

# Esta clase va a ser

- grabada

a

Clase 12. PYTHON

# Programación orientada a Objetos II

# Temario

11

## Programación Orientada a Objetos I

- ✓ ¿Qué es la POO?
- ✓ Relaciones entre clases

12

## Programación orientada a objetos II

- ✓ [Clases](#)
- ✓ [Atributos](#)
- ✓ [Métodos](#)
- ✓ [Clases anidadas](#)
- ✓ [Encapsulamiento](#)

13

## Herencias

- ✓ Herencia
- ✓ Herencia múltiple
- ✓ Poliformismo

# Objetivos de la clase

- **Conocer** los métodos y atributos.
- **Crear** los primeros objetos y anidarlos.
- **Identificar** los principios del encapsulamiento.

# Repositorio Github

Te dejamos el acceso al Repositorio de Github donde encontrarás todo el material complementario y scripts de la clase.

✓ [Repositorio Python](#)



# Retomando definiciones



¿Qué era la POO?

# ¿Qué era la POO?



Es un modo o **paradigma de programación**, que nos permite **organizar el código pensando el problema como una relación entre "cosas"**, denominadas **objetos**.

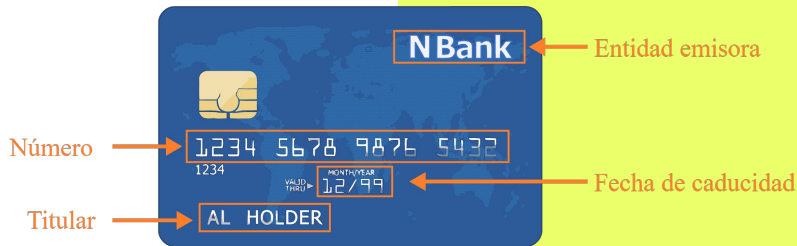
Los objetos **se trabajan utilizando las clases**. Estas nos permiten **agrupar un conjunto de variables y funciones** que veremos a continuación. 🙌



# Atributos

Cosas de lo más cotidianas como una computadora o un coche pueden ser representadas con **clases**. Estas clases **tienen diferentes características**, que en el caso de la computadora podrían ser la marca o modelo. Llamaremos a estas características, **atributos**.

Atributos



Métodos

Activar



Pagar

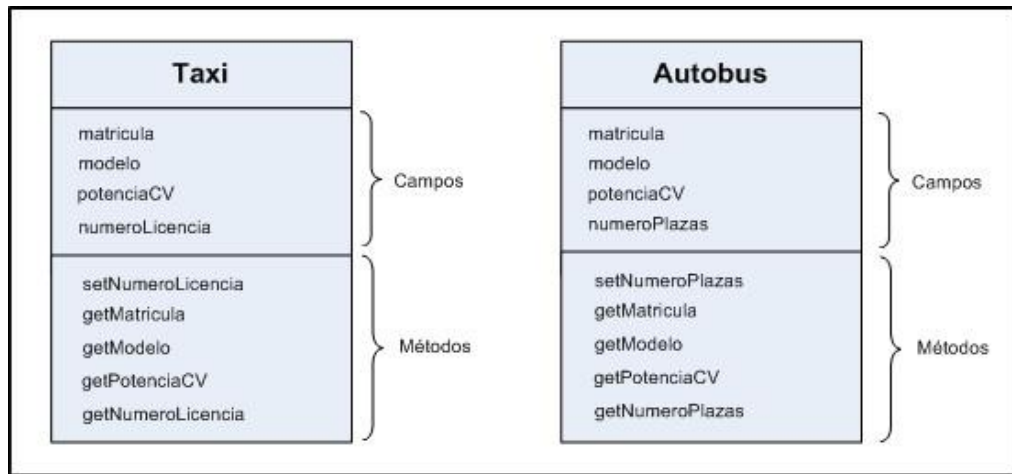


Anular



# Métodos

Por otro lado, las **clases** tienen un conjunto de funcionalidades. En el caso de la computadora podría ser imprimir o reproducir. Llamaremos a estas funcionalidades **métodos**.



