

Codo a Codo 4.0

FULL STACK PYTHON

html - css3 - bootstrap - javascript
vue.js - sql - python - django

Classes 29

Python – Partes 5



Temas que veremos las próximas clases

P00

Objetos

Clases

Abstracción

Encapsulamiento

Polimorfismo

Herencia

Programación orientada a objetos (POO)

Metodologías de programación

Prácticamente todos los lenguajes desarrollados en los últimos 25 años implementan la posibilidad de trabajar con POO (Programación Orientada a Objetos).

Python permite programar con las siguientes metodologías:

- **Programación Lineal (imperativa):** El código es desarrollado en una *secuencia lineal*, sin emplear funciones. Es apenas modificada por las bifurcaciones (condicionales) o repeticiones.
- **Programación Estructurada (funcional):** El código es desarrollado *modularmente a través de funciones* que agrupan actividades a desarrollar y luego son llamadas dentro del programa principal. Estas funciones pueden estar dentro del mismo archivo (módulo) o en una librería separada (funciones dentro de módulos que podemos importar).
- **Programación Orientada a Objetos:** Aplica la metodología de la programación orientada a objetos que tiene una sintaxis particular, donde se plantean *clases* y definen *objetos*.

Por ejemplo: Al hablar de un animal no estamos hablando de un objeto, no es un objeto hasta que no “exista”, tenga características o un comportamiento. A eso lo vamos a llamar clase, que es el “molde” para construir el objeto. También podemos hacer referencia a objetos intangibles (por ejemplo: una cuenta bancaria).

Programación orientada a objetos (POO)

El paradigma orientado a objetos

Un paradigma de programación es un estilo de desarrollo de programas, un modelo para resolver problemas computacionales o, mejor dicho, una forma distinta de pensar la programación. Los lenguajes de programación, necesariamente, se encuadran en uno o varios paradigmas a la vez a partir del tipo de órdenes que permiten implementar, algo que tiene una relación directa con su sintaxis.

- En el ***paradigma orientado a objetos*** el comportamiento del programa es llevado a cabo por objetos, entidades que representan elementos del problema a resolver y tienen atributos y comportamiento (pueden almacenar información y realizar acciones).
- Hace que el desarrollo de grandes proyectos de software sea más fácil y más intuitivo.
- Nos permite pensar sobre el software en términos de objetos del mundo real y sus relaciones.
- Nos acercamos más a la realidad, dado que trabajamos con elementos de la vida real.

En el caso de la programación el POO es de mediados de los 70 y su gran auge es a mediados de los 90. El lenguaje que vino a destacarse en este nuevo paradigma es JAVA (por excelencia orientado a objetos).

El concepto de objeto excede JAVA, Python, etc, aplica a varios lenguajes.

Objetos y clases

Clase

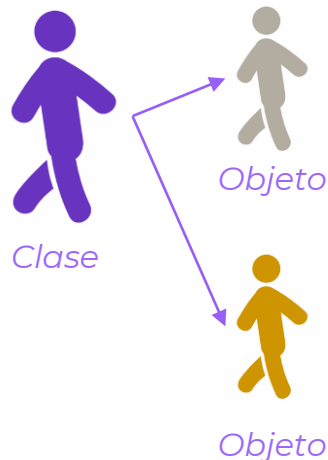
La programación orientada a objetos se basa en la definición de **clases**; a diferencia de la programación estructurada, que está centrada en las **funciones**.

Una clase es un **molde** del que luego se pueden crear múltiples objetos, con similares características. Esta plantilla o molde define los atributos (*variables*) y métodos (*funciones*) comunes a los objetos de ese tipo, pero luego, cada objeto tendrá sus propios valores y compartirán las mismas funciones.

Dado que las clases se usan para crear objetos, debemos declararlas antes de poder crear objetos (instancias) de esa clase. Al crear un objeto de una clase, se dice que se crea una **instancia de la clase** o un objeto propiamente dicho.

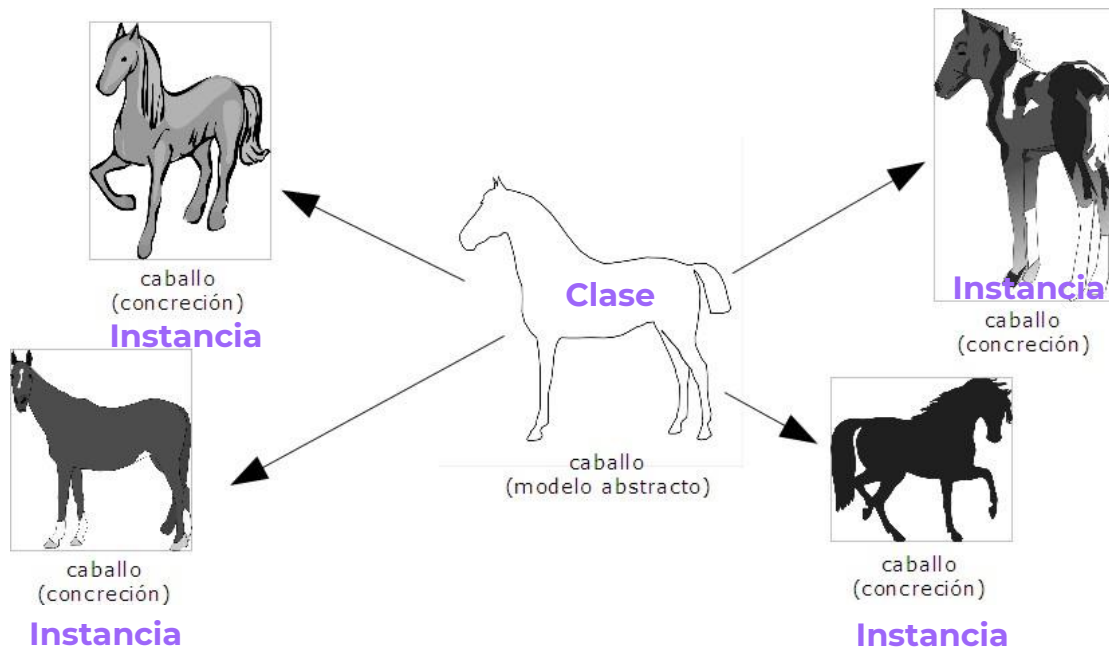
Una clase está formada por los **métodos** y las **variables** (atributos) que definen las características comunes a todos los objetos de esa clase. Precisamente la clave de la POO está en abstraer los métodos y los datos comunes a un conjunto de objetos y almacenarlos en una **clase**.

Una clase equivale a la **generalización de un tipo específico de objetos**. Una **instancia** es la concreción de una clase en un objeto. Las clases definen el tipo de datos (*type*).



