Introducción a POO en Python

Clase 1

Introducción

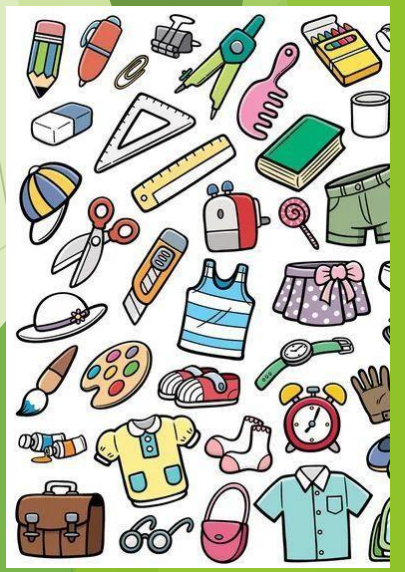
- En Python todo es un **"objeto"** y debe ser manipulado como tal. Pero ¿Qué es un objeto? ¿De qué hablamos cuando nos referimos a **"orientación a objetos"**? Nos enfocaremos primero, en cuestiones de **conceptos básicos**, para luego, ir introduciéndonos de a poco, en principios teóricos elementalmente necesarios, para implementar la orientación a objetos en la práctica.

Pensar en objetos

Un objeto es **"una cosa"**. Y, si una cosa es un **sustantivo**, entonces un objeto es un sustantivo.

Mira a tu alrededor y encontrarás decenas, cientos de objetos. Tu ordenador, es un objeto. Tú, eres un objeto. Tu llave es un objeto. Hasta tu celular, es otro objeto. Tu mascota también es un objeto.

Cuando pensamos en *"objetos"*, todos los sustantivos son objetos.



Describimos las cualidades de un objeto

Describir un objeto, es simplemente mencionar sus cualidades. Las cualidades son adjetivos. Un **adjetivo** es una cualidad del sustantivo. Entonces, para describir **"la manera de ser"** de un objeto, debemos preguntarnos ¿cómo es el objeto? Toda respuesta que comience por "el objeto es", seguida de un **adjetivo**, será una cualidad del objeto.

El objeto es verde **El objeto** es grande **El objeto** es feo

pelota
sustantivo

redonda
adjetivo

naranja
adjetivo

grande
adjetivo

Describimos las cualidades de un objeto

Ahora supongamos que en una conversación, empiezan a preguntarte **¿Qué es?** cada adjetivo que definimos a un objeto en particular. Tu respuesta debería ser, **"es un/una"** seguido de un sustantivo.

Por ejemplo:

– El objeto es verde. **¿Qué es verde?** Un color.

– El objeto es grande. **¿Qué es grande?** Un tamaño.

– El objeto es feo. **¿Qué es feo?** Un aspecto.

Describimos las cualidades de un objeto

Estos sustantivos que responden a la pregunta ¿Que es?, pueden pasar a formar parte de una locución adjetiva que especifique con mayor precisión, las descripciones anteriores:

– El objeto es de **color** verde.

– El objeto es de **tamaño** grande.

– El objeto es de **aspecto** feo.

Podemos decir entonces que una cualidad, es un **atributo** (derivado de "cualidad **atribuible** a un objeto") y que entonces, un objeto es un sustantivo que posee atributos, cuyas cualidades lo describen.

Describimos las cualidades de un objeto

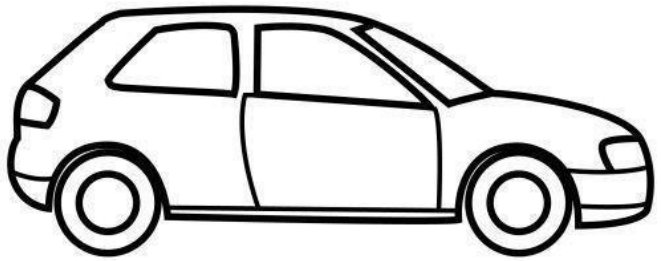
Veámoslo en el siguiente cuadro:

