Base de Datos

Programación Orientada a Objetos

Subsecretaría de Empleo
Chaco Gobierno de todos

Ministerio de Producción, Industria y Empleo
Chaco Gobierno de todos

CHACO
Gobierno de todos

INFORMATORIO

INTRODUCCIÓN

POO - OOP -

Programación Orientada a Objetos -

Object Oriented Programming

La Programación Orientada a Objetos (POO) es un paradigma de programación en el que se pueden pensar en problemas complejos como objetos. Desde el principio lo dijimos y hoy te lo vamos a explicar.

En Python, **todo es un objeto**, lo que significa que cada elemento de un programa de Python es un objeto, desde números y strings hasta funciones y módulos. Los objetos tienen atributos (características) y métodos (acciones) a los que se pueden acceder y utilizar. La POO en Python se basa en la creación de **clases**, que son plantillas para crear **objetos**. Las clases definen los **atributos** y **métodos** que tendrán los objetos creados a partir de ellas.

Este **paradigma de programación** se utiliza para organizar y estructurar el código de manera más eficiente. Python es un lenguaje de programación que admite la POO de forma nativa y proporciona todas las herramientas necesarias para implementarla.

En POO, los objetos son entidades que combinan datos y funciones relacionadas en una sola entidad. Estos **objetos** son **instancias de clases**, que son **plantillas o moldes para crear objetos**. Una clase define las características y el comportamiento que tendrán los objetos que se creen a partir de ella.

INFORMATORIO

Programación Orientada a Objetos VS Programación Estructurada

La Programación Orientada a Objetos se centra en la organización del código alrededor de objetos que encapsulan datos y comportamiento relacionado, promueve la reutilización de código a través de la herencia y el polimorfismo, y se basa en el concepto de abstracción para modelar sistemas complejos.

Por otro lado, la Programación Estructurada se basa en la organización del código en procedimientos o funciones, utiliza la división en módulos para la reutilización del código y se enfoca en la secuencia de instrucciones para resolver un problema.

Ambos paradigmas tienen sus ventajas y se utilizan en diferentes contextos según las necesidades del proyecto y las preferencias del desarrollador.

Procedimiento	Programación Estructurada	Programación Orientada a Objetos
Organización del código	En la programación estructurada, el código se organiza en procedimientos o funciones, que son bloques de código que realizan una tarea específica. Estos procedimientos se llaman secuencialmente y se pueden dividir en módulos para facilitar la reutilización del código.	En la POO, el código se organiza en objetos que combinan datos y funciones relacionadas. Los objetos son instancias de clases y se comunican entre sí mediante mensajes.
Abstracción	En la programación estructurada, la abstracción se logra utilizando funciones y procedimientos para dividir el código en partes más pequeñas y manejables.	La POO se basa en el concepto de abstracción, que permite representar entidades del mundo real en forma de objetos. Los objetos encapsulan datos y comportamiento en una sola entidad, lo que facilita la modelización de sistemas complejos.
Encapsulación y ocultamiento de datos	En la programación estructurada, los datos y las funciones están separados y se puede acceder directamente a los datos desde cualquier parte del programa.	En la POO, los objetos encapsulan datos y comportamiento relacionado en una sola entidad. Esto significa que los datos y las funciones que operan sobre esos datos están agrupados dentro del objeto y son accesibles a través de interfaces públicas. Esto permite el ocultamiento de datos, lo que significa que los detalles internos del objeto no son accesibles desde el exterior.
Herencia y polimorfismo	La programación estructurada no tiene conceptos directos de herencia y polimorfismo.	Estos son conceptos clave en la POO que permiten la reutilización de código y la creación de relaciones entre clases. La herencia permite que una clase herede atributos y métodos de una clase base, lo que promueve la reutilización de código y la especialización de clases. El polimorfismo permite que objetos de diferentes clases respondan de manera diferente a la misma llamada de método, lo que facilita el diseño y la extensibilidad del código.