Podemos crear muchos objetos desde una sola clase

Objetos y clases



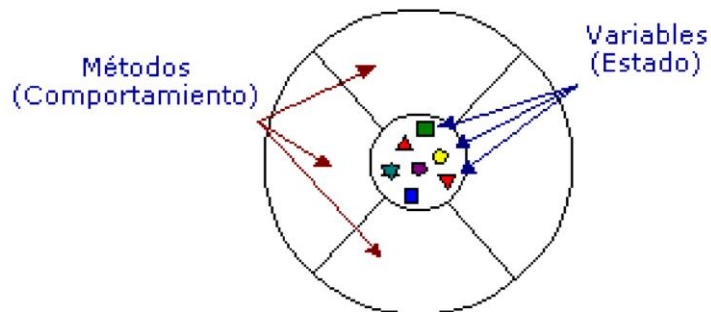
Cada uno de los objetos tiene su propia copia de las variables definidas en la clase de la cual son instanciados y comparten la misma implementación de los métodos.

No se puede crear un objeto sin previamente haber creado o definido la clase, porque la clase es el molde para ese objeto.

Objetos y clases

Objeto

- Es una **encapsulación genérica de datos** y de los procedimientos (*funciones*) para manipularlos. Debemos pensarlos como objetos del mundo real.
- Tienen un **estado** y un **comportamiento**: El **estado** de los objetos se determina a partir de una o más *variables (valores del atributo)* y el **comportamiento** con la implementación de *métodos*.



Representación común de los objetos de software

*Se habla de **encapsulación** porque esa información asociada al objeto podría ser vulnerable, son datos sensibles para el objeto. Ejemplo: si cambio el saldo de la cuenta del objeto "cliente", cambio información sensible.*

Objetos y clases

EJEMPLO DE CLASES Y OBJETOS

Clase:

Coche

Clase Coche

arrancar, ir, parar, girar

color, velocidad, carburante

nombre de la clase

métodos (funciones)

atributos (datos)

Objeto:

Ferrari

coche.ferrari

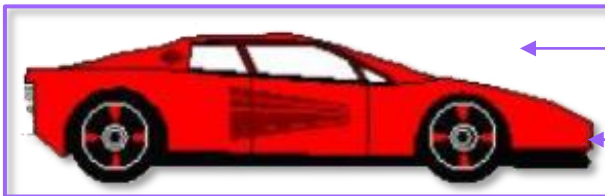
nombre del objeto

métodos

arrancar, ir, parar, girar

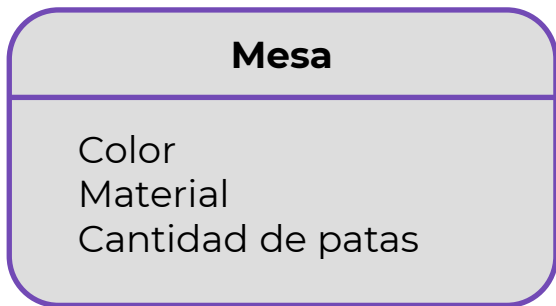
datos

rojo, 280 km/h, lleno



Objetos y clases

Clase



Objeto 1



Objeto 2



Un objeto es una entidad en un programa, usualmente un sustantivo.

Por ejemplo: una persona en particular.

- **Clase:** Persona;
- **Propiedades/Atributos:** Nombre; Edad; Dirección;
- **Comportamiento/Métodos:** Caminar; Hablar; Respirar;
- **Objeto:** José Pérez de 23 años que vive en la calle Cucha cucha 123;



Otro ejemplo: Crear una nueva variable denominada `mi_nombre` con el valor de "Matias". Esta variable es en realidad una referencia a un objeto. El tipo de objeto es `str` porque para poder crearla, necesitamos instanciar desde la clase `str` □ **`mi_nombre = "Matias"`**

Objetos y clases

Otros conceptos relacionados con clases y objetos

- **Atributos:** Son los *datos* que caracterizan al objeto. Son **variables** que almacenan datos relacionados al estado de un objeto.
- **Métodos (o funciones de miembro):** Son los que caracterizan su **comportamiento**, es decir, son todas las **acciones** (denominadas *operaciones*) que el objeto puede realizar por sí mismo. Estas operaciones hacen posible que el objeto responda a las solicitudes externas (o que actúe sobre otros objetos). Además, las operaciones están estrechamente ligadas a los atributos, ya que sus acciones pueden depender de, o modificar, los valores de un atributo.
- **Identidad:** El objeto tiene una *identidad*, que lo distingue de otros objetos, sin considerar su estado. Por lo general, esta identidad se crea mediante un **identificador** que deriva naturalmente de un problema (por ejemplo: un producto puede estar representado por un código, un automóvil por un número de modelo, etc.).

Mensajes y métodos

El modelado de objetos no sólo tiene en consideración los objetos de un sistema, sino también sus interrelaciones, es decir la interacción entre ellos.

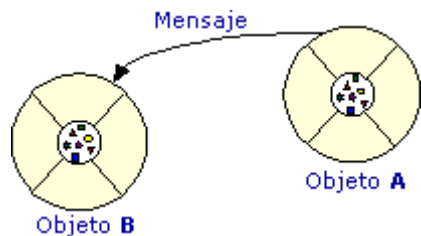
Mensaje

Los objetos interactúan enviándose mensajes unos a otros. Tras la recepción de un mensaje el objeto actuará. La acción puede ser el envío de otros mensajes, el cambio de su estado, o la ejecución de cualquier otra tarea que se requiera que haga el objeto.

Método

Un método se implementa en una clase, y determina cómo tiene que actuar el objeto cuando recibe un mensaje. El método es una acción que va a llevar adelante el objeto a través de la llamada de otro objeto.

Cuando un objeto A necesita que el objeto B ejecute alguno de sus métodos, el objeto A le manda un mensaje al objeto B.



Al recibir el mensaje del objeto A, el objeto B ejecutará el método adecuado para el mensaje recibido.

Objetos y clases

¿Cómo declaramos una clase y creamos objetos?

Para definir una clase conviene buscar un nombre de clase lo más próximo a lo que representa. La definimos con la palabra clave **class**, seguidamente del nombre de la clase y dos puntos.

Problema 1:

Implementar una clase llamada Persona que tendrá como atributo (variable) su nombre y dos métodos (funciones), uno de dichos métodos inicializará el atributo nombre y el siguiente método mostrará en la pantalla el contenido del mismo. Definir dos objetos de la clase Persona e incorporar una variable de clase (piernas).

Datos:


Clase: Persona

Variable: Nombre

Métodos: Inicializar e imprimir

Variable de clase: Piernas (2)

Objetos de la clase: 2



```
class Persona: #Creamos la clase
```

Esta clase tendrá como atributo (variable) su nombre y dos métodos (funciones), uno de dichos métodos inicializará el atributo nombre y el siguiente método mostrará en la pantalla el contenido del mismo. Además, definiremos un atributo de la clase llamado “piernas”.

```
piernas=2 #Variable de clase
```

Los métodos de una clase se definen utilizando la misma sintaxis que para la definición de funciones. Dentro del método diferenciamos los atributos del objeto antecediendo el identificador **self**.

```
def inicializar(self,nombre): #Constructor
    self.nombre=nombre

def imprimir(self): #Método para mostrar datos
    print("Nombre: {}".format(self.nombre))
```

*Todo método tiene como primer parámetro el identificador **self** que tiene la referencia del objeto que llamó al método.*

Con la asignación previa almacenamos en el atributo **nombre** el parámetro nom, los atributos siguen existiendo cuando finaliza la ejecución del método. Por ello cuando se ejecuta el método imprimir podemos mostrar el nombre que cargamos en el primer método.

