> Object oriented programming

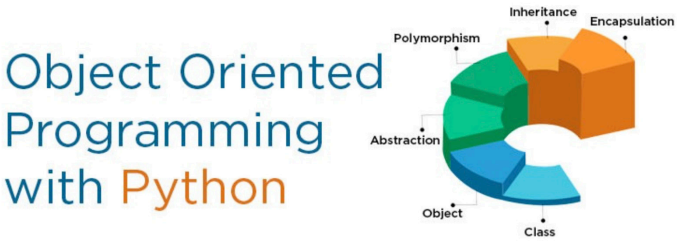
SEMANA 5

Programación orientada a objetos - poo

INTRODUCCIÓN

P.O.O.

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 la 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.

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.

Ejemplo:

```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 se puede, 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.

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.

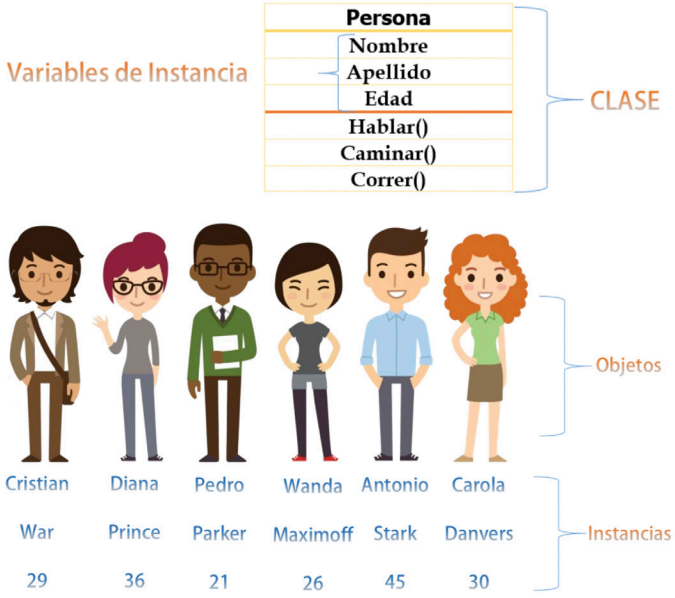
Objeto

Un objeto es una instancia de una clase. Una instancia representa una entidad específica y concreta dentro de un programa. Esta entidad concreta 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".

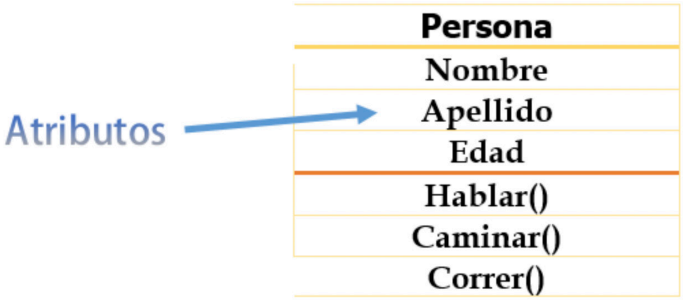
Por ejemplo, consideremos una clase "Persona". Podemos crear múltiples instancias de esta clase para representar personas individuales con diferentes nombres, edades, etc. Cada instancia tendrá su propio conjunto de atributos y métodos, y se pueden acceder y manipular de forma independiente.

Veámoslo de forma gráfica para comprenderlo mejor:



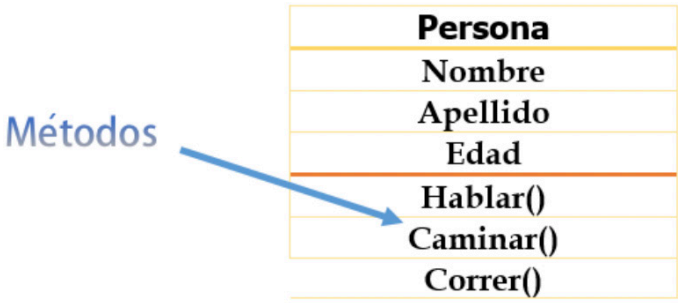
Atributo

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.



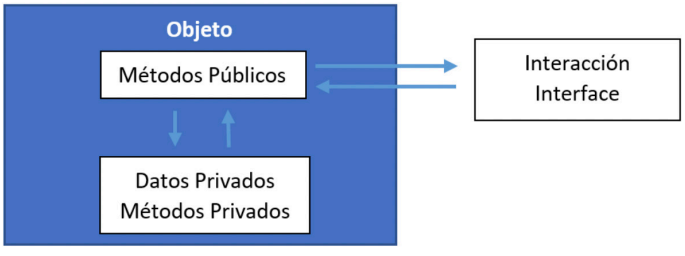
Método

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



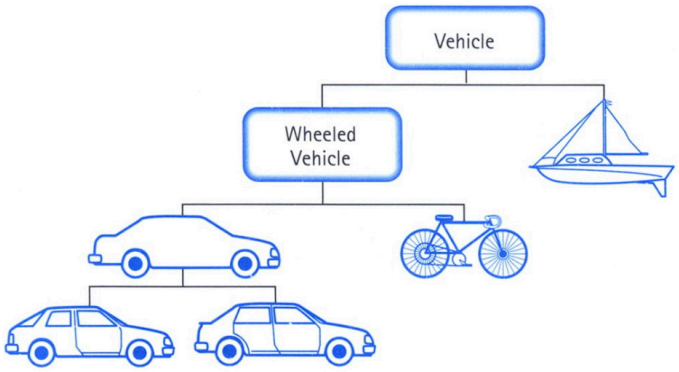
Encapsulación

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.



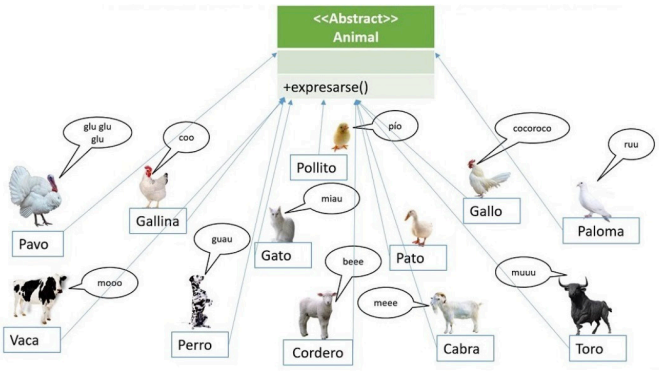
Herencia

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

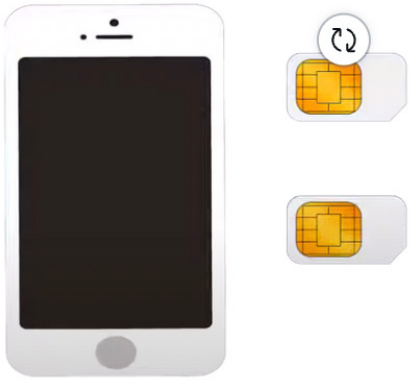
Una interfaz es una colección de métodos abstractos que definen un conjunto de acciones que una clase debe implementar. Una interfaz proporciona un contrato o especificación para las clases que la implementan, asegurando que tengan ciertos métodos y comportamientos comunes.
Dicho de otra manera, una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

Por ejemplo, pensemos en un control remoto para el televisor. 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 pisó 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 has comprendido, 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.
Seguiremos viendo otros conceptos en las siguientes páginas, pero ahora es momento de pasar al código, donde comenzaremos a crear clases.

Creando una clase

Como nos hemos adelantado 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.

