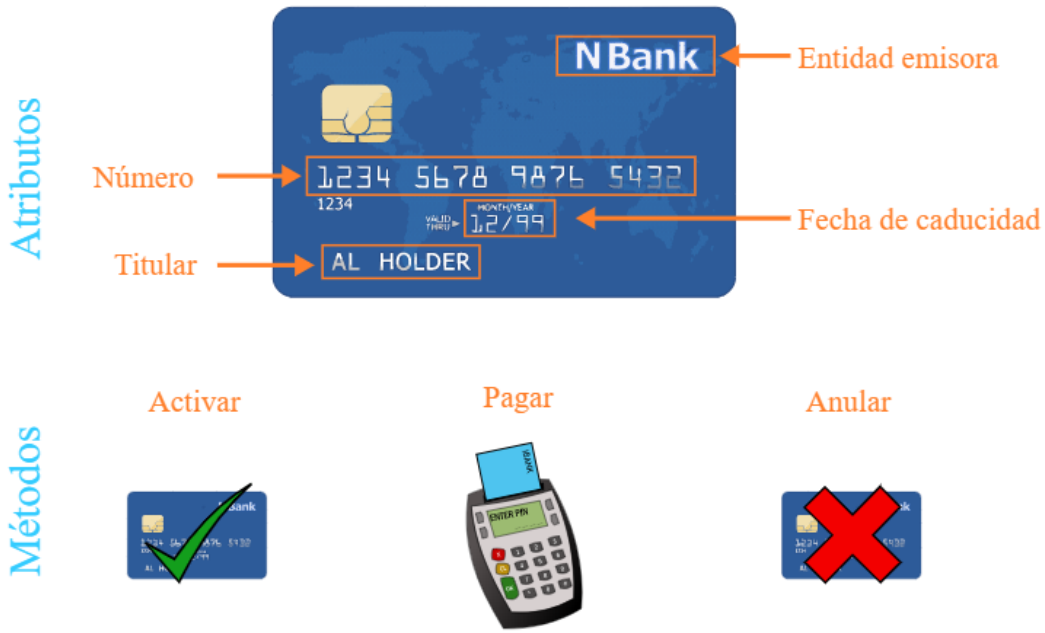
Python también permite *la programación orientada a objetos*, que es un paradigma de programación en la que los datos y las operaciones que pueden realizarse con esos datos se agrupan en unidades lógicas llamadas **objetos**.

Los objetos suelen representar conceptos del dominio del programa, como un estudiante, un coche, un teléfono, etc. Los datos que describen las características del objeto se llaman **atributos** y son la parte estática del objeto, mientras que las operaciones que puede realizar el objeto se llaman **métodos** y son la parte dinámica del objeto.

La programación orientada a objetos permite simplificar la estructura y la lógica de los grandes programas en los que intervienen muchos objetos que interactúan entre si.

Ejemplo. 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.



Programación Orientada a Objetos

Acceso a los atributos y métodos de un objeto

- `dir(objeto)`: Devuelve una lista con los nombres de los atributos y métodos del objeto `objeto`.

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.

En Python los tipos de datos primitivos son también objetos que tienen asociados atributos y métodos.

Ejemplo. Las cadenas tienen un método `upper` que convierte la cadena en mayúsculas. Para aplicar este método a la cadena `c` se utiliza la instrucción `c.upper()`.

```
>>> c = 'Python'
>>> print(c.upper())    # Llamada al método upper del objeto c (cadena)
PYTHON
```

Ejemplo. Las listas tienen un método `append` que convierte añade un elemento al final de la lista. Para aplicar este método a la lista `l` se utiliza la instrucción `l.append(<elemento>)`.

```
>>> l =  [1, 2, 3]
>>> l.append(4)       # Llamada al método append del objeto l (lista)
>>> print(l)
[1, 2, 3, 4]
```

Programación Orientada a Objetos

Clases (class)

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.

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>
```

Los atributos se definen igual que las variables mientras que los métodos se definen igual que las funciones. Tanto unos como otros tienen que estar indentados por 4 espacios en el cuerpo de la clase.

Ejemplo El siguiente código define la clase Saludo sin atributos ni métodos. La palabra reservada pass indica que la clase está vacía.

```
>>> class Saludo:
...     pass      # Clase vacía sin atributos ni métodos.
>>> print(Saludo)
<class '__main__.Saludo'>
```

Es una buena práctica comenzar el nombre de una clase con mayúsculas.

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'>
```

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.

```
>>> class Saludo:
...     pass       # Clase vacía sin atributos ni métodos.
>>> s = Saludo()    # Creación del objeto mediante instanciación de la clase.
>>> s
<__main__.Saludo object at 0x7fcfc7756be0>     # Dirección de memoria donde se crea el objeto
>>> type(s)
<class '__main__.Saludo'>     # Clase del objeto
```

Programación Orientada a Objetos

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.

```
>>> 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('Oscar')
Bienvenido Oscar
```

La razón por la que existe el parámetro self es porque Python traduce la llamada a un método de un objeto objeto.método(parámetros) en la llamada clase.método(objeto, parámetros), es decir, se llama al método definido en la clase del objeto, pasando como primer argumento el propio objeto, que se asocia al parámetro self.

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:
...     def __init__(self, id, cantidad = 0):    # Inicializador
            self.id = id                    # Creación del atributo id
...         self.saldo = cantidad           # Creación del atributo saldo
...         return
...     def mostrar_saldo(self):
...         print('El saldo es', self.saldo, '€')
...         return
>>> t = Tarjeta('1111111111', 1000)    # Creación de un objeto con argumentos
>>> t.muestra_saldo()
El saldo es 1000 €
```

Programación Orientada a Objetos

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:
...     pi = 3.14159                # Atributo de clase
...     def __init__(self, radio):
...         self.radio = radio      # Atributo de instancia
...     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
```

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€
```

Programación Orientada a Objetos

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.