OBJETO (sustantivo)	ATRIBUTO (locución adjetiva)	CUALIDAD DEL ATRIBUTO (adjetivo)
(el) Objeto	(es de) color	Verde
	(es de) tamaño	Grande
	(es de) aspecto	Feo

Objetos compuestos de objetos

Ahora que sabemos que los atributos de un objeto son sustantivos, nos preguntamos. ¿Estos sustantivos no pueden ser otros objetos?. Y en efecto, si pueden serlo. Veamos el ejemplo del automóvil:

Objeto: Automóvil



Objeto: Rueda

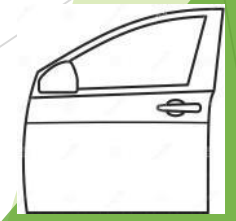
Objeto: Luz

Objeto: Estereo

Objeto: Motor

Objeto: Volante

Objeto: Puerta



Objetos compuestos de objetos

Si pueden observar cada una de estas partes pueden tener sus propios atributos.

Objeto Rueda:

- Color: Negro
- Rodado: 29 cm

Objeto Estereo:

- Marca: Sony
- Usb: Tiene

Objeto Automóvil:

- Pintura: Rojo
- Precio: $500.000

Objeto Motor:

- Marca: Honda
- Tipo de Válvulas: V6

Objeto Luz:

- Tipo: Trasera
- Color: Roja

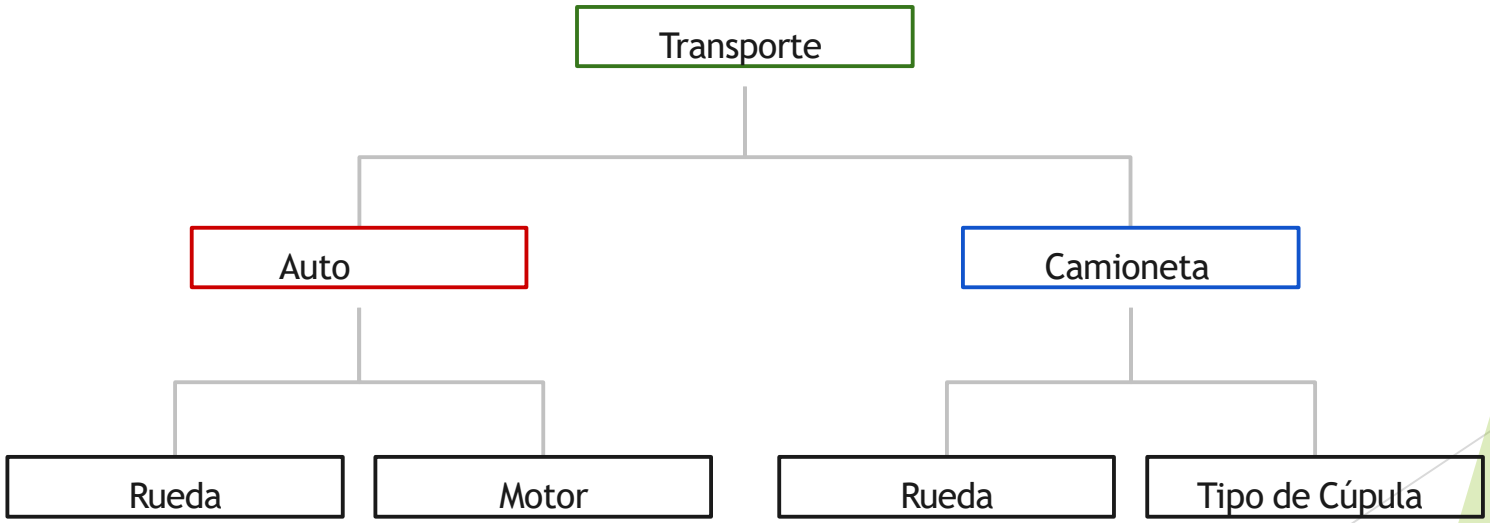
Objetos que comparten características con otros objetos

Hemos comenzado a entender un poco más que los objetos pueden tener atributos que sean otros objetos. Si bien, este objeto automóvil tiene sus características, también comparte características con otros tipos de "transportes", como por ejemplo un objeto Camioneta.

