



Jul. 2022

Unidad 3:



Objetivos de aprendizaje

Ser capaces de:

- Explicar diferentes conceptos de POO y la importancia de la misma
- Instanciar clases
- Definir métodos de instancia y pasarles argumentos
- Declarar atributos y métodos de clase
- Sobrescribir métodos
- Implementar la herencia múltiple

Introducción

- La programación orientada a objetos (POO) es un paradigma de programación basado en el concepto de objetos
- Los "objetos" son estructuras de datos que contienen datos en forma de atributos y funciones conocidos como métodos

Kindler, E.; Krivy, I. (2011).

POO según Steve Jobs

En "[Rolling Stone](#)" 16 de junio de 1994

"Los objetos son como las personas.

Son cosas vivas,
que respiran,
que tienen conocimientos en su interior
sobre cómo hacer las cosas
y que tienen memoria en su interior
para poder recordar cosas.

Y en lugar de interactuar con ellos a un nivel muy bajo,
se interactúa con ellos a un nivel de abstracción muy alto..."

Steve Jobs in 1994: The Rolling Stone Interview

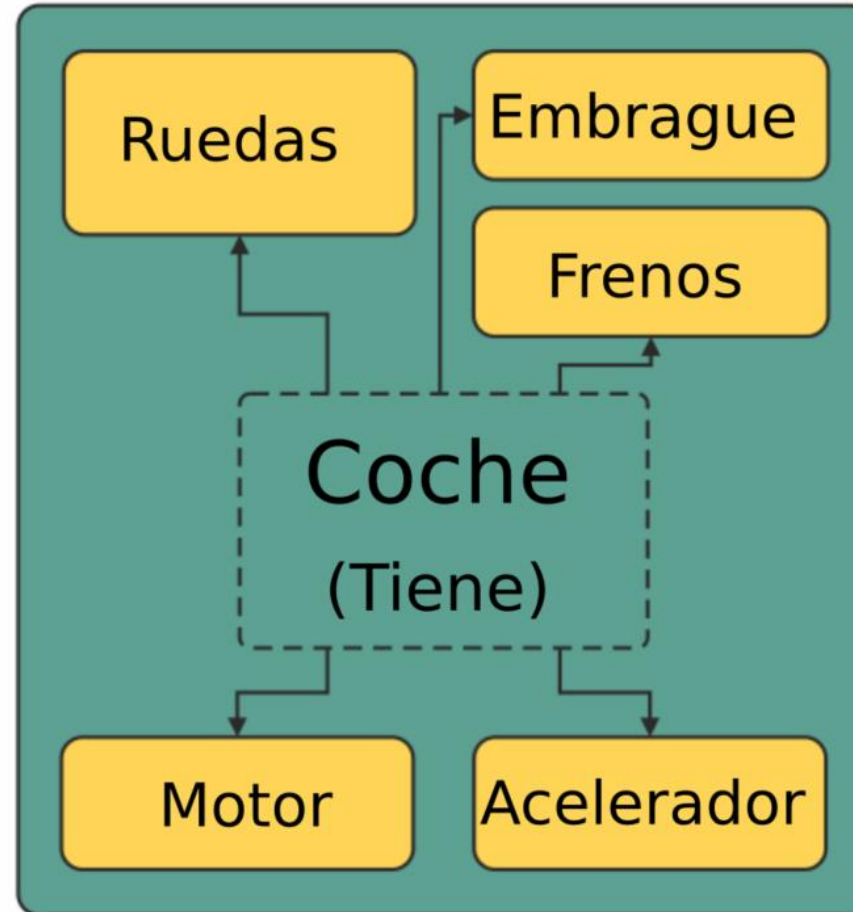
Even at one of the low points in his career, Jobs still had confidence in the limitless potential of personal computing

By JEFF GOODELL



Ejemplo de Objeto

- Múltiples atributos
- Comportamientos específicos
 - Arrancar
 - Acelerar
 - Desacelerar
 - cambiar de marcha



La Programación Orientada a Objetos (POO)

- Todo en Python es un objeto

La POO forma parte del núcleo de Python

- Python es totalmente compatible con este paradigma

- Los dos protagonistas son los objetos y las clases

- Las clases se utilizan para crear objetos
- Los objetos cuando son creados por una clase
 - heredan los atributos y métodos de la clase

```
<'int'>  
<'str'>  
<'bool'>  
<'list'>  
<'dict'>  
<'function'>
```

La POO ofrece las siguientes ventajas

- Hace que el código sea reutilizable
- Facilita el diseño/modelado de software
- Facilita las pruebas, la depuración y el mantenimiento
- Los datos están seguros gracias a la abstracción y la ocultación

Clase

Definir: Utilizando la palabra clave de Python *class*, seguida del nombre de la clase.

```
>>> class Persona:  
... pass  
...
```

la palabra clave *pass* se utiliza como marcador de posición

Objeto

- Instanciar un objeto de una clase es el acto de construir lo que una clase describe

```
>>> juan = Persona()  
>>> pedro = Persona()  
>>> jose = Persona()
```

```
>>> juan is jose  
  
>>> juan = pedro  
>>> juan is pedro
```

Objeto

- Instanciar un objeto de una clase es el acto de construir lo que una clase describe

```
>>> juan = Persona()  
>>> pedro = Persona()  
>>> jose = Persona()
```

```
>>> juan is jose  
Falso  
>>> juan = pedro  
>>> juan is pedro  
Verdadero
```

