



# **POO**

# **Programación**

# **Orientada**

# **a Objeto**

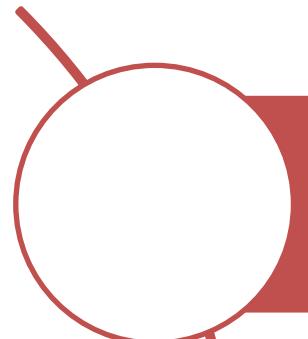
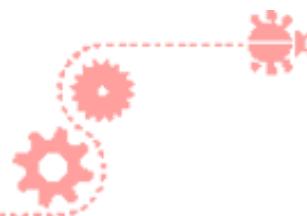
# Breve introducción al tema



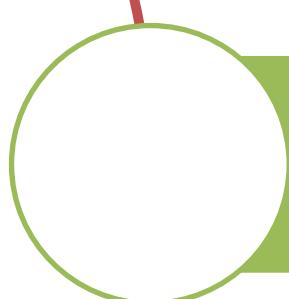
- La Programación Orientada a Objetos (POO) es un paradigma que organiza el código en clases (plantillas o modelos) y objetos (instancias de esas clases), permitiendo que los programas sean más modulares, reutilizables y fáciles de mantener.
- A diferencia de la programación estructurada, que se basa en funciones y procedimientos, la POO divide el problema en “entidades” con atributos (datos) y métodos (acciones) que interactúan entre sí.
- Python admite tanto programación estructurada como POO, por lo que aprender este paradigma facilita trabajar con otros lenguajes modernos



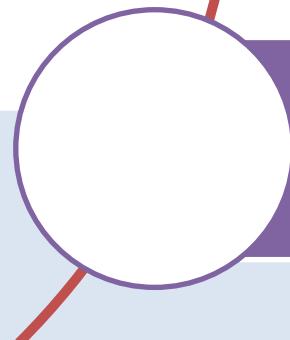
# ¿Qué es un paradigma de programación?



Es una manera o estilo de programación.



Existen varias formas de diseñar



Existen varias formas de trabajar para obtener los resultados necesarios.



# Paradigmas



**Imperativo:** programación estructurada, modular. Sucesión de instrucciones.

**Declarativo:** consiste en decirle a un programa lo que tiene **que hacer** en lugar de decirle cómo debería hacerlo

**POO:** se construyen modelos de objetos

**Reactiva:** los objetos reaccionan a los valores que recibe

# Ejemplos de los paradigmas de programación en la vida real

| Paradigma                        | Descripción  | Ejemplo de la vida real  |
|----------------------------------|--|--|
| <b>Imperativo</b>                | Se indica <b>paso a paso</b> qué hacer.                              | <b>Receta de cocina:</b> seguir instrucciones secuenciales para preparar un plato (cortar, mezclar, cocinar en orden exacto).                              |
| <b>Declarativo</b>               | Se dice <b>qué resultado se espera</b> , sin detallar cómo lograrlo. | <b>Pedir una pizza:</b> simplemente decís “quiero una pizza de muzzarella”, y el restaurante decide cómo prepararla.                                       |
| <b>Reactivo</b>                  | El sistema <b>responde a eventos o cambios</b> .                     | <b>Semáforo inteligente:</b> detecta el tráfico y cambia de color automáticamente cuando hay autos esperando.  |
| <b>POO (Orientado a Objetos)</b> | Se modelan entidades con <b>atributos y comportamientos</b> .        | <b>Auto:</b> tiene atributos (color, modelo, patente) y métodos (acelerar, frenar, girar). Cada auto es un objeto, pero todos se basan en la “clase Auto”. |

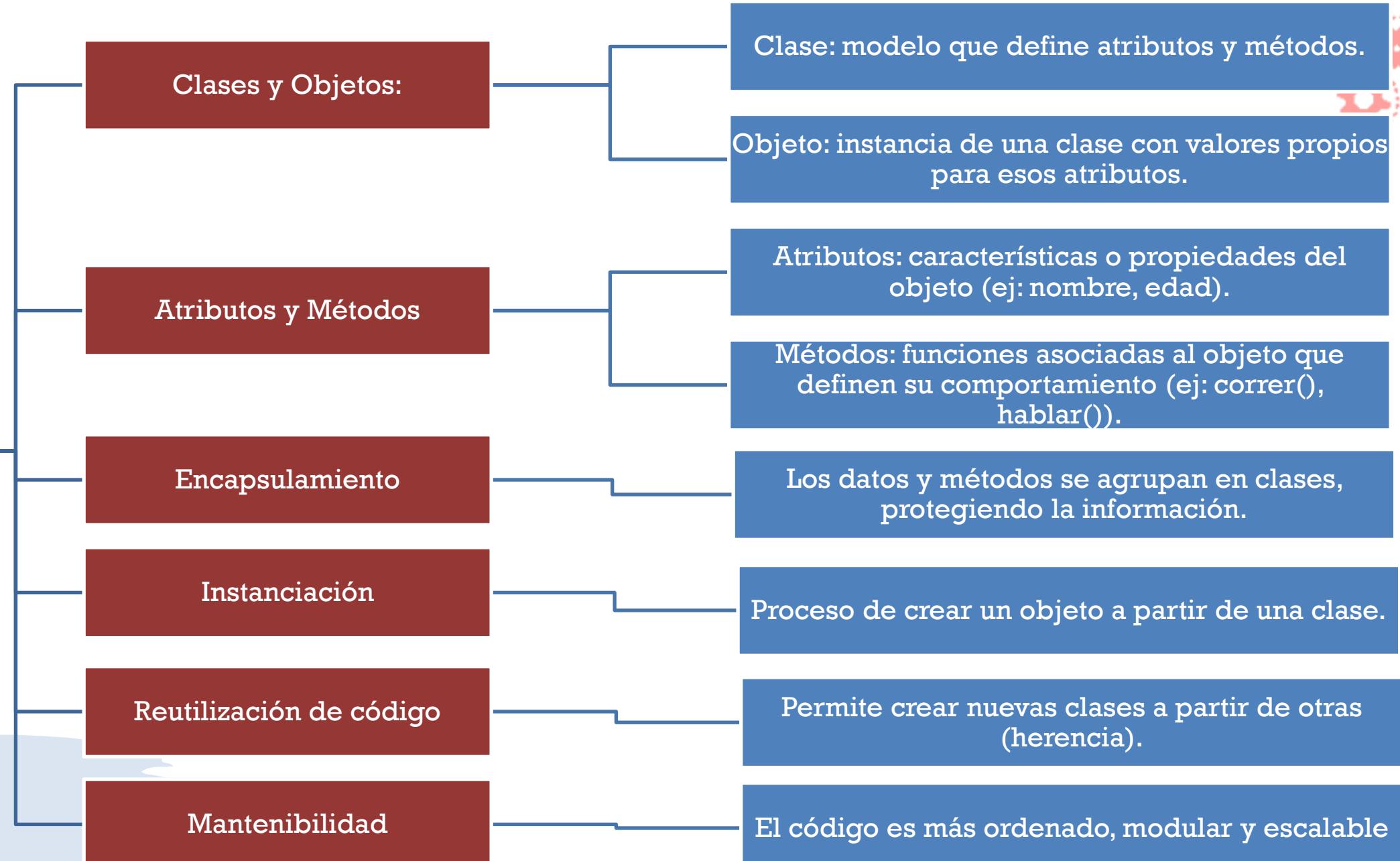
| Paradigma | Descripción  | Ejemplo de la vida real<br>(Completar) |
|-----------|--|--|
|           | Se modelan entidades con atributos y comportamientos.        |  |
|           | Se dice qué resultado se espera, sin detallar cómo lograrlo. |  |
|           | Se indica paso a paso qué hacer.                             |  |
|           | El sistema responde a eventos o cambios.                     |  |

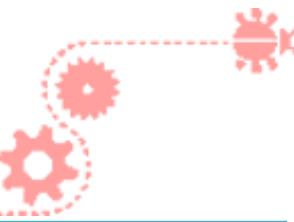
| Situación   | Paradigma | Justificación |
|---|-----------|---------------|
| 1. Seguir las instrucciones de un manual para armar un mueble.  |           |               |
| 2. Decirle a un jardinero: “quiero que el pasto quede corto” (él decide cómo hacerlo).                |           |               |
| 3. El sensor de movimiento enciende la luz automáticamente cuando entras a una habitación.            |           |               |
| 4. Una bicicleta que tiene color, rodado y frena cuando apretás el freno.                             |           |               |
| 5. Enviar una carta al correo y esperar que llegue a destino, sin preocuparte por la ruta que tomará. |           |               |
| 6. Configurar el despertador para que suene a las 7:00 am y reaccione en ese horario.                 |           |               |

La programación Orientada a objetos se define como un **paradigma de la programación**. Una manera de programar específica, donde se **organiza el código en unidades denominadas clases**, de las cuales se crean objetos que se relacionan entre sí para conseguir los objetivos de las aplicaciones.

Podemos entender la programación Orientada a objetos (POO) como una forma especial de programar, más cercana a como expresaríamos las cosas en la vida real que otros tipos de programación, que permite diseñar mejor las aplicaciones, llegando a mayores cotas de complejidad, sin que el código se vuelva inmanejable.

# Aspectos + importantes de la POO



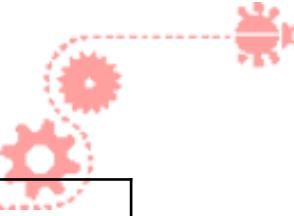


# Vamos con un ejemplo introductorio

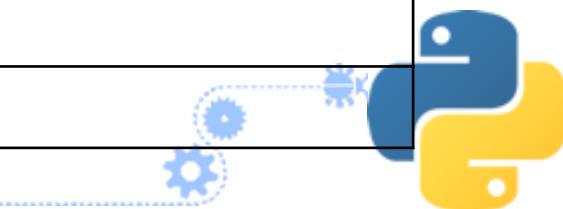
| Aspecto                        | Explicación simple  | Ejemplo  |
|--------------------------------|---|--|
| <b>Clase</b>                   | Es como un <b>molde o plano</b> que define cómo será algo.                      | El <b>plano de una casa</b> : indica cuántas habitaciones, baños y puertas tendrá.   |
| <b>Objeto</b>                  | Es el <b>producto real</b> creado a partir de la clase.                         | Una <b>casa construida</b> a partir de ese plano.  |
| <b>Atributo</b>                | Son las <b>características</b> del objeto.                                      | En una casa: color de las paredes, cantidad de ventanas, tamaño del jardín.  |
| <b>Método</b>                  | Son las <b>acciones</b> que puede hacer el objeto.                              | En una casa: abrir la puerta, encender la luz, cerrar las ventanas.  |
| <b>Encapsulamiento</b>         | Protege los datos para que no cualquiera los cambie sin control.                | La <b>llave de la puerta</b> : solo alguien autorizado puede entrar y modificar lo que hay dentro.                             |
| <b>Instanciación</b>           | Es el acto de <b>crear un objeto</b> a partir de la clase.                      | Construir una casa siguiendo el plano.   |
| <b>Reutilización de código</b> | Usar lo que ya creaste para no empezar de cero.                                 | Usar el <b>mismo plano</b> para construir varias casas en un barrio.   |
| <b>Mantenibilidad</b>          | Que el código (o el sistema) sea <b>fácil de modificar</b> sin romper lo demás. | Si querés cambiar el color de todas las casas, vas al plano y lo cambiás allí: todas las nuevas casas saldrán del nuevo color. |



# Ejercitación de comprensión de contenido



| Situación  | Aspecto Correcto |
|--|------------------|
| 1. Plano del “Departamento”.   |                  |
| 2. Construcción de 5 departamentos siguiendo el plano.               |                  |
| 3. Características propias de un departamento (color, piso, número). |                  |
| 4. Acciones posibles (abrir puerta, encender luces).                 |                  |
| 5. Solo el propietario puede acceder.                                |                  |
| 6. Construcción de más departamentos usando el mismo plano.          |                  |
| 7. Se agrega balcón al plano y afecta a los nuevos departamentos.    |                  |
| 8. Diseño de “Producto” con precio y fecha.                          |                  |
| 9. Fabricación de 100 productos.                                     |                  |
| 10. Cada producto con su número de serie y precio.                   |                  |
| 11. Acción de imprimir etiqueta.                                     |                  |
| 12. Solo el supervisor puede modificar el precio.                    |                  |
| 13. Nuevo lote de productos con el mismo diseño.                     |                  |
| 14. Se agrega código QR a todos los productos nuevos.                |                  |



# ¿Cómo se piensa en objetos?

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real.

Por ejemplo vamos a pensar en un **AUTO** para tratar de modelizarlo en un esquema de POO.

Diríamos que el **AUTO** es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca.

# Cómo se piensa en objetos

Además tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o estacionar (entre otras). Por tanto, pensar en objetos requiere analizar qué elementos vas a manejar en tus programas, tratando de identificar sus características y funcionalidades.

# Manejando el concepto de CLASE

En un esquema de programación orientada a objetos "EL AUTO" sería lo que se conoce como "Clase".

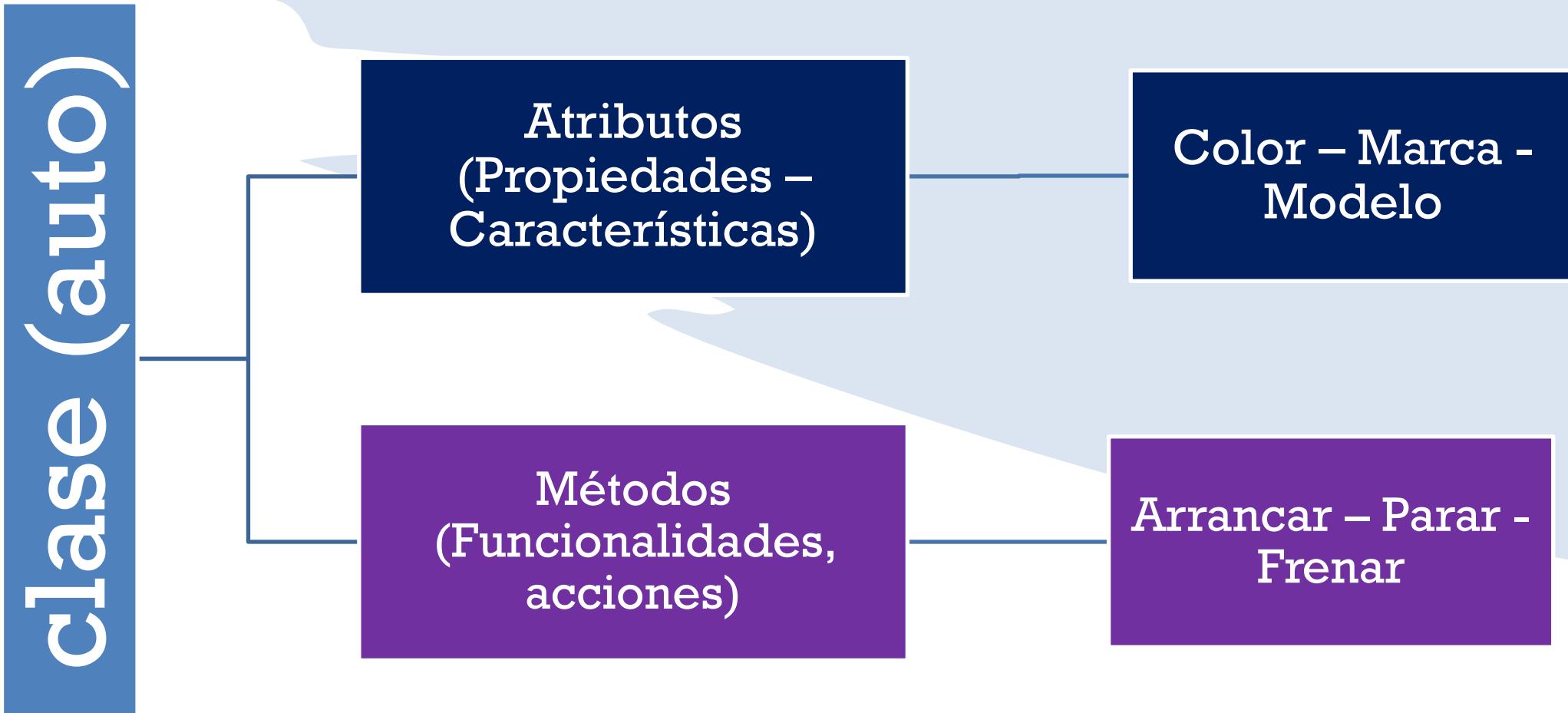
La **clase** **contiene** la **definición** de las **características** de un **modelo** (EL AUTO), como el color o la marca, junto con la **implantación** de sus **funcionalidades**, como arrancar o parar.

# Manejando el concepto de CLASE

Las características definidas en la clase las llamamos **Atributos** y las funcionalidades asociadas las llamamos **métodos**.

La clase AUTO describe como son todos los autos del mundo, es decir, qué propiedades tienen y que funcionalidades deben poder realizar y cómo se realizan.

La clase AUTO describe como son todos los autos del mundo, es decir, qué propiedades tienen y que funcionalidades deben poder realizar y cómo se realizan



# Manejando el concepto de OBJETOS

A partir de una clase podemos crear cualquier número de objetos de esa clase.

A partir de la clase “EL AUTO ” podemos crear un auto rojo que es de la marca Ford y modelo Fiesta, otro verde que es de la marca Seat y modelo Ibiza.

Por tanto, los objetos son ejemplares de una clase, o elementos concretos creados a partir de una clase.

La clase es como el molde y a los objetos como concreciones creadas a partir del molde de clase.