```
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 a usaremos será de esta forma; dentro del bloque de código para la clase "Persona" en la primer línea, 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.

```python
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 muestre 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 puedes 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 hemos mencionado antes, a nuestros atributos, no necesariamente, se tiene que declararlos con parámetros, se puede darle valores directamente:

```python
class nombre_de_la_clase:
    def __init__(self):
        self.atributo1 = 'Wanda'
        self.atributo2 = 26
        self.atributoN = 'Valor fijo'

    def nombre_del_metodo(self):
            print(f"Hola, mi nombre es {self.atributo1} y tengo {self.atributo2} años.")


variable1 = nombre_de_la_clase()
variable1.nombre_del_metodo()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL   Python

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

Cuando instanciamos un objeto, para este caso, 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 hemos mostrado en el primer ejemplo de la clase Persona, y luego, pasarle argumentos con valores variables cuando realizamos la instancia.

Otro punto a mencionar es cuando instanciamos. Puedes ver que, para crear un objeto de clase, y volviendo 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 hemos visto varias veces <<type>>

```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.")

persona1 = Persona("Wanda", 26)
persona1.saludar()
print(type(persona1))
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

<class '__main__.Persona'>
```

Siguiendo con el ejemplo de la clase "Persona", puedes ver que, en la línea 11 agregamos "type" para ver el tipo de objeto que es persona1, y la salida nos muestra <class '__main__.Persona'>

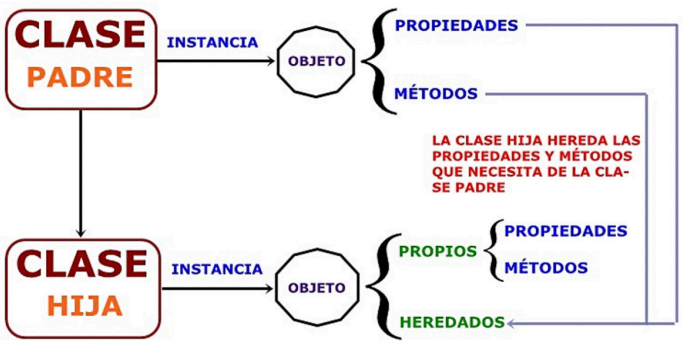
Así, cuando usábamos "type" para verificar si una variable o estructura era int o string o list, etc, por ejemplo, en este caso estamos verificando que nuestra variable es de tipo "Persona", por lo que ahora ya seguramente puedes comprender mucho más que, en Python, "todo es un objeto".

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 sabes crear una clase, inicializar sus atributos y definirlos. Además, has aprendido 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 hablaremos 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 hemos mencionado, estaremos 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.

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

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

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

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

```python
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.

Se puede observar que, cuando usamos el constructor en la clase hija, hemos puesto 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", recuerda 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:

```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}.")

    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.
```

Puedes ver el esquema completo donde podrás 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.

En las siguientes páginas seguiremos viendo más conceptos, donde el próximo concepto será el de herencia múltiple.

Usando herencia múltiple

En el esquema anterior pudimos observar varias cosas al realizar herencia.
Una de las cosas más importantes fue que pudimos modificar un método que heredamos de una clase padre a través de la función super(), y pudimos hacer que se comporte de otra manera en la clase hija.
Sin embargo, puede llegar un punto en el que necesitemos heredar más clases, ya que podemos requerir tener una nueva clase, pero con funcionalidades y características de otras clases y, para no repetir código, podemos ir heredando las veces que sean necesarias.

Un ejemplo de herencia múltiple sería el siguiente:

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

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

class Empleado():
    def __init__(self, nombre, puesto):
        self.nombre =nombre
        self.puesto = puesto

    def saludar(self):
        print(f"El puesto en el que me desempeño es el de {self.puesto}.")

class Profesor(Persona, Empleado):
    def __init__(self, nombre, puesto, antiguedad):
        Persona.__init__(self, nombre)
        Empleado.__init__(self, nombre, puesto)
        self.antiguedad = antiguedad

    def saludar(self):
        Persona.saludar(self)
        Empleado.saludar(self)
        print(f'Estoy en este puesto hace {self.antiguedad} años.')


profesor1 = Profesor('Informatorio', 'profesor', 11)
profesor1.saludar()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Hola, mi nombre es Informatorio.
El puesto en el que me desempeño es el de profesor.
Estoy en este puesto hace 11 años.
```

En este ejemplo, hemos aplicado la técnica de herencia múltiple en la creación de una clase (en este caso, la clase "Profesor"), que ha heredado de más de una clase padre. Al hacer esto, la clase hija se beneficia de los atributos y métodos de las clases padre que hemos especificado.