```
>>> class Tarjeta:
...     def __init__(self, id, cantidad = 0):
...         self.id = id
...         self.saldo = cantidad
...         return
...     def mostrar_saldo(self):      # Método de la clase Tarjeta que hereda la clase Tarjeta_descuento
...         print('El saldo es',  self.saldo, '€.')
...         return
...
>>> class Tarjeta_descuento(Tarjeta):
...     def __init__(self, id, descuento, cantidad = 0):
...         self.id = id
...         self.descuento = descuento
...         self.saldo = cantidad
...         return
...     def mostrar_descuento(self):   # Método exclusivo de la clase Tarjeta_descuento
...         print('Descuento de', self.descuento, '% en los pagos.')
...         return
...
>>> t = Tarjeta_descuento('0123456789', 2, 1000)
>>> t.mostrar_saldo()
El saldo es 1000 €.
>>> t.mostrar_descuento()
Descuento de 2 % en los pagos.
```

La principal ventaja de la herencia es que evita la repetición de código y por tanto los programas son más fáciles de mantener.

En el ejemplo de la tarjeta de crédito, el método `mostrar_saldo` solo se define en la clase madre. 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.

Jerarquía de clases

A partir de una clase derivada mediante herencia se pueden crear nuevas clases hijas aplicando de nuevo la herencia. Ello da lugar a una jerarquía de clases que puede representarse como un árbol donde cada clase hija se representa como una rama que sale de la clase madre.

Debido a la herencia, cualquier objeto creado a partir de una clase es una instancia de la clase, pero también lo es de las clases que son ancestros de esa clase en la jerarquía de clases.

El siguiente comando permite averiguar si un objeto es instancia de una clase:

- `isinstance(objeto, clase)`: Devuelve `True` si el objeto `objeto` es una instancia de la clase `clase` y `False` en caso contrario.

```
# Asumiendo la definición de las clases Tarjeta y Tarjeta_descuento anteriores.
>>> t1 = Tarjeta('1111111111', 0)
>>> t2 = t = Tarjeta_descuento('2222222222', 2, 1000)
>>> isinstance(t1, Tarjeta)
True
>>> isinstance(t1, Tarjeta_descuento)
False
>>> isinstance(t2, Tarjeta_descuento)
True
>>> isinstance(t2, Tarjeta)
True
```

Programación Orientada a Objetos

Sobrecarga y polimorfismo

Los objetos de una clase hija heredan los atributos y métodos de la clase madre y, por tanto, a priori tienen 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 presenten 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.

```
>>> class Tarjeta:
...     def __init__(self, id, cantidad = 0):
...         self.id = id
...         self.saldo = cantidad
...         return
...     def mostrar_saldo(self):
...         print('El saldo es {:.2f}€.'.format(self.saldo))
...         return
...     def pagar(self, cantidad):
...         self.saldo -= cantidad
...         return
>>> class Tarjeta_Oro(Tarjeta):
...     def __init__(self, id, descuento, cantidad = 0):
...         self.id = id
...         self.descuento = descuento
...         self.saldo = cantidad
...         return
...     def pagar(self, cantidad):
...         self.saldo -= cantidad * (1 - self.descuento / 1...00)
>>> t1 = Tarjeta('1111111111', 1000)
>>> t2 = Tarjeta_Oro('2222222222', 1, 1000)
>>> t1.pagar(100)
>>> t1.mostrar_saldo()
El saldo es 900.00€.
>>> t2.pagar(100)
>>> t2.mostrar_saldo()
El saldo es 901.00€.
```

Programación Orientada a Objetos

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 maninupar 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.

Resolver un problema siguiendo el paradigma de la programación orientada a objetos requiere un cambio de mentalidad con respecto a como se resuelve utilizando el paradigma de la programación procedimental.

La programación orientada a objetos es más un proceso de modelado, donde se identifican las entidades que intervienen en el problema y su comportamiento, y se definen clases que modelizan esas entidades. Por ejemplo, las entidades que intervienen en el pago con una tarjeta de crédito serían la tarjeta, el terminal de venta, la cuenta corriente vinculada a la tarjeta, el banco, etc. Cada una de ellas daría lugar a una clase.

Después se crean objetos con los datos concretos del problema y se hace que los objetos interactúen entre sí, a través de sus métodos, para resolver el problema. Cada objeto es responsable de una subtarea y colaboran entre ellos para resolver la tarea principal. Por ejemplo, la terminal de venta accede a los datos de la tarjeta y da la orden al banco para que haga un cargo en la cuenta vinculada a la tarjeta.

De esta forma se pueden abordar problemas muy complejos descomponiéndolos en pequeñas tareas que son más fáciles de resolver que el problema principal (*¡divide y vencerás!*).

Programación Orientada a Objetos

Clases/Objetos de Python

- Python es un lenguaje de programación orientado a objetos.
- Casi todo en Python es un objeto, con sus propiedades y métodos.
- Una clase es como un constructor de objetos o un "modelo" para crear objetos.

Crear una clase

Para crear una clase, utilice la palabra clave class:

```
class MyClass:
  x = 5

print(MyClass)
```

Crear objeto

Ahora podemos usar la clase llamada MyClass para crear objetos:

Ejemplo
Cree un objeto llamado p1 e imprima el valor de x:

```
p1 = MyClass()

print(p1.x)
```

La función __init__()

Los ejemplos anteriores son clases y objetos en su forma más simple y no son realmente útiles en aplicaciones de la vida real.

Para comprender el significado de las clases, debemos comprender la función __init__() incorporada.

Todas las clases tienen una función llamada __init__(), que siempre se ejecuta cuando se inicia la clase.

Use la función __init__() para asignar valores a las propiedades del objeto u otras operaciones que sean necesarias cuando se crea el objeto:

Ejemplo
Cree una clase llamada Persona, use la función __init__() para asignar valores para el nombre y la edad:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Nota: La __init__()función se llama automáticamente cada vez que la clase se usa para crear un nuevo objeto.

La función __str__()

La función __str__() controla lo que se debe devolver cuando el objeto de clase se representa como una cadena.

Si la función __str__() no está configurada, se devuelve la representación de cadena del objeto:

Ejemplo
La representación de cadena de un objeto SIN la función __str__():

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

p1 = Person("John", 36)

print(p1)
```

Ejemplo
La representación de cadena de un objeto CON la función __str__():