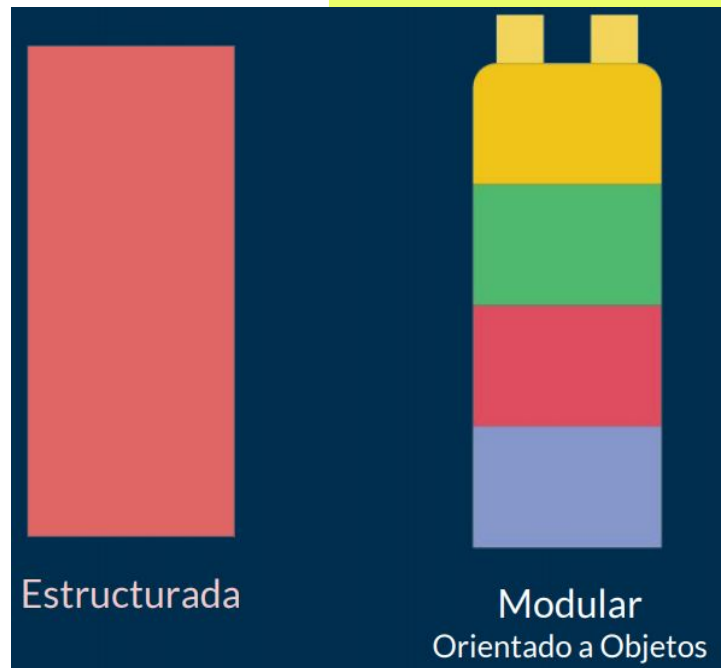
# Objetos

Por último, pueden existir diferentes tipos de computadoras. Llamaremos a estos diferentes tipos de computadoras objetos.

Es decir, **el concepto abstracto de computadora es la clase, pero cualquier tipo de computadora particular será el objeto.**

# ¿Para qué usar la POO?

En el mundo de la programación hay gran cantidad de aplicaciones que realizan tareas muy similares y es importante identificar los patrones que nos permiten no reinventar la rueda. La programación orientada a objetos intentaba resolver esto.





PARA RECORDAR

Python está diseñado para soportar el paradigma de la programación orientado a objetos (POO), donde todos los módulos se tratan como entidades que tienen atributos, y comportamiento. Lo cual, en muchos casos, ayuda a crear programas de una manera sencilla, además de que los componentes son reutilizables.

# Classes

# Uso de POO: clases



Lo primero es crear una clase, a la cual llamaremos “perro”. Se trata de una clase vacía y sin mucha utilidad práctica, pero es la mínima clase que podemos crear. El uso del PASS realmente no tiene utilidad, pero daría un error si después de los : no tenemos contenido.

```
1 class Perro:  
2     pass
```

# Instanciar una clase



Ahora que tenemos la clase, podemos crear uno o más objetos de la misma, como si de una variable normal se tratase.

Nombre de la variable igual a la clase con ( ). Dentro de los paréntesis irían los parámetros de entrada si los hubiera.

```
class Perro:  
    pass  
  
perro1 = Perro()  
perro2 = Perro()
```



# Atributos

# Asignando atributos

A la hora de añadir atributos a la clase, es importante distinguir dos tipos:

- ✓ **Atributos de instancia:** Pertenecen a la instancia de la clase o al objeto. Son atributos particulares de cada instancia, en nuestro caso de cada perro.
- ✓ **Atributos de clase:** Se trata de atributos que pertenecen a la clase, por lo tanto serán comunes para todos los objetos.

# Asignando atributos de instancia



Empecemos creando el nombre y la raza del perro. Esto se suele hacer por medio de un método `__init__` que será llamado automáticamente cuando se cree un objeto. No se le puede cambiar el nombre y tiene su definición propia, "**Constructor**".

```
class Perro:

    #Constructor de la clase
    def __init__(self, nombre, raza):

        #Atributos de la instancia
        self.nombre = nombre
        self.raza = raza
```



## Para pensar

¿Para qué crees que se utiliza la variable SELF?

¿Verdadero o falso?

# ¿Qué es el self?

Es una variable que representa la instancia de la clase, y deberá estar siempre ahí.