# Analicemos

# ejemplos

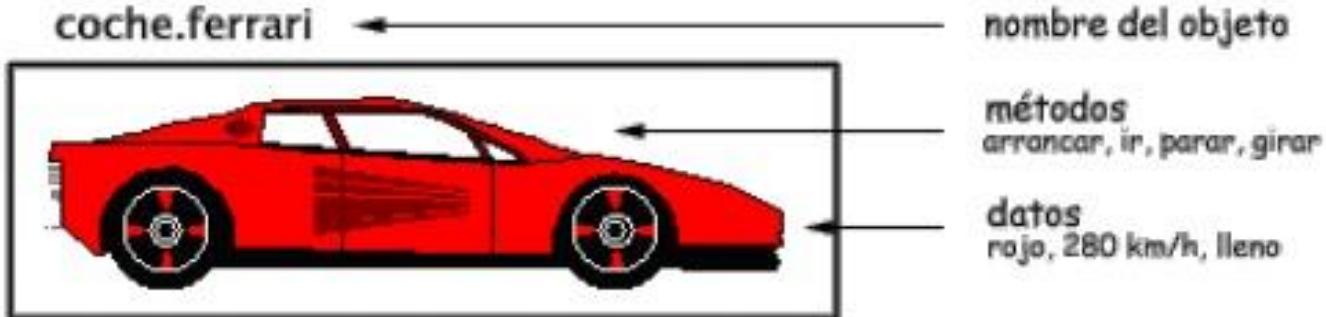


# EJEMPLO DE CLASES Y OBJETOS

**Clase:**  
*Coche*



♦ **Objeto:** *Ferrari*



# Computadora

numeroDeSerie

marca

procesador

sistemaOperativo

memoria

aula

encender()

ejecutarPrograma()

almacenarArchivo()

cerrarPrograma()

apagar()

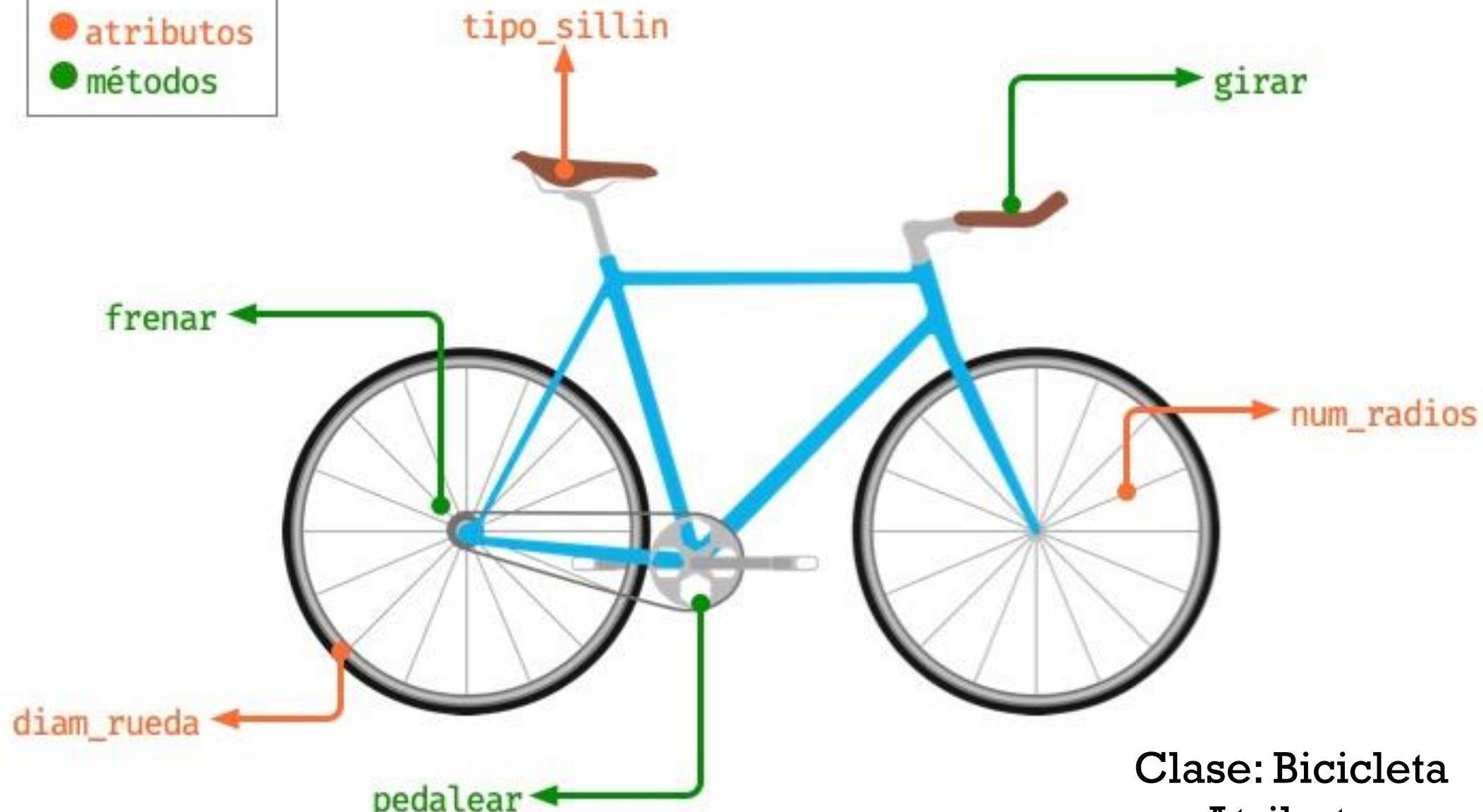


Identificador de la clase

Atributos

Métodos

● atributos  
● métodos



Clase: Bicicleta  
Atributos  
Métodos

# PERRO

Clase

Raza  
tamaño  
Años  
Color

Miembros  
de datos

atributos

Comer()  
Dormir()  
Sentar()  
Correr()

método

## Persona

nombre  
apellido materno  
apellido paterno  
sexo  
edad

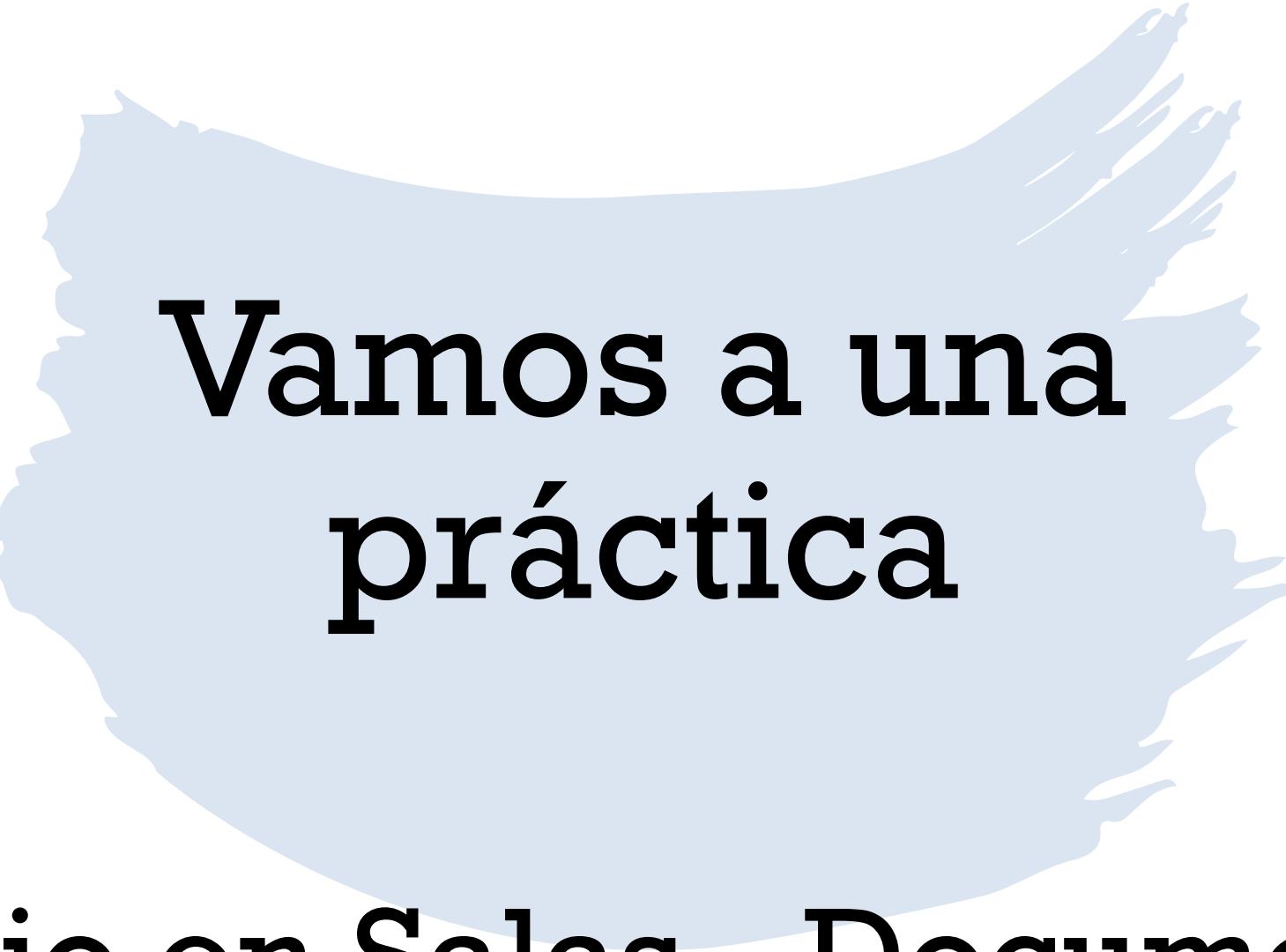
comer()  
beber()  
dormir()

Nombre

Atributos

Operaciones





**Vamos a una  
práctica**

**Trabajo en Salas - Documentar**

# Práctica N° 1



Clase: Monopatín

Atributos

Métodos



Clase: Gato

Atributos

Métodos



Clase Estudiante

Atributos

Métodos



## Situación Base

Una veterinaria quiere organizar la información de las mascotas que atiende.

Cada mascota debe registrar: nombre, especie, edad y si tiene vacunas al día.

Además, debe poder realizar estas acciones: presentarse (mostrar su información) y actualizar el estado de sus vacunas.

Tarea:

- Identifica la clase en esta situación.
- Enumera los atributos que tendría cada objeto.
- Escribe los métodos que podría tener esta clase.
- Indica qué sería la instancia en este contexto.



### Clase

- Mascota (es el modelo general para cualquier animal que atienda la veterinaria).

### Atributos:

- Nombre – Especie – Edad - Vacunas al día (True/False)

### Métodos:

- Presentarse (mostrar en pantalla nombre, especie y edad).
- Actualizar estado de vacunas (cambiar de “no vacunado” a “vacunado”).

### Instanciación:

Cuando se registra a una mascota en el sistema, por ejemplo:

- `Mascota1(nombre="Luna", especie="Perro", edad=3, vacunas=True)`
- `Mascota2(nombre="Michi", especie="Gato", edad=2, vacunas=False)`



# Situación 1 – Biblioteca

## Contexto:

Una biblioteca quiere registrar la información de los **libros**. Cada libro debe tener: título, autor, año de publicación y estado (disponible o prestado). Además, debe poder realizar estas acciones: mostrar su información y cambiar su estado a “prestado” o “disponible” según corresponda.

## Tarea:

1. Identifica **la clase**.
2. Enumera **los atributos**.
3. Escribe **los métodos** que tendría esta clase.
4. Explica **qué sería la instancia**.



## Situación 2 – Escuela

### Contexto:

Una escuela necesita guardar información de sus **alumnos**.

Cada alumno debe registrar: nombre, apellido, DNI y promedio general.

Debe poder realizar estas acciones: mostrar su ficha de alumno y actualizar su promedio cuando carga nuevas notas.

### Tarea:

1. Identifica **la clase**.
2. Enumera **los atributos**.
3. Escribe **los métodos** que tendría esta clase.
4. Explica **qué sería la instancia**.



# Situación 3 – Tienda Online

## Contexto:

Una tienda online gestiona los **productos** que vende.

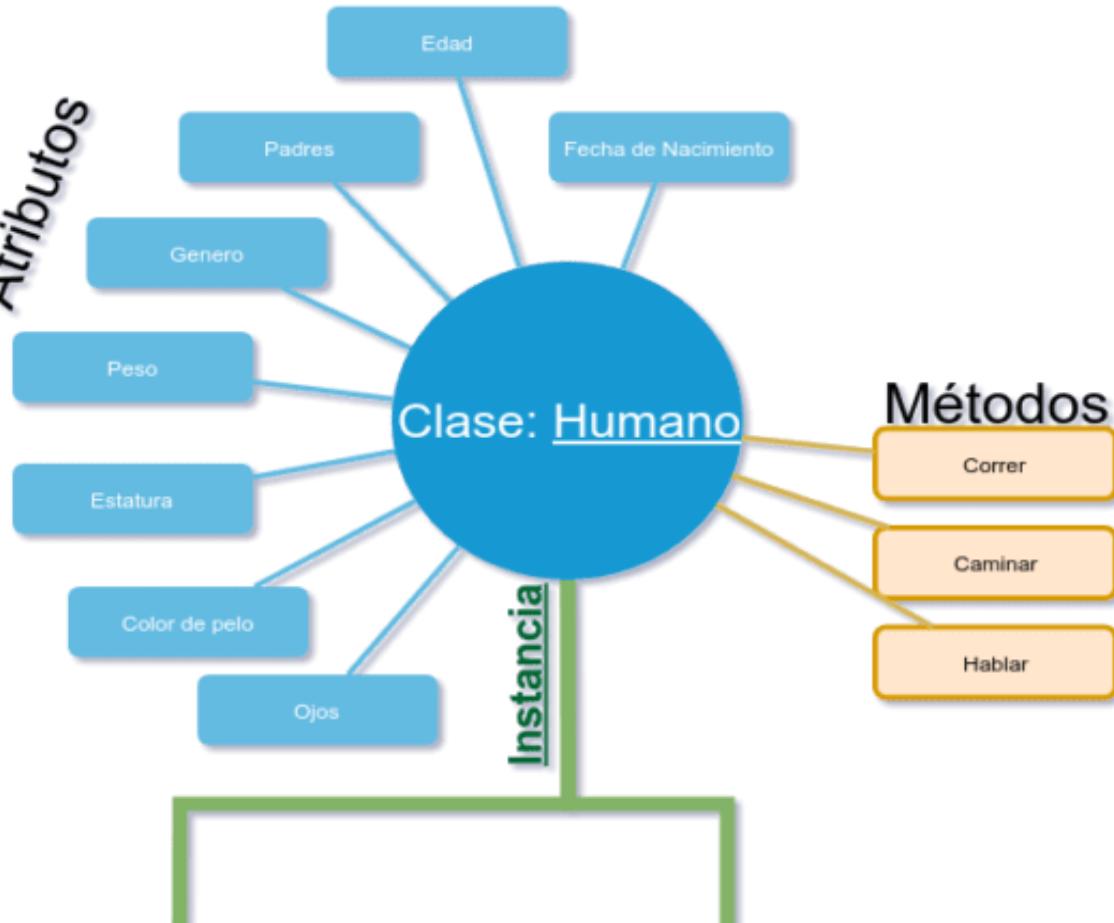
Cada producto tiene: nombre, precio, stock y categoría.

Debe poder realizar estas acciones: mostrar la información del producto, aplicar descuento al precio y actualizar el stock cuando se vende.

## Tarea:

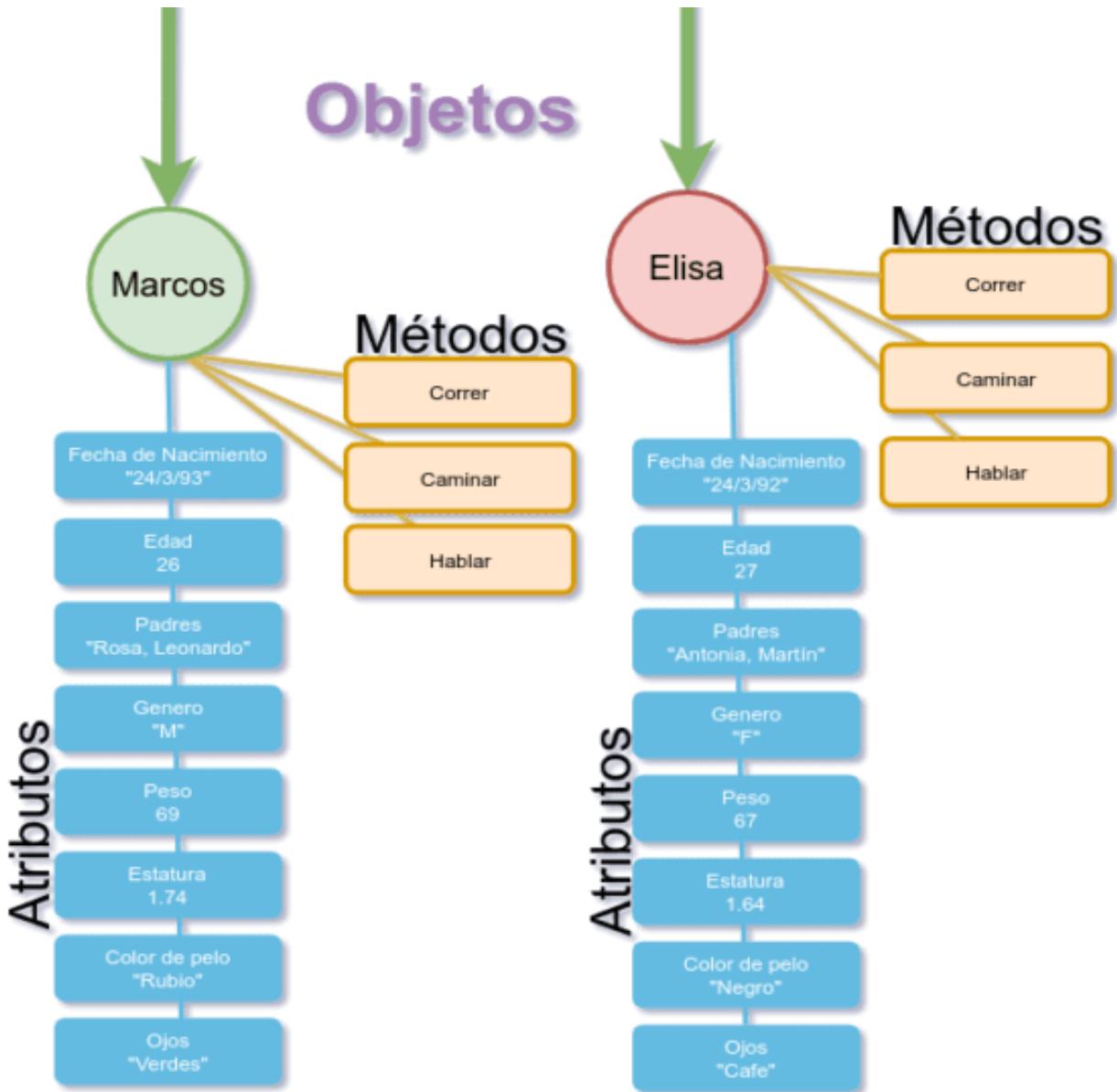
1. Identifica la **clase**.
2. Enumera los **atributos**.
3. Escribe los **métodos** que tendría esta clase.
4. Explica qué sería la **instanciación**.