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

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)
```

Métodos de objeto

Los objetos también pueden contener métodos. Los métodos en los objetos son funciones que pertenecen al objeto.

Vamos a crear un método en la clase Person:

Ejemplo

Inserte una función que imprima un saludo y ejecútela en el objeto p1:

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Nota: El parámetro selfes una referencia a la instancia actual de la clase y se utiliza para acceder a las variables que pertenecen a la clase.

El auto parámetro

El parámetro selfes una referencia a la instancia actual de la clase y se utiliza para acceder a las variables que pertenecen a la clase.

No tiene que ser nombrado self, puedes llamarlo como quieras, pero tiene que ser el primer parámetro de cualquier función en la clase:

Ejemplo

Usa las palabras mysillyobject y abc en lugar de self :

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Modificar propiedades de objetos

Puede modificar propiedades en objetos como este:

Ejemplo
Establezca la edad de p1 en 40:

```
p1.age = 40
```

Eliminar propiedades de objeto

Puede eliminar propiedades en objetos usando la palabra clave del:

Ejemplo
Elimine la propiedad de edad del objeto p1:

```
del p1.age
```

La declaración de pass

las definiciones class no pueden estar vacías, pero si por algún motivo tiene una definición classs en contenido, introdúzcala en la declaración passpara evitar que se produzca un error.

Ejemplo

```
class Person:
  pass
```

.

Python es un lenguaje orientado a objetos

En Python todo es un objeto. Cuando creas una variable y le asignas un valor entero, ese valor es un objeto; una función es un objeto; las listas, tuplas, diccionarios, conjuntos, … son objetos; una cadena de caracteres es un objeto. Y así podría seguir indefinidamente.

Pero, ¿por qué es tan importante la programación orientada a objetos? Bien, este tipo de programación introduce un nuevo paradigma que nos permite encapsular y aislar datos y operaciones que se pueden realizar sobre dichos datos.

Clases y objetos en Python

Básicamente, una clase es una entidad que define una serie de elementos que determinan un estado (datos) y un comportamiento (operaciones sobre los datos que modifican su estado).

Por su parte, un objeto es una concreción o instancia de una clase.

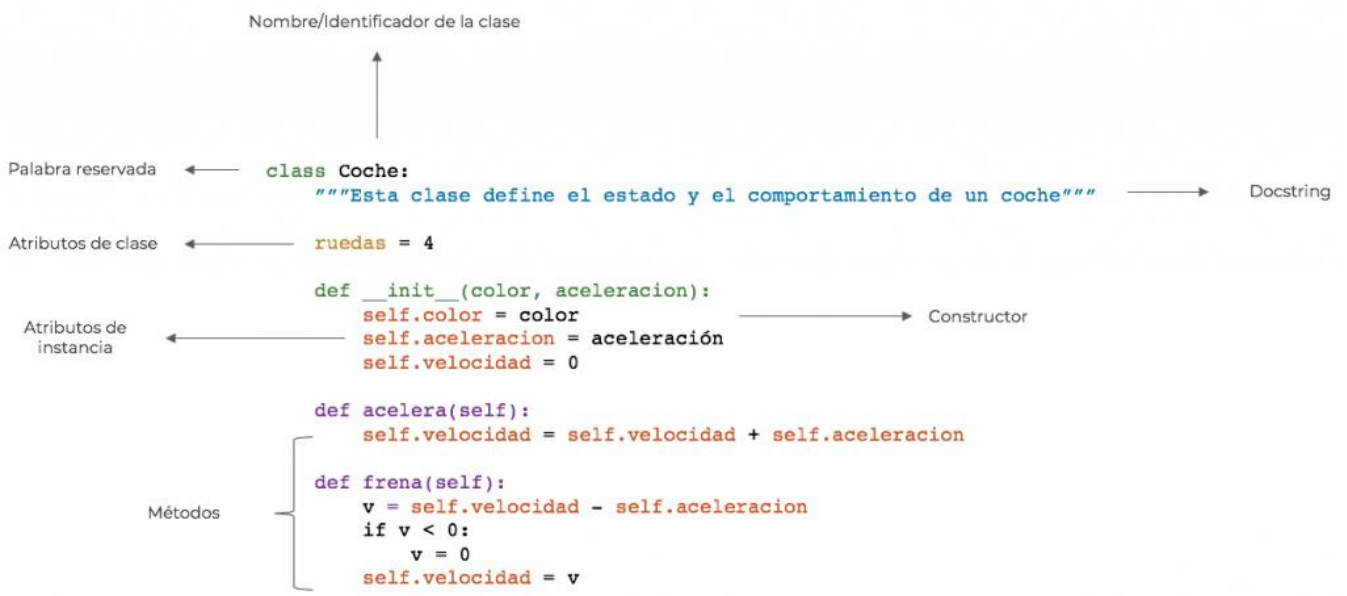
Seguro que si te digo que te imagines un coche, en tu mente comienzas a visualizar la carrocería, el color, las ruedas, el volante, si es diésel o gasolina, el color de la tapicería, si es manual o automático, si acelera o va marcha atrás, etc.

Pues todo lo que acabo de describir viene a ser una clase y cada uno de los de coches que has imaginado, serían objetos de dicha clase.

Como te decía, una clase engloba datos y funcionalidad. Cada vez que se define una clase en Python, se crea a su vez un tipo nuevo (¿recuerdas? tipo `int`, `float`, `str`, `list`, `tuple`, … todos ellos están definidos en una clase).

Para definir una clase en Python se utiliza la palabra reservada `class`. El siguiente esquema visualiza los elementos principales que componen una clase. Todos ellos los iremos viendo con detenimiento en las siguientes secciones:

Programación Orientada a Objetos



El esquema anterior define la clase `Coche` . Dicha clase establece una serie datos, como `ruedas`, `color`, `aceleración` o `velocidad` y las operaciones `acelera()` y `frena()`.

Cuando se crea una variable de tipo `Coche`, realmente se está instanciando un objeto de dicha clase. En el siguiente ejemplo se crean dos objetos de tipo `Coche`:

```
>>> c1 = Coche('rojo', 20)
>>> print(c1.color)
rojo
>>> print(c1.ruedas)
4
>>> c2 = Coche('azul', 30)
>>> print(c2.color)
azul
>>> print(c2.ruedas)
4
```

`c1` y `c2` son objetos, objetos cuya clase es `Coche`. Ambos objetos pueden `acelerar` y `frenar`, porque su clase define estas operaciones y tienen un `color`, porque la clase `Coche` también define este *dato*. Lo que ocurre es que `c1` es de `color` *rojo*, mientras que `c2` es de `color` *azul*.

NOTA: Es una convención utilizar la notación CamelCase para los nombres de las clases. Esto es, la primera letra de cada palabra del nombre está en mayúsculas y el resto de letras se mantienen en minúsculas.

Constructor de una clase en Python

Para crear un objeto de una clase determinada, es decir, *instanciar* una clase, se usa el nombre de la clase y a continuación se añaden paréntesis (como si se llamara a una función).