Los dos tienen Ruedas, Puertas, Volantes, Motor, etc.
Pero a nuestro objeto Camioneta se le puede sumar un atributo más, como por ejemplo "Tipo de Cúpula", que obviamente un Automóvil no la tiene.

Objetos que comparten características con otros objetos

Ahora tenemos una clasificación de transportes, que comparten atributos. A esto se le llama Herencia.
Y si pueden ver, el objeto Camioneta a pesar de tener las mismas características del Automóvil, tiene otra diferente. A esto se lo denomina Polimorfismo.



Los Objetos pueden hacer Acciones

Pasemos a lo siguiente. Tenemos que los objetos pueden tener características que describen o dicen de qué partes están compuestos. Pero también los mismos pueden realizar acciones, que para el caso de nuestro objeto Automóvil pueden ser:

Objeto Automóvil:

- Acelerar
- Frenar
- Girar

Todas estas acciones son Verbos, de los cuales son particulares del objeto. Así que también pueden ser heredadas.

Vocabulario en la Programación Orientada a Objetos

Como todo tecnicismo en la programación, cada cosa que mencionamos tiene su nombre particular en la programación orientada a objetos. En el cuadro siguiente vamos a estandarizar esto:

Cuando hablamos de…	**En la programación se denomina...**	**En la programación orientada a objetos es...**
“objeto”	Objeto	Un elemento
“atributos” (o cualidades)	Propiedades	Un elemento
e “acciones” que puede realizar el objeto	Métodos	Un elemento
“atributos-objeto”	Composición	Una técnica
que tienen nombres de atributos iguales y sin embargo, pueden tener valores diferentes	Polimorfismo	Una característica
objetos que son sub-tipos (o ampliación) de otros	Herencia	Una característica

A programar Objetos en Python

Con todo esto en mente. Comenzaremos a trasladarlo a código Python, para poder hacer uso de los mismos.

Recuerden siempre, que una buena práctica en la programación orientada a objetos o **POO**, realizar un diagrama de objetos con sus propiedades, métodos y herencias, siempre es fundamental para ahorrarse problemas a futuro.

Elementos y Características de la POO

Los elementos de la POO, pueden entenderse como los "materiales" que necesitamos para diseñar y programar un sistema, mientras que las características, podrían asumirse como las "herramientas" de las cuáles disponemos para construir el sistema con esos materiales.

Entre los elementos principales de la POO, podremos encontrar a:

- Clases
- Propiedades
- Métodos

Clases

Las clases son los modelos sobre los cuáles se construirán nuestros objetos.
En Python, una clase se define con la instrucción **class** seguida de un nombre genérico para el objeto.

```
class Objeto:
    #propiedades #metodos

class Automovil: #propiedades #metodos
```

Propiedades

Las propiedades, como hemos visto antes, son las características intrínsecas del objeto. Éstas, se representan a modo de variables, solo que técnicamente, pasan a denominarse "propiedades":

```
class Objeto:
    #propiedades propiedad =" #metodos

class Automovil: pintura = "" precio = "" #metodos
```

Métodos

Los métodos son "funciones", solo que técnicamente se denominan métodos, y representan acciones propias que puede realizar el objeto (y no otro):

```
class Automovil: #propiedades pintura = "" precio = "" #metodos
def acelerar(self):
     #generar el algoritmo de acelerar
```