En este caso, contamos con tres clases: "Persona", "Empleado" y "Profesor". La clase "Profesor" hereda tanto de "Persona" como de "Empleado". Aunque la clase "Profesor" tiene su propio método "saludar", también estamos heredando el método "saludar" de las clases padre que hemos incluido.

```python
class Profesor(Persona, Empleado):
    def __init__(self, nombre, puesto, antiguedad):
        Persona.__init__(self, nombre)
        Empleado.__init__(self, nombre, puesto)
        self.antiguedad = antiguedad

    def saludar(self):
        Persona.saludar(self)
        Empleado.saludar(self)
        print(f'Estoy en este puesto hace {self.antiguedad} años.')
```

De esta manera, se obtiene la salida de los 3 saludos heredados. Pero, ¿qué sucede si no se incluyen estos métodos heredados en el método propio de la clase "Profesor"? Lo veremos a continuación:

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

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

class Empleado():
    def __init__(self, nombre, puesto):
        self.nombre =nombre
        self.puesto = puesto

    def saludar(self):
        print(f"El puesto en el que me desempeño es el de {self.puesto}.")

class Profesor(Persona, Empleado):
    def __init__(self, nombre, puesto, antiguedad):
        Persona.__init__(self, nombre)
        Empleado.__init__(self, nombre, puesto)
        self.antiguedad = antiguedad

    def saludar(self):
        print(f'Estoy en este puesto hace {self.antiguedad} años.')


profesor1 = Profesor('Informatorio', 'profesor', 11)
profesor1.saludar()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Estoy en este puesto hace 11 años.
```

Continuamos con el mismo esquema, pero esta vez eliminamos la herencia en la clase "Profesor" para el método "saludar()". Al realizar una herencia (simple o múltiple) y tener uno o más métodos con el mismo nombre, el método de la clase hija reemplaza a los métodos heredados con el mismo nombre. Por lo tanto, cuando se busca un atributo o método en la clase actual, se buscará primero en la misma clase "Profesor" y luego en las clases padres de las que hereda. Si no se especifica la inclusión de los métodos "saludar()" de las clases anteriores, como en el ejemplo anterior donde se incluyeron los métodos de las clases padres y, por lo tanto, se mostraron los tres métodos en pantalla, solo se mostrará el mensaje de la clase hija "Profesor" al llamar al método "saludar()". Los mensajes de las clases padres con el mismo nombre no se mostrarán. Hablaremos más adelante sobre el orden en que se buscan los métodos a través del MRO.

Entendemos que puede resultar confuso, pero continuaremos proporcionándote ejemplos para que puedas comprenderlo mejor. Cuando empieces a relacionarlo con situaciones cotidianas, tendrás una mejor comprensión del tema. Por lo tanto, aquí tienes un ejemplo infalible.

¡Por supuesto! Para que puedas entenderlo sin problema, te mostrarémos un ejemplo en el que codificaremos la estructura de una familia.

Vayamos al principio. Piensa en todas las cosas que heredaste de tus antepasados, y en la relación que tienes con ellos. Ahora que lo tienes en mente, vamos a traducir esto a código para que lo puedas entender más fácilmente.

Para empezar, crearemos una clase para el abuelo. No le asignaremos atributos todavía, solo trabajaremos con métodos para simplificar el proceso. Supongamos que "Abuelo" tiene la habilidad de construir casas. Lo representamos en el código de la siguiente manera:

```python
class Abuelo():
    pass

    def construir_casas(self):
        print('Puedo construir casas tradicionales de ladrillo y cemento.')

abuelo = Abuelo()
abuelo.construir_casas()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Puedo construir casas tradicionales de ladrillo y cemento.
```

Ahora bien, vamos a pensar en que "Abuela" tiene otra habilidad, la cual va a ser pintar casas. Lo pasamos a código:

```python
class Abuela():
    pass

    def pintar_casas(self):
        print('Puedo pintar casas con rodillo y pincel.')

abuela = Abuela()
abuela.pintar_casas()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Puedo pintar casas con rodillo y pincel.
```

Ahora tenemos dos clases "abuelos", ambos con habilidades únicas, sin embargo, antes de pasar a "hijos" podemos agregar una característica que se va a pasar de generación en generación:

```python
class Abuelo():
    pass

    def saludar(self):
        print('Cuando saludo, paso la mano.')

    def construir_casas(self):
        print('Puedo construir casas tradicionales de ladrillo y cemento.')


class Abuela():
    pass

    def saludar(self):
        print('Cuando saludo, doy un abrazo.')

    def pintar_casas(self):
        print('Puedo pintar casas con rodillo y pincel.')

abuelo = Abuelo()
abuela = Abuela()

abuelo.saludar()
abuela.saludar()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Cuando saludo, paso la mano.
Cuando saludo, doy un abrazo.
```

El saludo es algo que se va a transmitir de generación en generación, tal vez otras habilidades no, pero saludar, todos los aprendemos.

Ahora que tenemos unas "ciertas bases" pasemos a un hijo. Vamos a dividir la pantalla de nuestro editor para que puedas visualizar mejor todo el contexto.

```python
class Abuelo():
    pass

    def saludar(self):
        print('Cuando saludo, paso la mano.')

    def construir_casas(self):
        print('Puedo construir casas tradicionales de ladri

class Abuela():
    pass

    def saludar(self):
        print('Cuando saludo, doy un abrazo.')

    def pintar_casas(self):
        print('Puedo pintar casas con rodillo y pincel.')
```

```python
class Hijo(Abuela, Abuela):
    pass


hijo = Hijo()
hijo.pintar_casas()
hijo.construir_casas()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Puedo pintar casas con rodillo y pincel.
Puedo construir casas tradicionales de ladrillo y cemento.
```

¿Comienzas a entender mejor el comportamiento de la herencia múltiple? Observa cómo pudimos crear una clase hija con muy poco código, heredando de otras clases padre. Esta es una de las características más poderosas de la POO, ya que permite la reutilización de código y ahorro de líneas de código desde el principio.

No obstante, como recordarás, introdujimos el método saludar() en ambas clases padre, lo que nos lleva a preguntarnos: ¿qué ocurriría si utilizamos el método saludar() en la clase hija?

```python
class Abuelo():
    pass

    def saludar(self):
        print('Cuando saludo, paso la mano.')

    def construir_casas(self):
        print('Puedo construir casas tradicionales de ladri

class Abuela():
    pass

    def saludar(self):
        print('Cuando saludo, doy un abrazo.')

    def pintar_casas(self):
        print('Puedo pintar casas con rodillo y pincel.')
```

```python
class Hijo(Abuela, Abuela):
    pass


hijo = Hijo()
hijo.pintar_casas()
hijo.construir_casas()
hijo.saludar()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Puedo pintar casas con rodillo y pincel.
Puedo construir casas tradicionales de ladrillo y cemento.
Cuando saludo, paso la mano.
```

En efecto, el ejemplo del abuelo que saluda nos lleva a preguntarnos acerca del MRO mencionado previamente. Para continuar con el ejemplo de la familia, es importante abordar este tema antes.

MRO – Método de resolución de orden en Python Method Order Resolution

En Python, cada clase se deriva de la clase object, que es el tipo más básico en el lenguaje. Por lo tanto, todas las clases, ya sean predefinidas o definidas por el usuario, se consideran clases derivadas, y todos los objetos se clasifican como instancias de la clase object.