Objetos distintos

Objetos iguales

Añadir atributos a un objeto

- Un atributo es una característica específica de un objeto
- Escribimos el nombre del objeto seguido de un punto (.) el nombre del atributo a añadir y asignándole un valor

```
>>> persona1 = Persona()  
>>> persona1.nombre = "Juan Perez"
```

dict contiene todos los atributos del objeto

Añadir atributos a un objeto

- Un atributo es una característica específica de un objeto
- Escribimos el nombre del objeto seguido de un punto (.) el nombre del atributo a añadir y asignándole un valor

```
>>> persona1 = Persona()  
>>> persona1.nombre = "Juan Perez"
```

dict contiene todos los atributos del objeto



Establecer atributos de esta manera es una mala práctica

El método `__init__`

- Constructor: Para añadir de forma adecuada atributos a un objeto

```
>>> class Persona:  
...     def __init__(self, nombre):  
...         self.name = nombre  
...  
>>>
```

name se refiere al nombre de la persona

self se refiere al objeto que estamos creando

self nos permite obtener o establecer los atributos del objeto dentro de nuestra función

Siempre es el primer argumento de un método de instancia.

E1: Añadir atributos a una clase

1. Crear una clase TelefonoMovil con sus principales atributos
 1. Tamaño de pantalla
 2. Capacidad de la memoria RAM
 3. Sistema Operativo
2. Instanciar los siguientes teléfonos móviles:



iPhone 13 con pantalla de 5.5, 6GB de RAM, y sistema operativo iOS



Samsung Galaxy con pantalla de 6.8, 8GB de RAM y sistema Android

Actividad 1: Definir una clase y objetos

Para una plataforma de noticias tecnológicas te han pedido que diseñes un sistema de plantillas para sus teléfonos móviles con una estructura similar a:

El nuevo teléfono [] tiene una pantalla de [] pulgadas de pantalla. [] de RAM y funciona con la última versión de []. Su mayor competidor es el teléfono [], que tiene una pantalla de [] pulgadas, [] de RAM y ejecuta Android.

- Genera la plantilla utilizando los valores de los atributos de los objetos creados en el ejercicio anterior

Definir métodos en una clase

Los objetos están compuestos por comportamientos conocidos como métodos

Reescribir la clase Persona para incluir el método `presentarse()`.

1. Crear un método `presentarse()` en nuestra clase Persona

```
class Persona:  
    # constructor
```

```
def presentarse(self):  
    print(f"Hola, me llamo {self.name} y tengo {self.age} años")
```

2. Instanciar un objeto y llamar al método que hemos definido

```
jorge=Persona("Jorge", 35)  
jorge.presentarse()
```

Pasar argumentos a los métodos de instancia

- Se puede pasar argumentos a los métodos de una clase al igual que con las funciones normales

```
def saludar(self, persona):  
    if persona.nombre == "Rogelio":  
        print(f"Buenos días {persona.nombre}")  
    else:  
        print(f"Hola {persona.nombre}")
```

- Llamar al método `saludar(persona)`

E2: Establecer atributos de instancia dentro de métodos de instancia

- Vamos a crear un método `cumpleaños()` que incremente la edad de la persona.
 1. Implementa el método `cumpleaños()`, que toma la edad y la incrementa en uno
 2. Crea una instancia de persona y comprueba la edad
 3. Llama al método `cumpleaños()` y comprueba nuevamente la edad

Actividad 2: Definir métodos en una clase

- Construye un programa para calcular la circunferencia y el área de círculos
 - La fórmulas de un círculo para calcular

la **circunferencia** es $2 * \pi * r$
el **área** es $\pi * r * r$

1. Definir la clase **Circulo** y el método constructor con el radio
2. Crear el método **área**, que devuelve el cálculo del área del círculo.
3. Crear el método **circunferencia**, que devuelve la circunferencia del círculo.
4. Solicita como **entrada del usuario** el radio.
5. Crea un bucle **while** para que la solicitud de entrada del usuario se ejecute por siempre
6. Redondea la salida a 2 decimales

Radio del circulo : 5
Area: 78.54
Circunferencia: 31.42
...

Atributos de clase frente a atributos de instancia

- Inicializar un objeto con atributos específicos aplica esos atributos sólo a ese objeto
- Los atributos declarados dentro del constructor se añaden como atributos de la instancia
La vinculación de esos atributos a la instancia ocurre en el método `__init__` donde añadimos atributos a `self`

E3: Declarar una clase con atributos de instancia

- Declararemos una clase **NavegadorWeb** que tiene atributos para
 - el historial
 - la página actual y
 - una bandera booleana que muestra si esta de incógnito o no
1. Inicializa los objetos de la clase
 2. Comprueba el atributo **pagina_actual** en diferentes instancias de la clase NavegadorWeb

Atributos de clase

- Los atributos de clase están ligados a la propia clase y son compartidos por todas las instancias
- También podemos definir estos atributos
- Cuando cambiamos el atributo de la clase a través de la misma se reflejará en todas las instancias existentes

E4: Extendiendo la clase con atributos

1. Añadir a la clase NavegadorWeb el atributo conectado (booleano a True)
2. Instanciamos un objeto NavegadorWeb.
3. Acceder a los atributos de la clase a través de la propia clase
4. Imprime los atributos `__dict__` de nuestras instancias
5. ¿Por qué no obtenemos un `AttributeError` cuando intentamos recuperar este atributo?