Ejemplos

```python
# Programación Estructurada

# Función para calcular el área de un círculo
def calcular_area_circulo(radio):
    area = 3.14 * radio * radio
    return area

# Función para imprimir el resultado
def imprimir_resultado(area):
    print("El área del círculo es:", area)

# Llamada a las funciones
radio = 5
area_circulo = calcular_area_circulo(radio)
imprimir_resultado(area_circulo)
```

```python
# Programación orientada a objetos

# Clase Circulo
class Circulo:
    def __init__(self, radio):
        self.radio = radio

    def calcular_area(self):
        area = 3.14 * self.radio * self.radio
        return area

    def imprimir_resultado(self, area):
        print("El área del círculo es:", area)

# Creación de objeto y llamada a métodos
radio = 5
circulo = Circulo(radio)
area_circulo = circulo.calcular_area()
circulo.imprimir_resultado(area_circulo)
```

Como podés ver hay una gran diferencia de organización y estructura del código. La programación estructurada divide el código en funciones, mientras que la POO encapsula datos y comportamiento en objetos.
Pero estamos dándote varios conceptos nuevos así que antes de darte más código, te vamos a dar la definición de estos nuevos conceptos.

Conceptos y Definiciones

Abstracción: La abstracción es un concepto fundamental en la programación orientada a objetos (POO) que se refiere a la capacidad de representar entidades complejas y sus características esenciales de manera simplificada, centrándose en los aspectos relevantes y ocultando los detalles innecesarios.

En términos más simples, la abstracción permite crear modelos simplificados de objetos del mundo real en forma de **clases** y **objetos** en el código. Estos modelos **encapsulan** las propiedades y comportamientos esenciales de los objetos, pero omiten los detalles específicos que no son relevantes para el problema en cuestión.

La abstracción se logra a través de la creación de clases, donde se definen **atributos** y **métodos** que representan las características y acciones de los objetos. Los atributos representan las propiedades o

datos asociados a un objeto, mientras que los métodos representan las operaciones o comportamientos que el objeto puede realizar.

Un ejemplo sencillo de abstracción sería una clase "Coche". Podemos definir los atributos esenciales de un coche, como la marca, el modelo y el año de fabricación, así como los métodos relevantes, como acelerar, frenar y girar. Estos detalles específicos son relevantes para el concepto de un coche, pero otros detalles, como la complejidad del motor o el funcionamiento interno, pueden ser abstractos y no se necesitan en el contexto de uso de la clase.

La abstracción permite simplificar la complejidad de un sistema al proporcionar una representación clara y concisa de los objetos y sus interacciones. Al enfocarse en lo esencial, se facilita el diseño, la comprensión y el mantenimiento del código. Además, la abstracción permite la **reutilización de código** a través de la creación de clases genéricas que pueden ser aplicadas a diferentes situaciones.

Clase: Una clase es una **plantilla o molde** que define la estructura y el comportamiento de los **objetos**. Representa un concepto abstracto y contiene atributos y métodos que describen las **características y acciones que los objetos** que esa **clase** pueden tener.

Objeto: Un objeto es una **instancia de una clase**. Una instancia representa una **entidad específica y concreta** dentro de un programa. Esta entidad concreta que se crea a partir de una clase específica. Tiene atributos que almacenan datos y métodos que definen su comportamiento y acciones. Cada instancia tiene su propio conjunto de atributos y puede tener un estado único y diferente al de otras instancias de la misma clase. Como sinónimo podemos decir “Vamos a instanciar un objeto” que es lo mismo que decir “vamos a crear un objeto”.

Por ejemplo, consideremos una clase "Persona". Podemos crear múltiples instancias de esta clase para representar personas individuales con diferentes nombres, edades, etc. Cada instancia tendrá su propio conjunto de atributos y métodos, y se pueden acceder y manipular de forma independiente.