Recordemos que una clase es un molde que nos permite definir objetos. Crearemos dos objetos de la clase Persona:

```
persona1=Persona()  
persona2=Persona()
```

Definimos un objeto llamado **persona1** y lo creamos asignándole el nombre de la clase con paréntesis abierto y cerrado al final (como cuando llamamos a una función).

Luego llamaremos a los métodos (funciones), para ello debemos disponer luego del nombre del objeto el operador . (punto) y por último el nombre del método (función).

En el caso que tenga parámetros se los enviamos (salvo el primer parámetro (self) que el mismo Python se encarga de enviar la referencia del objeto que se creó):

```
persona1.inicializar("Pedro") #Llamamos al constructor con un nombre  
persona1.imprimir() #Mostramos los datos
```

También podemos llamar a los métodos para el otro objeto creado:

```
persona2.inicializar("Carla")  
persona2.imprimir()
```



Ejercicio_1_POO.py

La declaración de clases es una de las ventajas fundamentales de la POO, ya que la reutilización de código (gracias a que está encapsulada en clases) es muy sencilla.

Objetos y clases

¿Cómo declaramos una clase y creamos objetos?

Problema 2:

Implementar una clase llamada Alumno que tenga como atributos su nombre y su nota. Definir los métodos para inicializar sus atributos, imprimirlos y mostrar un mensaje si su estado es “regular” (nota mayor o igual a 4). Definir dos objetos de la clase Alumno.

Datos:

Clase: Alumno

Variables: Nombre y nota

Métodos: Inicializar, imprimir y mostrar_estado

Objetos de la clase: 2

Declaramos la clase Alumno y definimos sus tres métodos, en el método inicializar llegan como parámetros aparte del self el nombre y nota del alumno:

```
class Alumno: #Creamos la clase

    def inicializar(self,nombre,nota): #Constructor
        self.nombre=nombre
        self.nota=nota

    def imprimir(self): #Método para mostrar los datos
        print("Nombre: {}".format(self.nombre))
        print("Nota: {}".format(self.nota))

    def mostrar_estado(self): #Método para ver si está aprobado
        if self.nota>=4:
            print("Regular")
        else:
            print("Desaprobado")
```

No hay problema que los atributos se llamen iguales a los parámetros ya que siempre hay que anteceder la palabra "self" al nombre del atributo: **self.nombre=nombre**

Tener en cuenta que cuando se crean los atributos en el método inicializar luego podemos acceder a los mismos en los otros métodos de la clase, por ejemplo, en el método **mostrar_estado** verificamos el valor almacenado en el atributo nota, creado dentro del método constructor inicializar.

Habíamos dicho que una clase es un **molde** que nos permite crear luego objetos de dicha clase, en este problema el molde **Alumno** lo utilizamos para crear dos objetos de dicha clase:

```
Obj 1 {
    alumno1=Alumno() #Creamos el objeto
    alumno1.inicializar("Diego",2) #Le damos 2 atributos (nombre y nota)
    alumno1.imprimir() #Imprimimos los datos
    alumno1.mostrar_estado() #Vemos si está aprobado
}

Obj 2 {
    alumno2=Alumno()
    alumno2.inicializar("Ana",10)
    alumno2.imprimir()
    alumno2.mostrar_estado()
}
```

Es fundamental la definición de objetos de una clase para que tenga sentido la declaración de dicha clase.

```
terminal
Nombre: Diego
Nota: 2
Desaprobado
Nombre: Ana
Nota: 10
Regular
```



Ejercicio_2_POO.py

Objetos y clases

Atributos de instancia

Dijimos que cuando creamos una clase estamos generando una *plantilla o molde* para un nuevo tipo de objeto que puede almacenar información y realizar acciones. Por ejemplo podemos crear la clase “Perro” con **class Perro** (se sugiere la notación CamelCase para los nombres de las clases):

- Agreguemos propiedades que todos los perros deberían tener (nombre, edad).
- Las propiedades de todos los objetos de tipo Perro deben definirse en un método denominado **__init__()**
 - **__init__()** define el estado inicial del objeto asignando valores a las propiedades del objeto.
 - Esto significa que **__init__()** inicializa *a cada nueva instancia* de la clase.
 - Le podemos pasar la cantidad de parámetros deseada, siempre y cuando el primero sea **self**.
- Los métodos se declaran de manera similar a las funciones con **def** como primer elemento.

```
class Perro:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
```

Objetos y clases

Atributos de clase

Los atributos de clase son atributos que tienen el mismo valor **para todas las instancias** (en este caso, para todos los objetos de tipo *perro*).

Estos atributos de clase pueden definirse por fuera del método `__init__()` y siempre se encontrarán directamente debajo de la definición del nombre de la clase.

Importante: cualquier modificación al valor de una variable de clase es **arrastrada** hacia las instancias.

Ejemplo: en la misma clase Perro podemos asignar un atributo “Género” que sea siempre el mismo para todos.

```
class Perro:
    # Atributo de Clase
    genero= "Canis"
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
```

Objetos y clases

Instancia

Para instanciar una clase en el interprete de Python escribimos el nombre de la clase con paréntesis. Esto hace que la instancia de Perro se coloque en una posición de memoria. Para poder almacenarla en una variable hacemos lo siguiente:

```
miMascota = Perro()  
otroPerro = Perro()
```

Para instanciar el perro con atributos de nombre y edad hacemos lo siguiente:

```
miMascota = Perro("Popey", 8)  
otroPerro = Perro("Bart", 5)
```

*Obviamos el parámetro **self** ya que es transparente al usuario en Python.*

Podemos acceder a cada atributo de cada instancia mediante el nombre de la variable y el nombre del atributo asociado:

```
miMascota.edad  
otroPerro.nombre
```

Si queremos cambiar un atributo podríamos hacer lo siguiente:

```
otroPerro.edad= 10  
otroPerro.genero= "Felis"
```

Métodos de instancias

- Los métodos de instancia son “funciones” definidas dentro de una clase que solo pueden ser llamadas desde la instancia de la clase.
- Como el método `__init__()`, en un método de instancia siempre el primer parámetro será ***self***.

```
class Perro:
    # Atributo de Clase
    genero= "Canis"
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
    # Método de instancia
    def imprimir(self):
        return f'{self.nombre} tiene {self.edad} años.'
    # Otro método de instancia
    def ladrar(self, sonido):
        return f'{self.nombre} dice {sonido}.'
```

Llamada a un método

Para llamar a cada método simplemente utilizamos el operador unario (el punto) y entre paréntesis pasamos los parámetros (que puede o no tener):

```
miMascota = Perro("Paka", 11)
miMascota.imprimir()
miMascota.ladRAR("Guau guau!")
```

Utilizando **print** podemos ver las propiedades de cada objeto:

```
print("Género:", miMascota.genero)
print(miMascota.imprimir())
print(miMascota.ladRAR("Guau, guau!"))
```

terminal
Género: Canis
Paka tiene 11 años.
Paka dice Guau, guau!.