## Atributos



## Métodos

## Objetos



## Atributos

## Métodos

## Atributos

## Métodos

# Clase Taza

- Atributos
  - Color
  - Forma
  - Tamaño



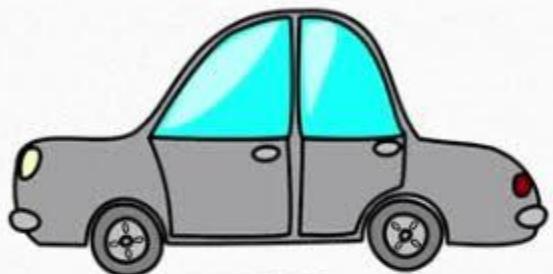
Objeto TazaMusical

- Métodos
  - Sostenerse
  - Almacenar

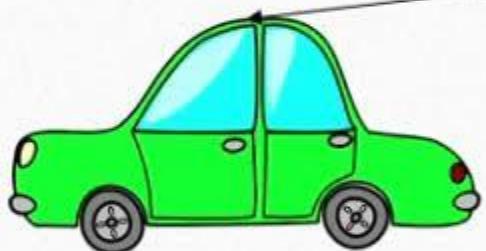


Objeto TazaMounstroComeGalletas

## ClaseCarro



ObjetoCarroVerde



Métodos

- Encender
- Acelerar
- Frenar

Atributos

- Color = Verde
- Puertas = 4
- MaxVelocidad = 70

ObjetoCarroAzul



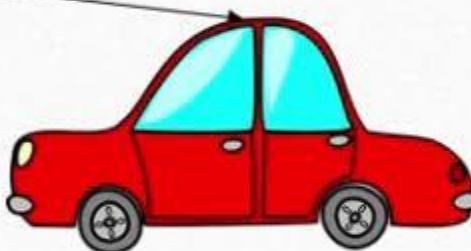
Métodos

- Encender
- Acelerar
- Frenar

Atributos

- Color = Azul
- Puertas = 2
- MaxVelocidad = 80

ObjetoCarroRojo



Métodos

- Encender
- Acelerar
- Frenar

Atributos

- Color = Rojo
- Puertas = 4
- MaxVelocidad = 100

# Objetos en el mundo real



Lavadora



Perro



Televisión



Persona



Factura

**¿Cómo hemos  
venido  
programando  
hasta ahora ???**



Archivo Editar Selección Ver Ir Ejecutar ... ⟲ ⟳ 🔎 PYTHON 🔍

EXPLORADOR ...

PYTHON > Carpeta1 > CFEI\_INTERMEDIO > Ciclos > Bucle2.py > ...

```
1 import os
2 os.system("cls")
3
4 email=input("Ingrese direccion de email: ")
5
6 conta=0
7 #recorre cada uno de los caracteres de una variable tipo texto
8 #el bucle se repite tantas veces como caracteres tenga
9 #la variable email.
10 for i in email:
11     if i=="@":
12         conta+=1
13
14 if conta==0:
15     print ('La direccion no tiene arroba')
16 elif conta==1:
17     print ('La direccion tiene formato correcto')
18 else:
19     print ('La direccion tiene mas de un arroba')
20
```

Se ejecuta en la terminal



```
op=0
while op!=7:
    os.system("cls")
    print ('-'*10), ' AGENDA DE CONTACTOS ', ('-'*10)
    print ('-'*46)
    print ('1 - AGREGAR NUEVO CONTACTO'
          '\n2 - LISTAR AGENDA'
          '\n3 - MODIFICAR CONTACTO'
          '\n4 - ELIMINAR CONTACTO'
          '\n5 - BUSQUEDA FINA'
          '\n6 - EXPORTAR DATOS'
          '\n7 - SALIR')
    op=int(input('Elija opcion: '))
    if op==1:
        agregarDatos()
    elif op==2:
        mostrarDatos()
    elif op==3:
        modificarDatos()
    elif op==4:
        eliminarDatos()
    elif op==5:
        buscarDatos()
    elif op==6:
```

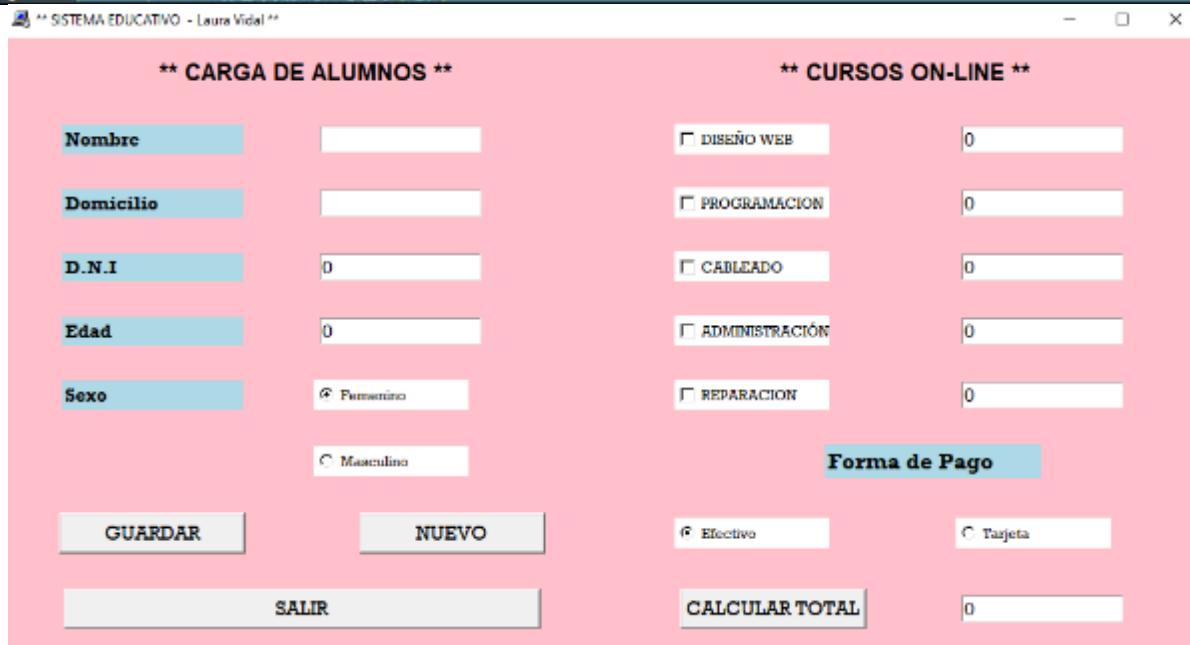
Luego comenzamos a aplicar funciones  
Funcionaban a través de un menú.  
Esto hacía que pareciera que no se ejecutara de arriba hacia abajo..  
Sino a través de una opción

Se ejecuta en la terminal

Practica\_general.py X

Practicas\_asistidas > Practica\_general.py > ...

```
151  
152     tarj=Radiobutton(marco,text="Tarjeta",variable=eot,value=2)  
153     tarj.grid(row=7,column=4,sticky="w",padx=10,pady=10)  
154     tarj.config(fg="black",bg="white",width=15,font=("Rockwell",10),anchor="w")  
155  
156 #FUNCIONALIDAD A LOS BOTONES  
157 def Total():  
158     if curso1.get()==1:  
159         valor1.set(1500)  
160     else:  
161         valor1.set(0)  
162     if curso2.get()==1:  
163         valor2.set(2000)  
164     else:  
165         valor2.set(0)
```



Luego pasamos  
a una interfaz  
gráfica.  
La lógica lo  
maneja el  
usuario a través  
de los objetos.

Se ejecuta en la  
ventana

**Pero en  
realidad....cuál es el  
estilo de  
programación que  
actualmente  
se utiliza ???**



# Programación Orientada a Objetos

**Es un estilo que se centra en, reconocer todos los elementos necesarios para el programa que queremos desarrollar.**

**A cada uno de ellos los rotula, los carátula como objetos.**



# Programación Orientada a Objetos



TENEMOS UNA  
ORDEN Y  
EJECUTAMOS  
ACCIONES



# Programación Orientada a Objetos

En esa realidad nuestra, parecida a un programa informático, existen objetos con los cuales interactuamos



Todas las acciones que realizamos en la vida diaria están vinculadas a un objeto.

Con ese objeto interactuamos y depende de lo que quedamos hacer con ese objeto



# Programación Orientada a Objetos

En los últimos programas hemos manipulado elementos que son objetos

Calculo de compras

Ingresar 1er valor: 0

Ingresar 2do valor: 0

Ingresar 3er valor: 0

Confirmar

TOTAL: 0

Nueva Suma

Servicios especiales:

Flete

Garantia extendida

Categoría:

Cliente

No cliente

Salir

¿Cuáles son esos objetos? 

# Programación Orientada a Objetos

Y que cosas hemos podido observar en esos objetos?

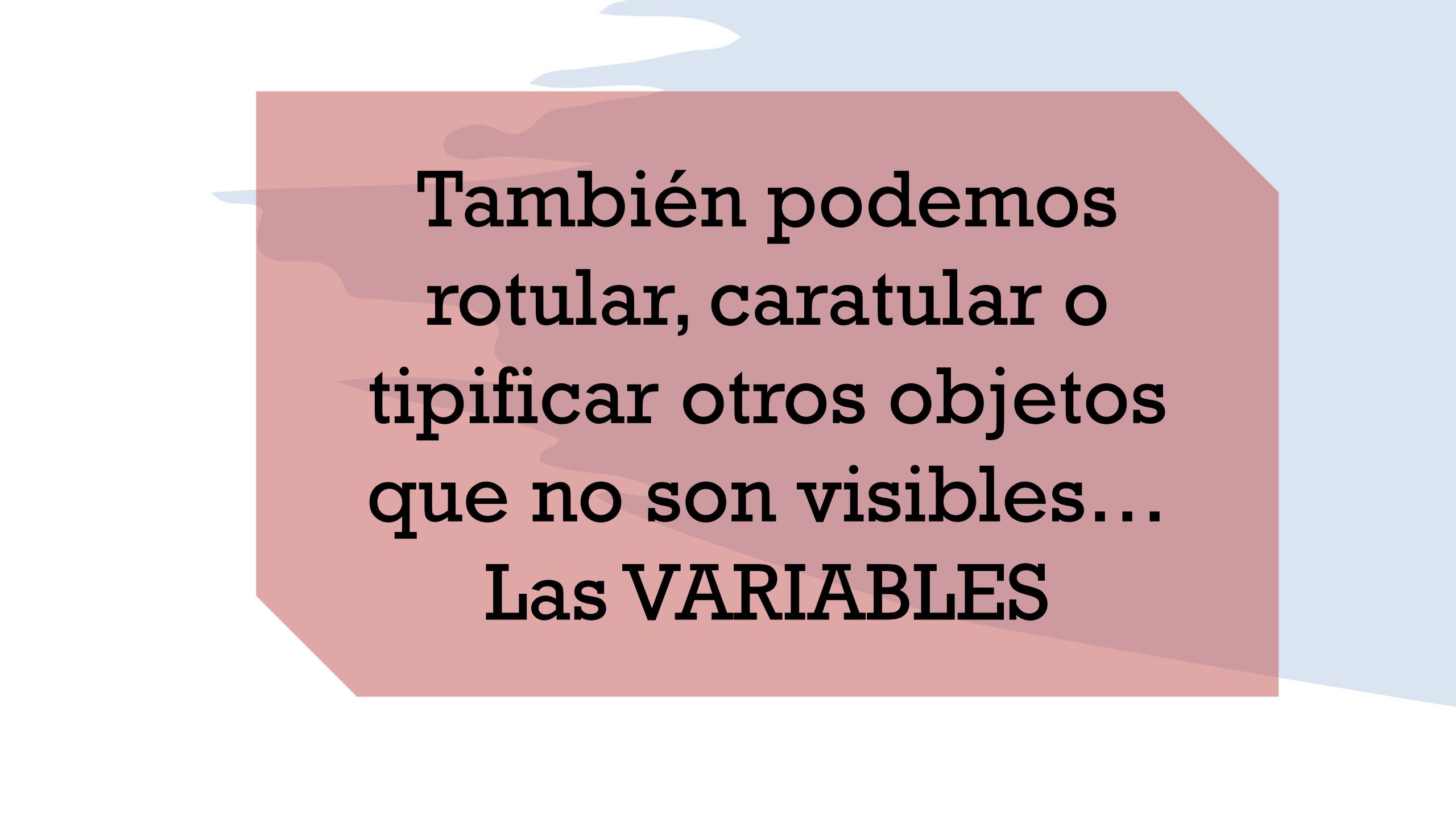
## Características

- Ubicación – Color – Tamaño – Contenido- Orientación

## Comportamientos

- Acciones que se desencadenan cuando hacemos **click** en un botón o marcamos una casilla. A cada acción hay un comportamiento





También podemos rotular, caratular o tipificar otros objetos que no son visibles....

Las VARIABLES

# Programación Orientada a Objetos

En un sentido mas amplio es también correcto decir que las variables como Valor1, Valor2, Valor3; también son objetos y tienen comportamientos como **GET** (con el que se toma su contenido) o **SET** (con el que se le da contenido).

```
### FUNCIONES PARA BUTTONS
def Calcular():
    suma=Valor1.get()+Valor2.get()+Valor3.get()
    if Cli.get()==1:
        suma=suma*0.9
    if Flete.get()==1:
        suma=suma*1.1
    if Garantia.get()==1:
        suma=suma*1.1
    Total.set(suma)

def Nuevo():
    Valor1.set(0)
    Valor2.set(0)
    Valor3.set(0)
    Total.set(0)
    #resetear los checks y radio buttons
    Cli.set(1)
    Flete.set(0)
    Garantia.set(0)
```

**GET – SET** son acciones, es decir comportamientos



# Programación Orientada a Objetos

Objetos definidos por Python

Objetos que se relacionan con la finalidad del programa y que nosotros definimos.

Variables, Etiquetas, Cuadros de texto, botones,etc.

El valor de cada compra, el comprador y su característica, etc.



# Programación Orientada a Objetos

## Diferencia entre objetos de Python y los definidos por el Programador

Objetos definidos por  
Python (botón - entry)

Ya **tienen** una forma o modelo  
**definido** por Python (**características**  
y **comportamientos**)

Objetos definidos por el  
Programador (cliente)

Deben ser modelados por el  
programador en base a la  
información que tenemos de ellos



# Programación Orientada a Objetos

**Tomemos como ejemplo un Sistema Educativo.....**

**¿Qué  
objetos  
podemos  
reconocer  
allí??**

Reconocer objetos y como es esto de modelarlos estableciendo sus características y sus comportamientos que creemos nosotros pueden llegar a tener. Esto nos permite entender el concepto de objetos, comportamientos y características

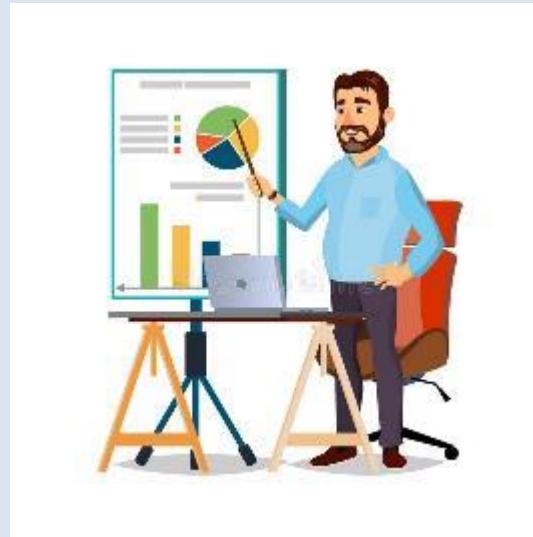


# Programación Orientada a Objetos

## Objetos de un Sistema Educativo



Alumnos

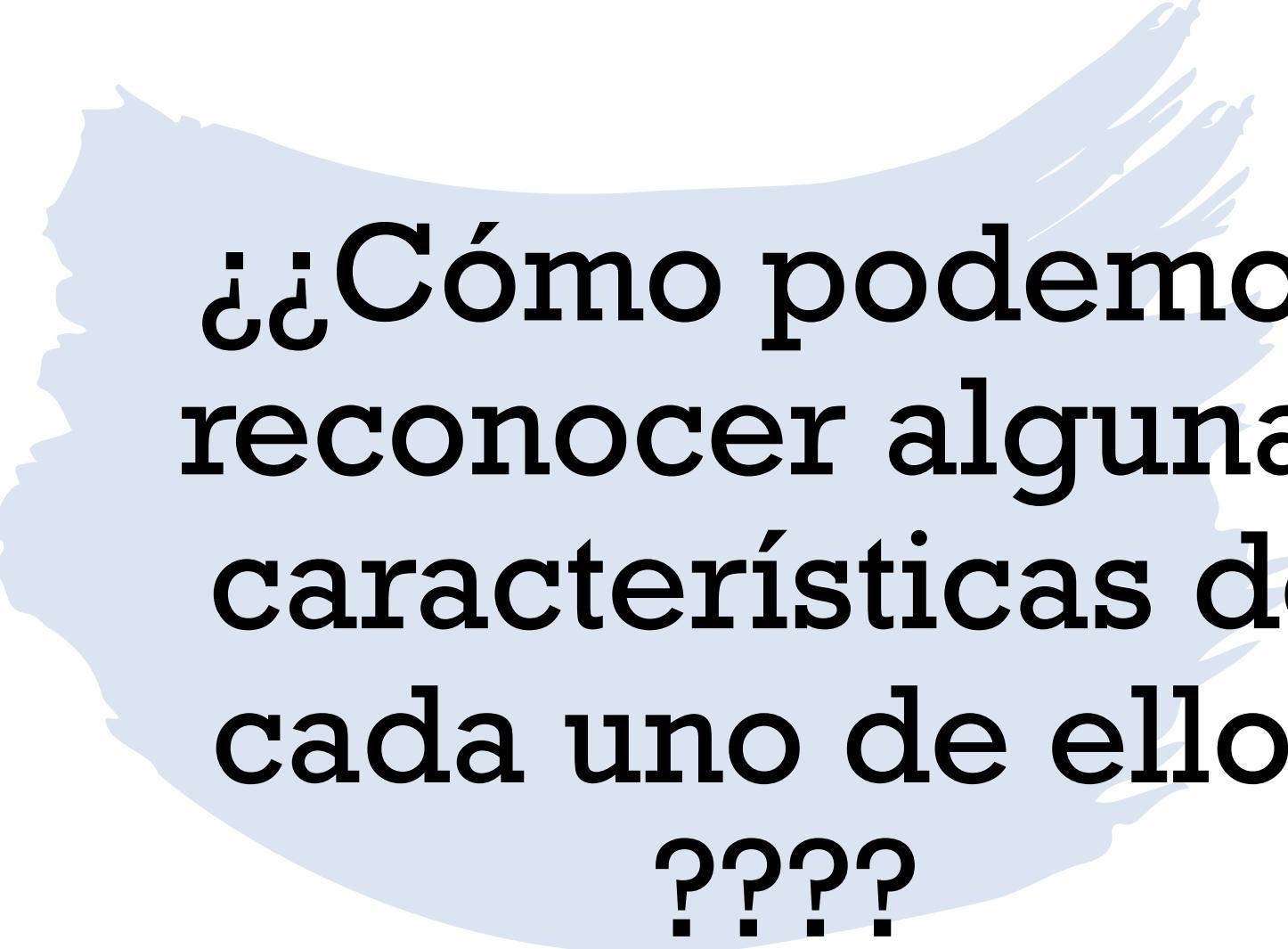


Docentes



Cursos





**¿¿Cómo podemos  
reconocer algunas  
características de  
cada uno de ellos**

????