Crear un Objeto

Las clases por sí mismas, no son más que modelos que nos servirán para crear objetos en concreto. Podemos decir que una clase, es el razonamiento abstracto de un objeto, mientras que el objeto, es su materialización. A la acción de crear objetos, se la denomina "instanciar una clase" y dicha instancia, consiste en asignar la clase, como valor a una variable:

```
auto1 = Automovil()
print auto1.pintura        #muestra el valor que auto1 tiene en la propiedad pintura
auto1.precio = 500000      #setea el valor de 500000 a la propiedad precio de auto1

auto2  =  Automovil()
print   auto2.pintura      #muestra el valor que auto2 tiene en la propiedad pintura
auto2.precio = 80000       #setea el valor de 80000 a la propiedad precio de auto2
```

Accediendo a los métodos y propiedades de un objeto

Una vez creado un objeto, es decir, una vez hecha la instancia de clase, es posible acceder a su métodos y propiedades. Para ello, Python utiliza una sintaxis muy simple: el nombre del objeto, seguido de punto y la propiedad o método al cuál se desea acceder:

```
auto1 = Automovil() argumento_1 = auto1.precio
precio_mas_iva = argumento_1*1.21 print precio_mas_iva

#realiza lo que esté programado en el método acelerar del auto1 auto1.acelerar()
```

Paradigma de Objetos

La **Programación Orientada a Objetos** *POO* (**OOP** en inglés) se trata de un paradigma de programación introducido en la década del 70' del siglo pasado.

Este paradigma de programación nos permite organizar el código de una manera que se asemeja bastante a como pensamos y utilizamos **objetos** en la vida cotidiana. Pero nosotros vamos a utilizar o crear **clases,** como si fueran una fábrica de objetos.

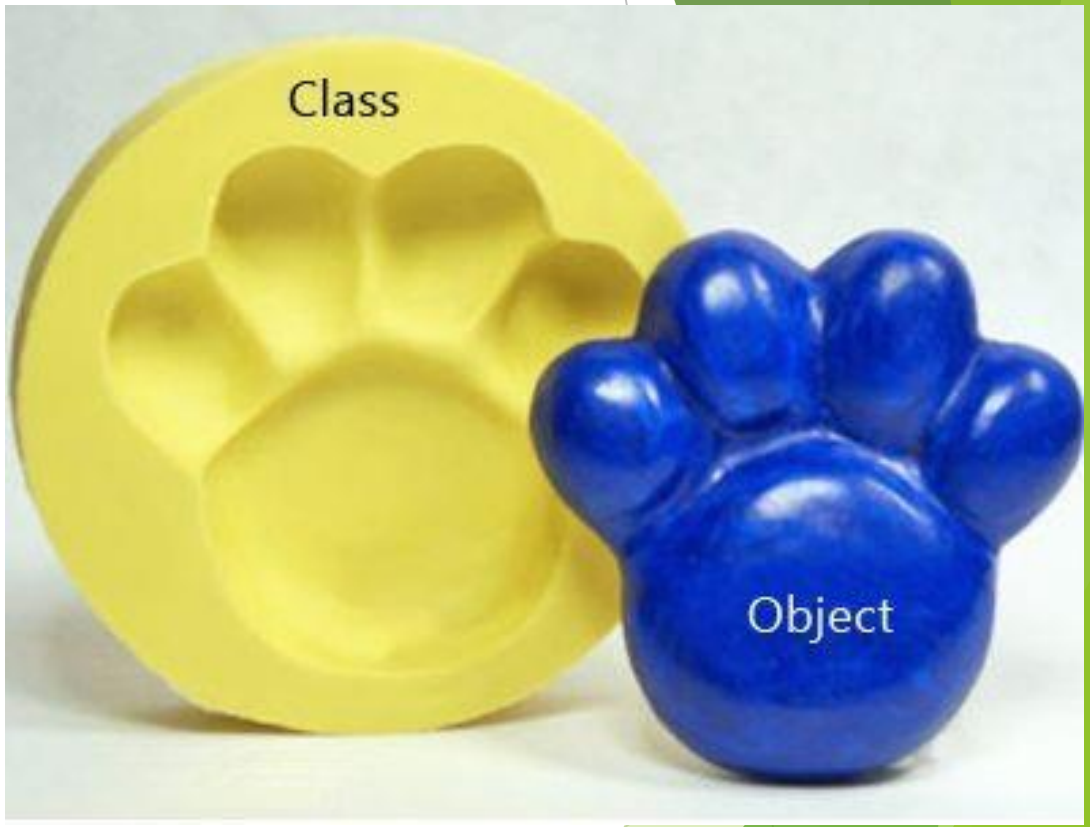
Las **clases** nos permiten agrupar un conjunto de variables y funciones, y que con las cuales podemos crear entidades que vivirán en nuestros programas.