Ejemplo_perro.py

Método `__init__` y `__del__` de la clase

El método `__init__` es un método especial de una clase en Python. Su objetivo fundamental es **inicializar** los atributos del objeto que creamos. Con este método podemos remplazar al método **inicializar**, utilizado en ejercicios anteriores.

El método `__del__` también es un método especial que será ejecutado cuando termine la ejecución del programa y el objeto sea eliminado.

Las ventajas de implementar el método `__init__` en lugar del método inicializar son:

1. El método `__init__` es el primer método que se ejecuta cuando se crea un objeto.
2. El método `__init__` se llama automáticamente. Es decir es imposible olvidarse de llamarlo ya que se llamará automáticamente al crear el objeto.
3. Quien utiliza POO en Python conoce el objetivo de este método.

Otras características son:

- Se ejecuta inmediatamente luego de crear un objeto.
- El método `__init__` no puede retornar dato.
- El método `__init__` puede recibir parámetros que se utilizan normalmente para inicializar atributos.
- El método `__init__` es un método opcional, de todos modos es muy común declararlo.

Método `__init__` y `__del__` de la clase

Sintaxis del constructor:

```
def __init__(self):  
    print('Método init llamado')
```

Debemos definir un método llamado `__init__` (es decir utilizamos dos caracteres de subrayado, la palabra `init` y seguidamente otros dos caracteres de subrayado). Lo mismo sucede con el método `__del__`

```
def __del__(self):  
    print('Método delete llamado')
```

Estos métodos se llamarán **automáticamente** al crear/instanciar al objeto, es decir que no debemos llamarlos en el programa principal.

Método `__init__` y `__del__` de la clase

Problema 3:

Confeccionar una clase que represente un empleado. Definir como atributos su nombre y su sueldo. En el método `__init__` cargar los atributos por teclado y luego en otro método imprimir sus datos y por último uno que imprima un mensaje si debe pagar impuestos (si el sueldo supera a 3000).

Datos:

Clase: Empleado

Variables: Nombre y sueldo

Métodos: `__init__`, `__del__`, imprimir, `paga_impuestos`

Creamos la clase el método `__init__`, que pedirá los datos para que se ingresen por teclado. Además crearemos el método para eliminar el objeto:

```
class Empleado:

    def __init__(self):
        self.nombre=input("Ingrese el nombre del empleado: ")
        self.sueldo=float(input("Ingrese el sueldo: "))

    def __del__(self):
        print('Método delete llamado')
```

Los otros dos métodos tienen por objeto mostrar los datos del empleado y mostrar una leyenda si paga impuestos o no:

```
def imprimir(self):
    print("Nombre: {}".format(self.nombre))
    print("Sueldo: {}".format(self.sueldo))

def paga_impuestos(self):
    if self.sueldo>3000:
        print("Debe pagar impuestos")
    else:
        print("No paga impuestos")
```

Desde el bloque principal creamos un objeto de la clase Empleado. No llamamos directamente al método `__init__` sino que se llama automáticamente luego de crear el objeto de la clase Empleado. Además debemos llamar explícitamente a estos dos métodos:

```
# Bloque Principal
empleado1=Empleado()
empleado1.imprimir()
empleado1.paga_impuestos()
```

```
Ingrese el nombre del empleado: Juan Pérez
Ingrese el sueldo: 3001
Nombre: Juan Pérez
Sueldo: 3001.0
Debe pagar impuestos
Método delete llamado
```

terminal

```
Ingrese el nombre del empleado: Juan López
Ingrese el sueldo: 2999
Nombre: Juan López
Sueldo: 2999.0
No paga impuestos
Método delete llamado
```

terminal

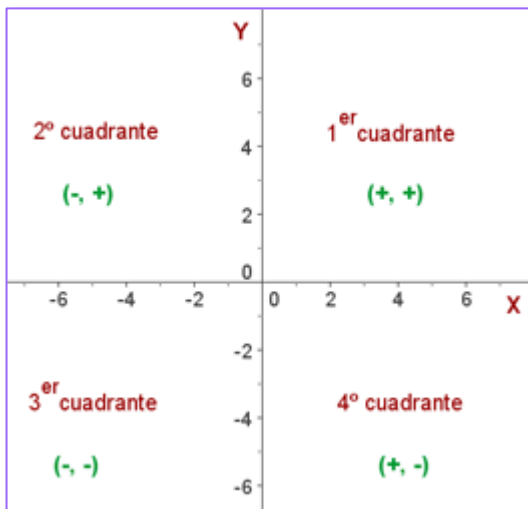


Ejercicio_3_POO.py

Método `__init__` y `__del__` de la clase

Problema 4:

Desarrollar una clase que represente un punto en el plano y tenga los siguientes métodos: inicializar los valores de x e y que llegan como parámetros, imprimir en que cuadrante se encuentra dicho punto (concepto matemático, primer cuadrante si x e y son positivas, si $x < 0$ e $y > 0$ segundo cuadrante, etc.)



Datos:

Clase: Punto

Variables: x e y

Métodos: `__init__`, `__del__`, `imprimir`, `imprimir_cuadrante`

En este problema el método `__init__` aparte del parámetro `self` que siempre va tenemos otros dos parámetros. Desde el bloque principal donde creamos un objeto de la clase `Punto` pasamos los datos a los parámetros.

```
class Punto:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __del__(self):
        print('Método delete llamado')

    def imprimir(self):
        print("Coordenada del punto: ({}:{}".format(self.x,self.y))

    def imprimir_cuadrante(self):
        if self.x>0 and self.y>0: print("Primer cuadrante")
        elif self.x<0 and self.y>0: print("Segundo cuadrante")
        elif self.x<0 and self.y<0: print("Tercer cuadrante")
        else: print("Cuarto cuadrante")

# Bloque principal
punto1=Punto(10,-30)
punto1.imprimir()
punto1.imprimir_cuadrante()
```

Recordemos que pasamos dos parámetros aunque el método `__init__` recibe 3. El parámetro **self** recibe la referencia de la variable `punto1` (es decir el objeto propiamente dicho).



Ejercicio_4_POO.py

Llamada de métodos desde otro método de la misma clase

Hasta ahora en todos los problemas planteados hemos llamado a los métodos desde donde definimos un objeto de dicha clase, por ejemplo:

```
empleado1=Empleado("Diego", 2000)
empleado1.paga_impuestos()
```

Utilizamos la sintaxis: **[nombre del objeto].[nombre del metodo]**, es decir antecedemos al nombre del método el nombre del objeto y el operador punto.