E5: Implementación de un contador para las instancias de una clase

Vamos a crear un contador que se incrementará cada vez que se instancie un nuevo objeto NavegadorWeb

1. Añade el atributo de clase `número_de_navegadores_web` que servirá de contador y comenzará en 0
2. Modifica el constructor para incrementar el contador cada vez que se crea una nueva instancia añadiendo la línea
3. Comprueba que el contador está en 0
4. Instala un nuevo objeto y comprueba el contador

Actividad 3: Creando atributos de clase

- Suponga que está diseñando un software para una compañía de ascensores que involucra un mecanismo de seguridad para prevenir que el elevador sea usado cuando se llenen más allá de su capacidad.

1. Declare la clase `ascensor` añadiendo un atributo de clase límite de ocupación de 8 personas
2. Añade el inicializador, que comprobará si se supera el límite de ocupación, e imprimirá un mensaje indicando cuántas personas deben bajar
3. Cree algunas instancias para probar

- La salida debería ser la siguiente:

```
Ocupantes del ascensor 1: 6
Se excede la capacidad del ascensor 2 ocupantes deben salir.
Ocupantes del ascensor 2: 10
```


E6: Creación de Métodos de Instancia

Para la clase `NavegadorWeb` implementaremos los métodos `navegar()` y `limpiar_historial()`

1. Añade el método `navegar()` a la clase

Si no estamos en modo incógnito cualquier llamada a `navegar` establecerá la página actual del navegador al argumento *nueva_pagina* y la añadirá al historial

2. Desde una instancia Llamar a `navegar()` debería cambiar `pagina_actual`

3. Crea el método `limpiar_historial` que borrará el historial del navegador hasta el último elemento dejando sólo la página actual en la lista

4. Añade al historial del navegador un par de páginas y luego llama al método `limpiar_historial()` para ver si funciona

Métodos de clase

- Difieren de los métodos de instancia en que están ligados a la propia clase y no a la instancia
- No tienen acceso a los atributos de la instancia
- Pueden ser llamados a través de la propia clase
- No requieren la creación de una instancia de la clase
- Los métodos de clase el primer parámetro es siempre la propia clase
- Pueden ser utilizados para devolver objetos de un tipo diferente o con diferentes atributos
- La definición de la función comienza con `@classmethod`
 - añade la función que está debajo como un método de clase
- Siguiendo línea declaramos la función que toma el argumento `cls` (clase)

Método de fábrica

- Un método de fábrica devuelve objetos
- Un caso de uso común para los métodos de clase es cuando se hacen métodos de fábrica
- Pueden ser utilizados para devolver objetos de un tipo diferente o con diferentes atributos

E7: Probando el método de fábrica

A la clase Navegador vamos a añadir un método de clase llamado `incognito()` que inicializa un objeto del navegador web en modo incógnito:

1. Imprime el método de la clase
2. Crea una instancia de la clase que se inicia en modo incógnito
3. Imprime la página actual de nuestra instancia para comprobar si se ha configurado
4. Confirmar que no se ha rastreado el historial
5. Llamar a los métodos de la clase a través de las instancias para conseguir el mismo efecto

E8: Accediendo a los Atributos de la Clase desde los Métodos de la Clase

- Los métodos de clase también tienen acceso a los atributos de la clase
- Dado que la mayoría de los navegadores actuales tienen una API de geolocalización simularemos esta funcionalidad en la clase
- En la clase NavegadorWeb crearemos un atributo `geo_coordenadas` que contiene la latitud y la longitud actuales
- También añadiremos un método de clase llamado `cambiar_geo_coordenadas()` que cambiará las coordenadas al ser llamado

E8: Accediendo a los Atributos de la Clase desde los Métodos de la Clase

1. Añade el atributo de clase `geo_coordenadas` igual a `latitud:39.466667` y `longitud: -0.375`
2. Añade el método de clase `cambiar_geo_coordenadas()` que toma el parámetro `nuevas_coordenadas` que es **un diccionario**.
3. El método de clase comprueba si la `latitud` y la `longitud` proporcionadas en los parámetros son válidas, recuerda que los valores validos para:
 - la `latitud` -> debe estar dentro del rango entre -90 y 90 grados
 - la `longitud` -> debe estar dentro del rango entre -180 y 180 grados
4. Crea `firefox` como una instancia de `NavegadorWeb` para comprobar sus `geocoordenadas`
5. Llamar a `cambiar_geo_coordenadas` en la clase
6. ¿Que sucedió?

Atributos de instancia VS atributos de clase

- Atributos de Instancia
 - Están vinculados a la instancia
 - Sólo se pueden recuperar a través de la instancia
 - La modificación de este valor sólo lo hace para la instancia actual
- Atributos de Clase
 - Están vinculados la clase
 - Se puede acceder a ellos a través de la instancia y de la clase
 - Al cambiar este valor, se modifica para todas las instancias

Métodos de instancia VS métodos de clase

- Métodos de Instancia
 - Están vinculados a la instancia
 - Sólo pueden ser llamados a través de una instancia
 - Toman la instancia propia como primer argumento
 - Tienen acceso a los atributos de la instancia y de la clase
- Métodos de Clase
 - Están vinculados a la clase
 - Se puede acceder a través de la instancia y de la clase
 - Toman la clase como primer argumento
 - Sólo tienen acceso a los atributos de la clase

Encapsulación y ocultación de información

- Uno de los conceptos clave de la POO
- La encapsulación es la agrupación de datos
 - con los métodos que operan sobre esos datos.
- Se utiliza para ocultar el estado interno de un objeto agrupando y proporcionando métodos que pueden obtener y establecer el estado del objeto a través de una interfaz.
- Esta ocultación del estado interno de un objeto se llama ocultación de información.
- En Python, la ocultación de información se realiza marcando los atributos como privados o protegidos
 - **atributos private** sólo deben usarse dentro de la clase y no accedidos externamente.
 - **atributos protegidos** similares a los privados, pueden ser utilizados en contextos muy específicos.
- Por defecto, todos los atributos son públicos.