```
obj = MiClase()
```

El código anterior crea una nueva instancia de la clase `MiClase` y asigna dicho objeto a la variable `obj`. Esto crea un objeto *vacío*, sin estado.

Sin embargo, hay clases (como nuestra clase `Coche`) que deben o necesitan crear instancias de objetos con un estado inicial.

Esto se consigue implementando el método especial `__init__()`. Este método es conocido como el constructor de la clase y se invoca cada vez que se instancia un nuevo objeto.

El método `__init__()` establece un primer parámetro especial que se suele llamar `self`. Pero puede especificar otros parámetros siguiendo las mismas reglas que cualquier otra función.

En nuestro caso, el constructor de la clase coche es el siguiente:

```
def __init__(self, color, aceleracion):
    self.color = color
    self.aceleracion = aceleracion
    self.velocidad = 0
```

Como puedes observar, además del parámetro `self`, define los parámetros `color` y `aceleracion`, que determinan el estado inicial de un objeto de tipo `Coche`.

En este caso, para instanciar un objeto de tipo coche, debemos pasar como argumentos el color y la aceleración como vimos en el ejemplo:

```
c1 = Coche('rojo', 20)
```

IMPORTANTE: A diferencia de otros lenguajes, en los que está permitido implementar más de un constructor, en Python solo se puede definir un método `__init__()`.

Atributos, atributos de datos y métodos

Una vez que sabemos qué es un objeto, tengo que decirte que la única operación que pueden realizar los objetos es referenciar a sus atributos por medio del operador `.`

Como habrás podido apreciar, un objeto tiene dos tipos de atributos: *atributos de datos* y *métodos*.

- Los atributos de datos definen el estado del objeto. En otros lenguajes son conocidos simplemente como atributos o miembros.
- Los métodos son las funciones definidas dentro de la clase.

Siguiendo con nuestro ejemplo de la clase `Coche`, vamos a crear el siguiente objeto:

```
>>> c1 = Coche('rojo', 20)
>>> print(c1.color)
rojo
>>> print(c1.velocidad)
0
>>> c1.acelera()
>>> print(c1.velocidad)
20
```

En la *línea 2* del código anterior, el objeto `c1` está referenciando al atributo de dato `color` y en la *línea 4* al atributo `velocidad`. Sin embargo, en la *línea 6* se referencia al método `acelera()`. Llamar a este método tiene una implicación como puedes observar y es que modifica el estado del objeto, dado que se incrementa su velocidad. Este hecho lo puedes apreciar cuando se vuelve a referenciar al atributo `velocidad` en la *línea 7*.

Atributos de datos

A diferencia de otros lenguajes, los atributos de datos no necesitan ser declarados previamente. Un objeto los crea del mismo modo en que se crean las variables en Python, es decir, cuando les asigna un valor por primera vez.

El siguiente código es un ejemplo de ello:

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 10)
>>> print(c1.color)
rojo
>>> print(c2.color)
azul
>>> c1.marchas = 6
>>> print(c1.marchas)
6
>>> print(c2.marchas)
Traceback (most recent call last):
File "<input>", line 1, in <module>
AttributeError: 'Coche' object has no attribute 'marchas'
```

Los objetos `c1` y `c2` pueden referenciar al atributo `color` porque está definido en la clase `Coche`. Sin embargo, solo el objeto `c1` puede referenciar al atributo `marchas` a partir de la *línea 7*, porque inicializa dicho atributo en esa línea. Si el objeto `c2` intenta referenciar al mismo atributo, como no está definido en la clase y tampoco lo ha inicializado, el intérprete lanzará un error.

Métodos

Los métodos son las funciones que se definen dentro de una clase y que, por consiguiente, pueden ser referenciadas por los objetos de dicha clase. Sin embargo, realmente los métodos son algo más.

Si te has fijado bien, pero bien de verdad, habrás observado que las funciones `acelera()` y `frena()` definen un parámetro `self`.

```
def acelera(self):
    self.velocidad = self.velocidad + self.aceleracion
```

No obstante, cuando se usan dichas funciones no se pasa ningún argumento. ¿Qué está pasando? Pues que `acelera()` está siendo utilizada como un método por los objetos de la clase `Coche`, de tal manera que cuando un objeto referencia a dicha función, realmente pasa su propia referencia como primer parámetro de la función.

NOTA: Por convención, se utiliza la palabra `self` para referenciar a la instancia actual en los métodos de una clase.

Sabiendo esto, podemos entender, por ejemplo, por qué todos los objetos de tipo `Coche` pueden referenciar a los atributos de datos `velocidad` o `color`. Son inicializados para cada objeto en el método `__init__()`.

Del mismo modo, el siguiente ejemplo muestra dos formas diferentes y equivalentes de llamar al método `acelera()`:

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 20)
>>> c1.acelera()
>>> Coche.acelera(c2)
>>> print(c1.velocidad)
20
>>> print(c2.velocidad)
20
```

Para la clase `Coche`, `acelera()` es una función. Sin embargo, para los objetos de la clase `Coche`, `acelera()` es un método.