¿Qué pasa si queremos llamar dentro de la clase a otro método que pertenece a la misma clase?, la sintaxis es la siguiente:

```
self.[nombre del metodo]
```

***Importante:** esto sólo se puede hacer cuando estamos dentro de la misma clase.*

Llamada de métodos desde otro método de la misma clase

Problema 5:

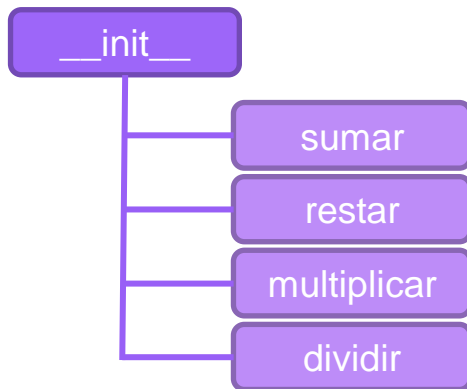
Plantear una clase Operaciones que solicite en el método `__init__` la carga de dos enteros e inmediatamente muestre su suma, resta, multiplicación y división. Hacer cada operación en otro método de la clase Operación y llamarlos desde el mismo método `__init__`


Datos:

Clase: Operacion

Variables: valor1 y valor2

Métodos: `__init__`, `__del__`, sumar, restar, multiplicar y dividir





Nuestro método `__init__` además de cargar los dos enteros procede a llamar a los métodos que calculan la suma, resta, multiplicación y división de los dos valores ingresados. La llamada de los métodos de la misma clase se hace antecediendo al nombre del método la palabra `self`:

```
class Operacion:

    def __init__(self):
        self.valor1=int(input("Ingrese primer valor:"))
        self.valor2=int(input("Ingrese segundo valor:"))
        self.sumar()
        self.restar()
        self.multiplicar()
        self.dividir()

    def __del__(self):
        print('Método delete llamado')
```

El método que calcula la suma de los dos atributos cargados en el método `__init__` define una variable local llamada `suma` y guarda la suma de los dos atributos. Posteriormente muestra la suma por pantalla:

```
def sumar(self):  
    suma=self.valor1+self.valor2  
    print("La suma es: {}".format(suma))
```

De forma similar los otros métodos calculan la resta, multiplicación y división de los dos valores ingresados:

```
def restar(self):  
    resta=self.valor1-self.valor2  
    print("La resta es: {}".format(resta))  
  
def multiplicar(self):  
    producto=self.valor1*self.valor2  
    print("El producto es: {}".format(producto))  
  
def dividir(self):  
    division=self.valor1/self.valor2  
    print("La division es: {}".format(division))
```

En el bloque principal de nuestro programa solo requerimos crear un objeto de la clase Operación ya que el resto de los métodos se llama en el método `__init__`:

```
operacion1=Operacion()
```



Ejercicio_5_POO.py

Llamada de métodos desde otro método de la misma clase

Problema 6:

Plantear una clase que administre dos listas de 5 nombres de alumnos y sus notas.

Mostrar un menú de opciones que permita:

1- Cargar alumnos.

2- Listar alumnos.

3- Mostrar alumnos con notas mayores o iguales a 7.

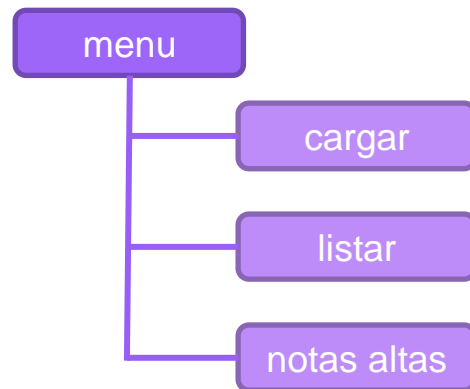
4- Finalizar programa.

Datos:

Clase: Alumnos

Variables: listas [nombres] y [notas]

Métodos: __init__, __del__, menu, cargar, listar, notas_altas y finalizar



Crearemos la clase y los métodos `__init__` y `__del__`:

```
class Alumnos:

    def __init__(self):
        self.nombres=[]
        self.notas=[]

    def __del__(self):
        print('Método delete llamado')
```

Con el método `menu` crearemos el menú de opciones:

```
def menu(self):
    opcion=0
    while opcion!=4:
        print("1- Cargar alumnos")
        print("2- Listar alumnos")
        print("3- Listado de alumnos con notas mayores o iguales a 7")
        print("4- Finalizar programa")
        opcion=int(input("Ingrese su opcion:"))
        if opcion==1: self.cargar()
        elif opcion==2: self.listar()
        elif opcion==3: self.notas_altas()
        else: self.finalizar()
```

Los demás métodos dependerán de la opción elegida en el menú:

Op 1:
Cargar
alumnos

```
def cargar(self):  
    for x in range(5):  
        nombre=input("Ingrese nombre del alumno:")  
        self.nombres.append(nombre)  
        nota=int(input("Nota del alumno:"))  
        self.notas.append(nota)
```

Op 2:
Mostrar
alumnos

```
def listar(self):  
    print("Listado completo de alumnos")  
    for x in range(5):  
        print(self.nombres[x],self.notas[x])  
    print("_____")
```

Op 3:
Mostrar
notas
altas

```
def notas_altas(self):  
    print("Alumnos con notas superiores o iguales a 7")  
    for x in range(5):  
        if self.notas[x]>=7:  
            print(self.nombres[x],self.notas[x])  
    print("_____")
```

Op 4:
Salir

```
def finalizar(self):  
    print("Programa finalizado!")  
    print("_____")
```

En el bloque principal
creamos el objeto y
llamamos al método
menú:

```
# Bloque principal  
alumnos=Alumnos()  
alumnos.menu()
```



Ejercicio_6_POO.py

Colaboración de clases

Normalmente en un problema resuelto con la metodología de programación orientada a objetos no interviene una sola clase, sino que hay **muchas clases que interactúan** y se comunican. Plantearemos un problema separando las actividades en dos clases.

Problema 7:

Un banco tiene 3 clientes que pueden hacer depósitos y extracciones. También el banco requiere que al final del día calcule la cantidad de dinero que hay depositado.

Lo primero que hacemos es identificar las clases: Cliente y Banco, luego debemos definir los atributos y los métodos de cada clase:

Cliente

atributos

nombre

monto

métodos

__init__

depositar

extraer

retornar_monto

Banco

atributos

3 Cliente (3 objetos de la clase Cliente)

métodos

__init__

operar

depositos_totales



Primero hacemos la declaración de la clase **Cliente**, en el método `__init__` inicializamos los atributos nombre con el valor que llega como parámetro y el atributo monto con el valor cero. Recordemos que en Python para diferenciar un atributo de una variable local o un parámetro le antecedemos la palabra clave `self` (es decir nombre es el parámetro y `self.nombre` es el atributo):

```
class Cliente:

    def __init__(self,nombre):
        self.nombre=nombre
        self.monto=0
```

El resto de los métodos de la clase Cliente quedará de la siguiente manera

```
def depositar(self,monto):
    self.monto=self.monto+monto

def extraer(self,monto):
    self.monto=self.monto-monto

def retornar_monto(self):
    return self.monto

def imprimir(self):
    print("{} tiene depositada la suma de {}".format(self.nombre,self.monto))
```

En el bloque principal no se requiere crear objetos de la clase Cliente, esto debido a que los clientes son atributos del Banco.