Encapsulación y ocultación de información

Python, implementa los modificadores de acceso a los atributos en los propios nombres de los atributos

Atributos protegidos, anteponemos al nombre del atributo un **guión bajo**, `_`.

Para indicar que está protegido:

```
self._velocidad = 300  
self._color = "negro"
```

Atributos privados, anteponemos al nombre del atributo un **doble guión bajo** `__`.

Esto hace que el atributo sea inaccesible desde fuera de la clase.

El atributo sólo puede ser obtenido y establecido desde dentro de la clase:

```
self.__velocidad = 300  
self.__color = "negro"
```

Encapsulación y ocultación de información

- Getters y Setters
 - Para cambiar el atributo privado `__velocidad`, debemos utilizar el método **setter** definido `cambiar_velocidad`.
 - Para obtener el atributo de velocidad desde fuera de la clase si es necesario, podemos utilizar el método **getter** “`obtener_velocidad`”

Actividad 4: Creación de métodos de clase y uso de la ocultación de información

Suponga que trabaja para una compañía de electrónica que tiene un nuevo dispositivo reproductor de música que quiere lanzar al mercado.

El software para este dispositivo necesita soportar actualizaciones over-the-air que permitan a los usuarios escuchar sus canciones favoritas sin problemas.

Crea una clase que represente un reproductor de música portátil, "ReproductorMusica".

La clase ReproductorMusica debe tener un método de reproducción, que establece la primera pista de la lista de reproducción como la que se está reproduciendo actualmente.

La lista de reproducción debe ser un atributo privado.

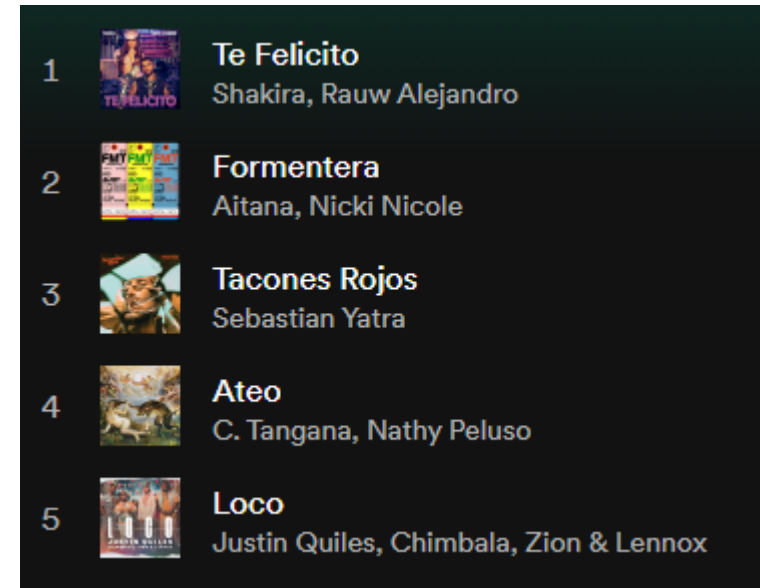
Además, debe tener un atributo de versión de firmware

y un método de clase actualizar_firmware que actualice la versión del firmware

Actividad 4: Creación de métodos de clase y uso de la ocultación de información

1. Define la clase ReproductorMusica añadiendo el atributo de clase `version_firmware`

2. Define el método inicializador y pon en la lista de reproducción las siguientes 5 canciones:



3. Asegúrate de que la lista de reproducción es `privada`.

Actividad 4: Creación de métodos de clase y uso de la ocultación de información

4. Define el método `play`, que establece el atributo `cancion_actual` al primer elemento de la lista de reproducción.
5. Define el método `listar_canciones` que devuelve la lista de reproducción del `ReproductorMusica`.
6. Añadir el método `actualizar_firmware` que comprueba antes de actualizar si la nueva versión del firmware que se proporciona es más reciente que la versión actual
7. Ejecutar el script para probar el reproductor