```
>>> print(Coche.acelera)
<function Coche.acelera at 0x10c60b560>
>>> print(c1.acelera)
<bound method Coche.acelera of <__main__.Coche object at 0x10c61efd0>>
```

Atributos de clase y atributos de instancia

Una clase puede definir dos tipos diferentes de atributos de datos: atributos de clase y atributos de instancia.

- Los atributos de clase son atributos compartidos por todas las instancias de esa clase.
- Los atributos de instancia, por el contrario, son únicos para cada uno de los objetos pertenecientes a dicha clase.

En el ejemplo de la clase `Coche`, `ruedas` se ha definido como un atributo de clase, mientras que `color`, `aceleracion` y `velocidad` son atributos de instancia.

Para referenciar a un atributo de clase se utiliza, generalmente, el nombre de la clase. Al modificar un atributo de este tipo, los cambios se verán reflejados en todas y cada una las instancias.

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 20)
>>> print(c1.color)
rojo
>>> print(c2.color)
azul
>>> print(c1.ruedas) # Atributo de clase
4
>>> print(c2.ruedas) # Atributo de clase
4
>>> Coche.ruedas = 6 # Atributo de clase
>>> print(c1.ruedas) # Atributo de clase
6
>>> print(c2.ruedas) # Atributo de clase
6
```

Si un objeto modifica un atributo de clase, lo que realmente hace es crear un atributo de instancia con el mismo nombre que el atributo de clase.

```
>>> c1 = Coche('rojo', 20)
>>> c2 = Coche('azul', 20)
>>> print(c1.color)
rojo
>>> print(c2.color)
azul
```

```
>>> c1.ruedas = 6 # Crea el atributo de instancia ruedas
>>> print(c1.ruedas)
6
>>> print(c2.ruedas)
4
>>> print(Coche.ruedas)
4
```

Herencia en Python

En programación orientada a objetos, la herencia es la capacidad de reutilizar una clase extendiendo su funcionalidad. Una clase que hereda de otra puede añadir nuevos atributos, ocultarlos, añadir nuevos métodos o redefinirlos.

En Python, podemos indicar que una clase hereda de otra de la siguiente manera:

```
class CocheVolador(Coche):
    ruedas = 6
    def __init__(self, color, aceleracion, esta_volando=False):
        super().__init__(color, aceleracion)
        self.esta_volando = esta_volando
    def vuela(self):
        self.esta_volando = True
    def aterriza(self):
        self.esta_volando = False
```

Como puedes observar, la clase `CocheVolador` hereda de la clase `Coche`. En Python, el nombre de la clase padre se indica entre paréntesis a continuación del nombre de la clase hija.

La clase `CocheVolador` redefine el atributo de clase `ruedas`, estableciendo su valor a `6` e implementa dos métodos nuevos: `vuela()` y `aterriza()`.

Fíjate ahora en la primera línea del método `__init__()`. En ella aparece la función `super()`. Esta función devuelve un objeto temporal de la superclase que permite invocar a los métodos definidos en la misma. Lo que está ocurriendo es que se está redefiniendo el método `__init__()` de la clase hija usando la funcionalidad del método de la clase padre.

Como la clase `Coche` es la que define los atributos `color` y `aceleracion`, estos se pasan al constructor de la clase padre y, a continuación, se crea el atributo de instancia `esta_volando` solo para objetos de la clase `CocheVolador`.

Al utilizar la herencia, todos los atributos (atributos de datos y métodos) de la clase padre también pueden ser referenciados por objetos de las clases hijas. Al revés no ocurre lo mismo.

Veamos todo esto con un ejemplo:

```
>>> c = Coche('azul', 10)
>>> cv1 = CocheVolador('rojo', 60)
>>> print(cv1.color)
rojo
>>> print(cv1.esta_volando)
False
>>> cv1.acelera()
>>> print(cv1.velocidad)
60
>>> print(CocheVolador.ruedas)
6
>>> print(c.esta_volando)
Traceback (most recent call last):
File "<input>", line 1, in <module>
AttributeError: 'Coche' object has no attribute 'esta_volando'
```

NOTA: Cuando no se indica, toda clase Python hereda implícitamente de la clase `object`, de tal modo que `class MiClase` es lo mismo que `class MiClase(object)`.

Las funciones isinstance() e issubclass()

Como ya vimos en otros tutoriales, la función incorporada `type()` devuelve el tipo o la clase a la que pertenece un objeto. En nuestro caso, si ejecutamos `type()` pasando como argumento un objeto de clase `Coche` o un objeto de clase `CocheVolador` obtendremos lo siguiente:

```
>>> c = Coche('rojo', 20)
>>> type(c)
<class 'objetos.Coche'>
>>> cv = CocheVolador('azul', 60)
>>> type(cv)
<class 'objetos.CocheVolador'>
```

Sin embargo, Python incorpora otras dos funciones que pueden ser de utilidad cuando se quiere conocer el tipo de una clase. Son: `isinstance()` e `issubclass()`.

- `isinstance(objeto, clase)` devuelve `True` si `objeto` es de la clase `clase` o de una de sus clases hijas. Por tanto, un objeto de la clase `CocheVolador` es instancia de `CocheVolador` pero también lo es de `Coche`. Sin embargo, un objeto de la clase `Coche` nunca será instancia de la clase `CocheVolador`.
- `issubclass(clase, claseinfo)` comprueba la herencia de clases. Devuelve `True` en caso de que `clase` sea una subclase de `claseinfo`, `False` en caso contrario. `claseinfo` puede ser una clase o una tupla de clases.

```
>>> c = Coche('rojo', 20)
>>> cv = CocheVolador('azul', 60)
>>> isinstance(c, Coche)
True
isinstance(cv, Coche)
True
>>> isinstance(c, CocheVolador)
False
>>> isinstance(cv, CocheVolador)
True
>>> issubclass(CocheVolador, Coche)
True
>>> issubclass(Coche, CocheVolador)
False
```

Herencia múltiple en Python

Python es un lenguaje de programación que permite herencia múltiple. Esto quiere decir que una clase puede heredar de más de una clase a la vez.

```
class A:
    def print_a(self):
        print('a')
class B:
    def print_b(self):
        print('b')

class C(A, B):
    def print_c(self):
        print('c')
c = C()
c.print_a()
c.print_b()
c.print_c()
```

El script anterior dará como resultado

```
a
b
c
```

Encapsulación: atributos privados

E*ncapsulación* (o *encapsulamiento*), en programación orientada a objetos, hace referencia a la capacidad que tiene un objeto de ocultar su estado, de manera que sus datos solo se puedan modificar por medio de las operaciones (métodos) que ofrece.

Si vienes de otros lenguajes de programación, quizá te haya resultado raro que no haya mencionado nada sobre atributos públicos o privados.

Bien, por defecto, en Python, todos los atributos de una clase (atributos de datos y métodos) son públicos. Esto quiere decir que desde un código que use la clase, se puede acceder a todos los atributos y métodos de dicha clase.