Veámoslo de forma gráfica para comprenderlo mejor:

INFORMATORIO

Variables de Instancia

Persona
Nombre
Apellido
Edad
Hablar()
Caminar()
Correr()

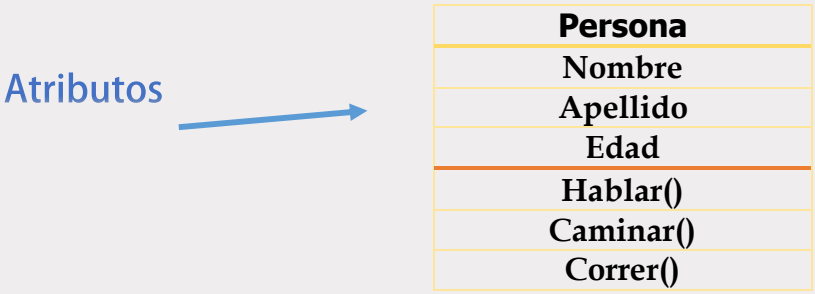
CLASE

Objetos

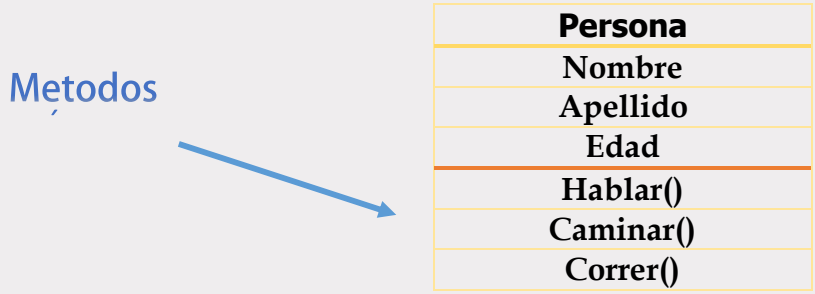
Cristian	Diana	Pedro	Wanda	Antonio	Carola
War	Prince	Parker	Maximoff	Stark	Danvers
29	36	21	26	45	30

Instancias

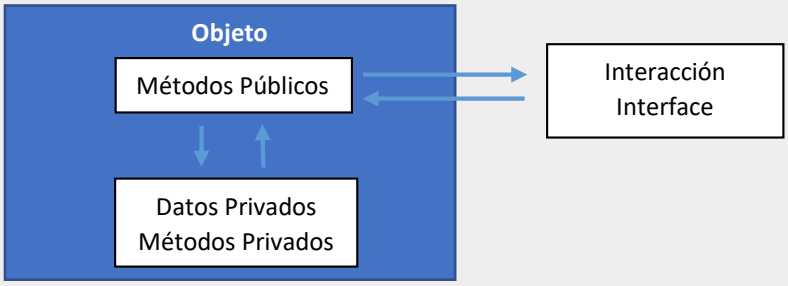
Atributo: Un **atributo** es una **variable** asociada a una **clase** u **objeto** que almacena datos. Los atributos **representan** las **características** o **propiedades** de los **objetos**. Pueden ser variables de cualquier tipo de datos, como enteros, cadenas, listas, etc.