El uso de `__init__` y el doble `__` no es una coincidencia. Cuando veas un método con esa forma, significa que está reservado para un uso especial del lenguaje. En este caso sería lo que se conoce como **constructor**.

# Asignando atributos de clase



Hasta ahora hemos definido atributos de instancia, ya que pertenecen a cada perro concreto. Ahora vamos a definir un atributo de clase, que será común para todos los perros. Por ejemplo, la especie de los perros es algo común para todos los objetos Perro.

```
class Perro:

    #Atributos de clase
    especie = "Mamífero"

    #Constructor de la clase
    def __init__(self, nombre, raza):

        #Atributos de la instancia
        self.nombre = nombre
        self.raza = raza
```

# Asignando atributos de clase



Dado que es un atributo de clase, no es necesario crear un objeto para acceder al atributo.

Se puede acceder también al atributo de clase desde el objeto.

😊 De esta manera, todos los objetos que se creen de la clase **perro** compartirán ese atributo de clase.

```
print(Perro.especie)  
# mamífero
```

```
mi_perro = Perro("Toby", "Bulldog")  
mi_perro.especie  
# 'mamífero'
```



# Estructura final de una clase con atributos

```
14 class Perro:
15
16     #Atributos de clase
17     especie = "Mamífero"
18
19     #Constructor de la clase
20     def __init__(self, nombre, raza):
21
22         #Atributos de la instancia
23         self.nombre = nombre
24         self.raza = raza
25
26
27 perro1 = Perro("Sammy", "Caniche")
28
29 print(f"Su nombre es: {perro1.nombre}")
30 print(f"Su raza es: {perro1.raza}")
31 print(f"Es un: {perro1.especie}")
```

PROBLEMS   OUTPUT   **TERMINAL**   DEBUG CONSOLE

Su nombre es: Sammy  
Su raza es: Caniche  
Es un: Mamífero



# Métodos

# Definiendo métodos

En realidad cuando usamos `__init__` ya estábamos definiendo un método especial.



A continuación, vamos a ver cómo definir métodos que le den alguna funcionalidad interesante a nuestra clase, siguiendo con el ejemplo de perro. 🐾

# Definiendo métodos



Vamos a codificar dos métodos: **LadRAR** y **caminar**.

El primero no recibirá ningún parámetro y el segundo recibirá el número de pasos que queremos andar. Como hemos indicado anteriormente, **self** hace referencia a la instancia de la clase.

```
class Perro:
    # Atributo de clase
    especie = 'mamífero'

    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza

    def ladra(self):
        print("Guau")

    def camina(self, pasos):
        print(f"Caminando {pasos} pasos")
```

# Definiendo métodos



Se puede definir un método con **def** y el **nombre**, y entre **( )** los **parámetros de entrada que recibe**, donde siempre tendrá que estar **self** el primero.

```
class Perro:
    # Atributo de clase
    especie = 'mamífero'

    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza

    def ladra(self):
        print("Guau")

    def camina(self, pasos):
        print(f"Caminando {pasos} pasos")
```

# Definiendo métodos



Por lo tanto, si creamos un objeto **mi\_perro**, podremos hacer uso de sus métodos llamándolos con **.** y el nombre del método. Como si de una función se tratase, pueden recibir y devolver argumentos.



```
mi_perro = Perro("Toby", "Bulldog")
mi_perro.ladra()
mi_perro.camina(10)

# Creando perro Toby, Bulldog
# Guau
# Caminando 10 pasos
```

Nuestro código quedaría de la siguiente manera



```
1 class Perro:
2
3     #Atributos de clase
4     especie = "Mamífero"
5
6     #Constructor de la clase
7     def __init__(self, nombre, raza):
8
9         #Atributos de la instancia
10        self.nombre = nombre
11        self.raza = raza
12
13    #Métodos
14    def ladrar(self):
15        print("Este perro ha ladrado :( \nEstá enojado")
16
17
18    def caminar(self, pasos):
19        print(f"Este perro ha caminado {pasos} pasos")
20
21
22 perro1 = Perro("Sammy", "Caniche")
23
24 print(f"Su nombre es: {perro1.nombre}")
25 print(f"Su raza es: {perro1.raza}")
26 print(f"Es un: {perro1.especie}")
27
28 perro1.ladrar()
29 perro1.caminar(5)
30
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