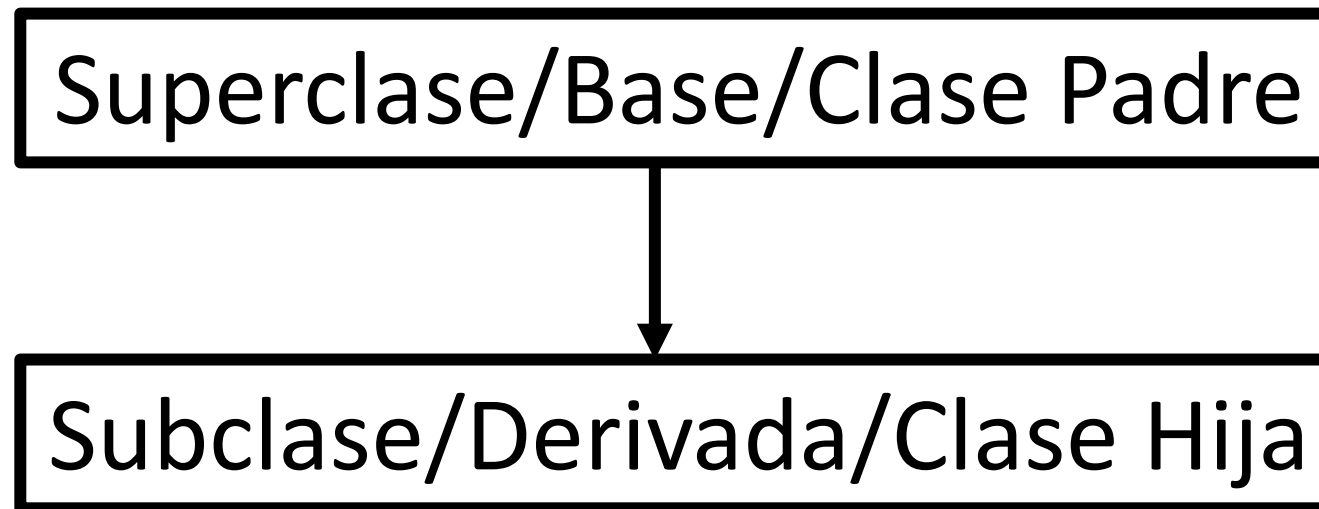
- Salida de la ejecución del script:

```
Tu lista de reproducción es: ['Te Felicito', 'Formentera', 'Tacones', 'Ateo', 'Loco']  
versión de firmware: 1.0  
versión de firmware: 2.0  
Reproduciendo actualmente: Formentera
```


Herencia de clases

Una característica clave de la POO.

Es un mecanismo que permite que la implementación de una clase derive de la implementación de otra clase.



La herencia en Python

- La sintaxis de la herencia es mínima
 - Se define la clase y se pasa la clase base como parámetro
- Hace que el código sea más reutilizable
 - Se definen las propiedades en la clase base y se hereda la funcionalidad en las clases derivadas
 - Un método o atributo añadido a una clase base se aplica automáticamente a todas sus subclases
- Añade flexibilidad a nuestro código
 - Se puede sustituir una instancia de la subclase en cualquier lugar donde se utilice una instancia de la superclase

Herencia de clases

- Un ejemplo práctico de herencia en el mundo real es el de **los felinos**



leones



leopardos



gatos



guepardos



tigres



ligres

- Todos los felinos comparten las mismas
 - Propiedades
 - peso
 - vida promedio
 - velocidad
 - Comportamientos
 - Cazar
 - hacer sonidos/ruidos
 - ...

Herencia de clases en el mundo de los felinos



Leon:

```
peso  
vida  
velocidad  
sonido
```

Ligre:

```
peso  
vida  
velocidad  
sonido
```



Guepardo:

```
peso  
vida  
velocidad  
sonido
```

Tigre:

```
peso  
vida  
velocidad  
sonido
```



Gato:

```
peso  
vida  
velocidad  
sonido
```

Leopardo:

```
peso  
vida  
velocidad  
sonido=
```



Herencia de clases en el mundo de los felinos



Leon:

```
Leon.peso=190  
Leon.vida=8  
Leon.velocidad=80  
Leon.sonido=rugido
```

Ligre:

```
Ligre.peso=400  
Ligre.vida=18  
Ligre.velocidad=50  
Ligre.sonido=rugido
```



Guepardo:

```
Guepardo.peso=72  
Guepardo.vida=10  
Guepardo.velocidad=90  
Guepardo.sonido=gañido
```

Tigre:

```
Tigre.peso=90  
Tigre.vida=8  
Tigre.velocidad=49  
Tigre.sonido=rugido
```



Gato:

```
Gato.peso=4  
Gato.vida=12  
Gato.velocidad=48  
Gato.sonido=ronroneo
```

Leopardo:

```
Leopardo.peso=31  
Leopardo.vida=12  
Leopardo.velocidad=58  
Leopardo.sonido=rugido
```



Herencia de clases en el mundo de los felinos

```
class Felino:  
    Felino.peso  
    Felino.vida  
    Felino.velocidad  
    Felino.sonido
```

```
class Leon(Felino):  
    pass
```

```
leon=Leon(190,8,80)
```

```
class Gato(Felino):  
    def hacer_sonido(self):  
        print("Miau...")
```

E9: Implementación de la Herencia de Clases

Definiremos la clase Felino de la que derivaremos nuestros felinos.

- La clase tendrá los métodos `hacer_sonido` y `imprimir_datos`, y los atributos `peso`, `vida media` y `velocidad`.
- El método constructor tomará los argumentos `peso`, `vida media` y `velocidad`, a partir de los cuales añadirá los atributos: `peso_en_kg`, `vida media_en_años` y `velocidad_en_kph` al objeto.
- El método `hacer_sonido` imprimirá “Grrrr....” una vocalización no amenazante que es común a varios felinos.
- El método `imprimir_datos` imprimirá datos sobre el felino como por ejemplo:

“El gato pesa 4kg, tiene una vida de 18 años y puede correr a una velocidad máxima de 48kph.”

E9: Implementación de la Herencia de Clases

Definiremos la clase Felino de la que derivaremos nuestros felinos.