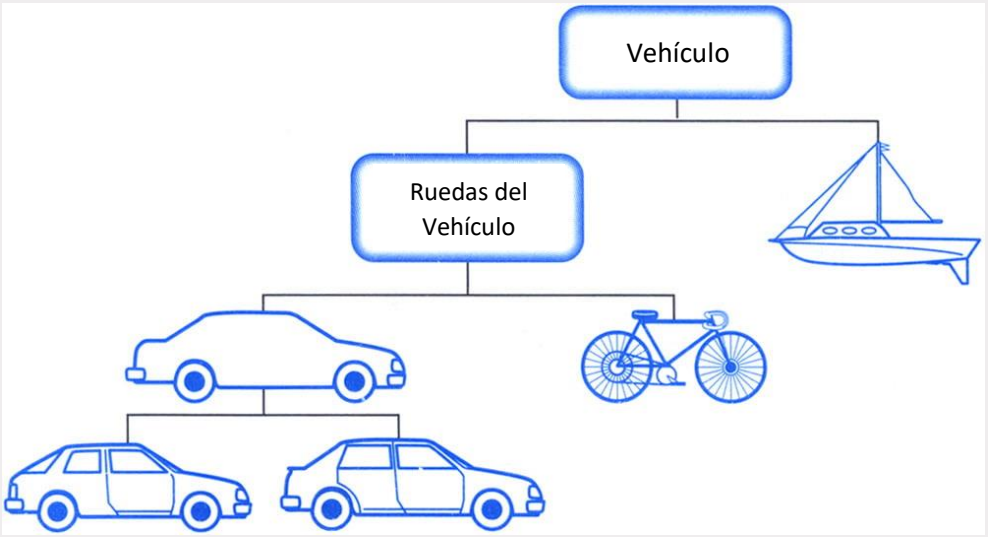
Método: Un método es una **función definida** en una clase que define el **comportamiento** de los **objetos** de esa clase. Representa las **acciones** que un objeto puede realizar o los **cálculos** que puede realizar. Los métodos tienen acceso a los atributos del objeto y pueden modificar su estado interno.



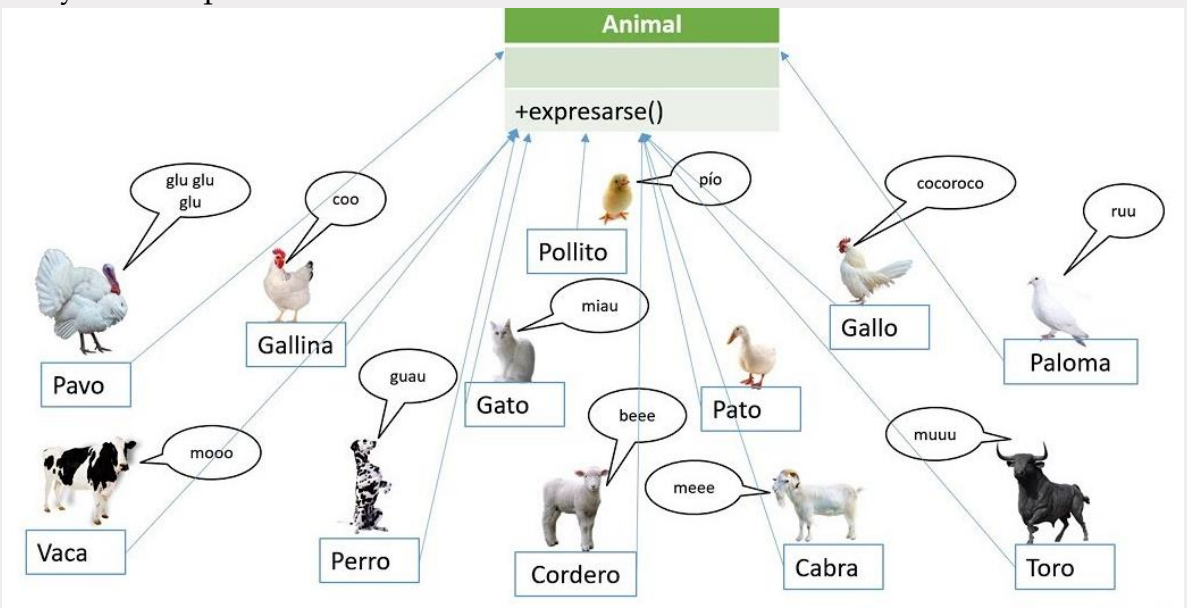
Encapsulación: La encapsulación es un principio de la POO que consiste en ocultar los detalles internos de un objeto y proporcionar una interfaz pública para interactuar con él. Los atributos y métodos internos de un objeto se mantienen privados y solo se acceden a través de métodos públicos, lo que garantiza la integridad y el control sobre el objeto.