```
Su raza es: Caniche
Es un: Mamífero
Este perro ha ladrado :(
Está enojado
Este perro ha caminado 5 pasos
```



# Trabajando con DrawIO

Crear un Diagrama de Clase, utilizando la herramienta de DrawIO

Duración: **10 minutos**



ACTIVIDAD EN CLASE

# Trabajando con DrawIO

## Consigna:

Crear una clase llamada **Alumno**, que posea como atributos de instancia el nombre y la nota del estudiante. Como métodos propios de la clase, se deberán definir correspondientemente el constructor, el método imprimir y resultado.

**Aclaración:** Tanto los atributos como métodos, son de tipo públicos.

**Importante:** En la siguiente actividad en clase, implementaremos nuestro Diagrama de Clase, directamente en Python.





# Break

¡10 minutos y volvemos!

# Métodos especiales


# ¿Qué son?

Los métodos mágicos de las clases Python son aquellos que comienzan y terminan con dos caracteres subrayados. Son de suma importancia ya que muchas de las operaciones que se hacen con clases en Python utilizan estos métodos, como puede ser la creación del objeto (`__init__`) o generar una cadena que los represente (`__str__`).

# Métodos especiales

Todos estos métodos tienen un primer parámetro que hace referencia al propio objeto, que por convenio se suele llamar **self**. Siendo el resto de los parámetros del método variables.

Veamos a continuación algunos de estos métodos y cuál es su finalidad. 🙌



```
1 class Perro:
2
3     #Atributos de clase
4     especie = "Mamífero"
5
6     #Constructor de la clase
7     def __init__(self, nombre, raza):
8
9         #Atributos de la instancia
10        self.nombre = nombre
11        self.raza = raza
12
13    #Métodos
14    def ladrar(self):
15        print("Este perro ha ladrado :( \nEstá enojado")
16
17
18    def caminar(self, pasos):
19        print(f"Este perro ha caminado {pasos} pasos")
20
21
22 perro1 = Perro("Sammy", "Caniche")
23
24 print(f"Su nombre es: {perro1.nombre}")
25 print(f"Su raza es: {perro1.raza}")
26 print(f"Es un: {perro1.especie}")
27
28 perro1.ladrar()
29 perro1.caminar(5)
30
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

```
Su raza es: Caniche
Es un: Mamífero
Este perro ha ladrado :(
Está enojado
Este perro ha caminado 5 pasos
```

# `_init_`



El método mágico más utilizado en las clases de python es el constructor. Empleado para crear cada una de las instancias de las clases. No tiene una cantidad fija de parámetros, ya que estos dependen de las propiedades que tenga el objeto. Por ejemplo, se puede crear un objeto Vector:

```
class Vector():  
    def __init__(self, data):  
        self._data = data  
  
v = Vector([1,2])  
print(v)
```

# **\_str\_**

Posiblemente el segundo método más utilizado, con el que se crea una representación del objeto con significado para las personas.

En el ejemplo anterior, **al imprimir el objeto, se ha visto una cadena que indica el tipo de objeto y una dirección**. Lo que generalmente no tiene sentido para las personas.

# \_\_str\_\_




Un problema que se puede solucionar con el método  
\_\_str\_\_.

```
class Vector():  
    def __init__(self, data):  
        self._data = data  
  
    def __str__(self):  
        return f"The values  
are: {self._data}"  
  
v = Vector([1,2])  
print(v)
```

# `_len_`

Podríamos usar la función `len` para obtener el número de elementos que tenga el objeto.