1. Definir la clase Felino con los atributos **peso**, **vida media** y **velocidad**; y los métodos **hacer_sonido** y **imprimir_datos** que imprimirá “Grrrr....” y “El *gato* pesa 4 Kg, tiene una vida de 12 años y puede correr a una velocidad máxima de 48 Km/h.” con los datos correspondientes respectivamente
2. Instanciar una instancia de Felino e interactuar con los diferentes métodos y atributos que tiene
3. Crea las subclases Leopardo, Gato y León, que heredarán de la clase Felino
4. Instanciar las nuevas clases Leopardo, Gato y León
5. Comprobar los atributos y métodos **hacer_sonido** y **imprimir_datos**

Sobrescritura de métodos

- Para añadir o cambiar la funcionalidad de una subclase.
- Sobrescribiendo el método `hacer_sonido` en nuestra subclase redefinimos la implementación original del método definido en la superclase
- Sobrescribir el método sonido para nuestras subclases:

```
class Gato(Felino):  
    def hacer_sonido(self):  
        print("Miau...")
```

```
class Leon(Felino):  
    def hacer_sonido(self):  
        print("Gr...")
```

Sobrescritura de `__init__()`

- Para añadir un atributo
- Muchos felinos grandes tienen un patrón en su pelaje
 - tienen manchas o rayas.
- Añadamos esto a nuestra subclase Gato

```
class Gato(Felino):  
    def __init__(self, peso, edad, velocidad):  
        self.pelaje_manchado=True
```

- Sólo añadirá el atributo `pelaje_manchado` a esta instancia

E10: Sobrescribir el método `__init__` para añadir un atributo

1. Sobrescribe el método inicializador y añade el atributo `pelaje_manchado`
2. Si inicializas la clase Gato recién modificada, al intentar acceder a los atributos originales debería dar un **Error**
3. Antes de añadir el atributo `pelaje_manchado` dentro del método `__init__` de la subclase se debe invocar el método `__init__` de la clase Felino

```
class Guepardo(Felino):  
    def __init__(self, peso, vida, velocidad):  
        Felino.__init__(self, peso, vida, velocidad)  
        self.pelaje_manchado = True
```

Otros métodos comúnmente sobrescritos

Métodos Dunder (doble guión bajo) o mágicos

- Métodos especiales de las clases que siempre llevan el prefijo y el sufijo de doble guion bajo
 - el método `__init__`
- Usados para añadir funcionalidad personalizada, personalizar las clases, cambiar el aspecto de la salida impresa
 - `__str__()`
 - `__del__()` destruye un objeto

El método `__str__()`

Por defecto cada objeto en Python tiene el método `__str__()`

Recuperar la cadena que contiene la representación legible del objeto

Se llama cada vez que se llama a `print()` sobre un objeto en Python

Reemplacemos el método `imprimir_datos()` de la clase Felino

```
def __str__(self):  
    return("cadena")
```

Cuando llamemos a `print()` sobre cualquier instancia de Felino debería tener el mismo resultado que cuando llamamos a `imprimir_datos()`

```
print(guepardo)
```

El método `__del__()`

- Es el método destructor, llamado cada vez que un objeto es destruido

```
def __del__(self):  
    print("No se ha dañado a ningún animal al borrar esta instancia.")
```

- Si llamamos a `del` en una instancia de Tigre debería imprimir ese mensaje:

```
>>> tigre = Tigre(72, 12, 120)  
>>> del tigre  
No se ha dañado a ningún animal al borrar esta instancia  
>>> tigre  
NameError: el nombre 'guepardo' no está definido
```

Herencia múltiple

- Es una característica que permite heredar atributos y métodos de más de una clase
- Herramienta potente y a la vez puede ser complicada
asegurarnos de entender lo que sucede cuando la usamos
- Python permite la herencia múltiple a diferencia de lenguajes como Java y C#
- El caso de uso más común son métodos/atributos que están destinados a ser utilizados por otras funciones
- Ejemplo:
 - una clase Logger tendría un método log()

E11: Implementación de la Herencia Múltiple

- En el mundo real, los leones y los tigres pueden aparearse naturalmente para crear un híbrido conocido como ligre (o tigon).
 - Los ligres son mucho más grandes que los leones o los tigres, **son sociales** como los leones, y tienen rayas y les gusta nadar al igual que los tigres
1. Definir las clases León y Tigre
 - clase Tigre con el atributo `patron_pelaje` y el método `nadar()`
 - clase León con el atributo `es_social`
 2. Definir la clase Ligre, que herede de las clases Tigre y León
 3. Observar lo que la clase Ligre ha heredado de las otras clases
 - Debe tener el atributo `patron_pelaje` y el método `nadar()` de la clase Tigre y el atributo `es_social` de la clase León

E11: Implementación de la Herencia Múltiple

La salida debería ser similar a:

```
El león pesa 190 Kg, tiene una vida de 14 años y puede correr a una velocidad máxima de 80 Km/h.  
El tigre pesa 310 Kg, tiene una vida de 26 años y puede correr a una velocidad máxima de 65 Km/h y el  
patrón del pelaje es a rayas  
Salpicando agua!!!  
El ligre es social? True  
Patrón de pelaje: a rayas
```

Actividad 6: Practicando la herencia múltiple

- Estamos en el año 2000. Estás trabajando para una empresa de telefonía móvil y te han encargado modelar el software de un teléfono móvil que tendrá una cámara incorporada
- Crear una clase llamada *Cámara* y una clase llamada *TeléfonoMóvil* que serán las clases base de una clase derivada llamada *CámaraTeléfono*
- *CámaraTeléfono* debe tener inicializado un atributo de memoria y debe tener un método `hacer_foto()` que imprima el mensaje *¡Sonríe!*

Actividad 6: Practicando la herencia múltiple

1. Crear una clase Camera que tenga un método hacer_foto().
2. Crear una clase TelefonoMovil que será inicializada con un atributo de memoria.
3. Crear una clase CamaraTelefono que herede de las clases TelefonoMovil y Camera.
4. Inicializar una instancia de la clase CamaraTelefono.
5. Llamar al método hacer_foto() en la instancia debería tener una salida como esta.
6. Imprime el atributo de memoria