# Programación Orientada a Objetos

Alumnos

**NOMBRE**  
**DOMICILIO**  
**DNI**  
**EDAD**

Docentes

**NOMBRE**  
**DOMICILIO**  
**DNI**  
**CATEGORIA**  
**ANTIGÜEDAD**  
**SUELDO**

Cursos

**NOMBRE**  
**DURACION**  
**MODALIDAD**  
**FECHA INICIO**



**Una vez analizado el tema de  
los objetos..**

**¿Podemos reconocer algunos  
comportamientos que cada  
uno de estos objetos pueden  
realizar?**

# Programación Orientada a Objetos

Que comportamientos podemos ver en ellos?

Alumnos

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR

Docentes

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR

Cursos

AGREGAR  
MODIFICAR  
ELIMINAR  
INSCRIBIR  
INICIAR



# Conclusión....

Todas las  
palabras son  
Verbo - Acciones  
**Comportamiento**

Característica  
describe un  
aspecto del  
objeto

El comportamiento es una acción  
que el objeto desarrolla

# Programación Orientada a Objetos

Mis Modelos

Alumnos

Docentes

Cursos

Características

NOMBRE  
DOMICILIO  
DNI  
EDAD

NOMBRE  
DOMICILIO  
DNI  
CATEGORIA  
ANTIGÜEDAD  
SUELDO

NOMBRE  
DURACION  
MODALIDAD  
FECHA INICIO

Comportamientos

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR

AGREGAR  
MODIFICAR  
ELIMINAR  
INSCRIBIR  
INICIAR



# Programación Orientada a Objetos

Modelos de  
Python

## Entry

texto1

texto2

texto3

| CARACTERISTICAS |
|-----------------|
| WIDTH           |
| FG              |
| BG              |
| TEXTVARIABLE    |

| COMPORTAMIENTOS |
|-----------------|
| ENTRY           |
| GRID            |
| CONFIG          |

## Acciones

```
#ENTRYS
texto1=Entry(marco,textvariable=Valor1)
texto1.grid(row=0,column=1,sticky="w",pady=10, padx=10)
texto1.config(fg="red", bg="white", width=15, font=("Arial",12))
texto2=Entry(marco,textvariable=Valor2)
texto2.grid(row=1,column=1,sticky="w",pady=10, padx=10)
texto2.config(fg="red", bg="white", width=15, font=("Arial",12))
texto3=Entry(marco,textvariable=Valor3)
texto3.grid(row=2,column=1,sticky="w",pady=10, padx=10)
texto3.config(fg="red", bg="white", width=15, font=("Arial",12))
```



En este ejemplo vimos que hay un modelo Entry que me dice como lo tengo que hacer, no hay otra manera

### Variable – Textvariable

Un modelo es una referencia que yo quiero seguir para que las cosas se hagan de la forma que yo quiero, crear un Entry con dichas características y comportamientos

```
#ENTRYS  
texto1=Entry(marco, textvariable=Valor1)  
texto1.grid(row=0, column=1, sticky="w", pady=10, padx=10)
```

# Ejercicio

Tomemos un objeto DEFINIDO en Python y analicemos sus CARACTERÍSTICAS Y COMPORTAMIENTOS

Pensemos una situación problemática a resolver (vida diaria – similar al Sist. Educativo)  
Detectemos los objetos que necesitaríamos  
De esos objetos definamos CARACTERÍSTICAS  
De esos objetos definamos COMPORTAMIENTOS

# Programación Orientada a Objetos

Alumno 1



NOMBRE  
DOMICILIO  
DNI  
EDAD

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR

Alumno 2



NOMBRE  
DOMICILIO  
DNI  
EDAD

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR

Alumno N



NOMBRE  
DOMICILIO  
DNI  
EDAD

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR



En el punto anterior podemos ver que vamos a tener varios alumnos que tienen las mismas características y comportamientos (y los docentes también).

**NO SE PUEDE** estar redefiniendo las características de cada alumno y de todos los objetos...**POR SEPARADO.**

**SOLUCIÓN:** trabajar con un sistema de modelo

Modelo: tenía características y comportamientos

**Modelo Alumno**

**Python CLASE**



**Python  
ATRIBUTO**

|                  |
|------------------|
| <b>NOMBRE</b>    |
| <b>DOMICILIO</b> |
| <b>DNI</b>       |
| <b>EDAD</b>      |

**Python  
METODO**

|                  |
|------------------|
| <b>AGREGAR</b>   |
| <b>MODIFICAR</b> |
| <b>ELIMINAR</b>  |
| <b>LISTAR</b>    |

**Alumno2 instancia  
de Clase ALUMNO**



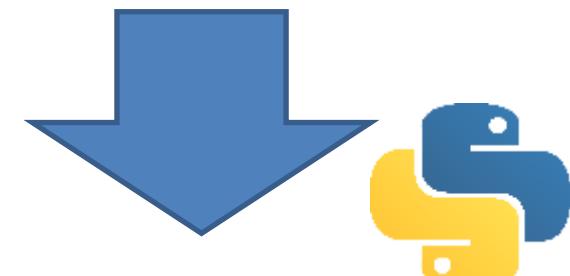
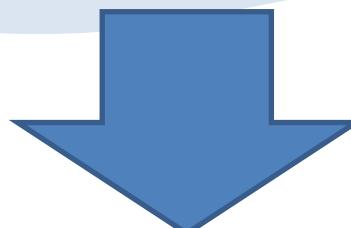
**VA A  
TENER**

**VA A  
TENER**

**AlumnoN instancia  
de Clase ALUMNO**



Voy a crear una instancia (UNA COPIA) de la clase ALUMNO que tomará los atributos y los comportamientos de la clase ALUMNO





# Python

Programación Orientada a Objetos

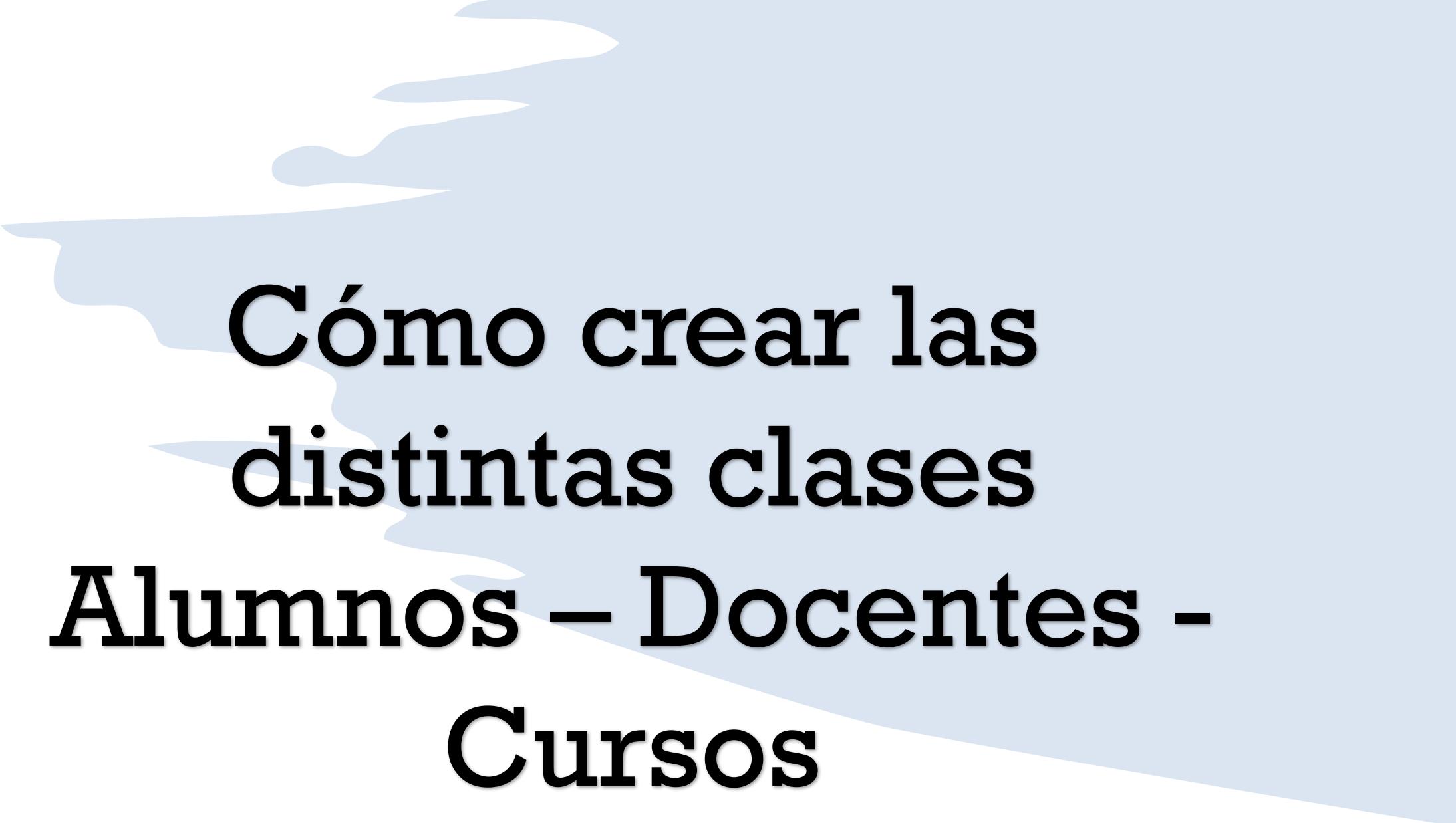
Clase N°  
10

# Programación Orientada a Objetos

## ¿Cómo se crea una CLASE??

```
class Alumnos():  
  
    def <ATRIBUTOS>  
  
    def <METODOS>
```





**Cómo crear las  
distintas clases**

**Alumnos – Docentes –  
Cursos**

# Programación Orientada a Objetos

Mis Modelos

Alumnos

Docentes

Cursos

Características

NOMBRE  
DOMICILIO  
DNI  
EDAD

NOMBRE  
DOMICILIO  
DNI  
CATEGORIA  
ANTIGÜEDAD  
SUELDO

NOMBRE  
DURACION  
MODALIDAD  
FECHA INICIO

Comportamientos

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR

AGREGAR  
MODIFICAR  
ELIMINAR  
INSCRIBIR  
INICIAR



```
class Alumnos():
    def __init__(self,id=0,nombre="",domicilio="",dni=0,edad=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni
        self.edad=edad

    def Agregar(self):
        print("Se agrego el alumno ",self.nombre)

    def Modificar(self):
        print("Se modiflico el alumno ",self.nombre)

    def Eliminar(self):
        print("Se elimino el alumno ",self.nombre)
```

Alumno1=Alumnos(0,"Raul Lopez")

Alumno1.Agregar()

Alumno1=Alumnos(0,"Juan Perez")

## Este es el original

Que tiene métodos

Agregar

Modificar

Eliminar

Y tiene atributos

Nombre, Domicilio, dni, edad

```
class Alumnos():
    def __init__(self,id=0,nombre="",domicilio="",dni=0,edad=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni
        self.edad=edad

    def Agregar(self):
        print("Se agrego el alumno ",self.nombre)

    def Modificar(self):
        print("Se modiflico el alumno ",self.nombre)

    def Eliminar(self):
        print("Se elimino el alumno ",self.nombre)
```

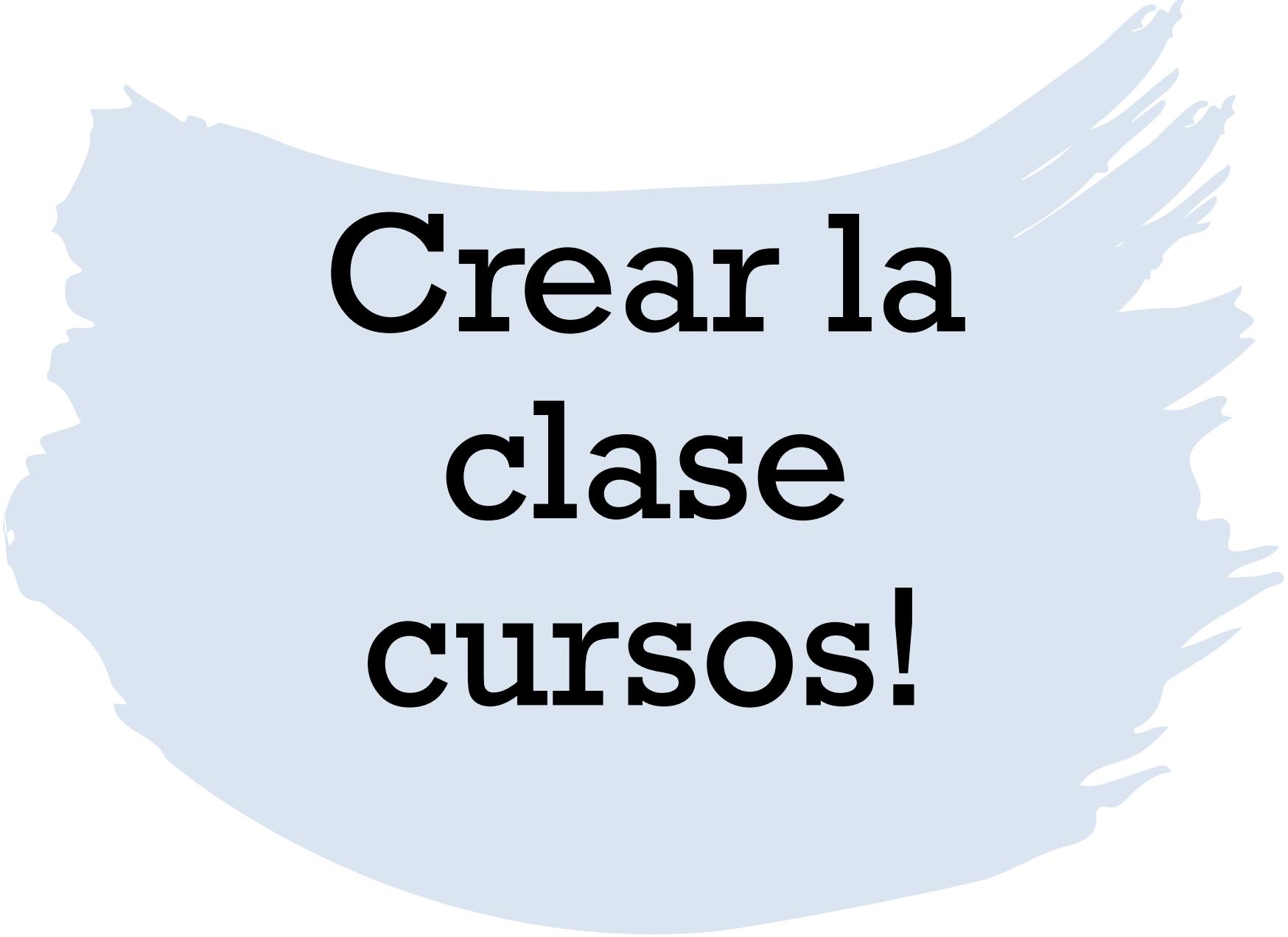
Ese original se puede copiar tantas veces como se quiera

Crean tantos objetos alumnos como se desee y éstos tomarán los métodos y atributos como si fuesen propios (INSTANCIA)

# A ejecutar este código



**Agregar tres  
objetos que son  
copias del mismo  
modelo  
Ejecutar!!!!**



**Crear la  
clase  
cursos!**



# Python

Programación Orientada a Objetos

Clase N°  
11

# Módulos en Python

```
class Alumnos():
    def __init__(self,id=0,nombre="",domicilio="",dni=0,edad=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni
        self.edad=edad

    def Agregar(self):
        print("Se agrego el alumno ",self.nombre)

    def Modificar(self):
        print("Se modiflico el alumno ",self.nombre)

    def Eliminar(self):
        print("Se elimino el alumno ",self.nombre)
```

```
Alumno1=Alumnos(0,"Raul Lopez")
Alumno1.Agregar()
```

# instancia



# Módulos en Python – Repasemos...

Creamos una clase

```
class <<Nombre>>():  
    def <ATRIBUTOS>  
    def <METODOS>
```

Copia de la clase  
(instancia)

```
Alumno1=Alumnos(0,"Raul  
Lopez")  
Alumno1.Agregar()
```

```

import os
os.system("cls")

#Definición de la clase Alumnos
#Con sus atributos y métodos
class Alumnos():
    def __init__(self, id=0,nombre="",domicilio="",dni=0,edad=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni
        self.edad=edad
    #def __init__ definimos los atributos de la clase
    def Agregar(self):
        print("Se agrego el alumno ",self.nombre,self.domicilio)
    def Modificar(self):
        print("Se modificó el alumno ",self.nombre)
    def Eliminar(self):
        print("Se eliminó el alumno ",self.nombre)
    #def Agregar(self) definimos los métodos
Alumno1=Alumnos(0,"Juan Perez","Cipolletti")
Alumno1.Agregar()
Alumno1.Modificar()
Alumno1.Eliminar()

```

#ejecuto el método Agregar  
 #Alumno1 está vinculado a todos los comportamientos  
 #por qué? porque creamos una instancia en Alumno1  
 #que es una copia de la clase Alumnos

```

#definimos los atributos (características)
#Id:estructura de base de datos. Clave principal de BDD
#Este método se escribe “__init__”
# y para recordarlo lo puede asociar con “inicializar”.
#El self hace referencia al nombre del objeto en el que se encuentra escrito
#SELF representa al objeto que se crea desde el modelo,
desde la clase

```

# Clase Alumnos

#La forma de usar la clase sería:  
 #Vamos a crear un objeto alumno  
 #Aca creamos un objeto Alumno1 pero se copie de la clase Alumno  
 #Vemos que la clase alumno tiene argumentos id, nombre..  
 #Le pasamos los argumentos (cómo parámetros) que queremos  
 #Los parámetros que no pasamos toman el valor inicial

```

import os
os.system("cls")
class Docentes():
    def __init__(self,
id=0,nombre="",domicilio="",dni=0,categoria=0,antiguedad=0,sueldo=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni
        self.categoría=categoría
        self.antiguedad=antiguedad
        self.sueldo=sueldo
    #def__init__ definimos los atributos de la clase
    def Agregar(self):
        print("Se agrego el docente ",self.nombre, "con ",self.antiguedad,
años de experiencia)
    def Modificar(self):
        print("Se modificó el docente ",self.nombre, "con ",self.antiguedad,
años de experiencia)
    def Eliminar(self):
        print("Se eliminó el docente ",self.nombre, "con ",self.antiguedad,
años de experiencia)

Docente1=Docentes(0,"Ing Perez",antiguedad=10)
Docente2=Docentes(1,"Ing Garcia",antiguedad=5)
Docente3=Docentes(2,"Ing Fernandez",antiguedad=7)

Docente1.Modificar()
Docente2.Agregar()
Docente3.Eliminar()

```

#definimos los atributos (características)  
#Id:estructura de base de datos. Clave principal de BDD  
#Este método se escribe “\_\_init\_\_”  
# y para recordarlo lo puede asociar con “inicializar”.  
#El self hace referencia al nombre del objeto en el que se encuentra escrito  
#SELF representa al objeto que se crea desde el modelo,  
desde la clase

# Clase Docente

#La forma de usar la clase sería:  
#Vamos a crear un objeto docente  
#Aca creamos un objeto Docente1 pero se copie de la clase Docente  
#Vemos que la clase docente tiene argumentos id, nombre...  
#Le pasamos los argumentos (cómo parámetros) que queremos  
#Los parametros que no pasamos toman el valor inicial  
#Si no se sigue el numero de orden de los parámetros  
#hay que referenciar el atributo

#ejecuto el método Agregar  
#Docente está vinculado a todos los comportamientos  
#por qué? porque creamos una instancia en Docente1  
#que es una copia de la clase Docente

```

import os
os.system("cls")

#Definición de la clase Alumnos
#Con sus atributos y métodos
class Alumnos():
    def __init__(self, id=0,nombre="",domicilio="",dni=0,edad=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni
        self.edad=edad
    #def __init__ definimos los atributos de la clase
    def Agregar(self):
        print("Se agrego el alumno ",self.nombre,self.domicilio)
    def Modificar(self):
        print("Se modificó el alumno ",self.nombre)
    def Eliminar(self):
        print("Se eliminó el alumno ",self.nombre)
    #def Agregar(self) definimos los métodos
Alumno1=Alumnos(0,"Juan Perez","Cipolletti")
Alumno1.Agregar()
Alumno1.Modificar()
Alumno1.Eliminar()

```

#ejecuto el método Eliminar  
 #Micurso está vinculado a todos los comportamientos  
 #por qué? porque creamos una instancia en Curso1  
 #que es una copia de la clase curso

#definimos los atributos (características)  
 #Id:estructura de base de datos. Clave principal de BDD  
 #Este método se escribe “\_\_init\_\_”  
 # y para recordarlo lo puede asociar con “inicializar”.  
 #El self hace referencia al nombre del objeto en el que se encuentra escrito  
 #SELF representa al objeto que se crea desde el modelo,  
 desde la clase

# Clase Curso

#La forma de usar la clase sería:  
 #Vamos a crear un objeto curso  
 #Aca creamos un objeto Curso1 pero se copie de la clase curso  
 #Vemos que la clase curso tiene argumentos id, nombre..  
 #Le pasamos los argumentos (cómo parámetros) que queremos

Diferencia bien marcada (cosas diferentes)

Clase

Instancia

Por un lado: el código que me permite crear la clase

Y el uso específicamente de esa clase

Ubicarse en archivo diferentes

La clase debería ir en un archivo .py  
(claseAlumnos.py)

```
Alumno1=Alumnos(0,"Raul Lopez")  
Alumno1.Agregar()
```

En otro archivo...