No obstante, hay una forma de indicar en Python que un atributo, ya sea un dato o un método, es interno a una clase y no se debería utilizar fuera de ella. Algo así como los miembros privados de otros lenguajes. Esto es usando el carácter guión bajo `_atributo` antes del nombre del atributo que queramos ocultar.

En cualquier caso, el atributo seguirá siendo accesible desde fuera de la clase, pero el programador está indicando que es privado y no debería utilizarse porque no se sabe qué consecuencias puede tener.

También es posible usar un doble guión bajo `__atributo`. Esto hace que el identificador sea literalmente reemplazado por el texto `_Clase__atributo`, donde `Clase` es el nombre de la clase actual.

Un ejemplo nunca está de más.

```
class A:
    def __init__(self):
        self._contador = 0 # Este atributo es privado
    def incrementa(self):
        self._contador += 1
    def cuenta(self):
        return self._contador


class B(object):
    def __init__(self):
        self.__contador = 0 # Este atributo es privado
    def incrementa(self):
        self.__contador += 1
    def cuenta(self):
        return self.__contador
```

En el ejemplo anterior, la clase `A` define el atributo privado `_contador`. Un ejemplo de uso de la clase sería el siguiente:

```
>>> a = A()
>>> a.incrementa()
>>> a.incrementa()
>>> a.incrementa()
>>> print(a.cuenta())
3
>>> print(a._contador)
3
```

Como puedes observar, es posible acceder al atributo privado, aunque no se debiera.

En cambio, la clase `B` define el atributo privado `__contador` anteponiendo un doble guión bajo. El resultado de hacer el mismo experimento cambia:

```
>>> b = B()
>>> b.incrementa()
>>> b.incrementa()
>>> print(b.cuenta())
2
>>> print(b.__contador)
Traceback (most recent call last):
File "<input>", line 1, in <module>
AttributeError: 'B' object has no attribute '__contador'
>>> print(b._B__contador)
2
```

Si te fijas, no se puede acceder al atributo `__contador` fuera de la clase. Este identificador se ha sustituido por `_B__contador`.

Polimorfismo

Polimorfismo es la capacidad de una entidad de referenciar en tiempo de ejecución a instancias de diferentes clases.

Imagina que tenemos las siguientes clases que representan animales:

```
class Perro:
    def sonido(self):
        print('Guauuuuu!!!')
class Gato:
    def sonido(self):
        print('Miaaauuuu!!!')
class Vaca:
    def sonido(self):
        print('Múuuuuuuu!!!')
```

Las tres clases implementan un método llamado `sonido()`. Ahora observa el siguiente script:

```
def a_cantar(animales):
    for animal in animales:
        animal.sonido()
    if __name__ == '__main__':
        perro = Perro()
        gato = Gato()
        gato_2 = Gato()
        vaca = Vaca()
        perro_2 = Perro()
        granja = [perro, gato, vaca, gato_2, perro_2]
        a_cantar(granja)
```

En él se ha definido una función llamada `a_cantar()`. La variable `animal` que se crea dentro del bucle `for` de la función es *polimórfica*, ya que en tiempo de ejecución hará referencia a objetos de las clases `Perro`, `Gato` y `Vaca`. Cuando se invoque al método `sonido()`, se llamará al método correspondiente de la clase a la que pertenezca cada uno de los animales.

Programación Orientada a Objetos

Las cuatro cláusulas principales en la programación orientada a objetos (POO) son:

Encapsulación: La encapsulación es el principio de la POO que permite ocultar los detalles internos de una clase y proporcionar una interfaz para interactuar con los objetos. Se logra mediante el uso de modificadores de acceso como "public", "private" y "protected" para controlar el acceso a los miembros de una clase (atributos y métodos). La encapsulación ayuda a mantener la integridad de los datos y promueve la reutilización del código.

Herencia: La herencia es un mecanismo en la POO que permite crear nuevas clases basadas en clases existentes. La clase existente se denomina clase base o superclase, y la nueva clase se denomina clase derivada o subclase. La herencia permite la reutilización de código y la extensión de funcionalidades. La clase derivada hereda los atributos y métodos de la clase base, y puede agregar nuevos atributos y métodos, o modificar los existentes.

Polimorfismo: El polimorfismo es la capacidad de un objeto de tomar muchas formas diferentes. En la POO, el polimorfismo permite que objetos de diferentes clases respondan de manera diferente a un mismo mensaje o método. Esto se logra mediante el uso de herencia y la implementación de métodos con el mismo nombre en diferentes clases. El polimorfismo facilita la flexibilidad y modularidad del código.

Abstracción: La abstracción es el proceso de identificar las características esenciales de un objeto o sistema y representarlas de manera simplificada en un modelo. En la POO, la abstracción se logra mediante la creación de clases abstractas e interfaces. Una clase abstracta define características comunes para un conjunto de clases relacionadas, mientras que una interfaz define un conjunto de métodos que una clase concreta debe implementar. La abstracción ayuda a ocultar los detalles innecesarios y enfocarse en los aspectos importantes de un sistema.

Ejemplos Herencia

```
# Definición de la clase base (superclase)
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def comer(self):
        print(f"{self.nombre} está comiendo.")

    def dormir(self):
        print(f"{self.nombre} está durmiendo.")


# Definición de una clase derivada (subclase) que hereda de Animal
class Perro(Animal):
    def __init__(self, nombre, raza):
        super().__init__(nombre)
        self.raza = raza

    def ladrar(self):
        print(f"{self.nombre} está ladrando.")


# Creación de instancias de las clases
animal_generico = Animal("Animal genérico")
animal_generico.comer()
animal_generico.dormir()

perro1 = Perro("Fido", "Labrador")
perro1.comer()
perro1.dormir()
perro1.ladrar()
print(f"Raza del perro: {perro1.raza}")
```

En este ejemplo, tenemos una clase base llamada "Animal" que tiene dos métodos: "comer" y "dormir". Luego, creamos una clase derivada llamada "Perro" que hereda de la clase "Animal" y agrega un método adicional llamado "ladrar". La clase "Perro" también tiene un atributo adicional llamado "raza".