Pero si probamos con el objeto nos encontraremos con un error. 🤖




```
class Vector():  
  
    def __init__(self, data):  
        self._data = data  
  
    def __str__(self):  
        return f"The values are:  
{self._data}"  
  
    def __len__(self):  
        return len(self._data)  
  
v = Vector([1,2])  
len(v)
```



# `__len__`


Esto es así porque la clase no ha implementado aún el método `__len__`. Un problema se puede solucionar de tomar fácil al incluir este método.



```
class Vector():  
  
    def __init__(self, data):  
        self._data = data  
  
    def __str__(self):  
        return f"The values are:  
{self._data}"  
  
    def __len__(self):  
        return len(self._data)  
  
v = Vector([1,2])  
len(v)
```

# \_\_getitem\_\_

Si se desea que los usuarios puedan leer los elementos mediante el uso de corchetes es necesario implementar el método `__getitem__` en la clase. Este elemento necesita un parámetro adicional, que es la posición del elemento a leer. La función debe retornar el valor leído.



```
class Vector():  
    def __init__(self, data):  
        self._data = data  
  
    def __str__(self):  
        return f"The values  
are: {self._data}"  
  
    def __len__(self):  
        return len(self._data)  
  
    def __getitem__(self,  
pos):  
        return self._data[pos]  
  
v = Vector([1,2])  
v[1]
```



PARA RECORDAR

El método `__getitem__` solo nos sirve para leer, ya que para modificar, se tiene que crear también el método `__setitem__`, el cual veremos a continuación.



# `__setitem__`

Este es el complemento del método anterior. En este caso es necesario pasar dos parámetros adicionales, la posición y el valor a reemplazar.

```
class Vector():  
    def __init__(self, data):  
        self._data = data  
  
    def __str__(self):  
        return f"The values are: {self._data}"  
  
    def __len__(self):  
        return len(self._data)  
  
    def __getitem__(self, pos):  
        return self._data[pos]  
  
    def __setitem__(self, pos, value):  
        self._data[pos] = value  
  
v = Vector([1,2])  
v[1] = 20  
print(v)
```



# `__iter__`

Hace que nuestra clase sea iterable, lo que permite usarla por ejemplo en bucles tipo for, es necesario implementar este método. A continuación, podemos ver una implementación de este método en el que devuelve una cadena con el índice y el valor de cada elemento.

```
class Vector():
    def __init__(self, data):
        self._data = data

    def __str__(self):
        return f"The values are: {self._data}"

    def __len__(self):
        return len(self._data)

    def __getitem__(self, pos):
        return self._data[pos]

    def __setitem__(self, pos, value):
        self._data[pos] = value

    def __iter__(self):
        for pos in range(0, len(self._data)):
            yield f"Value[{pos}]: {self._data[pos]}"

v = Vector([1,2])

for vec in v:
    print(vec)
```



# `_iter_`

Cuando necesitamos que nuestra clase sea iterable, lo que permite usarla por ejemplo, en bucles tipo **for**, es necesario implementar este método. A continuación, podemos ver una implementación en el que devuelve una cadena con el índice y el valor de cada elemento.

```
class Vector():
    def __init__(self, data):
        self._data = data

    def __str__(self):
        return f"The values are: {self._data}"

    def __len__(self):
        return len(self._data)

    def __getitem__(self, pos):
        return self._data[pos]

    def __setitem__(self, pos, value):
        self._data[pos] = value

    def __iter__(self):
        for pos in range(0, len(self._data)):
            yield f"Value[{pos}]: {self._data[pos]}"

v = Vector([1,2])

for vec in v:
    print(vec)
```

# Clases anidadas

# Relación entre clases

Cuando vimos el por qué trabajar por medio de este paradigma, dijimos que era fundamental para **separar a nuestro problema en problemas más pequeños (Clases)**, pero estos “problemas” no van a estar aislados, estarán **relacionados entre sí**. Por ejemplo, los **Perros**, suelen ser la mascota de una **Persona**.