En la programación orientada a objetos se definen estructuras de datos u **objetos**, cada uno con sus propios **atributos**. Cada objeto también puede contener sus propios **métodos**(funciones).

Paradigma de Objetos

Existen diversos lenguajes que trabajan con POO, pero los más populares son los lenguajes basados en clases en los que los *objetos son instancias de clases*.

Los principios fundamentales de la programación orientada a objetos son la encapsulación, la abstracción, la herencia y el polimorfismo.



Aspectos básicos de una clase

- Las clases se definen vía la palabra reservada `class` seguida de su nombre y, entre paréntesis, el nombre de las clases de las cuales debe heredar, en caso de haber.

```
class
MiClase:
pass
```

- La convención para nombrar una clase difiere de las funciones y otras sentencias; en las clases cada término se escribe con su primera letra en mayúscula y no se emplean guiones bajos.

Recordá que pass no ejecuta ninguna operación, sino que simplemente rellena un lugar que es requerido sintácticamente.

Ahora bien, todas las clases necesitan que se ejecute un codigo cada vez que se haga uso de ella y en donde el programador tendrá la posibilidad de inicializar lo que sea necesario.

```
class MiClase:
    def __init__(self):
        pass
```

A las funciones definidas dentro de una clase se las conoce con el nombre de *método*. Todas las funciones que empiezan y terminan con doble guión bajo son métodos especiales que utilizará Python y que no están diseñados para ser llamados manualmente (con algunas excepciones).

La función `__init__()` es un método que será invocado automáticamente por Python cada vez que se cree un objeto a partir de una clase. También denominado constructor en muchas bibliografías.

```
mi_objeto = MiClase()
```

Se dice que `mi_objeto` es una *instancia* de la clase `MiClase`. Para ilustrarlo mejor, prueba el siguiente código:

```
class MiClase:
    def___init__(self):
        print("Has creado una instancia de
MiClase.")

mi_objeto = MiClase()
```

Habrás observado que si bien el método `__init__()` requiere de un argumento, no hemos especificado ninguno al crear una instancia.

Esto se debe a que Python coloca el primer argumento de todos los métodos de una clase automáticamente (por convención se lo llama `self`). `self` es una referencia a la instancia actual de la clase.

Por ejemplo, consideremos el siguiente código, el cual crea dos instancias de `MiClase`.

```
mi_objeto = MiClase()
otro_objeto = MiClase()
```

En ambos casos, para crear la instancia Python automáticamente llama al método `__init__()`.

Sin embargo, en la primera línea `self` será una referencia a `mi_objeto`, mientras que en la segunda, a `otro_objeto`.

Por lo general estarás utilizando el método `__init__()` para inicializar objetos dentro de tu clase. Por ejemplo:

```
class MiClase:
    def __init__(self):
        self.a = 1

mi_objeto = MiClase()
print(mi_objeto.a)
```

Se dice que a es un atributo de `mi_objeto` o, en general, de `MiClase`. También podemos cambiar el valor de un atributo desde fuera de la clase:

```
mi_objeto = MiClase()
mi_objeto.a += 1
print(mi_objeto.a)  # Imprime 2
```

El método `__init__()` puede tener argumentos posicionales y por nombre como cualquier otra función convencional, que serán especificados al momento de crear una instancia de la clase.

```
class MiClase:
    def___init__(self, a):
        self.a = a

mi_objeto = MiClase(5)
print(mi_objeto.a)  # Imprime 5
```

