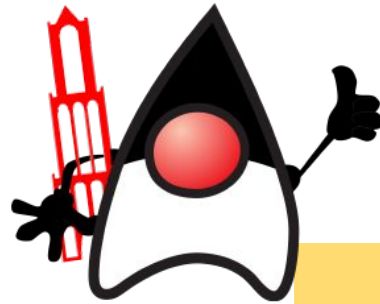




# Curso FullStack Python

Codo a Codo 4.0



# Python

## ***Parte 6***



# Temas que veremos las próximas clases

P.O.O

*Objetos*

*Clases*

*Abstracción*

*Encapsulamiento*

*Polimorfismo*

*Herencia*

# 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!"))
```

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

**terminal**



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

**terminal**

Ingrese el sueldo: 3001

Nombre: Juan Pérez

Sueldo: 3001.0

Debe pagar impuestos

Método delete llamado

Ingrese el nombre del empleado: Juan López

**terminal**

Ingrese el sueldo: 2999

Nombre: Juan López

Sueldo: 2999.0

No paga impuestos

Método delete llamado

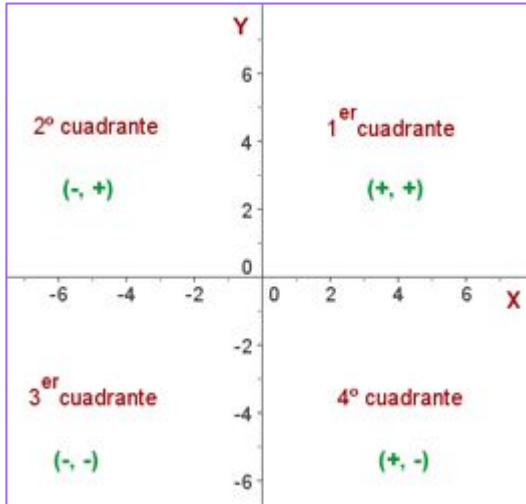


*Ejercicio\_3\_POO.py*

# 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:

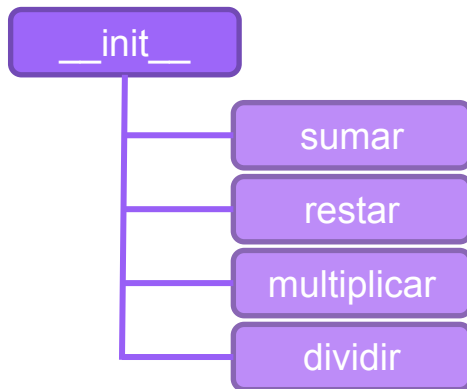
*Plantear una clase Operaciones que solicite en el método `__init__` la carga de dos enteros e inmediatamente muestre su suma, resta, multiplicación y división. Hacer cada operación en otro método de la clase Operación y llamarlos desde el mismo método `__init__`*


### 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.*

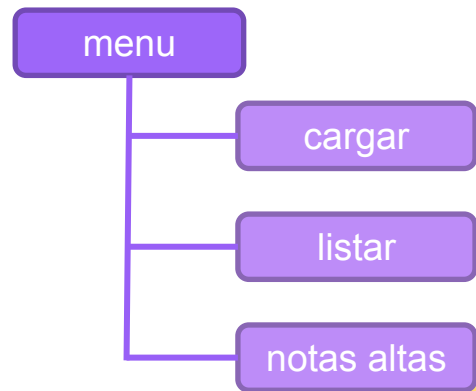
*4- Finalizar programa.*

### 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_dados=JuegoDeDados()  
juego_dados.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](https://pythones.net/propiedades-en-python-oop/#Propiedades_de_atributos_de_clase_en_Python_Getter_Setter_y_Deleter)