# Relación entre clases

Es muy simple. Pero en la próxima slide veremos otra variante de Clases Anidadas.



```
12
13 def __str__(self):
14     return "Nombre:" +self.nombre +", Raza:" +self.raza +", Especie:" +self.especie
15
16 #Métodos
17 def ladrar(self):
18     print("Este perro ha ladrado :( \nEstá enojado")
19
20
21 def caminar(self, pasos):
22     print(f"Este perro ha caminado {pasos} pasos")
23
24 class Persona:
25
26     def __init__(self, nombre, apellido, perro):
27         self.nombre = nombre
28         self.apellido = apellido
29         self.perro = perro
30
31
32     def __str__(self):
33         return "Mi nombre es: " +self.nombre + " " +self.apellido +"\n" +self.perro.__str__()
34
35 perro1 = Perro("Sammy", "Caniche")
36 persona1 = Persona("Nicolas", "Perez", perro1)
37
38 print(persona1)
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
TypeError: can only concatenate str (not "Perro") to str
PS C:\Users\nico\Desktop\Clase xx> & C:\Users\nico\AppData\Local\Microsoft\WindowsApps\python3.9.exe "c:/Users/nico/Desktop/Clase xx/example.py"
Mi nombre es: Nicolas Perez
Nombre:Sammy, Raza:Caniche, Especie:Mamifero
```

# Relación entre clases



Esta es otra forma de realizar clases internas a otras clases. No es lo más típico y usado, pero puede ser útil para cuando hay una relación muy estricta entre dos clases.

```
41 class Sueldo:
42
43     def __init__(self, sueldo):
44         self.sueldo = sueldo
45
46     def __str__(self):
47         return f"\nSUELDO: {self.sueldo}"
48
49
50 class Empleado:
51
52     def __init__(self, nombre, puesto):
53         self.nombre = nombre
54         self.puesto = puesto
55         self.sueldo = Sueldo(1200)
56
57
58     def __str__(self):
59         return f"NOMBRE: {self.nombre}\nPUESTO: {self.puesto}\n" + self.sueldo.__str__()
60
61 s1 = Sueldo(200)
62 emp1 = Sueldo.Empleado("Nico", "Profe")
63
64 print("RESULTADO 1: " + s1.__str__())
65 print("RESULTADO 2: " + emp1.__str__())
```



## Para pensar

¿Cómo explicarías los últimos print?

¿Qué se representa en el código de la siguiente slide?

Contesta mediante el chat de Zoom

```

41 class Sueldo:
42
43     def __init__(self, sueldo):
44         self.sueldo = sueldo
45
46     def __str__(self):
47         return f"\nSUELDO: {self.sueldo}"
48
49
50 class Empleado:
51
52     def __init__(self, nombre, puesto):
53         self.nombre = nombre
54         self.puesto = puesto
55         self.sueldo = Sueldo(1200)
56
57
58     def __str__(self):
59         return f"NOMBRE: {self.nombre}\nPUESTO: {self.puesto}\n" + self.sueldo.__str__()
60
61 s1 = Sueldo(200)
62 emp1 = Sueldo.Empleado("Nico", "Profe")
63
64 print("RESULTADO 1: " + s1.__str__())
65 print("RESULTADO 2: " + emp1.__str__())

```

```

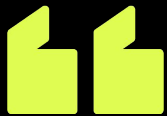
RESULTADO 1:
SUELDO: 200
RESULTADO 2: NOMBRE: Nico
PUESTO: Profe
SUELDO: 1200

```

# Encapsulamiento

# ¿De qué se trata?

Es un concepto relacionado con la programación, orientada a objetos, y 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 😞

[ellibrodepython.com](http://ellibrodepython.com)



PARA RECORDAR

Esto es genial para “aprender”, pero en la práctica no nos gustaría que desde la clase Perro le cambien el nombre a una instancia de la clase Sueldo, por dar un ejemplo. Es por eso que es fundamental “defender” a los datos y tornarlos privados.





# Para pensar

¿Qué creen que ocurre en este caso?

Pista

Atención a los `_` `_`

```
class Persona:

    tipo = "Humano" #Dato al que pueden acceder
    __sueldo = 2000 #No quiero que accedan a mi cuenta

    def __init__(self, nombre, apellido):
        self.nombre = nombre #NO me molesta que sepan mi nombre
        self.__apellido = apellido #No quiero que sepan mi apellido

    def __soy_feliz(self):
        print("No les importa :-)")

    def edad(self):
        return 31 #Soy joven aún no miento con mi edad

persona1 = Persona("Nicolas", "Lopez")

print(f"Resultado1: {persona1.tipo}\n")
print(f"Resultado2: {persona1.__sueldo}\n")
print(f"Resultado3: {persona1.nombre}\n")
print(f"Resultado4: {persona1.__apellido}\n")
print(f"Resultado5: {persona1.__soy_feliz()}\n")
print(f"Resultado6: {persona1.edad()}\n")
```