# Módulos en Python

## Archivo claseAlumnos.py

```
class Alumnos():
    def __init__(self,id=0,nombre="",domicilio="",dni=0,edad=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni
        self.edad=edad

    def Agregar(self):
        print("Se agrego el alumno ",self.nombre)

    def Modificar(self):
        print("Se modiflico el alumno ",self.nombre)

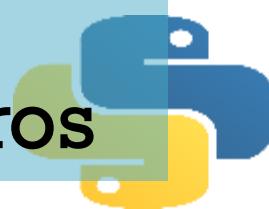
    def Eliminar(self):
        print("Se elimino el alumno ",self.nombre)
```

## Archivo UsarClases.py

```
from claseAlumnos import Alumnos
import os

os.system("cls")
Alumno1=Alumnos(0,"Raul Lopez")
Alumno1.Agregar()
```

Import  
Un código que  
desarrollamos nosotros



# Módulos en Python

Creamos la Carpeta4  
Que contendrá los  
archivos  
`claseAlumnos.py`  
`claseCurso.py`  
`claseDocente.py`  
`UsarClase.py`

## ✓ Carpeta4

- > `_pycache_`
- ✚ `claseAlumnos.py`
- ✚ `claseCursos.py`
- ✚ `claseDocentes.py`
- ✚ `UsarClases.py`



# Módulos en Python

Módulo  
claseAlumnos

Módulo  
claseDocentes

Módulo  
claseCursos

```
from claseAlumnos import Alumnos  
from claseDocentes import Docentes  
from claseCursos import Cursos  
import os
```

```
os.system("cls")  
Alumno1=Alumnos(0,"Raul Lopez")  
Alumno1.Agregar()
```

```
Docente1=Docentes(0,"Ing. Perez",antiguedad=10)  
Docente1.Modificar()
```

```
Curso1=Cursos(0,"Fortran","2 meses","Presencial")  
Curso1.Eliminar()
```



claseAlumnos.py Carpeta3 X

claseAlumnos.py Carpeta4

...

Carpeta3 > claseAlumnos.py > ...

```
1
2 class Alumnos():
3     def __init__(self,id=0,nombre="",domicilio="",dni=0,edad=0):
4         self.id=id
5         self.nombre=nombre
6         self.domicilio=domicilio
7         self.dni=dni
8         self.edad=edad
9
10    def Agregar(self):
11        print("Se agrego el alumno ",self.nombre)
12
13    def Modificar(self):
14        print("Se modiflico el alumno ",self.nombre)
15
16    def Eliminar(self):
17        print("Se elimino el alumno ",self.nombre)
18
19 Alumno1=Alumnos("Raui Garcia")
20 Alumno1.Agregar()
21
22
```

carpeta3

claseAlumnos.py X

Carpeta4 > claseAlumnos.py > Alumnos > \_\_init\_\_

```
1
2 class Alumnos():
3     def __init__(self,id=0,nombre="",domicilio="",dni=0,edad=0):
4         self.id=id
5         self.nombre=nombre
6         self.domicilio=domicilio
7         self.dni=dni
8         self.edad=edad
9
10    def Agregar(self):
11        print("Se agrego el alumno ",self.nombre)
12
13    def Modificar(self):
14        print("Se modiflico el alumno ",self.nombre)
15
16    def Eliminar(self):
17        print("Se elimino el alumno ",self.nombre)
18
19
20
21
```

carpeta4

```
import os
os.system("cls")

#Definición de la clase Alumnos
#Con sus atributos y métodos
class Alumnos():

    def __init__(self, id=0,nombre="",domicilio="",dni=0,edad=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni
        self.edad=edad
    def Agregar(self):
        print("Se agrego el alumno ",self.nombre,self.domicilio)
    def Modificar(self):
        print("Se modificó el alumno ",self.nombre)
    def Eliminar(self):
        print("Se eliminó el alumno ",self.nombre)
#def Agregar(self) definimos los métodos
```

# Clase Alumno

# Clase Docente

```
import os
os.system("cls")
def __init__(self,
id=0,nombre="",domicilio="",dni=0,categoria=0,antiguedad=0,sueldo=0):
    self.id=id
    self.nombre=nombre
    self.domicilio=domicilio
    self.dni=dni
    self.categoría=categoría
    self.antiguedad=antiguedad
    self.sueldo=sueldo
#def __init__ definimos los atributos de la clase
def Agregar(self):
    print("Se agrego el docente ",self.nombre, "con ",self.antiguedad," años de experiencia")
def Modificar(self):
    print("Se modificó el docente ",self.nombre, "con ",self.antiguedad," años de experiencia")
def Eliminar(self):
    print("Se eliminó el docente ",self.nombre, "con ",self.antiguedad," años de experiencia")
```

# Clase Curso

```
import os
os.system("cls")
class Cursos():
    def __init__(self, id=0,nombre="",duracion="",modalidad=""):
        self.id=id
        self.nombre=nombre
        self.duracion=duracion
        self.modalidad=modalidad
    def Agregar(self):
        print("Se agrego el curso ",self.nombre, "de tipo
",self.modalidad)
    def Modificar(self):
        print("Se modificó el curso ",self.nombre, "de tipo
",self.modalidad)
    def Eliminar(self):
        print("Se elimino el curso ",self.nombre, "de tipo
",self.modalidad)
#def Agregar(self) definimos los métodos
```

# UsarClase

```
from claseAlumnos import Alumnos
from claseDocente import Docentes
from claseCurso import Cursos
import os
os.system("cls")
Alumno1=Alumnos(0,"Juan Perez","Cipolletti")
Alumno1.Agregar()
```

```
Docente1=Docentes(0,"Ing Perez",antiguedad=10)
Docente2=Docentes(1,"Ing Garcia",antiguedad=5)
Docente3=Docentes(2,"Ing Fernandez",antiguedad=7)
Docente1.Modificar()
Docente2.Agregar()
Docente3.Eliminar()
```

```
Micurso1=Cursos(0,"Fortran","2 meses","presencial")
Micurso1.Eliminar()
```

Carpeta4 > claseDocentes.py > Docentes > \_\_init\_\_

```
1
2 class Docentes():
3     def __init__(self,id=0,nombre="",domicilio="",dni=0,cat
4                     antiguedad=0,sueldo=0):
5         self.id=id
6         self.nombre=nombre
7         self.domicilio=domicilio
8         self.dni=dni
9         self.categoría=categoría
10        self.antiguedad=antiguedad
11        self.sueldo=sueldo
12
13
14    def Agregar(self):
15        print("Se agrego el docente ",self.nombre," con ",
16              "años de experiencia")
17
18    def Modificar(self):
19        print("Se modiflico el docente ",self.nombre," con "
20              "años de experiencia")
21
22    def Eliminar(self):
23        print("Se elimino el docente ",self.nombre," con ",
24              "años de experiencia")
25
26
```

Carpeta4 > UsarClases.py > ...

```
1
2 from claseAlumnos import Alumnos
3 from claseDocentes import Docentes
4 from claseCursos import Cursos
5 import os
6
7 os.system("cls")
8 Alumno1=Alumnos(0,"Raul Lopez")
9 Alumno1.Agregar()
10
11 Docente1=Docentes(0,"Ing. Perez",antiguedad=10)
12 Docente1.Modificar()
13
14 Curso1=Cursos(0,"Fortran","2 meses","Presencial")
15 Curso1.Eliminar()
16
17
```

# Crear la carpeta4

Crear el archivo UsarClase.py

Modificar el archivo ClaseAlumnos.py

Modificar el archivo ClaseDocente.py

Modificar el archivo ClaseCurso.py

Ejecutar el nuevo código

En una carpeta de ejercicios  
 adicionales  
 Practicar!

Pensemos una situación problemática a resolver  
(vida diaria – similar al Sist. Educativo)  
Detectemos los objetos que necesitaríamos  
De esos objetos definamos CARACTERÍSTICAS  
De esos objetos definamos COMPORTAMIENTOS

# En una carpeta de ejercicios adicionales Practicar!

```
graph TD; A(( )) --> B(( )); B --> C(( ));
```

Crear los archivos de las clases con sus atributos y métodos

Crear el archivo UsarClase.py

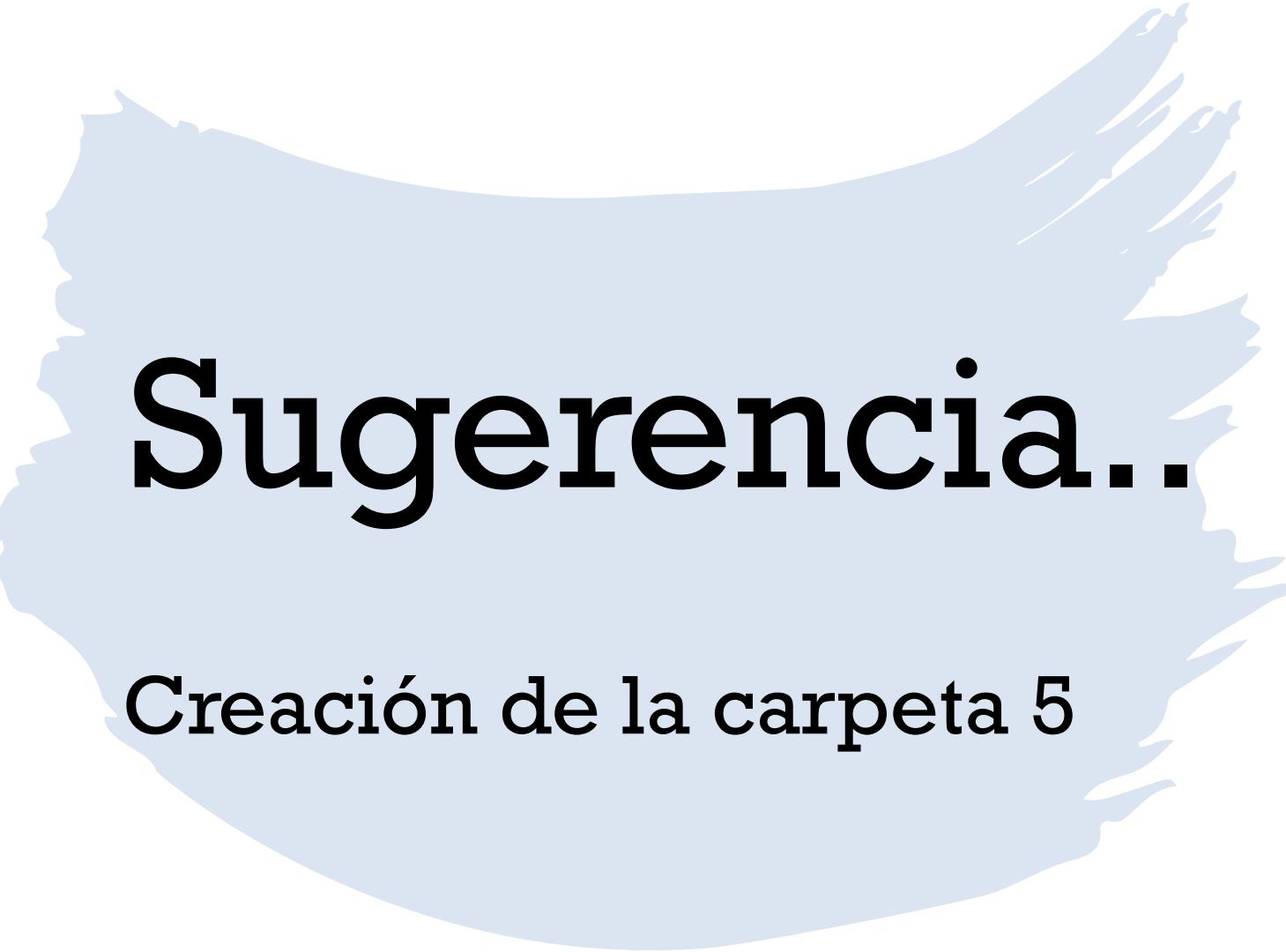
Ejecutar el código



# Python

Programación Orientada a Objetos

Clase N°  
12



# **Sugerencia..**

**Creación de la carpeta 5**

# ¿Qué es la herencia en POO?

Se trata de uno de los pilares fundamentales de la programación orientada a objetos

Es el mecanismo por el cual una clase permite heredar las características (atributos y métodos) de otra clase. La **herencia** permite **que** se puedan definir nuevas clases basadas de unas ya existentes a fin de reutilizar el código, generando así una jerarquía de clases dentro de una aplicación

# ¿Qué es la herencia en POO? Ejemplos

Es la relación entre una clase general y otra clase más específica.

Por ejemplo: Si declaramos una clase párrafo derivada de una clase texto, todos los métodos y variables asociadas con la clase texto, son automáticamente heredados por la subclase párrafo.

# Sigamos con el ejemplo del sistema educativo

Mis Modelos

Alumnos

Docentes

Cursos

Características

NOMBRE  
DOMICILIO  
DNI  
EDAD

NOMBRE  
DOMICILIO  
DNI  
CATEGORIA  
ANTIGÜEDAD  
SUELDO

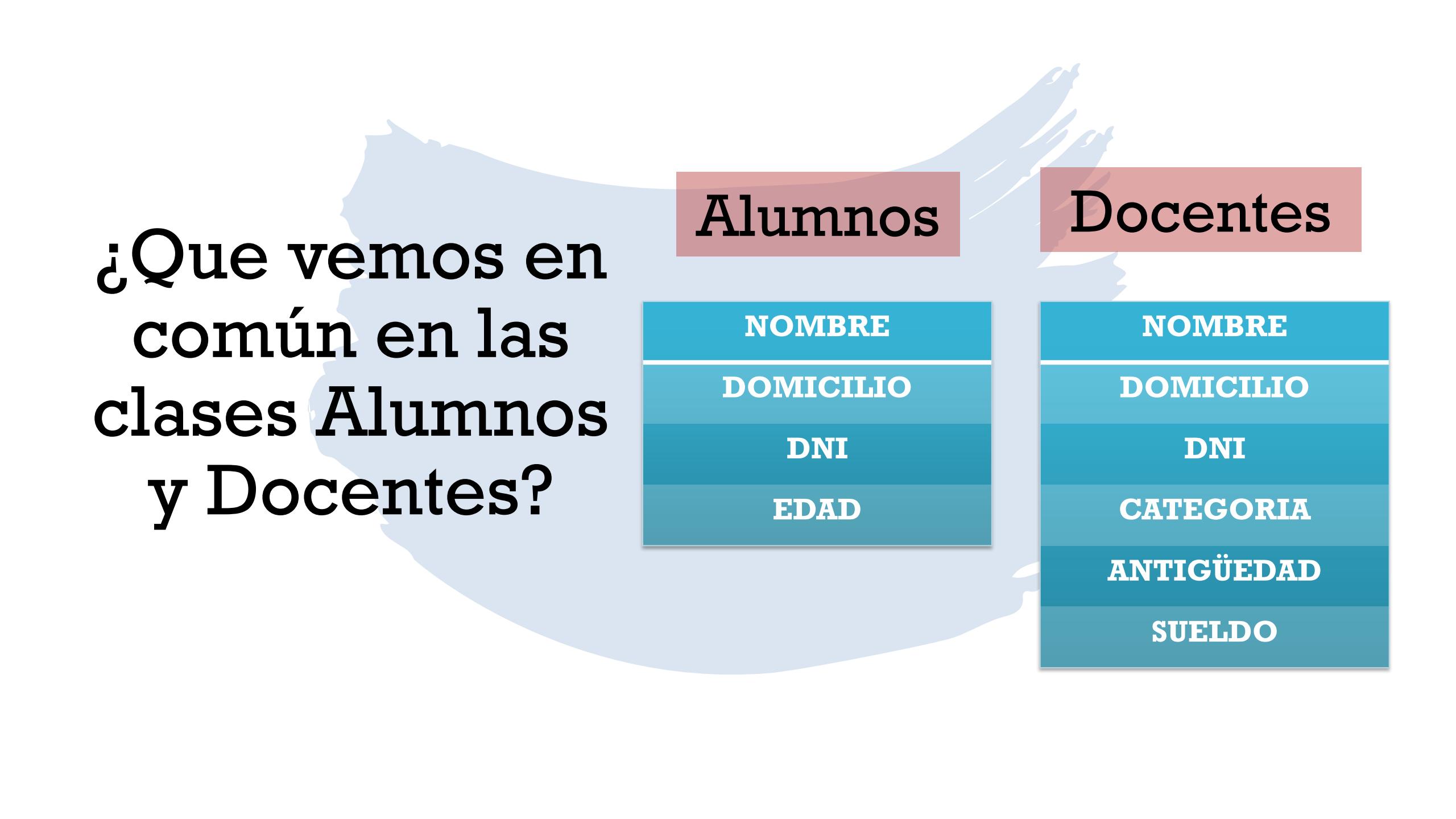
NOMBRE  
DURACION  
MODALIDAD  
FECHA INICIO

Comportamientos

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR

AGREGAR  
MODIFICAR  
ELIMINAR  
LISTAR

AGREGAR  
MODIFICAR  
ELIMINAR  
INSCRIBIR  
INICIAR



¿Que vemos en común en las clases Alumnos y Docentes?