Creamos instancias de las clases "Animal" y "Perro" y llamamos a los métodos correspondientes. Observa cómo el objeto "perro1" tiene acceso tanto a los métodos de la clase base "Animal" como a su propio método "ladrar" y su atributo "raza".

La salida del programa será:

```
Animal genérico está comiendo.
Animal genérico está durmiendo.
Fido está comiendo.
Fido está durmiendo.
Fido está ladrando.
Raza del perro: Labrador
```

Otro Ejemplo

```
# Definición de la clase base (superclase)
class Vehiculo:
    def __init__(self, nombre):
        self.nombre = nombre

    def moverse(self):
        print(f"{self.nombre} está en movimiento.")


# Definición de una clase derivada (subclase) que hereda de Vehiculo
class Automovil(Vehiculo):
    def __init__(self, nombre, color):
        super().__init__(nombre)
        self.color = color

    def sonar_bocina(self):
        print("¡Beep beep!")


# Creación de instancias de las clases
auto_rojo = Automovil("Coche Rojo", "Rojo")
auto_rojo.moverse()
auto_rojo.sonar_bocina()
print(f"Color del coche: {auto_rojo.color}")
```

Programación Orientada a Objetos

En este ejemplo, tenemos una clase base llamada "Vehiculo" que tiene un método llamado "moverse". Luego, creamos una clase derivada llamada "Automovil" que hereda de la clase "Vehiculo" y agrega un método adicional llamado "sonar_bocina". La clase "Automovil" también tiene un atributo adicional llamado "color".

Creamos una instancia de la clase "Automovil" llamada "auto_rojo" y llamamos a los métodos correspondientes. Observa cómo el objeto "auto_rojo" tiene acceso tanto al método de la clase base "Vehiculo" como a su propio método "sonar_bocina" y su atributo "color".

La salida del programa será:

```
Coche Rojo está en movimiento.
¡Beep beep!
Color del coche: Rojo
```

Ejemplo Polimorfismo

```
# Definición de la clase base (superclase)
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hacer_sonido(self):
        pass


# Definición de clases derivadas (subclases) que heredan de Animal
class Perro(Animal):
    def hacer_sonido(self):
        return "Woof woof!"


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


# Función que utiliza el polimorfismo
def hacer_ruido(animal):
    print(animal.hacer_sonido())


# Creación de instancias de las clases
perro = Perro("Fido")
gato = Gato("Misifu")

# Llamada a la función hacer_ruido con diferentes objetos
hacer_ruido(perro)
hacer_ruido(gato)
```

En este ejemplo, tenemos una clase base llamada "Animal" con un método llamado "hacer_sonido" que no tiene implementación en la clase base.

Luego, creamos dos clases derivadas llamadas "Perro" y "Gato", que heredan de la clase "Animal". Ambas clases implementan su propio método "hacer_sonido" que devuelve el sonido característico de cada animal.

Finalmente, tenemos una función llamada "hacer_ruido" que toma un objeto de tipo "Animal" como argumento y llama al método "hacer_sonido" del objeto. Esta función demuestra el polimorfismo, ya que puede funcionar con diferentes objetos (en este caso, un perro y un gato) y producir resultados diferentes según el tipo de objeto.

La salida del programa será:

```
Woof woof!

Miau miau!
```

Otro ejemplo

```
# Definición de la clase base (superclase)
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hacer_sonido(self):
        pass


# Definición de clases derivadas (subclases) que heredan de Animal
class Perro(Animal):
    def hacer_sonido(self):
        return "¡Woof woof!"


class Gato(Animal):
    def hacer_sonido(self):
        return "¡Miau miau!"


# Función que utiliza el polimorfismo
def hacer_ruido(animal):
    print(f"{animal.nombre} dice: {animal.hacer_sonido()}")


# Creación de instancias de las clases
perro = Perro("Fido")
gato = Gato("Misifu")

# Llamada a la función hacer_ruido con diferentes objetos
hacer_ruido(perro)
hacer_ruido(gato)
```

En este ejemplo, tenemos una clase base llamada "Animal" con un método llamado "hacer_sonido" que no tiene implementación en la clase base.

Luego, creamos dos clases derivadas llamadas "Perro" y "Gato", que heredan de la clase "Animal". Ambas clases implementan su propio método "hacer_sonido" que devuelve el sonido característico de cada animal.

Después, tenemos una función llamada "hacer_ruido" que toma un objeto de tipo "Animal" como argumento. La función imprime el nombre del animal junto con el sonido que hace, utilizando el método "hacer_sonido" del objeto. Esta función demuestra el polimorfismo, ya que puede funcionar con diferentes objetos (en este caso, un perro y un gato) y producir resultados diferentes según el tipo de objeto.

La salida del programa será:

```
Fido dice: ¡Woof woof!
Misifu dice: ¡Miau miau!
```

Ejemplos encapsulamiento

```
# Definición de la clase
class CuentaBancaria:
    def __init__(self, titular, saldo_inicial):
        self.__titular = titular
        self.__saldo = saldo_inicial

    def depositar(self, cantidad):
        self.__saldo += cantidad
        print(f"Se han depositado {cantidad} unidades.")

    def retirar(self, cantidad):
        if self.__saldo >= cantidad:
            self.__saldo -= cantidad
            print(f"Se han retirado {cantidad} unidades.")
        else:
            print("Fondos insuficientes.")

    def consultar_saldo(self):
        print(f"Saldo actual: {self.__saldo}")


# Creación de una instancia de la clase
cuenta = CuentaBancaria("Juan", 1000)

# Llamada a los métodos públicos
cuenta.consultar_saldo()
cuenta.depositar(500)
cuenta.consultar_saldo()
cuenta.retirar(200)
cuenta.consultar_saldo()

# Intento de acceso directo a los atributos privados (encapsulados)
print(cuenta.__titular)  # Esto generará un error
print(cuenta.__saldo)    # Esto generará un error
```

Programación Orientada a Objetos

@mihifidem
If you want to master something, teach it.

En este ejemplo, tenemos una clase llamada CuentaBancaria que representa una cuenta bancaria con atributos privados (__titular y __saldo) que están encapsulados, es decir, no se pueden acceder directamente desde fuera de la clase.