Herencia: La herencia es un mecanismo en la POO que permite que una **clase herede atributos y métodos de otra clase**. La clase que hereda se llama **clase derivada o subclase**, y la **clase de la cual se hereda se llama clase base o superclase**. La herencia permite reutilizar y extender la funcionalidad de las clases existentes, promoviendo la modularidad y la jerarquía.



Polimorfismo: El polimorfismo es la capacidad de un objeto para **responder de diferentes formas según el contexto**. Permite que objetos de **diferentes clases** se comporten de **manera similar** o respondan de **manera diferente** a la misma llamada de método. El polimorfismo promueve la flexibilidad y la interoperabilidad en el diseño de sistemas.



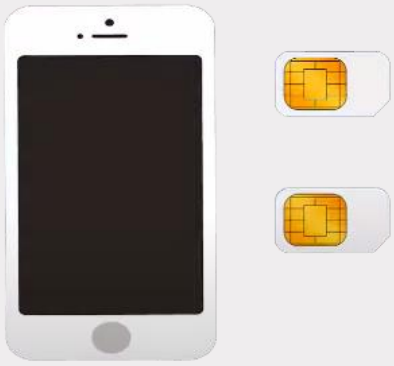
Constructor: Un constructor es un **método especial de una clase** que se llama automáticamente al **crear una instancia** (objeto) de esa clase. Se utiliza para **inicializar** los **atributos** de la clase y configurar su estado inicial. En muchos lenguajes de programación, incluido Python, el constructor se llama __init__().

```
class Circulo:
    def __init__(self, radio):
```

Interfaz: Una interfaz es una colección de métodos abstractos que definen un conjunto de acciones que una clase debe implementar. Una interfaz proporciona un contrato o especificación para las clases que la implementan, asegurando que tengan ciertos métodos y comportamientos comunes.
Dicho de otra manera, un interfaz define como se comporta un objeto y lo que se puede hacer con él.

Por ejemplo, pensemos en un control remoto para la tv. Todos los controles nos ofrecen la misma interfaz, mismas funcionalidades o métodos. Es decir, estos controles siempre deben tener <<subir volumen>>, <<bajar volumen>>, <<siguiente canal>>, <<anterior canal>>.

Agregación: La agregación es una relación entre objetos donde un objeto contiene o tiene una referencia a otros objetos. Los objetos agregados pueden existir de forma independiente y pueden estar asociados con varios objetos a la vez. La agregación se basa en la composición de objetos para formar relaciones más complejas. Para que lo entiendas mejor, supongamos un celular, el cual, puede funcionar por sí solo, sin embargo, para que uno pueda realizar llamadas, mandar mensajes, etc, necesita un chip… en este caso, el celular, puede llevar 1 o 2 chips. ¿Ahora, qué pasa si el celular se destruye? Supongamos que el chip no estaba puesto o que al celular lo piso un auto, destruyendo prácticamente todo, pero lo más probable es que al chip lo podamos sacar del celular roto y usarlo inmediatamente con otro celular. Vamos a ir con el siguiente concepto y si no lo comprendiste, lo vas a comprender mejor.



Composición: En la composición, un objeto se compone de otros objetos más pequeños que son esenciales para su funcionamiento. Los objetos más pequeños se conocen como componentes o partes, se crean como instancias de otras clases, y el ciclo de vida de los objetos más pequeños está totalmente controlado por el objeto contenedor. Esto significa que los objetos más pequeños existen solo mientras el objeto contenedor exista y se destruyen cuando el objeto contenedor se destruye.
Sigamos con el ejemplo del celular, pero ahora vamos a mirarlo desde el lado de la batería, donde la batería (como en la mayoría de los celulares nuevos), viene integrada, por lo que si miramos en el ejemplo anterior, si el celular se destruye, la batería también lo hará.
Y esta es la principal diferencia entre Agregación y Composición, donde en la Composición el objeto está relacionado directamente con el contenedor, por lo que mientras que el contenedor exista, el objeto más pequeño existirá (lo que implica una relación más fuerte), sin embargo, en la Agregación si el contendor deja de existir, el objeto más pequeño de todas formas podrá existir (lo que implica una relación más débil).