**Alumnos**

|                  |
|------------------|
| <b>NOMBRE</b>    |
| <b>DOMICILIO</b> |
| <b>DNI</b>       |
| <b>EDAD</b>      |

**Docentes**

|                   |
|-------------------|
| <b>NOMBRE</b>     |
| <b>DOMICILIO</b>  |
| <b>DNI</b>        |
| <b>CATEGORIA</b>  |
| <b>ANTIGÜEDAD</b> |
| <b>SUELDO</b>     |

# Herencia en Python

Atributos en  
común

## Alumnos

|           |
|-----------|
| NOMBRE    |
| DOMICILIO |
| DNI       |
| EDAD      |

## Docentes

|            |
|------------|
| NOMBRE     |
| DOMICILIO  |
| DNI        |
| CATEGORIA  |
| ANTIGÜEDAD |
| SUELDO     |

Estos atributos nombre, domicilio y DNI que se comparten.. Me permite “Pensar” en una entidad superior en donde Alumno y Docente son una misma cosa.. Es decir PERSONAS! De esta manera responden a un patrón que está por encima de la configuración de las clases



Esta relación planteada nos permite reconfigurar la forma de nuestros modelos pensando en un elemento superador.

Pensemos....

Estos atributos (en común) no están bien ubicados ni en docentes ni en alumnos..  
Sino que deberían ubicarse en una nueva clase...

# Herencia en Python

Clase con  
atributos en  
común

Persona

NOMBRE  
DOMICILIO  
DNI

NUEVA  
CLASE

Alumnos

NOMBRE  
DOMICILIO  
DNI

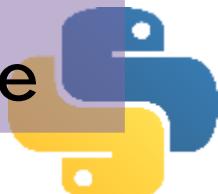
EDAD

Docentes

NOMBRE  
DOMICILIO  
DNI

CATEGORIA  
ANTIGÜEDAD  
SUELDO

Esta clase  
está por  
encima  
de  
Alumnos  
y  
Docente



Esta nueva clase <<Persona>> que está por encima de las clases <<Alumnos>> y <<Docentes>> solamente tiene esos tres atributos: Nombre – Domicilio – DNI  
Pero....

Las clases <<Alumnos>> y <<Docentes>> ya no los tendrán en la creación de la clase!..

Esto no quiere decir que ya no los necesiten.. Si necesitan que estén también.. Y ahí nace el concepto de Herencia.

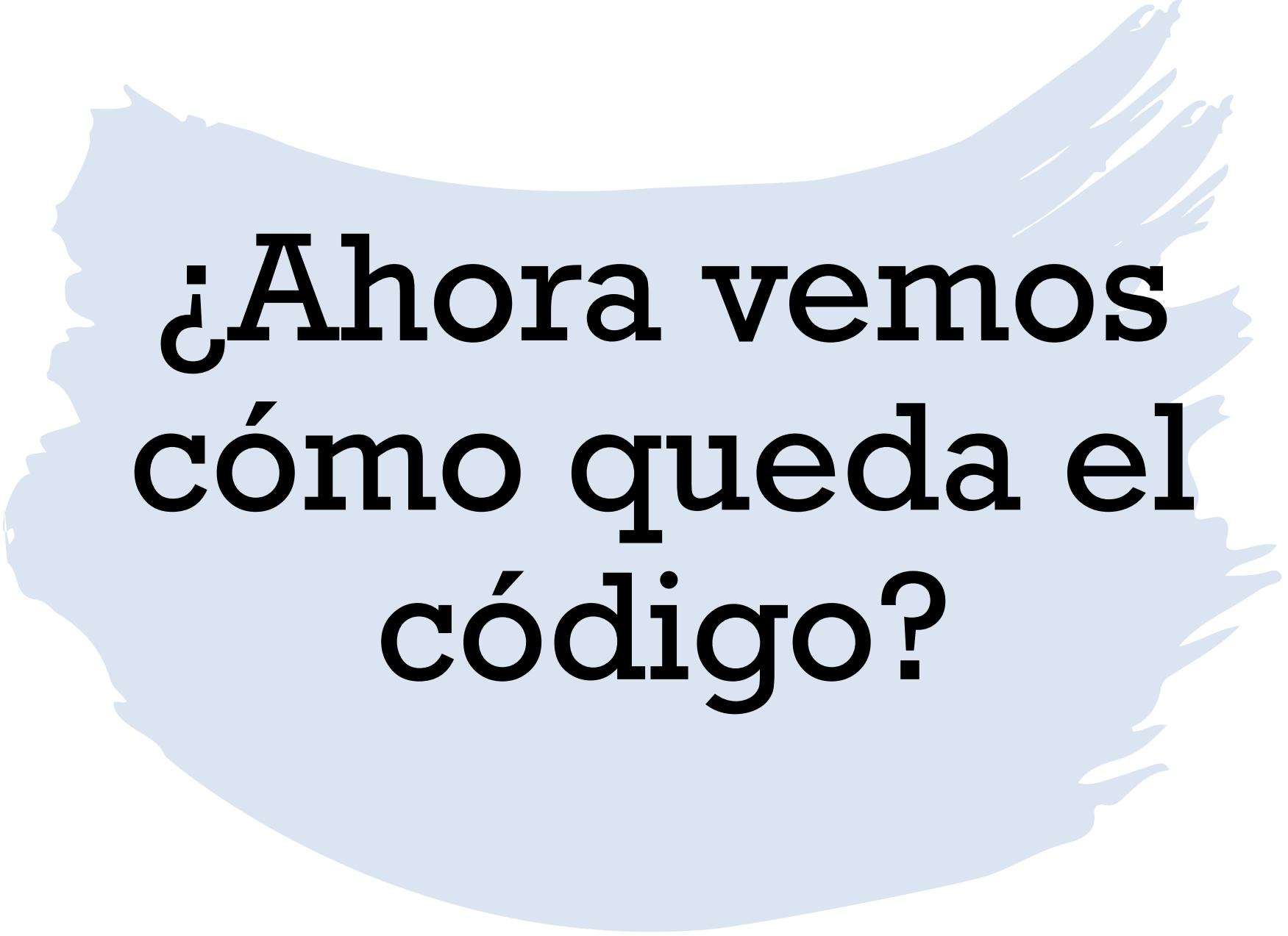
Qué significa esto?.. que <<Alumno>> hereda de <<Persona>> estos atributos <<Alumno tiene estos atributos no porque estén en el código sino porque lo hereda de <<Personas>>

Y Con <<Docentes>> que sucede?

Lo mismo.. Docente también va a tener nombre, domicilio y dni.. No porque estén definidos dentro de la clase...

Sino porque lo hereda de <<Persona>>

De esta forma Sintetizo la definición de nombre, domicilio y dni en una sola clase y no lo replico en Docente y Alumnos.



**¿Ahora vemos  
cómo queda el  
código?**

```
import os
os.system("cls")

#Definición de la clase Alumnos
#Con sus atributos y métodos
class Persona():
    def __init__(self,
id=0,nombre="",domicilio="",dni=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni
```



Clase persona

```
import os
os.system("cls")
#From el archivo clasePersona importo a Persona
from clasePersona import Persona

class Alumnos(Persona):
    def __init__(self,id=0,nombre="",domicilio="",dni=0,edad=0):
        Persona.__init__(self, id,nombre,domicilio,dni)
        #Saco los atributos y los importo de la clase persona
        #self.id=id
        #self.nombre=nombre
        #self.domicilio=domicilio
        #self.dni=dni
        self.edad=edad
    #def __init__ definimos los atributos de la clase
    def Agregar(self):
        print("Se agrego el alumno ",self.nombre,self.domicilio)
    def Modificar(self):
        print("Se modificó el alumno ",self.nombre)
    def Eliminar(self):
        print("Se eliminó el alumno ",self.nombre)
```

claseAlumnos

```
import os
os.system("cls")

#From el archivo clasePersona importo a Persona
from clasePersona import Persona

#Definición de la clase Docentes
#Con sus atributos y métodos
class Docentes(Persona):
    def __init__(self, id=0,nombre="",domicilio="",dni=0,categoria=0,antiguedad=0,sueldo=0):
        Persona.__init__(self,id,nombre,domicilio,dni)
        #self.id=id
        #self.nombre=nombre
        #self.domicilio=domicilio
        #self.dni=dni
        self.categoría=categoría
        self.antiguedad=antiguedad
        self.sueldo=sueldo
    #def__init__ definimos los atributos de la clase
    def Agregar(self):
        print("Se agrego el docente ",self.nombre, "con ",self.antiguedad," años de experiencia")
    def Modificar(self):
        print("Se modificó el docente ",self.nombre, "con ",self.antiguedad," años de experiencia")
    def Eliminar(self):
        print("Se eliminó el docente ",self.nombre, "con ",self.antiguedad," años de experiencia")
```



## claseDocente

```
from claseAlumnos import Alumnos  
from claseDocente import Docentes  
from claseCurso import Cursos  
import os  
os.system("cls")  
Alumno1=Alumnos(0,"Juan Perez","Cipolletti")  
Alumno1.Agregar()
```

```
Docente1=Docentes(0,"Ing Perez",antiguedad=10)  
Docente2=Docentes(1,"Ing Garcia",antiguedad=5)  
Docente3=Docentes(2,"Ing Fernandez",antiguedad=7)  
Docente1.Modificar()  
Docente2.Agregar()  
Docente3.Eliminar()
```

Esos atributos están ahora definidos en la clase persona pero la clase alumnos puede hacer uso de ellos, gracias a la HERENCIA

## Usar clase

Sacamos lo de curso.. Pero solamente para probar que las modificaciones anteriores no influyen en el resultado pero sí en la optimización de la programación

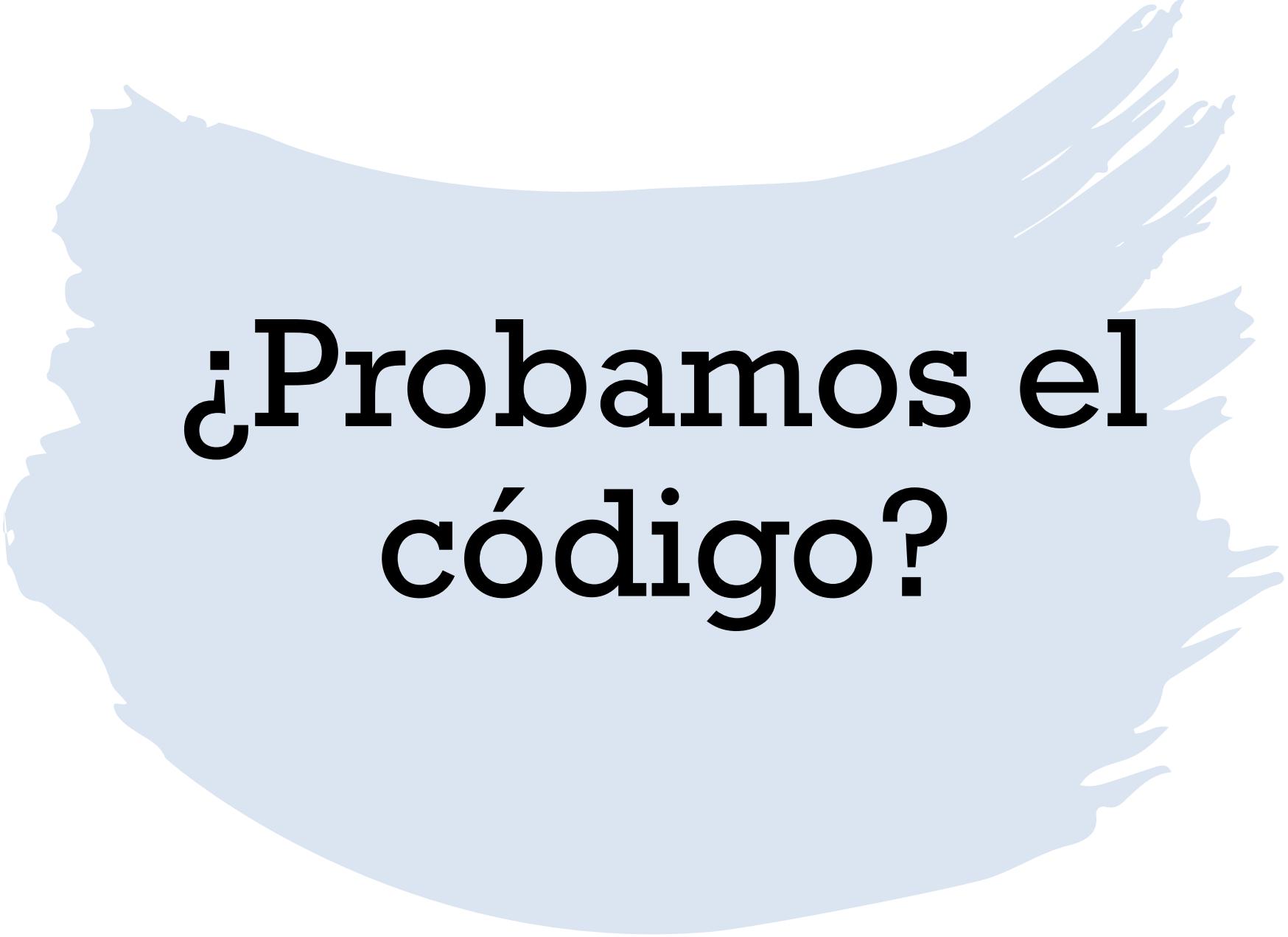
```
from claseAlumnos import Alumnos  
from claseDocente import Docentes  
from claseCurso import Cursos  
import os  
os.system("cls")  
Alumno1=Alumnos(0,"Juan Perez","Cipolletti")  
Alumno1.Agregar()  
  
Docente1=Docentes(0,"Ing Perez",antiguedad=10)  
Docente2=Docentes(1,"Ing Garcia",antiguedad=5)  
Docente3=Docentes(2,"Ing Fernandez",antiguedad=7)  
Docente1.Modificar()  
Docente2.Agregar()  
Docente3.Eliminar()
```

Esos atributos están ahora definidos en la clase persona pero la clase Docente puede hacer uso de ellos, gracias a la HERENCIA

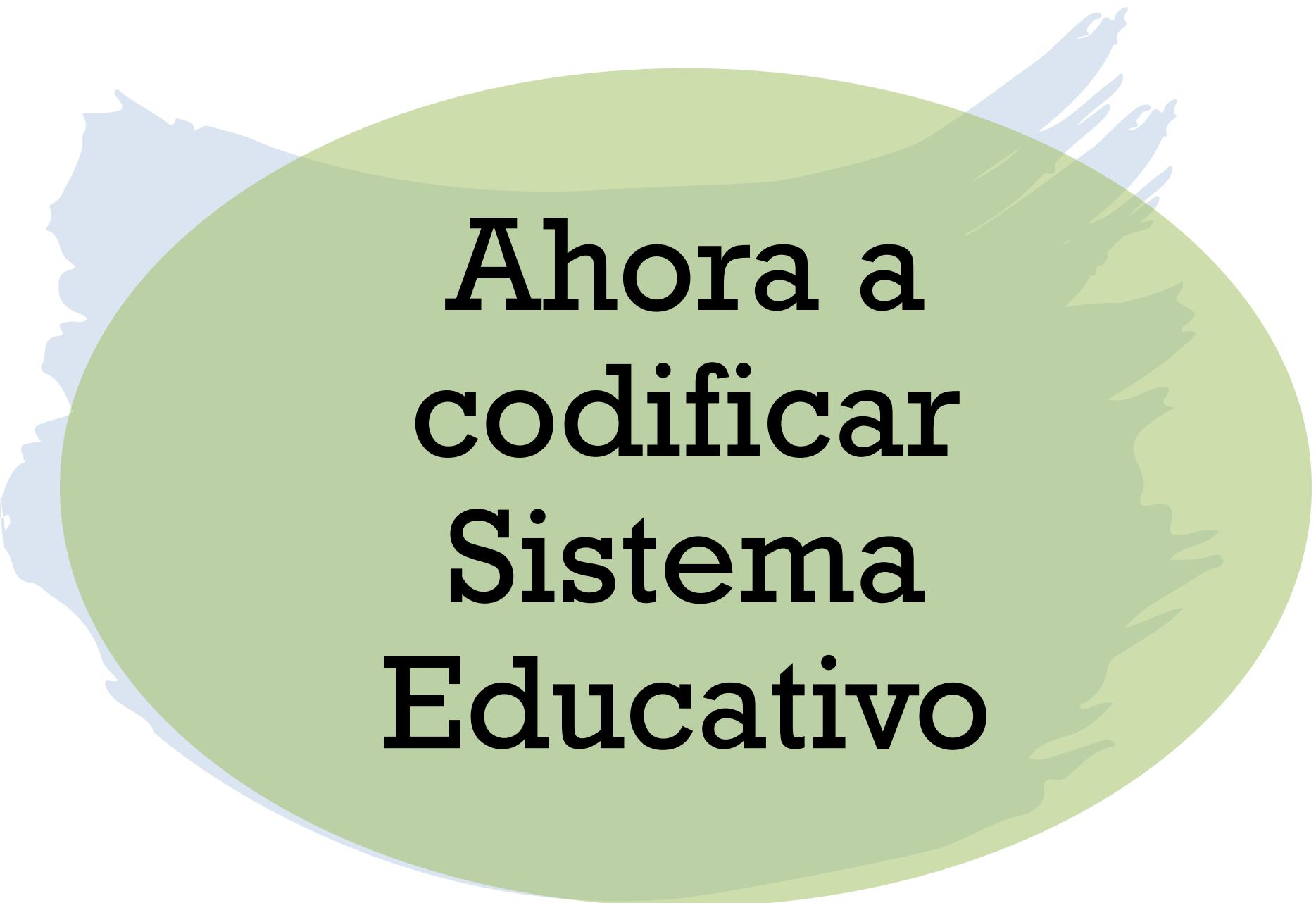
Antigüedad es propio de la clase Docente

## Usarclase

Sacamos lo de curso.. Pero solamente para probar que las modificaciones anteriores no influyen en el resultado pero sí en la optimización de la programación.