- La salida debería ser :

```
¡Sonrie!  
Memoria restante 198k
```

Diferencia entre Herencia y Polimorfismo

1. H -> Se aplica básicamente a las clases.
PM -> Se aplica básicamente a las funciones o métodos
2. H -> apoya el concepto de reutilización y reduce la longitud del código
PM -> permite al objeto decidir qué forma de la función implementar tanto en tiempo de compilación / ejecución
3. H -> puede ser simple, múltiple, ..
PM -> en tiempo de compilación (sobrecarga) y en tiempo de ejecución (overriding)
4. H -> La clase bicicleta puede heredar de la clase vehículos de dos ruedas que a su vez puede ser una subclase de vehículos.

PM -> La clase moto puede tener el nombre de método set_color(), que cambia el color de la moto basado en el nombre del color que se ha introducido

Resumen

- La POO
 - Hace que el código sea más reutilizable
 - Facilita el diseño del software
 - Hace que el código sea más fácil de probar, depurar y mantener
 - Añade seguridad a los datos de una aplicación
- Los comportamientos de un objeto se conocen como métodos
 - Se añade un método a una clase definiendo una función dentro de ella
 - Para estar vinculada a los objetos necesita tomar el argumento self
- Cubrimos
 - los atributos de clase
 - los métodos de clase
 - la encapsulación
 - las palabras clave que permiten ocultar información
 - la herencia
 - Herencia multiple
 - Sobrecarga de métodos (init, str, del)

Ejercicios Extras:

Ejercicio Herencia

- Clase Vehículo
 - Atributos: Nombre y color (privado para la clase)
 - Metodos: getColor, setColor, getNombre
- Clase Coche(heredada de Vehiculo)
 - Metodos: getDescripcion
- Probar con:
 - Ford Mustang
 - GT350
 - Rojo




Herencia Multiple

- El objeto ClaseHija es capaz de acceder a los métodos super1 y 2

```
class SuperClase1():  
    def metodo_super1(self):  
        print("Llamada al Metodo super UNO")
```

```
class SuperClase2():  
    def metodo_super2(self):  
        print("Llamada al Metodo super DOS")
```

```
class ClaseHija(SuperClase1, SuperClase2):  
    def metodo_hijo(self):  
        print("Método hijo")
```



```
c.metodo_super1() #Llamada al Metodo super UNO  
c.metodo_super2() #Llamada al Metodo super DOS  
c.metodo_hijo()   #Método hijo
```