```
1  print(issubclass(list, object))
2  print(issubclass(tuple, object))
3  print(issubclass(dict, object))
4  print(issubclass(set, object))
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN

True
True
True
True
```

```
1  print(isinstance(3, object))
2  print(isinstance(3.14, object))
3  print(isinstance('Hola mundo', object))
4  print(isinstance(False, object))
```

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

True
True
True
True
```

Usando la función "issubclass" o "isinstance" podemos verificar que todo es un derivado de object, el tipo más básico de Python y la razón por la que al principio de la cursada, te hemos dicho que en Python todo es un objeto.

Es decir, la SuperClase de la que se hereda todo, es object.

Ahora bien, ¿qué hace el MRO en la herencia múltiple? Este método de resolución se refiere al algoritmo utilizado para determinar el orden en el que se resuelven los atributos y métodos heredados en una jerarquía de clases.
Cuando trabajamos con herencia múltiple, es necesario establecer un orden claro para resolver los atributos y métodos heredados cuando se producen un conflictos, como en el caso de "Hijo" y el método heredado "saludar()", es decir, cuando dos o más clases en la jerarquía tienen un atributo o método con el mismo nombre.
Python utiliza un algoritmo llamado "C3 Linearization" para determinar el orden de resolución. El algoritmo garantiza que el orden de resolución sea consistente y evita problemas con la ambigüedad en la herencia múltiple.

¿Cómo trabaja C3 Linearization?

1. Se crea una lista de todas las clases involucradas en la herencia múltiple, incluida la clase principal y todas las clases padre en el orden en que se definen en la declaración de la clase hija.

2. Se realiza una búsqueda en profundidad (depth-first search) en el árbol de herencia, comenzando por la clase principal, para determinar el orden de las clases.

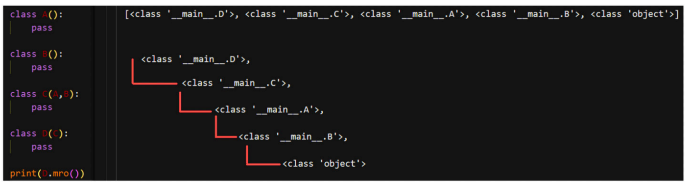
3. Durante la búsqueda en profundidad, se aplican las siguientes reglas para determinar el orden:de resolución, se utiliza ese orden para buscar y resolver los atributos y métodos heredados en caso de conflictos.

a) El orden de resolución debe preservar la propiedad de "linealización", lo que significa que el orden de las clases no debe introducir ciclos ni romper el orden de aparición.

b) Se da prioridad a las clases más específicas antes que a las clases más generales. Esto significa que las clases más cercanas en la jerarquía tienen prioridad sobre las clases más lejanas.

c) Si hay múltiples clases en el mismo nivel de la jerarquía, se respeta el orden en el que se definen en la declaración de la clase hija (de izquierda a derecha).

4. Una vez que se determina el orden

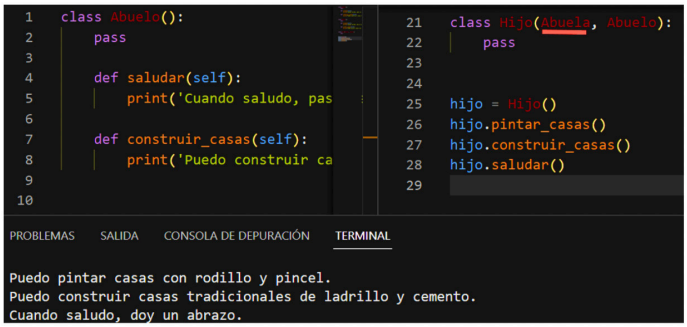


Como se puede apreciar al utilizar la función .**mro()**, la resolución del MRO nos permite entender la herencia que se ha estado implementando.

Así que, ahora que hemos visto este punto de, cómo se realiza el orden de resolución, podemos volver a nuestro ejemplo del "Hijo".

```
class Hijo(Abuelo, Abuela):
    pass


hijo = Hijo()
hijo.pintar_casas()
hijo.construir_casas()
hijo.saludar()
```

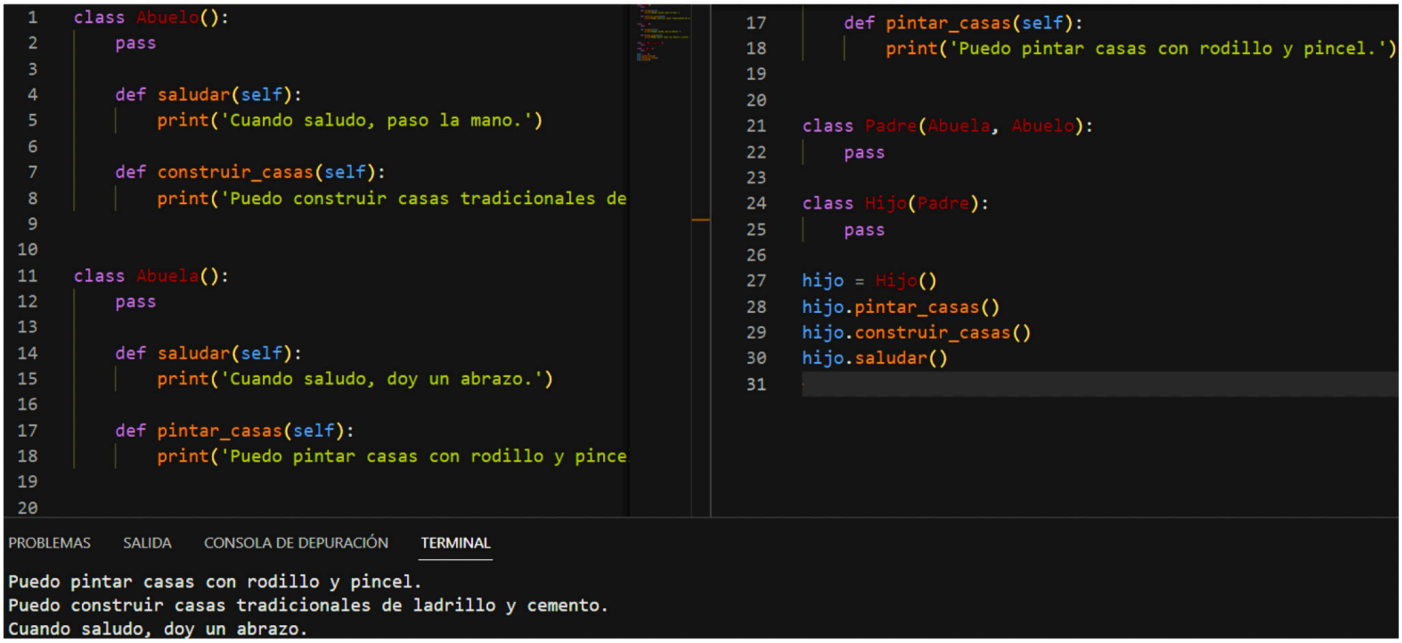
Análisis del orden de la herencia en la clase Hijo
Como se puede ver en el método "saludar()", la pantalla muestra que el saludo pertenece al abuelo. Si se observa el orden de nuestra herencia, se hereda primero la clase Abuelo y luego la clase Abuela. Es importante recordar que cada clase de "abuelos" tiene el método "saludar()", pero se comporta de manera diferente ya que el "saludo" de cada abuelo es distinto. Entonces, ¿qué sucede si cambiamos el orden de la herencia en la clase Hijo? Teniendo en cuenta esta información, exploramos esta posibilidad.