Luego creamos la clase Banco, que tendrá 3 objetos de la clase Cliente:

```
def __init__(self):  
    self.cliente1=Cliente("Juan")  
    self.cliente2=Cliente("Ana")  
    self.cliente3=Cliente("Diego")
```

Con el método **operar()** llamamos a algunos métodos de la clase Cliente: los 3 clientes depositaron diferentes montos y uno de ellos realizó una extracción:

```
def operar(self):  
    self.cliente1.depositar(100)  
    self.cliente2.depositar(150)  
    self.cliente3.depositar(200)  
    self.cliente3.extraer(150)
```

Con el método **depositos_totales()** sumamos los montos de los 3 clientes, los guardamos en la variable **total** que imprimiremos y llamamos al método **imprimir** de la clase Cliente:

```
def depositos_totales(self):  
    total=self.cliente1.retornar_monto()+self.cliente2.retornar_monto()+self.cliente3.retornar_monto()  
    print("El total de dinero del banco es: {}".format(total))  
    self.cliente1.imprimir()  
    self.cliente2.imprimir()  
    self.cliente3.imprimir()
```


En el programa principal creamos el objeto Banco y llamaremos a los métodos **operar()** y **depositos_totales()**:

```
banco1=Banco()  
banco1.operar()  
banco1.depositos_totales()
```

```
El total de dinero del banco es: 300  
Juan tiene depositada la suma de 100  
Ana tiene depositada la suma de 150  
Diego tiene depositada la suma de 50
```

terminal



Ejercicio_7_POO.py

¿Cómo es, entonces, el flujo del programa?

1. Se crea el objeto de tipo Banco que tendrá 3 clientes: Juan, Ana y Diego.
2. Al llamar al método operar() de la clase Banco se llama a los métodos depositar() y extraer() de la clase Cliente.
3. Al llamar al método depositos_totales() de la clase Banco se llama al método retornar_monto() de la clase Cliente y al método imprimir() de la misma clase que mostrará el nombre y lo que tiene depositado.

Colaboración de clases

Problema 8:

Plantear un programa que permita jugar a los dados. Las reglas de juego son: se tiran tres dados y si los tres salen con el mismo valor se debe mostrar un mensaje que diga "ganó", sino "perdió".

Lo primero que hacemos es identificar las clases: Dado y JuegoDeDados, luego debemos definir los atributos y los métodos de cada clase:

Dado

atributos

valor

métodos

tirar

imprimir

retornar_valor

JuegoDeDados

atributos

3 Dado (3 objetos de la clase Dado)

métodos

__init__

jugar



Importamos el módulo "random" de la biblioteca estándar de Python ya que requerimos utilizar la función randint: **import random**

La clase Dado define un método tirar que almacena en el atributo valor un número aleatorio comprendido entre 1 y 6.

Los otros dos métodos de la clase Dado tienen por objetivo mostrar el valor del dado y retornar dicho valor a otra clase que lo requiera.

```
class Dado:

    def tirar(self):
        self.valor=random.randint(1,6)

    def imprimir(self):
        print("Valor del dado: {}".format(self.valor))

    def retornar_valor(self):
        return self.valor
```

La clase JuegoDeDados define tres atributos de la clase Dado, en el método `__init__` crea dichos objetos:

```
class JuegoDeDados:

    def __init__(self):
        self.dado1=Dado()
        self.dado2=Dado()
        self.dado3=Dado()

    def jugar(self):
        self.dado1.tirar()
        self.dado1.imprimir()
        self.dado2.tirar()
        self.dado2.imprimir()
        self.dado3.tirar()
        self.dado3.imprimir()
        if self.dado1.retornar_valor()==self.dado2.retornar_valor() and
self.dado1.retornar_valor()==self.dado3.retornar_valor():
            print("Ganó")
        else:
            print("Perdió")
```

En el bloque principal se crea el objeto JuegoDeDados y se llama al método jugar() del mismo método:

```
juego_datos=JuegoDeDados()  
juego_datos.jugar()
```

```
Valor del dado: 3  
Valor del dado: 2  
Valor del dado: 6  
Perdió
```

terminal

```
Valor del dado: 3  
Valor del dado: 3  
Valor del dado: 3  
Ganó
```

terminal



Ejercicio_8_POO.py

¿Cómo es, entonces, el flujo del programa?

1. Se crea el objeto de tipo JuegoDeDados que tendrá 3 dados (objetos).
2. Al llamar al método jugar() de la clase JuegoDeDados se llama a los métodos tirar() e imprimir() de la clase Dado. En el primer caso se genera un número aleatorio entre 1 y 6, simulando la tirada del dado y en el segundo se muestra el valor del dado.
3. El mismo método jugar() también llama al método retornar_valor() de cada objeto Dado que devolverá el valor de cada uno de ellos. Ese valor devuelto se compara para determinar si los 3 dados son iguales (ganó) o no (perdió) dentro de una estructura condicional.

Acotación

Para cortar una línea en varias líneas en Python podemos encerrar entre paréntesis la condición:

```
if (self.dado1.retornar_valor()==self.dado2.retornar_valor()  
    and self.dado1.retornar_valor()==self.dado3.retornar_valor()):
```

O agregar una barra al final:

```
if self.dado1.retornar_valor()==self.dado2.retornar_valor() and \  
    self.dado1.retornar_valor()==self.dado3.retornar_valor():
```

Variables de clase

Hemos visto cómo definimos atributos en una clase anteponiendo la palabra clave self:

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

Los atributos son independientes por cada objeto o instancia de la clase, es decir si definimos tres objetos de la clase Persona, todas las personas tienen un atributo nombre pero cada uno tiene un valor independiente.

En algunas situaciones necesitamos almacenar datos que sean compartidos por todos los objetos de dicha clase, en esas situaciones debemos emplear variables de clase.

Para definir una variable de clase lo hacemos dentro de la clase pero fuera de sus métodos:

```
class Persona:
    variable=20

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

```
# Bloque principal
persona1=Persona("Juan")
persona2=Persona("Ana")
persona3=Persona("Luis")

print(persona1.nombre) # Juan
print(persona2.nombre) # Ana
print(persona3.nombre) # Luis

print(persona1.variable) # 20
Persona.variable=5
print(persona2.variable) # 5
```

	terminal
Juan	
Ana	
Luis	
20	
5	

Se reserva solo un espacio para la variable "variable", independientemente que se definan muchos objetos de la clase Persona. La variable "variable" es compartida por todos los objetos persona1, persona2 y persona3.

Para modificar la variable de clase hacemos referencia al nombre de la clase y seguidamente el nombre de la variable:

```
Persona.variable=5
```



Variables_de_clase.py

Variables de clase

Problema 9:

Definir una clase Cliente que almacene un código de cliente y un nombre.

En la clase Cliente definir una variable de clase de tipo lista que almacene todos los clientes que tienen suspendidas sus cuentas corrientes.

Imprimir por pantalla todos los datos de clientes y el estado que se encuentra su cuenta corriente.

Cliente

atributos

código
nombre

métodos

__init__
imprimir
esta_suspendido
suspender

variables de clase

suspendidos (lista)

Se crearán 4 clientes:

1. Juan
2. Ana
3. Diego (cuenta suspendida)
4. Pedro (cuenta suspendida)

La clase Cliente define una variable de clase llamada suspendidos que es de tipo lista y por ser variable de clase es compartida por todos los objetos que definamos de dicha clase.