Asociación: Es una relación débil donde una clase se asocia con otra clase, pero no hay una dependencia fuerte entre ellas. Puede haber una asociación unidireccional o bidireccional, y una clase puede tener una referencia a otra clase como atributo. Por ejemplo, una clase "Persona" puede estar asociada con una clase "Dirección" a través de un atributo de tipo "Dirección".

Dependencia: Es una relación donde un objeto de una clase depende de otro objeto de otra clase para realizar una acción. La dependencia puede ser temporal y no implica una relación de pertenencia o composición. Por ejemplo, una clase "Pedido" puede tener una dependencia con una clase "Cliente" para obtener información del cliente.

Ahora que ya vimos algunos conceptos (no significa que no vamos a ver más conceptos más adelante) es momento que vayamos al código, donde vamos a comenzar a crear clases.

Creando una clase

Como adelantamos en páginas anteriores (en el código que te mostramos para comparar POO con PE), para definir una clase vamos a utilizar la palabra reservada class, seguida del nombre que va a tener nuestra clase.

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

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")
```

Como en el ejemplo, la estructura básica que vamos a usar será de esta forma; dentro del bloque de código para la **clase** "**Persona**" primero que nada, tenemos la palabra reservada **def** y luego el método especial **__init__** (doble guión bajo, init, doble guión bajo) que se llama **constructor** y se utiliza para **inicializar** los atributos de la clase.
Después vienen los parámetros que tendrá esta clase, donde el primer **parámetro** "**self**" se refiere a la instancia actual de la clase (dentro del paréntesis y separados por coma).
En general, "**self**" (algo así como "uno mismo"), siempre va a estar como primer parámetro.
Luego declaramos los parámetros que vamos a utilizar, para este caso, "nombre" y "edad" (cerramos paréntesis y colocamos dos puntos. No olvides la indentación siempre).
En las siguientes líneas de código, asignaremos, este caso, a los atributos de nuestra clase los parámetros declarados. No necesariamente debemos asignar los parámetros declarados, podemos asignarles a nuestros atributos valores fijos, sin embargo, haciéndolo con valores fijos, cuando llamemos a métodos de nuestra clase, esos valores no serán variables, no van a cambiar a menos que lo hagamos de forma manual.
La forma de asignar los valores a nuestros atributos, será como la vemos en el ejemplo: la palabra reservada "self", un punto y el nombre del atributo, luego la asignación del valor, que, en este caso, son los parámetros.

El **método** "saludar" es una **función** que se define **dentro de la clase** y tiene acceso a los atributos de la clase utilizando la notación "**self.nombre**" y "**self.edad**" (es decir "self.atributo").

Para el método se sigue la misma forma, donde primero usamos la palabra reservada **def**, seguida del **nombre del método** y entre paréntesis nuestra palabra "**self**".
Dentro del bloque de código del método, podemos trabajarlo tal cual lo veníamos haciendo con funciones.

```
class nombre_de_la_clase:
    def __init__(self, parametro1, parametro2, parametroN):
        self.atributo1 = parametro1
        self.atributo2 = parametro2
        self.atributoN = parametroN

    def nombre_del_metodo(self):
            bloque de codigo
```

Como te contábamos antes, esta sería, en general, la forma básica para nuestra clase.

Si seguimos con el ejemplo de la **clase Persona** (que creamos anteriormente), podríamos **llamar** al **método** en una **instancia** de la **clase** para que imprima un saludo con el nombre y la edad de la persona.