¡Exactamente como pensaste! Primero, el hijo hereda el método "saludar()" de la Abuela y luego del Abuelo. Este es el método que entra en conflicto, ya que ambas clases tienen el mismo nombre de método. En este caso, la MRO (Resolución de Orden de Métodos) solucionará que el método que se heredará primero será el de la clase que se encuentre a la izquierda en el código.

Ahora, ¿y qué pasa si heredamos en una clase hija algo que ya viene heredado de otra clase hija con una clase padre anterior? ¿Confuso? Veamos el código del ejemplo y los explicamos.

(Veremos el código en la siguiente página)



```python
class Abuelo():
    pass

    def saludar(self):
        print('Cuando saludo, paso la mano.')

    def construir_casas(self):
        print('Puedo construir casas tradicionales de

class Abuela():
    pass

    def saludar(self):
        print('Cuando saludo, doy un abrazo.')

    def pintar_casas(self):
        print('Puedo pintar casas con rodillo y pince

    def pintar_casas(self):
        print('Puedo pintar casas con rodillo y pincel.')


class Padre(Abuela, Abuelo):
    pass

class Hijo(Padre):
    pass

hijo = Hijo()
hijo.pintar_casas()
hijo.construir_casas()
hijo.saludar()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL
Puedo pintar casas con rodillo y pincel.
Puedo construir casas tradicionales de ladrillo y cemento.
Cuando saludo, doy un abrazo.
```

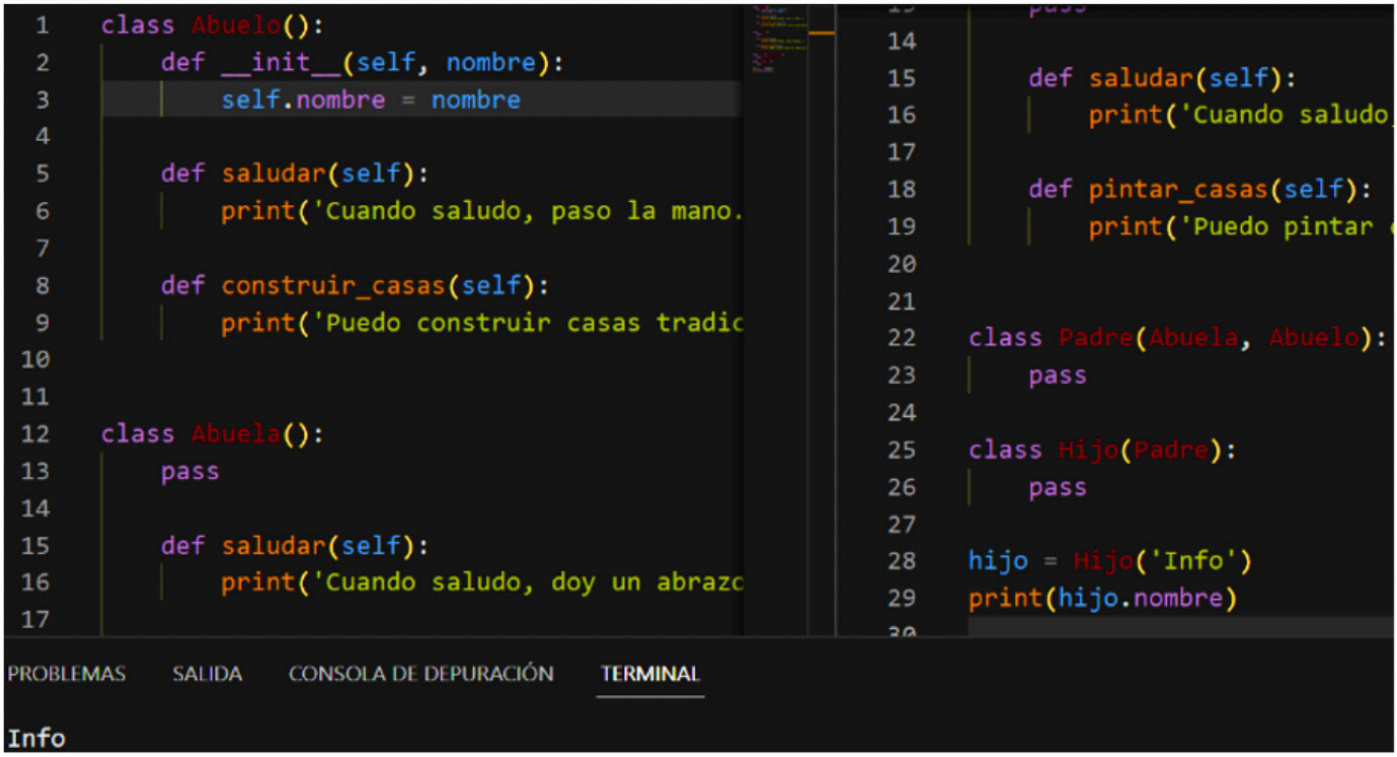
A la izquierda de la pantalla seguimos con la clase Abuelo y Abuela, a la derecha lo que hicimos fue agregar una clase más, "Padre", que es la que hereda (como en el ejemplo anterior) de Abuela y Abuelo, pero la clase Hijo hereda de Padre y por ende trae consigo las herencias anteriores de Abuela y Abuelo.