Una clase también puede incluir funciones, aunque recuerda, siempre el primer argumento debe ser `self`.

```
class MiClase:
    def___init__(self, a):
        self.a = a

    def imprimir_a(self):
        print(self.a)

mi_objeto = MiClase(5)
mi_objeto.imprimir_a()
```

Otros ejemplos:

```
class MiClase:
    def___init__(self, a):
        self.a = a

    def imprimir_a(self):
        print(self.a)

    def sumar_e_imprimir(self, n):
        self.a += n
        print(self.a)

mi_objeto = MiClase(5)
mi_objeto.sumar_e_imprimir(7)

  # Imprime 12
```

Ejercicio: Cumpleaños

Crea una clase “Persona”. Con atributos nombre y edad. Además, hay que crear un método “cumpleaños”, que aumente en 1 la edad de la persona cuando se invoque sobre un objeto creado con “Persona”.

Tendríamos que lograr ejecutar el siguiente código con la clase creada:

```
p = Persona("Juan", 21)
p.cumpleaños()
print(f"{p.nombre} cumple {p.edad} años")
```

Encapsulamiento

El encapsulamiento 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.

Para la gente que conozca otros lenguajes, le resultará un término muy familiar, pero en Python es algo distinto.

Python por defecto no oculta los atributos y métodos de una clase al exterior.

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

mi_clase = Clase("Que tal")
print(mi_clase.atributo_clase) # 'Hola'
print(mi_clase.atributo_instancia) # 'Que  tal'
```

Ambos atributos son accesibles desde el exterior. Sin embargo, esto es algo que tal vez no queremos que suceda con ciertos métodos o atributos.

La encapsulación consiste en denegar el acceso a los atributos y métodos internos de la clase desde el exterior. En Python no se aconseja para no complejizar el código. Pero se puede simular.

Si que queremos que los atributos y métodos solo pertenezcan a la clase, y solamente puedan ser accedidos por la mecánica interna del objeto, podemos usar la doble __(doble guion bajo) para nombrarlos. Esto hará que Python los interprete como “privados”, de manera que no podrán ser accedidos desde el exterior.

```
class Clase:
    atributo_clase = "Hola"   # Accesible desde el exterior
    __atributo_clase = "Hola" # No accesible

    # No accesible desde el exterior
    def___mi_metodo(self):
        print("Haz algo")
        self.__variable = 0

    # Accesible desde el exterior
    def metodo_normal(self):
        # El método si es accesible desde el interior
        self.__mi_metodo()

objeto = Clase()
print(objeto.atributo_clase)     # Ok!
objeto.metodo_normal()    # Ok!
#print(objeto.__atributo_clase)  # Error! Atributo no accesible
#objeto.__mi_metodo()     # Error! Método no accesible
```

Acceder a atributos privados (GET-SET-DEL)

Muchas veces, el público que decide aprender Python viene con un bagaje de conocimiento de otro lenguaje. Y piensan como si Python funcionase igual que: C, C++, Java, etc. Y eso en algunos aspectos es válido y en otros no tanto. Python es Python. Java es Java.

En Python cuando queremos acceder a atributos privados podemos fabricarnos los métodos *get, set* y *del.*

Pero hay otra alternativa al uso de estos métodos basada en las propiedades Python que simplifica la tarea. Las propiedades en Python son un tipo especial de atributo a los que se accede a través de llamadas a métodos. Con ello, es posible ocultar los métodos *"get", "set"* y *"del"* de manera que sólo es posible acceder mediante estas propiedades por ser públicas.

No es obligatorio definir métodos *"get"*, *"set"* y *"del"* para todas las propiedades. Es recomendable sólo para aquellos atributos en los que sea necesario algún tipo de validación anterior a establecer, obtener o borrar un valor. Para que una propiedad sea sólo de lectura hay que omitir las llamadas a los métodos "set" y "del".