```
 9   persona1 = Persona("Wanda", 26)
10   persona1.saludar()

PROBLEMAS    SALIDA    CONSOLA DE DEPURACIÓN

Hola, mi nombre es Wanda y tengo 26 años.
```

Como podés ver, **creamos una instancia** de la clase Persona llamada "persona1" con el nombre "Wanda" y la edad "26" (colocando estos argumentos en el orden correspondiente en el que se encontraban los parámetros).
Luego, llamamos al método "saludar" en esa instancia, lo que imprime el saludo correspondiente.

Sin embargo, como te contamos antes, a nuestros atributos, no necesariamente, se tiene que declararlos con parámetros, se puede darle valores directamente:

```
14  class nombre_de_la_clase:
15      def __init__(self):
16          self.atributo1 = 'Wanda'
17          self.atributo2 = 26
18          self.atributoN = 'Valor fijo'
19
20      def nombre_del_metodo(self):
21              print(f"Hola, mi nombre es {self.atributo1} y tengo {self.atributo2} años.")
22
23
24  variable1 = nombre_de_la_clase()
25  variable1.nombre_del_metodo()

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL        Python

Hola, mi nombre es Wanda y tengo 26 años.
```

Pero como podés ver, cuando realizamos una instancia tampoco podemos pasarle valores variables, ya que los valores fueron declarados dentro de la clase, por lo que, a menos que queramos que el valor de un atributo siempre sea fijo, la forma de declarar los valores de los atributos, es como parámetros, como te mostramos en el primer ejemplo de la clase Persona, y luego, pasarle argumentos con valores variables cuando realizamos la instancia.

Otro punto a tocar es cuando instanciamos. Podés ver que, para crear un objeto de clase, y volvamos al ejemplo, de la clase Persona, tenemos que hacerlo declarando un variable y asignándole la clase que queremos (en este caso, la clase Persona). Esto lo podemos comprobar con una función de Python que ya la vimos varias veces <<type>>

```
1   class Persona:
2       def __init__(self, nombre, edad):
3           self.nombre = nombre
4           self.edad = edad
5
6       def saludar(self):
7           print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")
8
9   persona1 = Persona("Wanda", 26)
10  persona1.saludar()
11  print(type(persona1))
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

<class '__main__.Persona'>
```

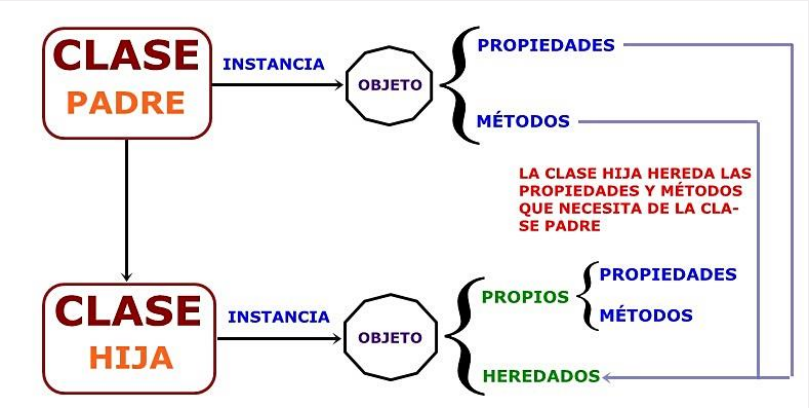
Seguimos con el ejemplo de la clase Persona, pero como ves, en la línea 11 agregamos *type* para ver el tipo de objeto que es *persona1*, y la salida nos muestra **<class '__main__.Persona'>**

Así, cuando usábamos type para verificar si una variable o estructura era **int** o **string** o **list**, etc, por ejemplo, en este caso estamos verificando que nuestra variable es de tipo **Persona**, por lo que ahora ya seguramente podés comprender mucho más que en Python "todo es un objeto" (frase que la escuchaste al principio del cursado).
Además, aparece __main__, que nos indica que esta clase (Persona en este caso) se definió dentro del módulo principal *main* o pertenece al módulo principal.

Ahora ya sabés crear una clase, inicializar sus atributos y definirlos. Además, ya aprendiste a crear métodos que se podrán usar en esa clase. Los métodos pueden ser la cantidad que necesitemos, no hay un límite, la cantidad va a estar basada en lo que vamos a requerir para nuestra clase. Y por eso ahora vamos a hablar de **herencia**.

Usando herencia

La herencia sirve para heredar atributos y métodos de una clase Padre. Al referirnos a una clase padre, nos referimos a una clase mayor, que está "un escalón más arriba".



Por lo que si definimos una nueva clase en base a una clase Padre, como te decíamos, vamos a estar heredando sus atributos y métodos. Vamos a plasmarlo en código, siguiendo con el ejemplo de Persona.

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

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")

class Estudiante(Persona):
    def __init__(self, nombre, edad, grado):
        super().__init__(nombre, edad)
        self.grado = grado

    def estudiar(self):
        print(f"{self.nombre} está estudiando en el grado {self.grado}.")

estudiante1 = Estudiante('Informatorio', 12, 'A')
estudiante1.estudiar()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Informatorio está estudiando en el grado A.
```