Contesta mediante el chat de Zoom



# Para pensar

¿Qué creen que ocurre en este caso entonces?

```
85 |
86 persona1 = Persona("Nicolas", "Lopez")
87
88 print(f"Resultado1: {persona1.tipo}\n")
89 #print(f"Resultado2: {persona1.__sueldo}\n")
90 print(f"Resultado3: {persona1.nombre}\n")
91 #print(f"Resultado4: {persona1.__apellido}\n")
92 #print(f"Resultado5: {persona1.__soy_feliz()}\n")
93 print(f"Resultado6: {persona1.edad()}\n")
```

PROBLEMS

OUTPUT

TERMINAL

DEBUG CONSOLE

AttributeError: 'Persona' object has no attribute '\_\_soy\_feliz'

PS C:\Users\nico\Desktop\Clase xx> & C:/Users/nico/AppData/Local/Microsoft/

RESULTADO 1:

SUELDO: 200

RESULTADO 2: NOMBRE: Nico

PUESTO: Profe

SUELDO: 1200

Resultado1: Humano

Resultado3: Nicolas

Resultado6: 31

Contesta mediante el chat de Zoom

# Visibilidad de los datos

Efectivamente, todo lo que esté privado ( **con doble \_\_** ) será imposible de acceder desde el exterior de la clase





# ¿Cómo encapsular en Python?

Hasta el momento sabemos que **la encapsulación es el ocultamiento de datos del estado interno para proteger la integridad del objeto**. En el siguiente ejemplo, CLIENTE es una clase, y hemos encapsulado (al anteponer `__`) la variable `accountNumber`.

```
1 | class Customer:
2 |     def __init__(self):
3 |         self.__accountNumber = 4321
4 |     def getAccountNumber(self):
5 |         return self.__accountNumber
```



# ¿Cómo encapsular en Python?

Del mismo modo, podemos ocultar el acceso a los métodos. Para eso, necesitamos anteponer el nombre del método con `__` (subrayado doble).

```
1 class Customer:
2     def __init__(self):
3         self.__accountNumber = 4321
4
5     def __processAccount(self):
6         print("Processing Account")
7
8     def getAccountNumber(self):
9         return self.__accountNumber
```



# Implementación en Python

Clases desde DrawIO a Python

Duración: **20 minutos**



ACTIVIDAD EN CLASE

# Implementación en Python

**Consigna:** Implementar la Clase de Alumno, creada en la actividad anterior, directamente en Python.

Aclaraciones Generales:

- ✓ El método *imprimir*, deberá mostrar por pantalla el nombre y la nota del estudiante.
- ✓ El método *resultado*, evalúa la nota correspondiente del estudiante. Si el estudiante saca menos de 5 puntos desapueba la materia, más de 5 puntos aprueba. **Tip:** Para la realización de este apartado, es necesario trabajar con estructuras condicionales.
- ✓ Se deberá instanciar, al menos, dos objetos pertenecientes a la clase **Alumno**.

# Repositorio Github

Te dejamos el acceso al Repositorio de Github donde encontrarás todo el material complementario y scripts de la clase.

✓ [Repositorio Python](#)







# #Codertraining

¡No dejes para mañana lo que puedes practicar hoy! Te invitamos a revisar la [Guía de Ejercicios Complementarios](#), donde encontrarás un ejercicio para poner en práctica lo visto en la clase de hoy.



**¿Aún quieres conocer más?**  
**Te recomendamos el**  
**siguiente material**



MATERIAL AMPLIADO

# Recursos multimedia

## Ejemplos de repaso



[EjemplodeRepaso](#)

Disponible en nuestro repositorio.

¿Preguntas?

# Resumen de la clase hoy

- ✓ POO.
- ✓ Clases, atributos y métodos.
- ✓ Métodos especiales.
- ✓ Clases anidadas.
- ✓ Encapsulamiento.

**Opina y valora**  
esta clase

**Muchas gracias.**

**#DemocratizandoLaEducación**