**¿Probamos el  
código?**



**Ahora a  
codificar  
Sistema  
Educativo**

# En la carpeta de ejercicios adicionales Practicar!

Pensemos una situación problemática a resolver  
(vida diaria – similar al Sist. Educativo)

Detectemos los objetos que necesitaríamos

De esos objetos definamos CARACTERÍSTICAS

De esos objetos definamos COMPORTAMIENTOS

Apliquemos el concepto de  
 herencia

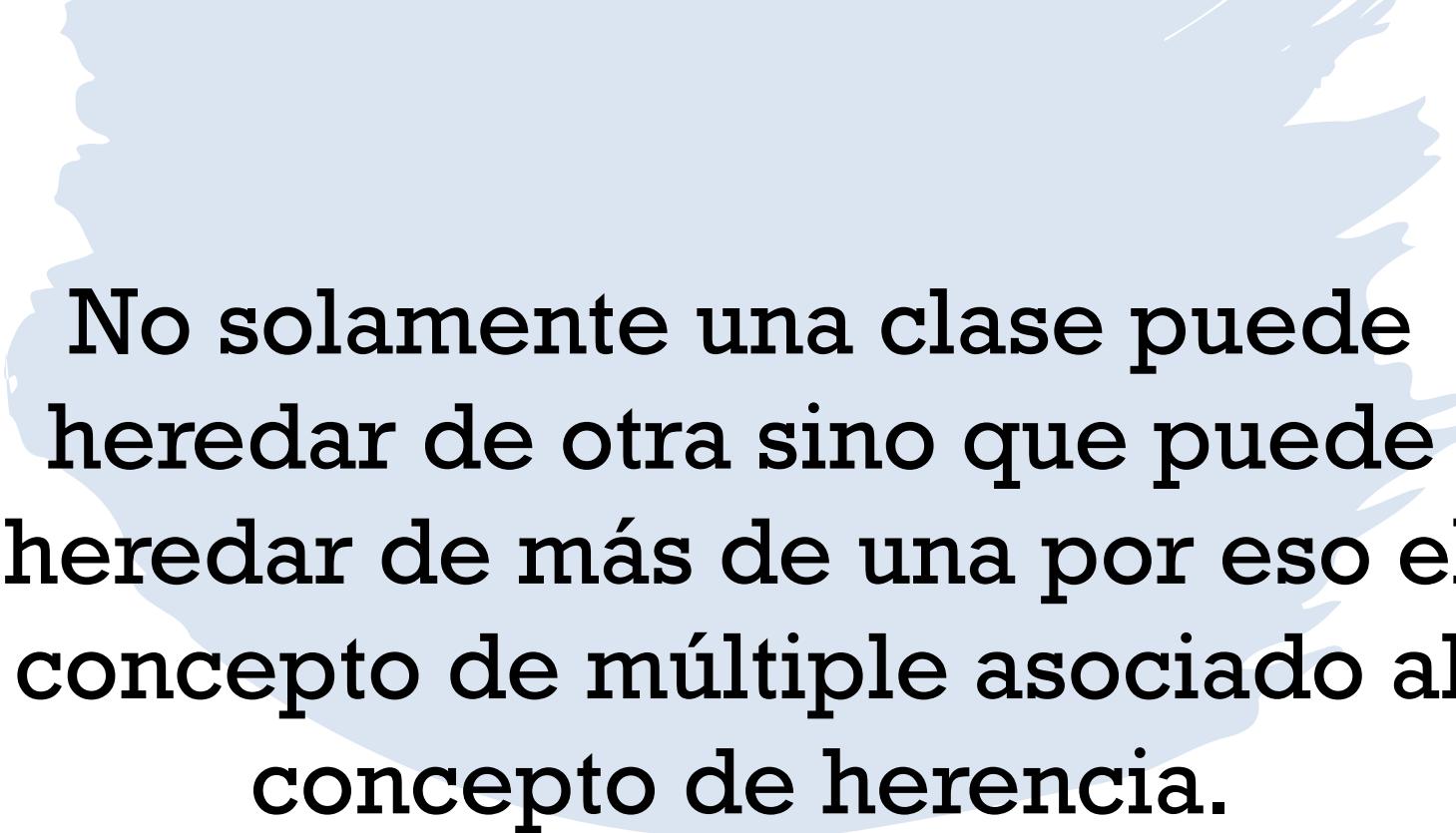


# Python

Programación Orientada a Objetos

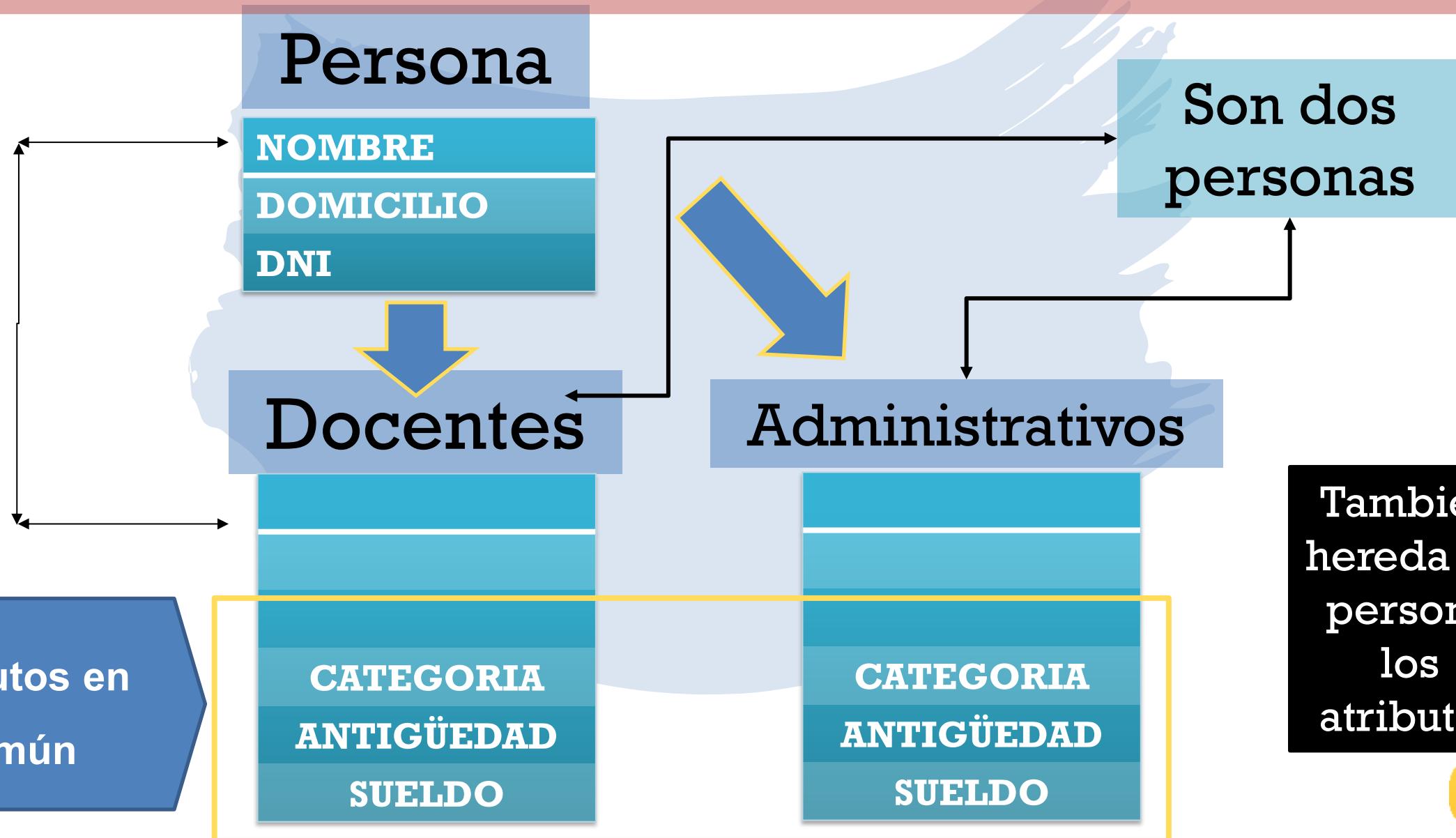
Clase N°  
13

# Herencia múltiple



No solamente una clase puede heredar de otra sino que puede heredar de más de una por eso el concepto de múltiple asociado al concepto de herencia.

# Módulos en Python



# Herencia múltiple

Los docentes y los administrativos son 2 personas dentro del sistema que realizan diferentes actividades, propias de cada rol.

Podemos decir que también hereda de la clase Persona estos atributos.

Los docentes y los administrativos al ser Empleados de este sistema educativo.. También comparten atributos: categoría, antigüedad y sueldo

# Herencia múltiple

Entonces volvemos a encontrar como cuando se analizaron los sujetos y a los objetos alumnos y docentes que tenían algo en común y pensamos en una entidad superior que los une.

En este caso también podemos decir lo mismo.. Es decir pensar en una nueva entidad llamada empleado. Empleado es una nueva clase

# Herencia múltiple

Esto quiere decir que una clase puede heredar atributos de más de una clase.

El concepto de herencia me permite heredar de una clase... ahora... cuando estamos en presencia de que una clase puede heredar atributos de más de una clase... decimos que es una herencia múltiple

# Herencia múltiple

El docente tiene cosas en común con el alumno y también con el empleado.

Persona

NOMBRE

DOMICILIO

DNI

Empleado

CATEGORIA

ANTIGÜEDAD

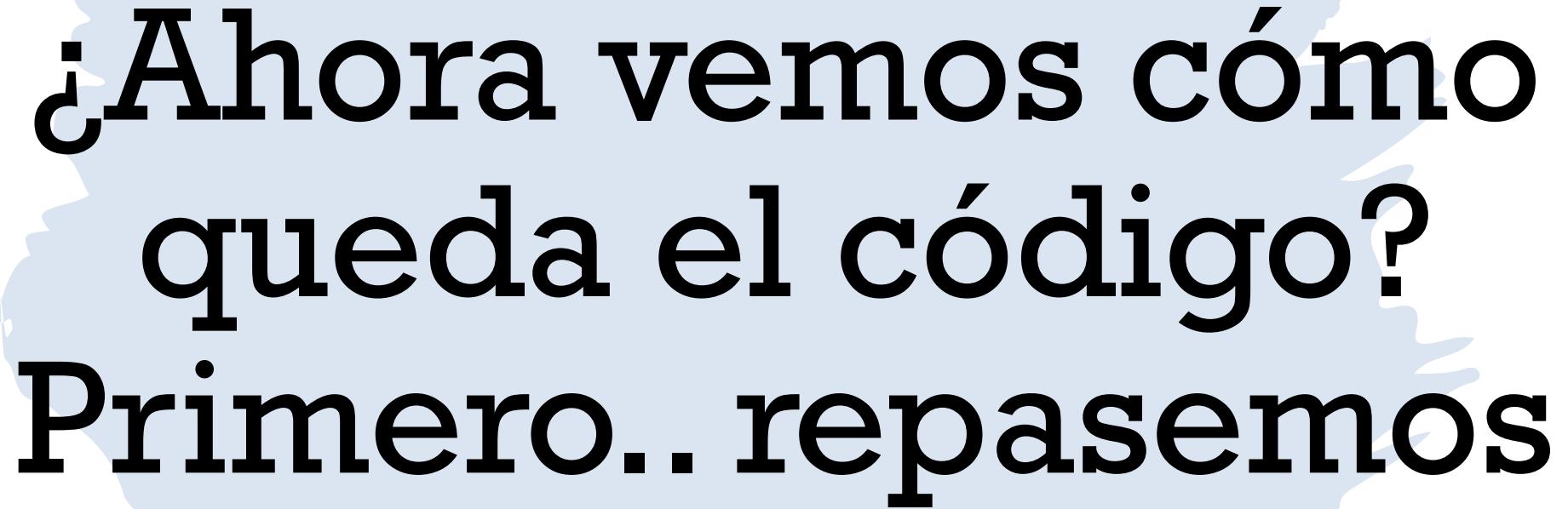
SUELDO

Docentes

Queda vacía porque hereda de dos clases... de persona y de empleado

Estamos en presencia de la herencia múltiple





**¿Ahora vemos cómo  
quedó el código?  
Primero.. repasemos**

clasePersona.py

Carpeta6 > clasePersona.py > ..

```
1
2 class Persona():
3     def __init__(self,id=0,nombre="",domicilio="",dni)
4         self.id=id
5         self.nombre=nombre
6         self.domicilio=domicilio
7         self.dni=dni
8
9
10
11
```

claseEmpleados.py

Carpeta6 > claseEmpleados.py > Empleados

```
1 class Empleados():
2     def __init__(self, categoria=0, antiguedad=0, sueldo=0):
3         self.categoria=categoría
4         self.antiguedad=antiguedad
5         self.sueldo=sueldo
6
7
8
```

Ambos grupos de atributos son  
los mismos que antes estaban  
definidos dentro de la clase  
**DOCENTES.**

```
class Empleado():
    def __init__(self, categoria=0,
antiguedad=0, sueldo=0):
        self.categoría=categoría
        self.antiguedad=antiguedad
        self.sueldo=sueldo
```

```
import os
os.system("cls")

#From el archivo clasePersona importo a Persona
from clasePersona import Persona
from claseEmpleado import Empleado

#Definición de la clase Docentes
#Con sus atributos y métodos
class Docentes(Persona,Empleado):
    def __init__(self, id=0,nombre="",domicilio="",dni=0,categoria=0,antiguedad=0,sueldo=0):
        Persona.__init__(self,id,nombre,domicilio,dni)
        Empleado.__init__(self,categoria,antiguedad,sueldo)

    def Agregar(self):
        print("Se agrego el docente ",self.nombre, "con ",self.antiguedad," años de
experiencia")
    def Modificar(self):
        print("Se modificó el docente ",self.nombre, "con ",self.antiguedad," años de
experiencia")
    def Eliminar(self):
        print("Se eliminó el docente ",self.nombre, "con ",self.antiguedad," años de
experiencia")
)
```

claseDocentes.py X

```
Carpeta6 > claseDocentes.py > Docentes
1 from clasePersona import Persona
2 from claseEmpleados import Empleados
3
4 class Docentes(Persona, Empleados):
5     def __init__(self, id=0, nombre="", dc,
6                  antiguedad=0, sueldo=0):
7         Persona.__init__(self, id, nombre)
8         Empleados.__init__(self, categor
9
10    def Agregar(self):
11        print("Se agrego el docente ", s
12        "años de experiencia")
13
14    def Modificar(self):
15        print("Se modiflico el docente "
16        "años de experiencia")
17
18    def Eliminar(self):
19        print("Se elimino el docente ", 
20        "años de experiencia")
21
22
```

clasePerson.py X

Carpeta6 > clasePerson.py > \_

```
1
2 class Persona():
3     def __init__(self, id=0, nombre=
4                     self.id=id
5                     self.nombre=nombre
6                     self.domicilio=domicilio
7                     self.dni=dni
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

claseEmpleados.py X

Carpeta6 > claseEmpleados.py > Empleados

```
1 class Empleados():
2     def __init__(self, categoria=0, an
3                     self.categoria=categoria
4                     self.antiguedad=antiguedad
5                     self.sueldo=sueldo
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

I



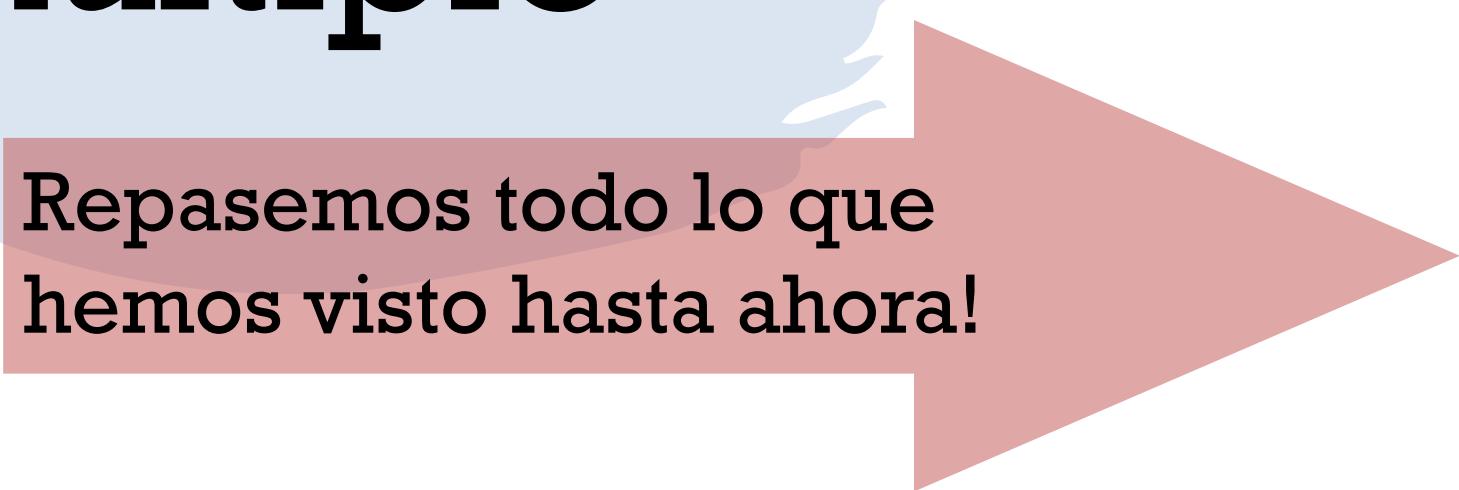
# Python

Programación Orientada a Objetos

Clase N°  
14



# **Herencia & Herencia múltiple**



**Repasemos todo lo que  
hemos visto hasta ahora!**

# Programación Orientada a Objetos

## ¿Cómo se crea una CLASE??

```
class Alumnos():

    def <ATRIBUTOS>

    def <METODOS>
```



```
import os
os.system("cls")
class Alumnos():
    def __init__(self, id=0,nombre="",domicilio="",dni=0,edad=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni
        self.edad=edad

    def Agregar(self):
        print("Se agrego el alumno ",self.nombre,self.domicilio)
    def Modificar(self):
        print("Se modificó el alumno ",self.nombre)
    def Eliminar(self):
        print("Se eliminó el alumno ",self.nombre)

Alumno1=Alumnos(0,"Juan Perez","Cipolletti")
Alumno1.Agregar()
Alumno1.Modificar()
Alumno1.Eliminar()
```

Atributos

Métodos

Instancia

```
import os
os.system("cls")
#From el archivo clasePersona importo a Persona
from clasePersona import Persona
```

```
class Alumnos(Persona):
    def __init__(self,id=0,nombre="",domicilio="",dni=0,edad=0):
        Persona.__init__(self, id,nombre,domicilio,dni)
```

## Herencia

```
import os
os.system("cls")

#From el archivo clasePersona importo a Persona
from clasePersona import Persona
from claseEmpleado import Empleado

#Definición de la clase Docentes
#Con sus atributos y métodos
class Docentes(Persona,Empleado):
    def __init__(self,
id=0,nombre="",domicilio="",dni=0,categoria=0,antiguedad=0,sueldo=0):
        Persona.__init__(self,id,nombre,domicilio,dni)
        Empleado.__init__(self,categoria,antiguedad,sueldo)
```

## Herencia múltiple

# **Herencia....!**

**Hemos aplicado herencia  
y herencia múltiple de los  
atributos... y que pasa  
con los métodos....?  
¿También se heredan?**

**SIIIIIIII**

# Herencia....!

```
class Docentes():

    def __init__(self, id=0,nombre="",domicilio="",dni=0,
                 self.id=id
                 self.nombre=nombre
                 self.domicilio=domicilio
                 self.dni=dni
                 self.categoría=categoría
                 self.antiguedad=antiguedad
                 self.sueldo=sueldo

    def Agregar(self):
        print("Se agrego el docente ",self.nombre, "con "
    def Modificar(self):
        print("Se modificó el docente ",self.nombre, "con "
    def Eliminar(self):
        print("Se eliminó el docente ",self.nombre, "con "

Docente1=Docentes(0,"Ing Perez",antiguedad=10)
Docente2=Docentes(1,"Ing Garcia",antiguedad=5)
Docente3=Docentes(2,"Ing Fernandez",antiguedad=7)
```

```
4   class Alumnos():

5
6       def __init__(self, id=0,nombre="",domicilio="",dni=0,
7                     self.id=id
8                     self.nombre=nombre
9                     self.domicilio=domicilio
10                    self.dni=dni
11                    self.edad=edad

12
13       def Agregar(self):
14           print("Se agrego el alumno ",self.nombre,self.dom
15       def Modificar(self):
16           print("Se modificó el alumno ",self.nombre)
17       def Eliminar(self):
18           print("Se eliminó el alumno ",self.nombre)
19       #def Agregar(self) definimos los métodos

20
21       Alumno1=Alumnos(0,"Juan Perez","Cipolletti")
22
23       Alumno1.Agregar()
24       Alumno1.Modificar()
25       Alumno1.Eliminar()
```

Si bien tienen métodos iguales, se llaman igual y hacen lo mismo. La esencia de esos métodos no es la misma. Agregar un alumno no es lo mismo que agregar un docente.