```
class Empleado:
    def___init_(self, nombre, salario):
        self.__nombre = nombre
        self.__salario = salario

    def getnombre(self):
        return self.__nombre

    def getsalario(self):
        return self.__salario

    def setnombre(self, nombre):
        self.__nombre = nombre

    def setsalario(self, salario):
        self.__salario = salario

    def delnombre(self):
        del self.__nombre

    def delsalario(self):
        del self.__salario
```

Vemos que nombre y salario son valores que se pasan al momento de generar la instancia. Pero luego esos valores son “privados”. ¿Entonces?

Si se quieren manipular, vamos a tener que armarnos métodos para poder manipularlos ya que internamente vamos a poder seguir trabajando con ellos.

En este caso los métodos *“get”* nos permiten obtener los valores, los *“set”* grabar valores y los *“del”* borrar.

¿Pero esto es lo que se busca en Python? No. Básicamente la filosofía del lenguaje es: “Simple es mejor que complejo”, no te compliques la vida.

La clase empleado sin tanta parafernalia:

```
class Empleado:
    def___init__(self, nombre, salario):
        self.nombre = nombre
        self.salario = salario
```

No sólo es más corta, sino que su funcionalidad es completamente equivalente, es más fácil de leer porque es obvia (se puede leer de un vistazo), y hasta es más eficiente.

Lo único diferente es que en esta última versión los atributos son públicos y modificables sin ningún método especial. Como vimos en la primera parte teórica del módulo.

Herencia

Cuando una clase B hereda de una clase A, aquélla conserva todos los métodos y atributos de ésta.

```
class ClaseA:
    def __init__(self):
        self.a = 1

class ClaseB(ClaseA):
    pass

mi_objeto = ClaseB()
print(mi_objeto.a)
```

En este código, `mi_objeto` es una instancia de `ClaseB`, que no ha definido ningún atributo `a`. Sin embargo, lo hereda de la `ClaseA`. La `ClaseB` podría definir nuevos atributos y funciones, e incluso reemplazar a los de su clase padre.

```
class ClaseB(ClaseA):
    def __init__(self):
        self.b = 2
```

En este caso, definimos el método de inicialización `__init__()` y creamos un nuevo atributo `b`. No obstante, esto reemplaza la antigua definición de `__init__()` en `ClaseA` de modo que, por ejemplo, `self.a` no se habrá definido. Siempre que reemplazamos un método, para permitir que las funciones con el mismo nombre en las clases padres se ejecuten debemos emplear la función `super()` del siguiente modo.

```
class ClaseB(ClaseA):
    def __init__(self):
        super().__init__()
        self.b = 2
```

Como alternativa también es posible llamar a la función _int_() de la clase padre directamente

```
class ClaseB(ClaseA):
    def __init__(self):
        ClaseA.__init__(self)
        self.b = 2
```

Ejercicio: Estudiante persona

Crear una clase “Persona” que sea la clase padre de otra clase “Estudiante”.

La clase “Persona” su método__init__() debe de estar preparado para recibir nombre y apellido. Además, esta clase , debe tener un método para mostrar `nombre_completo()` el cual debe mostrar el nombre acompañado del apellido.

La otra clase “Estudiante”, debe de poder heredar de “Persona”, y además recibir los argumentos nombre y apellido. También la clase “Estudiante”, recibe el valor “carrera”, y además contar con un método `mostrar_carrera()`. Las dos clases son obligatorias.

Ayuda:

```
class Persona():
    def __init__(self,nom,ape):
        #Completar código


class Estudiante(Persona):
    def __init__(self,nom,ape,carr):
        super().__init__(nom,ape)
        #Completar código
```

Y luego al usar “Estudiante”:

```
>>> x = Estudiante("Juan","Salvo","Ing Química")
>>> x.nombre_completo()
>>>"Juan Salvo"
>>> x.mostrar_carrera()
>>> "Ing Química"
```