Como puedes observar, al instanciar un objeto de clase Hijo y llamar a los métodos heredados, llama a todos sin problema, pero el método saludar(), realiza la misma acción que si "Padre" llamara a saludar(), por el orden de herencia que se está recibiendo.
Así que estamos hablando de una herencia en cascada, donde cada atributo y método vendrá de herencias anteriores y los métodos con el mismo nombre (como en este caso), tendrán la acción de la última herencia, o herencia anterior (en este caso, llamará al saludo de la abuela).

MRO fue resolviendo a medida que se realizaron las herencias.
La flexibilidad que ofrece Python para herencias en POO es muy amplia, por lo que te recomendamos que practiques con situaciones de la vida cotidiana para comprender aún mejor.

Cada problema que tengamos que resolver, va a tener varias formas de resolverse, sin embargo, lo que hay que cuidar es la legibilidad y simpleza del código. Hay que tener en cuenta que el problema se lo debe resolver antes de volcarlo al código.

Otra cosa que también nos permite Python es agregar atributos dinámicamente. Sigamos con el ejemplo de la familia, y ahora agregaremos atributos iniciales:



Aquí hemos agregado un atributo en una de las primeras clases (Abuelo) y lo estamos accediendo desde la última clase creada (hijo), para que puedas seguir visualizando como se van heredando atributos y métodos desde clases anteriores.
Ahora vamos a agregar un atributo dinámicamente, es decir, fuera de las clases, al objeto instanciado.

Entonces, si queremos usar ese atributo agregado dinámicamente en otra instancia, nos indica un error donde directamente dice que la clase Hijo no tiene ese atributo, y, es justamente porque en la clase no está declarado este atributo, así como está declarado nombre en la clase "Abuelo", por lo que esto es especialmente útil cuando solo vamos a requerir usar un atributo de manera "local" en un solo objeto instanciado.

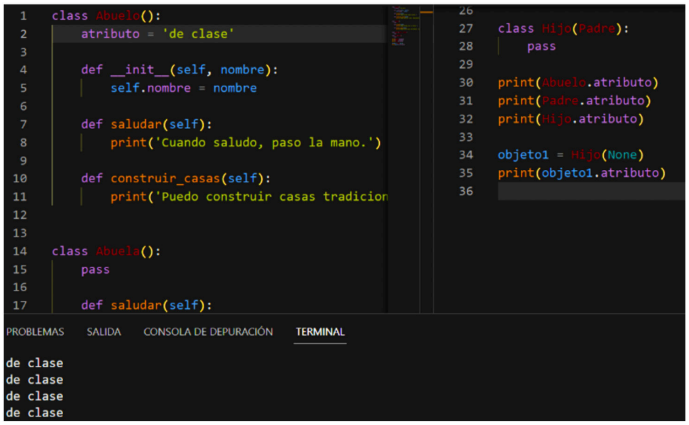
De esta forma hemos demostrado que por más herencia que haya desde clases anteriores, tenemos más recursos para poder usar sobre un objeto instanciado, sin necesidad que estén en los demás objetos de la misma u otra clase.

Entonces, siguiendo con los atributos, tenemos otros tipos de atributos aparte de los que estamos manejando hasta el momento. Recientemente vimos un atributo creado dinámicamente, el cual era accesible, solo desde el objeto instanciado.
Sin embargo, el atributo que venimos viendo mayormente, es el atributo de instancia, el cual puede acceder cualquier objeto instanciado de esa clase como, por ejemplo:

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

Al cual acceden todos los objetos instanciados de esa clase, y si es por herencia, también los heredan las subclases, o clases hijas.
Y tenemos otro atributo más, el cual es el atributo de clase, que no necesariamente, tiene que ser accedido solo desde un objeto como el atributo de instancia. Este tipo de atributo lo lleva la clase, por lo que podemos llamarlo usando solo la clase, sin importar si instanciamos o no un objeto:

De esta forma, podemos acceder al atributo de clase llamandolo directamente desde la clase sin tener que instanciar un objeto para llamar a un atributo.
Como te imaginarás, esto lo pueden heredar el resto de las clases que tengan herencia, así como todos las instancias que se realicen.



Tanto las clases, como los objetos instanciados, pueden acceder a este atributo de clase.
- A modo de aclaración: habrás observado que usamos la palabra reservada None como argumento para el parámetro nombre que se está heredando desde la clase Abuelo.

None lo podemos usar cuando necesitamos completar un espacio de memoria, como en este caso, para el atributo nombre, pero lo queremos hacer sin ningún valor en particular; en ese caso usamos None. -

Aprendiendo a ocultar - Encapsulamiento

En el código anterior con el atributo de clase que creamos, este puede ser accesible desde la propia clase sin tener que instanciar un objeto, pero ¿qué pasa si le asignamos un nuevo valor? ¿se puede? Veámoslo:

```
class Abuelo():
    atributo = 'de clase'

    def __init__(self, nombre):
        self.nombre = nombre

    def saludar(self):
        print('Cuando saludo, paso la mano.')

    def construir_casas(self):
        print('Puedo construir casas tradicior


class Abuela():
    pass

    def saludar(self):
```

```
class Hijo(Padre):
    pass


Abuelo.atributo = 'cambiando el valor'
print(Abuelo.atributo)
print(Padre.atributo)
print(Hijo.atributo)

objeto1 = Hijo(None)
print(objeto1.atributo)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

cambiando el valor
cambiando el valor
cambiando el valor
cambiando el valor
```

Efectivamente el valor está cambiando, por lo que si en algún momento necesitáramos que esto no pase, podemos convertir un atributo en privado o protegido; veámoslo:

Atributo protegido

En el caso de usar esta nomenclatura para un atributo, realmente lo que se está haciendo no es quitar la inaccesibilidad totalmente, sino que le damos el significado por convención que, si bien, se puede acceder y modificar al atributo desde fuera de la clase, es recomendable no hacerlo directamente. Más que inaccesible, en tal caso, estamos advirtiendo a otros programadores que no modifiquen este atributo. La forma de hacerlo es poniendo **un guion bajo delante del nombre** del atributo:

```
class Abuelo():
    _atributo = 'Este es un atributo de clase, protegido'

    def __init__(self, nombre):
        self._nombre = 'Este es un atributo de instancia protegido'

    def saludar(self):
        print('Cuando saludo, paso la mano.')

    def construir_casas(self):
        print('Puedo construir casas tradicionales de ladrillo y ceme


class Abuela():
    pass

    def saludar(self):
```

```
class Hijo(Padre):
    pass


print(Abuelo._atributo)

objeto = Hijo(None)
print(objeto._nombre)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  Python

Este es un atributo de clase, protegido
Este es un atributo de instancia protegido
```

Esta técnica se puede aplicar a atributos de clase, atributos de instancia e incluso a métodos.
¿Te gustaría intentarlo en los métodos? Sin embargo, es importante tener en cuenta que estos también se pueden modificar fuera de la clase, siguiendo el mismo proceso que el ejemplo anterior.

Atributo privado

Para este caso, no se puede acceder desde afuera, sin embargo, existe una forma de hacerlo, pero vamos con el ejemplo primero. La forma de hacerlo en este caso es poner **doble guion bajo delante del nombre** del atributo:

```
class Abuelo():
    __atributo = 'Este es un atributo de clase, privado'

    def __init__(self, nombre):
        self.__nombre = 'Este es un atributo de instancia, privado'

    def saludar(self):
        print('Cuando saludo, paso la mano.')

    def construir_casas(self):
        print('Puedo construir casas tradicionales de ladrillo y ceme


class Abuela():
    pass

    def saludar(self):
```

```
class Hijo(Padre):
    pass


print(Abuelo.__atributo)

objeto = Hijo(None)
print(objeto.__nombre)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  Python

AttributeError: type object 'Abuelo' has no attribute '__atributo'
Traceback (most recent call last):
  File "c:\Users\Pc\Desktop\Info2023\repaso\poo3.py", line 32, in <module>
    print(Abuelo.__atributo)
          ^^^^^^^^^^^^^^^^^
AttributeError: type object 'Abuelo' has no attribute '__atributo'
```

En Python, podemos encontrar el mecanismo conocido como name mangling
Si intentamos acceder a un atributo y recibimos un error, probablemente sea porque el nombre del atributo fue modificado internamente por Python. Debido a este mecanismo, es difícil acceder directamente al atributo. Aunque no existen atributos privados en Python, utilizamos la nomenclatura de name mangling para evitar

conflictos, especialmente en casos donde hay varios atributos con nombres similares.

Pero como te hemos dicho, de igual manera se puede acceder. Esto se puede hacer conociendo la clase desde donde viene el atributo:

```python
class Abuelo():
    __atributo = 'Este es un atributo de clase, privado'

    def __init__(self, nombre):
        self.__nombre = 'Este es un atributo de instancia, privado'

    def saludar(self):
        print('Cuando saludo, paso la mano.')

    def construir_casas(self):
        print('Puedo construir casas tradicionales de ladrillo y cemer


class Abuela():
    pass

    def saludar(self):
```

```python
class Hijo(Padre):
    pass



print(Abuelo._Abuelo__atributo)

objeto = Hijo(None)
print(objeto._Abuelo__nombre)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL        Python

Este es un atributo de clase, privado
Este es un atributo de instancia, privado
```

De este modo, podemos observar que, en realidad, no hay atributos completamente inaccesibles. No obstante, podemos utilizar estas nomenclaturas para minimizar el riesgo de errores o conflictos.

Atributo mágico

Por último, tenemos otra nomenclatura que ya la hemos visto y la venimos viendo desde que arrancamos con POO. Estos tipos de atributos empiezan y acaban con dos guiones bajos que se indican como atributos "mágicos", de uso especial y que residen en espacios de nombres que puede manipular el usuario. Solamente deben usarse en la manera que se describe en la documentación de Python y debe evitarse la creación de nuevos atributos de este tipo. Algunos ejemplos de nombres "singulares" de este tipo son:

Nombre	Descripción
__init__	método de inicialización de objetos (Constructor)
__del__	método de destrucción de objetos (Destructor)
__doc__	cadena de documentación de módulos, clases...
__class__	nombre de la clase
__str__	método que devuelve una descripción de la clase como cadena de texto
__repr__	método que devuelve una representación de la clase como cadena de texto
__module__	módulo al que pertenece la clase

Sin embargo, no vamos a explicar cada uno de estos name mangling, pero si te los brindamos en esta lista para que sepas que existen y los puedas explorar y usar según el proyecto que se requiera hacer.

Polimorfismo

En programación orientada a objetos se denomina polimorfismo a la capacidad que tienen los objetos, de una clase, de responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación.
Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa. Esto se puede conseguir a través de la herencia, donde un objeto de una clase hija, es al mismo tiempo un objeto de la clase padre, como lo venimos viendo.

En otras palabras, objetos de diferentes clases pueden compartir el mismo nombre de método, pero cada clase puede proporcionar su propia implementación específica.
Entonces, la definición de este término puede resultarte conocida. Esto es porque, básicamente, ya la hemos usado cuando vimos los ejemplos de la clase Persona, Empleado, Profesor.

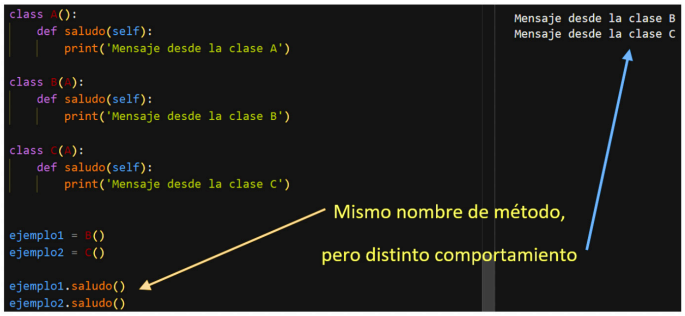
Te recordaremos un ejemplo:

```python
class Profesor(Persona, Empleado):
    def __init__(self, nombre, puesto, antiguedad):
        Persona.__init__(self, nombre)
        Empleado.__init__(self, nombre, puesto)
        self.antiguedad = antiguedad

    def saludar(self):
        Persona.saludar(self)
        Empleado.saludar(self)
        print(f'Estoy en este puesto hace {self.antiguedad} años.')
```