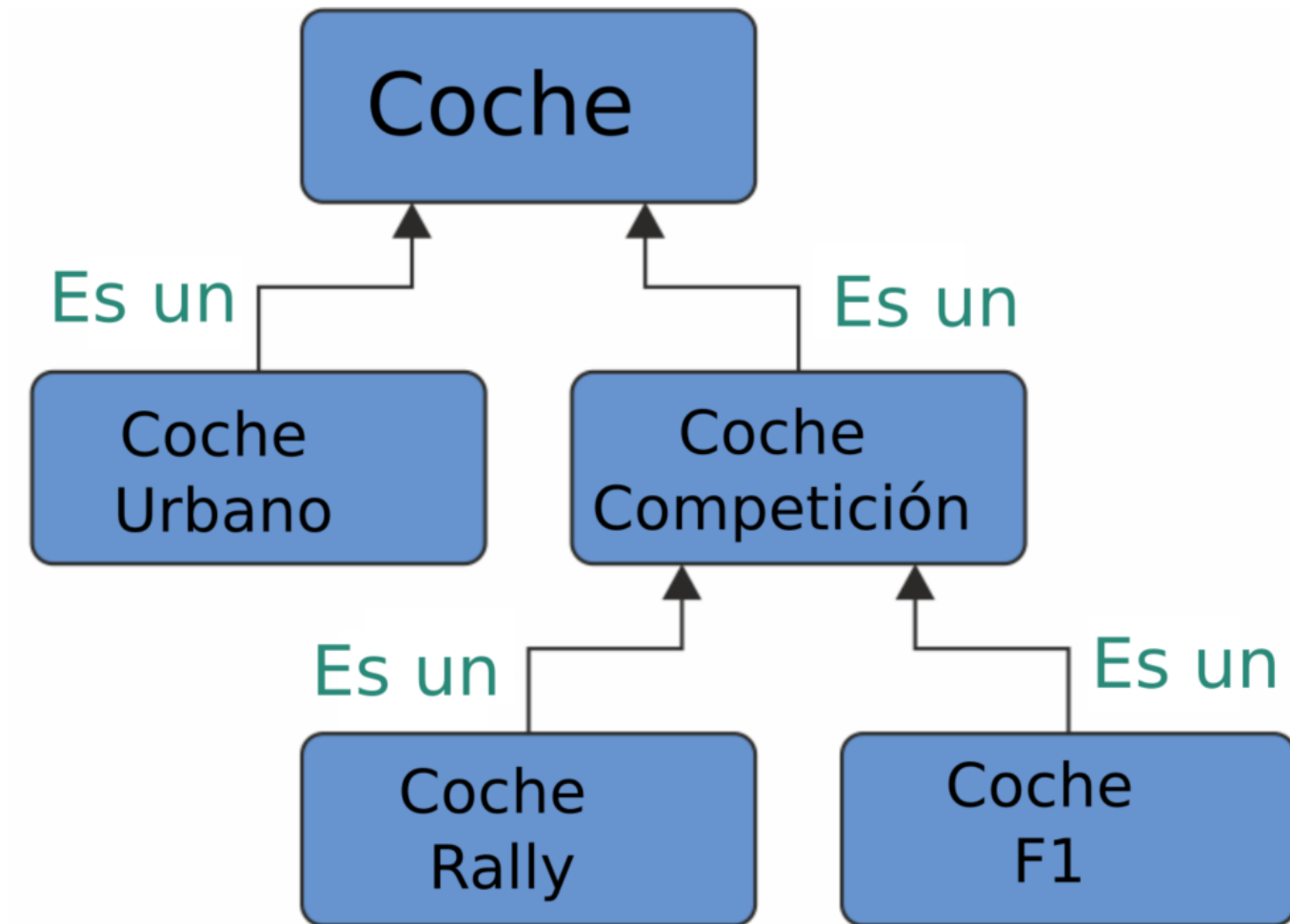

Anulación de métodos –overriding–

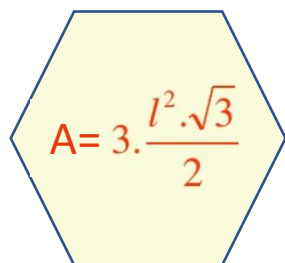
- La subclase define el mismo nombre de método y el mismo número de parámetros que el método de la clase base

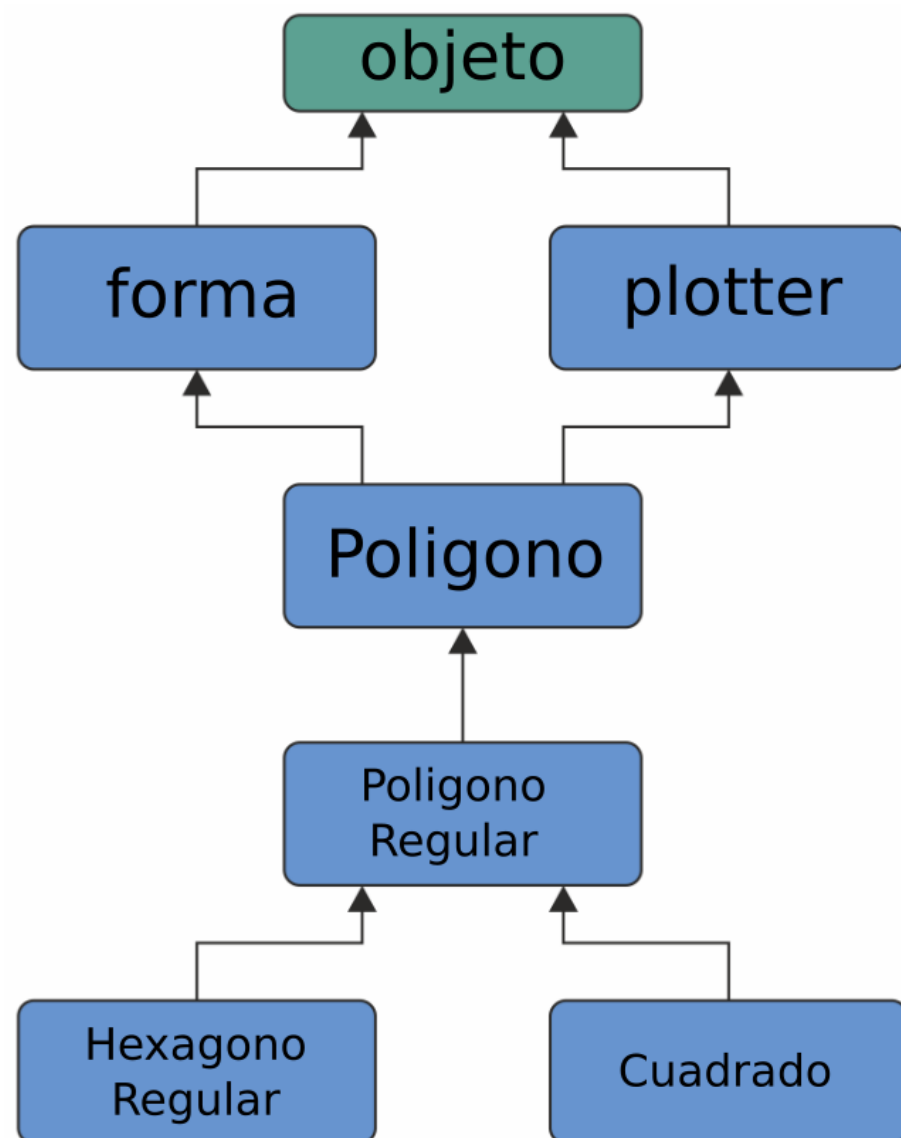
```
class A():  
    def __init__(self):  
        self.__x=1  
  
    def m1(self):  
        print("m1 de A")
```

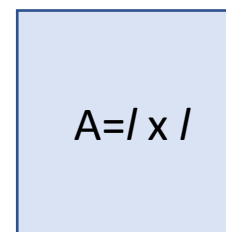
```
class B():  
    def __init__(self):  
        self.__y=1  
  
    def m1(self):  
        print("m1 de B")
```

```
c=B()  
c.m1() #m1 de B
```




$$A = 3 \cdot \frac{l^2 \cdot \sqrt{3}}{2}$$




$$A = l \times l$$