Se puede apreciar que la clase “Estudiante” hereda de la clase "Persona" utilizando el nombre de la clase Padre dentro del paréntesis.

```
class Estudiante(Persona):
```

Luego podemos ver el constructor __init__ en la clase "Estudiante" de la misma forma normal de declarar parámetros (que en este caso será: *nombre*, *edad*, *grado*)

```
def __init__(self, nombre, edad, grado):
```

Sin embargo, en la siguiente línea de código, podemos ver una nueva palabra reservada: **super()**

```
super().__init__(nombre, edad)
```

super(): Esta potente función integrada en Python proporciona una forma de acceder y usar métodos y atributos de clases principales, o padres o como también podemos llamarlas, superclases. Comúnmente la usamos para evitar redundancia de código y hacerlo más organizado y fácil de entender.

Al trabajar con herencia, esta función es muy útil para extender o modificar el comportamiento de un método de la superclase en la subclase (o clase hija) sin tener que reescribir completamente el método. Podemos llamar al método de la superclase y luego agregar o modificar el comportamiento, según el caso que estemos tratando.

Cómo pudiste observar, cuando usamos el constructor en la clase hija, pusimos los parámetros que íbamos a requerir para dicha clase, y cuando usamos la función super() y luego usamos el constructor pusimos los parámetros de la clase padre (sin llamar a “self”, ya que “self” recordá que hace referencia a la clase propia, que en este caso sería la clase hija).

Bien, ahora, supongamos que queremos modificar método “saludar()” de la clase padre:

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

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")

class Estudiante(Persona):
    def __init__(self, nombre, edad, grado):
        super().__init__(nombre, edad)
        self.grado = grado

    def estudiar(self):
        print(f"{self.nombre} está estudiando en el grado {self.grado}.")

    def saludar(self):
        print (f"Hola, soy {self.nombre} y soy un estudiante de {self.grado}.")

persona1 = Persona("Wanda", 26)
persona1.saludar()
estudiante1 = Estudiante('Informatorio', 12, 'A')
estudiante1.saludar()
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

Hola, mi nombre es Wanda y tengo 26 años.
Hola, soy Informatorio y soy un estudiante de A.
```

Te pasamos el esquema completo donde podemos observar bien como hemos modificado el método saludar de la clase padre, para que en la clase hija se comporte de otra manera (con otro mensaje de salida).

De esta forma podemos recrearlo tantas veces como necesitemos realizar herencias y modificar métodos.

Este apunte lo dividiremos en 2 partes, donde en la siguiente parte veremos más conceptos, donde lo primero que vamos a estar mostrando son las herencias múltiples.