La clase tiene métodos públicos para realizar operaciones en la cuenta, como depositar, retirar y consultar_saldo. Estos métodos pueden acceder a los atributos privados y modificarlos de manera controlada.

En el programa principal, creamos una instancia de la clase CuentaBancaria y realizamos algunas operaciones, como consultar el saldo, depositar y retirar dinero.

Sin embargo, cuando intentamos acceder directamente a los atributos privados (__titular y __saldo), obtenemos errores porque están encapsulados y no se puede acceder a ellos desde fuera de la clase.

La salida del programa será:

```
Saldo actual: 1000
Se han depositado 500 unidades.
Saldo actual: 1500
Se han retirado 200 unidades.
Saldo actual: 1300
AttributeError: 'CuentaBancaria' object has no attribute '__titular'
AttributeError: 'CuentaBancaria' object has no attribute '__saldo'
```

Otro ejemplo

```
# Definición de la clase
class Juguete:
    def __init__(self, nombre):
        self.__nombre = nombre

    def jugar(self):
        print(f"¡{self.__nombre} está siendo jugado!")

    def obtener_nombre(self):
        return self.__nombre


# Creación de una instancia de la clase
juguete = Juguete("Carrito")

# Llamada a los métodos públicos
print(f"Juguete: {juguete.obtener_nombre()}")
juguete.jugar()

# Intento de acceso directo al atributo privado (encapsulado)
print(juguete.__nombre)  # Esto generará un error
```

Programación Orientada a Objetos

En este ejemplo, tenemos una clase llamada Juguete que representa un juguete con un atributo privado (__nombre) que está encapsulado, es decir, no se puede acceder directamente desde fuera de la clase.

La clase tiene métodos públicos, como jugar y obtener_nombre, que permiten interactuar con el juguete de manera controlada.

En el programa principal, creamos una instancia de la clase Juguete y llamamos a los métodos públicos para jugar con el juguete y obtener su nombre.

Sin embargo, cuando intentamos acceder directamente al atributo privado (__nombre), obtenemos un error porque está encapsulado y no se puede acceder a él desde fuera de la clase.

La salida del programa será:

```
Juguete: Carrito
¡Carrito está siendo jugado!
AttributeError: 'Juguete' object has no attribute '__nombre'
```

Ejemplos Abstracción

```
from abc import ABC, abstractmethod

# Definición de la clase abstracta (interfaz)
class FiguraGeometrica(ABC):
    @abstractmethod
    def calcular_area(self):
        pass

    @abstractmethod
    def calcular_perimetro(self):
        pass


# Definición de una clase concreta que implementa FiguraGeometrica
class Rectangulo(FiguraGeometrica):
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def calcular_area(self):
        return self.base * self.altura

    def calcular_perimetro(self):
        return 2 * (self.base + self.altura)


# Creación de una instancia de la clase
rectangulo = Rectangulo(5, 3)

# Llamada a los métodos de la clase concreta
print("Rectángulo:")
print("Área:", rectangulo.calcular_area())
print("Perímetro:", rectangulo.calcular_perimetro())
```

En este ejemplo, tenemos una clase abstracta llamada FiguraGeometrica, que sirve como una interfaz para representar figuras geométricas. La clase abstracta define dos métodos abstractos: calcular_area y calcular_perimetro. Estos métodos deben ser implementados en las clases concretas que heredan de la clase abstracta.

Luego, tenemos una clase concreta llamada Rectangulo que hereda de FiguraGeometrica. Esta clase implementa los métodos abstractos para calcular el área y el perímetro de un rectángulo.

En el programa principal, creamos una instancia de la clase Rectangulo y llamamos a los métodos calcular_area y calcular_perimetro. Estos métodos están implementados en la clase Rectangulo, pero se definen a través de la abstracción de la clase FiguraGeometrica.

La salida del programa será:

```
Rectángulo:
Área: 15
Perímetro: 16
```

Otros ejemplo

```
from abc import ABC, abstractmethod

# Definición de la clase abstracta (interfaz)
class Animal(ABC):
    @abstractmethod
    def hacer_sonido(self):
        pass

    @abstractmethod
    def moverse(self):
        pass


# Definición de una clase concreta que implementa Animal
class Perro(Animal):
    def hacer_sonido(self):
        return "¡Guau guau!"

    def moverse(self):
        return "El perro corre."


# Creación de una instancia de la clase
perro = Perro()

# Llamada a los métodos de la clase concreta
print("Perro:")
print(perro.hacer_sonido())
print(perro.moverse())
```

En este ejemplo, tenemos una clase abstracta llamada Animal, que representa la interfaz común para todos los animales. La clase abstracta define dos métodos abstractos: hacer_sonido y moverse. Estos métodos deben ser implementados en las clases concretas que heredan de la clase abstracta.

Luego, tenemos una clase concreta llamada Perro que hereda de Animal. Esta clase implementa los métodos abstractos para hacer el sonido característico de un perro y describir cómo se mueve.

En el programa principal, creamos una instancia de la clase Perro y llamamos a los métodos hacer_sonido y moverse. Estos métodos están implementados en la clase Perro, pero se definen a través de la abstracción de la clase Animal.

La salida del programa será:

```
Perro:
¡Guau guau!
El perro corre.
```

Clases abstractas en Python

Un concepto importante en programación orientada a objetos es el de las clases abstractas. Unas clases en las que se pueden definir tanto métodos como propiedades, pero que no pueden ser instancias 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 puede ser instanciadas**. Simplemente proporciona una interfaz para las subclases derivadas y 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 define una interfaz común para las subclases. Proporciona atributos y métodos comunes para todas las subclases evitando así la necesidad de duplicar código. Imponiendo además lo 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 modulo abc (Abstract Base Classes). Un módulo que se encuentra en la librería estándar del lenguaje, por lo que no es necesario instalar. Así para definir una clase privada 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 que si la clase no hereda de ABC o contiene por lo menos un método abstracto, Python permitirá instancias las clases.

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

Animal()
```

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

Animal()
```

Métodos en las subclases

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

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

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

class Gato(Animal):
    def mover(self):
        print('Mover gato')

g = Gato() # Error
```

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.

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

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

class Gato(Animal):
  def mover(self):
    print('Mover gato')

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

g = Gato()
g.mover()
g.comer()
```

```
Mover gato
Animal come
Gato come
```