```
class Cliente:
    suspendidos=[] #Variable de Clase

    def __init__(self,codigo,nombre):
        self.codigo=codigo #Variable de Instancia
        self.nombre=nombre #Variable de Instancia
```

En el método imprimir mostramos el código, nombre del cliente y si se encuentra suspendida su cuenta corriente.

```
def imprimir(self):
    print("Codigo: {}".format(self.codigo))
    print("Nombre: {}".format(self.nombre))
    self.esta_suspendido()
```

El método suspender lo que hace es agregar el código de dicho cliente a la lista de clientes suspendidos.

```
def suspender(self):
    Cliente.suspendidos.append(self.codigo)
```

El método que analiza si está suspendido el cliente verifica si su código se encuentra almacenado en la variable de clase suspendidos.

```
def esta_suspendido(self):  
    if self.codigo in Cliente.suspendidos:  
        print("Esta suspendido")  
    else:  
        print("No esta suspendido")  
    print("_____")
```

Dentro del cuerpo principal del programa crearemos los 4 clientes (objetos)...

```
cliente1=Cliente(1,"Juan")  
cliente2=Cliente(2,"Ana")  
cliente3=Cliente(3,"Diego")  
cliente4=Cliente(4,"Pedro")
```

... y luego suspenderemos al cliente 3 y 4:

```
cliente3.suspender()  
cliente4.suspender()
```

Imprimiremos los datos de los 4 clientes:

```
cliente1.imprimir()  
cliente2.imprimir()  
cliente3.imprimir()  
cliente4.imprimir()
```

```
Codigo: 1  
Nombre: Juan  
No esta suspendido
```

```
Codigo: 2  
Nombre: Ana  
No esta suspendido
```

```
Codigo: 3  
Nombre: Diego  
Esta suspendido
```

```
Codigo: 4  
Nombre: Pedro  
Esta suspendido
```

terminal

*Es importante remarcar que todos los objetos acceden a una única lista llamada **suspendidos** gracias a que se definió como **variable de clase**.*



Ejercicio_9_POO.py

Podemos imprimir la variable de clase suspendidos de la clase Cliente:

```
print(Cliente.suspendidos)
```

```
[3, 4]
```

terminal

Método especial `__str__`

- Si llamamos a nuestra instancia **miMascota** mediante **`print(miMascota)`** veremos información sobre la misma que no es clara para el ojo humano inexperto:

`<__main__.Perro object at 0x000001CD92161DF0>`

Imprime la dirección de memoria donde está almacenado el objeto

- Entonces se puede agregar información relacionada con el significado que le hemos dado previamente. Eso lo hacemos con el método **`__str__()`**.
- Tanto el método **`__init__`** como el **`__str__`** se denominan **métodos mágicos** en Python y se escriben entre dobles guiones bajos (Dunder, Double Underscores).
- Ahora al realizar **`print(miMascota)`**, obtendremos la descripción del objeto.

```
# Se puede reemplazar el método imprimir() con __str__()
def __str__(self):
    return f'{self.nombre} tiene {self.edad} años.'
```

*El método `__str__` nos trae una cadena que **describe** al objeto, con información del objeto.*

Paka tiene 11 años.

terminal

Método especial `__str__`

Podemos hacer que se ejecute un método definido por nosotros cuando pasamos un objeto a la función **`print`** o cuando llamamos a la función **`str (convertir a string)`**. ¿Qué sucede cuando llamamos a la función **`print`** y le pasamos como parámetro un objeto?

```
class Persona:
    def __init__(self,nom,ape):
        self.nombre=nom
        self.apellido=ape

#Programa principal
persona1=Persona("José", "Rodríguez")
print(persona1)
```

Nos muestra algo parecido a esto:

```
<__main__.Persona object at 0x0000016AE124B5B0>
```

Método especial `__str__`

Python nos permite redefinir el método que se debe ejecutar. Esto se hace definiendo en la clase el método especial `__str__`

En el ejemplo anterior si queremos que se muestre el nombre y apellido separados por coma cuando llamemos a la función ***print*** el código que debemos implementar es el siguiente:

```
class Persona:
    def __init__(self,nom,ape):
        self.nombre=nom
        self.apellido=ape

    def __str__(self):
        cadena=self.nombre + " " + self.apellido
        return cadena
```

Como vemos debemos implementar el método `__str__` y retornar un **string**, este luego será el que imprime la función `print`.

Método especial __str__

En el programa principal ejecutaremos lo siguiente:

```
#Programa principal
persona1=Persona("José", "Rodríguez")
persona2=Persona("Ana", "Martínez")
print("{} - {}".format(persona1, persona2))
```

José Rodríguez - Ana Martínez

terminal



metodo_str.py

Método especial `__str__`

Problema 10:

Definir una clase llamada *Punto* con dos atributos *x* e *y*.

Crearle el método especial `__str__` para retornar un string con el formato *(x,y)*.

Punto:

atributos

x

y

métodos

`__init__`

`__str__`

La clase *Punto* define dos métodos especiales. El método `__init__` donde inicializamos los atributos *x* e *y*.

Y el segundo método especial que definimos es el `__str__` que debe retornar un string.

```
def __init__(self, x, y):  
    self.x=x  
    self.y=y  
  
def __str__(self):  
    return "({},{})".format(self.x, self.y)
```

Luego en el programa principal después de definir dos objetos de la clase Punto procedemos a llamar a la función print y le pasamos cada uno de los objetos.

Hay que tener en cuenta que cuando pasamos a la función print el objeto punto1 en ese momento se llama el método especial **__str__** que tiene por objetivo retornar un string que nos haga más legible lo que representa dicho objeto.

```
# Programa principal
punto1=Punto(10,3)
punto2=Punto(3,4)
print(punto1)
print(punto2)
```

```
(10,3)
(3,4)
```

terminal



Ejercicio_10_POO.py

Método especial __str__

Problema 11:

Declarar una clase llamada Familia. Definir como atributos el nombre del padre, madre y una lista con los nombres de los hijos.

Definir el método especial __str__ que retorne un string con el nombre del padre, la madre y de todos sus hijos.

Familia:

atributos

padre

madre

hijos (lista)

métodos

__init__

__str__

Para resolver este problema el método `__init__` recibe en forma obligatoria el nombre del padre, madre y en forma opcional una lista con los nombres de los hijos.

Si no tiene hijos la familia, el atributo hijos almacena una lista vacía.