El conjunto de datos que tiene un alumno no es el mismo que el conjunto de datos que tiene un docente.

Si bien se crea la clase persona y podemos trasladar los atributos a las clases.. En este caso no se puede heredar los métodos porque tendrían una solución errónea

Una coincidencia sería entre los docentes y los Administrativos.

Podemos ver que Clase Docente y la Clase Administrativo heredan los atributos de la clase Empleados

Ahora vamos a ver como sería la Clase Administrativo

The screenshot shows a Python IDE interface with two code files open:

- claseDocente.py** (Left Window):

```
1 import os
2 os.system("cls")
3
4 class Docentes():
5     def __init__(self, id=0,nombre="",domicilio="",dni=0,categoria=0,antiguedad=0,sueldo=0):
6         self.id=id
7         self.nombre=nombre
8         self.domicilio=domicilio
9         self.dni=dni
10        self.categoría=categoría
11        self.antiguedad=antiguedad
12        self.sueldo=sueldo
13
14    def Agregar(self):
15        print("Se agrego el docente ",self.nombre, "con ",self.antiguedad)
16    def Modificar(self):
17        print("Se modificó el docente ",self.nombre, "con ",self.antiguedad)
18    def Eliminar(self):
19        print("Se eliminó el docente ",self.nombre, "con ",self.antiguedad)
20
21 Docente1=Docentes(0,"Ing Perez",antiguedad=10)
22 Docente2=Docentes(1,"Ing Garcia",antiguedad=5)
23 Docente3=Docentes(2,"Ing Fernandez",antiguedad=7)
24
25 Docente1.Modificar()
26 Docente2.Agregar()
27 Docente3.Eliminar()
```
- claseAdministrativo.py** (Right Window):

```
8 #Definición de la clase Administrativo
9 #Con sus atributos y métodos
10 class Administrativo(Persona,Empleado):
11     def __init__(self, id=0,nombre="",domicilio="",dni=0,categoría=0,antiguedad=0,sueldo=0):
12         Persona.__init__(self,id,nombre,domicilio,dni)
13         Empleado.__init__(self,categoría,antiguedad,sueldo)
14
15
16
```

A large text overlay on the right side of the interface reads:

# No agregamos los métodos

Pero.. Qué tienen en común los docentes y los administrativos?  
La categoría y el sueldo. Ambos se pueden actualizar

Esto nos permite pensar en el método Actualizar que se van a incorporar dentro de la clase Empleados para trabajar el concepto de HERENCIA pero en MÉTODOS

Entonces.. Vamos a crear un método que nos permita actualizar la categoría y otro método que nos permita actualizar el sueldo

# ¿Dónde van esos métodos?

Dentro de la clase  
Empleado

```
class Empleado():
    def __init__(self, categoria=0, antiguedad=0, sueldo=0):
        self.categoría=categoría
        self.antiguedad=antiguedad
        self.sueldo=sueldo
    def Actualizarsueldo(self):
        print("Se actualizó el sueldo del empleado ",self.nombre,
con una antiguedad de ",self.antiguedad)
    def ActualizarCategoria(self):
        print("Se actualizó la categoria del empleado
",self.nombre," con una antiguedad de ",self.antiguedad)
```

La intención de agregar estos dos métodos dentro de la clase empleados es poder ver que se pueden ejecutar estos métodos... pero no se ejecutan desde la clase Empleado sino desde clases que heredan no solamente atributos sino también métodos de la clase Empleado

Se podrá ejecutar estos métodos desde las clases Docentes y/o de la clase Administrativos, ya que ambas clases heredan de la clase Empleados

**Repasemos....**

claseDocente.py • claseAdministrativo.py X

D ▾ II ...

claseAdministrativo.py X

NIVELINTERMEDIO > CARPETAS > claseAdministrativo.py > ...

```
1 import os
2 os.system("cls")
3
4 #From el archivo clasePersona importo a Persona
5 from clasePersona import Persona
6 from claseEmpleado import Empleado
7
8 #Definición de la clase Administrativo
9 #Con sus atributos y métodos
10 class Administrativo(Persona,Empleado):
11     def __init__(self, id=0,nombre="",domicilio="",dni=0,categoria=0,
12                  antiguedad=0,sueldo=0):
13         Persona.__init__(self,id,nombre,domicilio,dni)
14         Empleado.__init__(self,categoría,antiguedad,sueldo)
15
16 
```

NIVELINTERMEDIO > CARPETAS > claseAdministrativo.py > Administrativo > \_\_init\_\_

```
1 import os
2 os.system("cls")
3
4 #From el archivo clasePersona importo a Persona
5 from clasePersona import Persona
6 from claseEmpleado import Empleado
7
8 #Definición de la clase Administrativo
9 #Con sus atributos y métodos
10 class Administrativo(Persona,Empleado):
11     def __init__(self, id=0,nombre="",domicilio="",dni=0,categoryId=0,
12                  antiguedad=0,sueldo=0):
13         Persona.__init__(self,id,nombre,domicilio,dni)
14         Empleado.__init__(self,categoryId,antiguedad,sueldo)
15
16 
```

# Clase Empleado

```
class Empleado():
    def __init__(self,categoryId=0, antiguedad=0,sueldo=0):
        self.categoryId=categoryId
        self.antiguedad=antiguedad
        self.sueldo=sueldo

    def Actualizarsueldo(self):
        print("Se actualizó el sueldo del empleado ",self.nombre," con una antiguedad de ",
              ",self.antiguedad)

    def ActualizarCategoria(self):
        print("Se actualizó la categoria del empleado ",self.nombre," con una antiguedad de ",
              ",self.antiguedad)
```

Que tenemos que hacer con los métodos que hemos definido en la clase Empleados?

Absolutamente nadaaaaaaa..

Porque en Administrativo hay una herencia respecto de Empleados

Y en Docentes hay una herencia con respecto a Empleados

```
from claseAlumnos import Alumnos
from claseDocente import Docentes
from claseAdministrativo import Administrativo
import os
os.system("cls")
Alumno1=Alumnos(0,"Juan Perez","Cipolletti")
Alumno1.Agregar()

Docente1=Docentes(0,"Ing Perez",antiguedad=10)
Docente2=Docentes(1,"Ing Garcia",antiguedad=5)
Docente3=Docentes(2,"Ing Fernandez",antiguedad=7)
Docente1.Modificar()
Docente2.Agregar()
Docente3.Actualizarsueldo()
```

# Ejecutar UsarClases

```
Admin1=Administrativo(0,"Laura Fernandez",antiguedad=5)
Admin1.ActualizarCategoria()
```

```
2 from claseDocentes import Docentes  
3 from claseAdministrativos import Administrativos  
4 import os  
5  
6 os.system("cls")  
7 Alumno1=Alumnos(0,"Raul Lopez")  
8 Alumno1.Agregar()  
9  
10 Docente1=Docentes(0,"Ing. Perez",antiguedad=10)  
11 Docente1.Modificar()  
12 Docente1.ActualizarSueldo()  
13  
14 Admin1=Administrativos(0,"Laura Garcia",antiguedad=10)  
15 Admin1.ActualizarCategoria()
```

MODIFICAR es un método propio de la clase Docente. NO ES HEREDADO

ACTUALIZARSUELDO es un método heredado de la clase EMPLEADO

ACTUALIZARCATEGORIA es un método heredado de la clase EMPLEADO

UsarClases.py X

```
Carpeta7 > UsarClases.py > ...
1 from claseAlumnos import Alumnos
2 from claseDocentes import Docentes
3 from claseAdministrativos import Administrativos
4 import os
5
6 os.system("cls")
7 Alumno1=Alumnos(0,"Raul Lopez")
8 Alumno1.Agregar() 
9
10 Docente1=Docentes(0,"Ing. Perez",antiguedad=10)
11 Docente1.Modificar()
12 Docente1.ActualizarSueldo()
13
14 Admin1=Administrativos(0,"Laura Garcia",antiguedad=10)
15 Admin1.ActualizarCategoria()
16
```

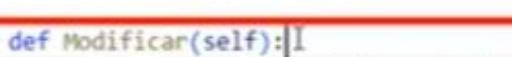
claseAlumnos.py X

```
Carpeta7 > claseAlumnos.py > ...
1 from clasePersona import Persona
2
3 class Alumnos(Persona):
4     def __init__(self,id=0,nombre="",domicilio=""):
5         Persona.__init__(self,id,nombre,domicilio)
6         self.edad=edad
7
8 def Agregar(self):
9     print("Se agrego el alumno ",self.nombre)
10
11 def Modificar(self):
12     print("Se modiflico el alumno ",self.nombre)
13
14 def Eliminar(self):
15     print("Se elimino el alumno ",self.nombre)
16
```

UsarClases.py X

```
Carpeta7 > UsarClases.py > ...
1 from claseAlumnos import Alumnos
2 from claseDocentes import Docentes
3 from claseAdministrativos import Administrativos
4 import os
5
6 os.system("cls")
7 Alumno1=Alumnos(0,"Raul Lopez")
8 Alumno1.Agregar()
9
10 Docente1=Docentes(0,"Ing. Perez",antiguedad=10)
11 Docente1.Modificar() 
12 Docente1.ActualizarSueldo()
13
14 Admin1=Administrativos(0,"Laura Garcia",antiguedad=10)
15 Admin1.ActualizarCategoria()
16
17
```

claseDocentes.py X

```
Carpeta7 > claseDocentes.py > Docentes > Modificar
1 from clasePersona import Persona
2 from claseEmpleados import Empleados
3
4 class Docentes(Persona, Empleados):
5     def __init__(self,id=0,nombre="",domicilio="",
6                  antiguedad=0,sueldo=0):
7         Persona.__init__(self,id,nombre,domicilio)
8         Empleados.__init__(self,categoria,antiguedad,sueldo)
9
10 def Agregar(self):
11     print("Se agrego el docente ",self.nombre,
12          "años de experiencia")
13
14 def Modificar(self): 
15     print("Se modiflico el docente ",self.nombre,
16          "años de experiencia")
17
```

```
UsarClases.py x
Carpeta7 > UsarClases.py > ...
1 from claseAlumnos import Alumnos
2 from claseDocentes import Docentes
3 from claseAdministrativos import Administrativos
4 import os
5
6 os.system("cls")
7 Alumno1=Alumnos(0,"Raul Lopez")
8 Alumno1.Agregar()
9
10 Docente1=Docentes(0,"Ing. Perez",antiguedad=10)
11 Docente1.Modificar()
12 Docente1.ActualizarSueldo() ←
13
14 Admin1=Administrativos(0,"Laura Garcia",antiguedad=10)
15 Admin1.ActualizarCategoria()
16
17
```

```
claseDocentes.py x
Carpeta7 > claseDocentes.py > Docentes > Modificar
1
2
3
4 class Docente(Persona, Empleados):
5     def __init__(self,id=0,nombre=" ",domicilio=" ",
6                  antiguedad=0,sueldo=0):
7         Persona.__init__(self,id,nombre,domicilio)
8         Empleados.__init__(self, categoria,antiguedad,sueldo)
9
10
11 def Agregar(self):
12     print("Se agrego el docente ",self.nombre,
13           "años de experiencia")
14
15 def Modificar(self):
16     print("Se modifico el docente ",self.nombre,
17           "años de experiencia")
18
19 def Eliminar(self):
20     print("Se elimino el docente ",self.nombre)
```

```
UsarClases.py x
Carpeta7 > UsarClases.py > ...
1 from claseAlumnos import Alumnos
2 from claseDocentes import Docentes
3 from claseAdministrativos import Administrativos
4 import os
5
6 os.system("cls")
7 Alumno1=Alumnos(0,"Raul Lopez")
8 Alumno1.Agregar()
9
10 Docente1=Docentes(0,"Ing. Perez",antiguedad=10)
11 Docente1.Modificar()
12 Docente1.ActualizarSueldo()
13
14 Admin1=Administrativos(0,"Laura Garcia",antiguedad=10) ←
15 Admin1.ActualizarCategoria()
16
17
```

```
claseAdministrativos.py x
Carpeta7 > claseAdministrativos.py > ...
1 from clasePersona import Persona
2 from claseEmpleados import Empleados
3
4 class Administrativo(Persona, Empleados):
5     def __init__(self,id=0,nombre=" ",domicilio=" ",
6                  antiguedad=0,sueldo=0):
7         Persona.__init__(self,id,nombre,domicilio)
8         Empleados.__init__(self, categoria,antiguedad,sueldo)
9
10
11
12
```

El método `ActualizarSueldo()` no está definido en la clase `docentes`.  
Entonces como hace?  
Busca en algunas de estas dos clases que se heredan.  
Lo busca en `persona` y como no está lo busca en `Empleados`.  
Porque Busca de izquierda a derecha en las clases heredadas.

Lo mismo sucede con el método `ActualizarCategoria()`  
No existe en la clase `Administrativo` entonces lo busca en `Persona` y luego en `Empleados`

# ¿Cómo hace la búsqueda?

Primero se va a lo cercano.. Por ejemplo..

Docente3.Actualizarsueldo()

En este caso.. Lo va a buscar en la Clase Docente, no en la clase Empleado porque Docente es lo que tiene más cerca. Luego a entrar a Docente se “entera” que hay una herencia con Empleado y lo va a buscar a Empleado

```
def Actualizarsueldo(self):  
    print("Se actualizó el sueldo del empleado ",self.nombre," con una antigüedad de ",self.antiguedad)
```

Primero lo busca en su propia clase y luego en las heredadas.. Siempre de izquierda a derecha si estamos en presencia de herencia múltiple

La secuencia o cadena de Herencias puede ser tan larga como sea necesario...

Clase1

Clase2 (Clase1)  
Clase3 (Clase2)

.....

ClaseN (Clase N-1)

```

from claseAlumnos import Alumnos
from claseDocente import Docentes
from claseAdministrativo import Administrativo
import os
os.system("cls")
Alumno1=Alumnos(0,"Juan Perez","Cipolletti")
Alumno1.Agregar()

```

```

Docente1=Docentes(0,"Ing Perez",antiguedad=10)
Docente2=Docentes(1,"Ing Garcia",antiguedad=5)
Docente3=Docentes(2,"Ing Fernandez",antiguedad=7)
Docente2.Agregar()
Docente3.Actualizarsueldo()

```

```

Admin1=Administrativo(0,"Laura Fernandez",antiguedad=5)
Admin1.ActualizarCategoria()

```

```

import os
os.system("cls")

from clasePersona import Persona
from claseEmpleado import Empleado
class Administrativo(Persona,Empleado):
    def __init__(self, id=0,nombre="",domicilio="",dni=0,categoria=0,
                 antiguedad=0,sueldo=0):
        Persona.__init__(self,id,nombre,domicilio,dni)
        Empleado.__init__(self,categoria,antiguedad,sueldo)

```

```

class Empleado():
    def __init__(self,categoria=0, antiguedad=0,sueldo=0):
        self.categoria=categoria
        self.antiguedad=antiguedad
        self.sueldo=sueldo
    def Actualizarsueldo(self):
        print("Se actualizó el sueldo del empleado ",self.nombre," con una antiguedad de "
              ",self.antiguedad)
    def ActualizarCategoria(self):
        print("Se actualizó la categoria del empleado ",self.nombre," con una antiguedad de "
              ",self.antiguedad)

```

## UsarClase

1

2

3

4

## Administrativo

## Empleado

3

```

import os
os.system("cls")

```

#Definición de la clase Alumnos  
#Con sus atributos y métodos

```

class Persona():
    def __init__(self,id=0,nombre="",domicilio="",dni=0):
        self.id=id
        self.nombre=nombre
        self.domicilio=domicilio
        self.dni=dni

```

```

import os
os.system("cls")

```

```

from clasePersona import Persona
from claseEmpleado import Empleado

```

```

class Docentes(Persona,Empleado):
    def __init__(self, id=0,nombre="",domicilio="",dni=0,
                 categoria=0,antiguedad=0,sueldo=0):
        Persona.__init__(self,id,nombre,domicilio,dni)
        Empleado.__init__(self,categoria,antiguedad,sueldo)

    def Agregar(self):
        print("Se agrego el docente ",self.nombre, "con ",self.antiguedad," años de experiencia")
    def Modificar(self):
        print("Se modificó el docente ",self.nombre, "con ",self.antiguedad," años de experiencia")

```

```

import os
os.system("cls")

```

```

from clasePersona import Persona
class Alumnos(Persona):

```

```

    def __init__(self, id=0,nombre="",domicilio="",dni=0,edad=0):
        Persona.__init__(self, id,nombre,domicilio,dni)

```

```

    def Agregar(self):
        print("Se agrego el alumno ",self.nombre,self.domicilio)

```

```

    def Modificar(self):
        print("Se modificó el alumno ",self.nombre)

```

```

    def Eliminar(self):
        print("Se eliminó el alumno ",self.nombre)

```

## Persona

3

## Docente

## Alumno



# Python

Programación Orientada a Objetos

Clase N°  
15 - 16