```
class Familia:

    def __init__(self, padre, madre, hijos=[]):
        self.padre=padre
        self.madre=madre
        self.hijos=hijos
```

El método especial `__str__` genera un string con los nombres del padre, madre y todos los hijos.

```
def __str__(self):  
    cadena=self.padre+", "+self.madre  
    for hijo in self.hijos:  
        cadena=cadena+", "+hijo  
    return cadena
```

En el programa principal crearemos y mostraremos 3 objetos Familia, algunos con un hijo, dos hijos o sin hijos:

```
# Programa principal  
familia1=Familia("Pablo","Ana",["Pepe","Julio"])  
familia2=Familia("Jorge","Carla")  
familia3=Familia("Luis","Maria",["Marcos"])  
  
print(familia1)  
print(familia2)  
print(familia3)
```

```
Pablo,Ana,Pepe,Julio  
Jorge,Carla  
Luis,Maria,Marcos
```

terminal



Ejercicio_11_POO.py

Objetos dentro de objetos

Al ser las clases un nuevo tipo de dato se pueden poner en colecciones e incluso utilizarse dentro de otras clases.

```
class Pelicula:

    # Constructor de clase
    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print('Se ha creado la película:', self.titulo)

    def __str__(self):
        return '{} ({}).format(self.titulo, self.lanzamiento)
```

continúa...

Objetos dentro de objetos

```
class Catalogo:

    peliculas = [] # Esta lista contendrá objetos de la clase Pelicula

    def __init__(self, peliculas=[]):
        Catalogo.peliculas = peliculas

    def agregar(self, p): # p será un objeto Pelicula
        Catalogo.peliculas.append(p)

    def mostrar(self):
        for p in Catalogo.peliculas:
            print(p) # Print toma por defecto str(p)
```

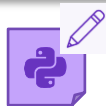
continúa....

Objetos dentro de objetos

```
#Programa principal
p = Pelicula("El Padrino", 175, 1972)
c = Catalogo([p]) # Añado una lista con una película desde el principio
c.mostrar()
c.agregar(Pelicula("El Padrino: Parte 2", 202, 1974)) # Añadimos otra
c.mostrar()
```

```
Se ha creado la película: El Padrino
El Padrino (1972)
Se ha creado la película: El Padrino: Parte 2
El Padrino (1972)
El Padrino: Parte 2 (1974)
```

terminal



Ejercicio_12_POO.py

Encapsulación

El **encapsulamiento o encapsulación** 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. Python por defecto no oculta los atributos y métodos de una clase al exterior, por ejemplo:

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

mi_clase = Clase("Que tal")
print(mi_clase.atributo_clase)
print(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
print(mi_clase.atributo_clase) # Ok!
mi_clase.metodo_normal() # Ok!
```

Y como curiosidad, podemos hacer uso de **dir** para ver el listado de métodos y atributos de nuestra clase. Podemos ver claramente como tenemos el metodo_normal y el atributo de clase, pero no podemos encontrar __mi_metodo ni __atributo_clase.

```
print(dir(mi_clase))

#['_Class__atributo_clase', '_Class__mi_metodo', '_Class__variable',
#'_class__', '_delattr__', '_dict__', '_dir__', '_doc__', '_eq__',
#'_format__', '_ge__', '_getattribute__', '_gt__', '_hash__', '_init__',
#'_init_subclass__', '_le__', '_lt__', '_module__', '_ne__', '_new__',
#'_reduce__', '_reduce_ex__', '_repr__', '_setattr__', '_sizeof__',
#'_str__', '_subclasshook__', '_weakref__', 'atributo_clase', 'metodo_normal']
```

Pues bien, en realidad si que podríamos acceder a **__atributo_clase** y a **__mi_metodo** haciendo un poco de trampa. Aunque no se vea a simple vista, si que están pero con un nombre distinto, para de alguna manera ocultarlos y evitar su uso. Pero podemos llamarlos de la siguiente manera, pero por lo general **no es una buena idea**.

```
print(mi_clase._Class__atributo_clase)
# 'Hola'
mi_clase._Class__mi_metodo()
# 'Haz algo'
```

Fuente del ejemplo:

<https://ellibrodepython.com/encapsulamiento-poo>



encapsulamiento.py

Encapsulación: atributos privados

La **encapsulación** consiste en denegar el acceso a los atributos y métodos internos de la clase desde el exterior, para **protegerlos**. En Python no existe, pero se puede simular precediendo atributos y métodos con **dos barras bajas** `__` como indicando que son “especiales”. En el caso de los atributos quedarían así:

```
class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera."

e = Ejemplo()
print(e.__atributo_privado)
```

Y en los métodos...

```
class Ejemplo:
    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera.")

e = Ejemplo()
e.__metodo_privado()
```

Encapsulación: atributos privados

¿Qué sentido tiene esto en Python? Ninguno, porque se pierde toda la gracia de lo que en esencia es el lenguaje: **flexibilidad** y **polimorfismo** sin control (veremos esto más adelante).

Sea como sea, para acceder a esos datos se deberían crear métodos públicos que hagan de interfaz. En otros lenguajes les llamaríamos **getters y setters** y es lo que da lugar a las *propiedades*, que no son más que atributos protegidos con interfaces de acceso.

```
class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera."

    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera.")

    def atributo_publico(self):
        return self.__atributo_privado

    def metodo_publico(self):
        return self.__metodo_privado()
```

```
e = Ejemplo()
print(e.atributo_publico())
e.metodo_publico()
```

```
Soy un atributo inalcanzable desde fuera.
Soy un método inalcanzable desde fuera.
```

terminal

Getters y Setters en Python

- Los **getters** serían las funciones que nos permiten acceder a una variable privada. En Python se declaran creando una función con el decorador **@property**.
- Los **setters** serían las funciones que usamos para sobrescribir la información de una variable y se generan definiendo un método con el nombre de la variable sin guiones y utilizando como decorador el nombre de la variable sin guiones más ".setter".

```
class ListadoBebidas:

    def __init__(self):
        self.__bebida = 'Naranja'
        self.__bebidas_validas = ['Naranja', 'Manzana']

    @property
    def bebida(self):
        return "La bebida oficial es: {}".format(self.__bebida)

    @bebida.setter
    def bebida(self, bebida):
        self.__bebida = bebida
```

Getters y Setters en Python

En este ejemplo declaramos dos variables, una llamada `_bebida` y una lista llamada `_bebidas_validas`. Para recuperar la información de la variable `_bebida` tendremos que hacerlo con el objeto y el nombre de la función `bebida`.

```
#Programa principal
bebidas= ListadoBebidas()
print(bebidas.bebida)
bebidas.bebida = 'Limonada'
print(bebidas.bebida)
```



Ejercicio_13_POO.py

```
La bebida oficial es: Naranja
La bebida oficial es: Limonada
```

terminal

Para ampliar (ejemplo): https://pythones.net/propiedades-en-python-oop/#Propiedades_de_atributos_de_clase_en_Python_Getter_Setter_y_Deleter