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

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

class Empleado():
    def __init__(self, nombre, puesto):
        self.nombre =nombre
        self.puesto = puesto

    def saludar(self):
        print(f"El puesto en el que me desempeño es el de {self.puesto}.")

class Profesor(Persona, Empleado):
    def __init__(self, nombre, puesto, antiguedad):
        Persona.__init__(self, nombre)
        Empleado.__init__(self, nombre, puesto)
        self.antiguedad = antiguedad

    def saludar(self):
        Persona.saludar(self)
        Empleado.saludar(self)
        print(f'Estoy en este puesto hace {self.antiguedad} años.')


profesor1 = Profesor('Informatorio', 'profesor', 11)
profesor1.saludar()
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Hola, mi nombre es Informatorio.
El puesto en el que me desempeño es el de profesor.
Estoy en este puesto hace 11 años.
```

Con el término polimorfismo también podemos referirnos a una sobrecarga de métodos (Overriding Methods), donde el lenguaje es el encargado de determinar que método ejecutar entre varios métodos que tengan el mismo nombre.
Lo vemos con un ejemplo:



Entonces, lo que estamos haciendo es sobrescribir un método heredado, de esta manera cuando instanciamos un objeto con una clase hija, podemos usar el mismo nombre de método, pero con distinto comportamiento debido a la clase de la que se instancia el objeto.

También podemos hablar de sobrecarga de operadores (Overloading Operators), que se enfoca en esencia al ámbito de los operadores aritméticos, binarios, de comparación y lógicos.

```
85  class Punto:
86      def __init__(self,x = 0,y = 0):
87          self.x = x
88          self.y = y
89      def __add__(self, parametro):
90          x = self.x + parametro.x
91          y = self.y + parametro.y
92          return x, y
93
94  punto1 = Punto(4,6)
95  punto2 = Punto(1,-2)
96  print(punto1 + punto2)
PROBLEMAS 1   SALIDA   CONSOLA DE DEPURACIÓN
(5, 4)
```

En este ejemplo, la clase "Punto" define el método __add__, que suma las coordenadas "x" e "y" de dos objetos "Punto" y devuelve un nuevo objeto "Punto" con las coordenadas sumadas.

Al utilizar el operador + entre dos objetos "Punto" (punto1 + punto2), Python invoca automáticamente el método __add__ y pasa punto1 como el primer argumento (self) y punto2 como el segundo argumento (parámetro). El método __add__ realiza la suma de las coordenadas y devuelve un nuevo objeto "Punto" que representa la suma.

El uso del método __add__ permite realizar operaciones de suma personalizadas en objetos de una clase definida por el usuario y proporciona una forma de interactuar con el operador + de manera significativa en el contexto de la clase.

Operadores y decoradores

Operadores

En Python, los operadores son símbolos especiales que se utilizan para realizar operaciones en variables y objetos. Por ejemplo, los operadores aritméticos como +, -, * y / se utilizan para realizar operaciones matemáticas, mientras que los operadores de comparación como ==, <, > se utilizan para realizar comparaciones entre valores.
Las clases en Python pueden sobrecargar estos operadores mediante la implementación de métodos especiales. Estos métodos especiales tienen nombres predefinidos que comienzan y terminan con doble guion bajo (__). Al implementar estos métodos especiales, puedes definir el comportamiento personalizado para los operadores en objetos de la clase.

Algunos ejemplos de operadores especiales que se pueden sobrecargar en una clase son:
__add__: sobrecarga el operador de suma (+).

__sub__: sobrecarga el operador de resta (-).
__mul__: sobrecarga el operador de multiplicación (*).
__div__: sobrecarga el operador de división (/).
__eq__: sobrecarga el operador de igualdad (==).
__lt__: sobrecarga el operador de menor que (<).

Sobrecargar estos operadores permite definir el comportamiento personalizado para los objetos de la clase cuando se utilizan operaciones relacionadas.

Decoradores

Los decoradores son una característica poderosa de Python que permiten modificar el comportamiento de una función o método existente sin modificar su implementación original. Los decoradores se definen utilizando la sintaxis de @ **seguida del nombre del decorador encima de la definición de una función**.

Un decorador es en realidad una función que toma otra función como argumento y devuelve una función modificada o envuelta. Los decoradores se utilizan comúnmente para agregar funcionalidades adicionales a una función, como registrar eventos, realizar validaciones previas o aplicar transformaciones a la salida.

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

    @property
    def nombre(self):
        return self._nombre

    @property
    def edad(self):
        return self._edad

    @edad.setter
    def edad(self, nueva_edad):
        if nueva_edad > 0:
            self._edad = nueva_edad


persona = Persona("Info", 11)
print(persona.nombre)
print(persona.edad)

persona.edad = 11
print(persona.edad)

persona.edad = -5
print(persona.edad) # No se actualiza la edad porque no es válida
```

```
Info
11
11
11
```

En este ejemplo, se utiliza el decorador @property para definir los métodos nombre y edad como propiedades de lectura. Esto permite acceder a ellos como atributos sin llamar explícitamente a los métodos. Además, se utiliza el decorador @edad.setter para definir un método que actúa como un setter para la propiedad edad, lo que permite establecer un nuevo valor para la edad con validación adicional.

Los operadores y decoradores son mecanismos poderosos en Python que permiten personalizar el comportamiento de las clases y las funciones de manera flexible. Los operadores especiales pueden ser sobrecargados en clases para definir operaciones personalizadas, mientras que los decoradores se utilizan para modificar el comportamiento de las funciones sin modificar su implementación original. Estas características son fundamentales en la programación orientada a objetos en Python y brindan flexibilidad y capacidad de personalización en el diseño de programas.

Resumen

La programación orientada a objetos (POO) es un paradigma fundamental en la programación que nos permite organizar y estructurar nuestro código de manera más eficiente y modular. A través de la POO, podemos modelar el mundo real y representar sus entidades y relaciones en forma de objetos.

En este recorrido por los conceptos de la POO, has aprendido sobre las características clave, como la encapsulación, la abstracción, la herencia y el polimorfismo. Hemos explorado cómo definir clases, crear objetos, y trabajar con atributos y métodos. También has visto cómo se establecen las relaciones entre las clases, como la herencia, la composición, la agregación, la asociación y la dependencia.

Además, has aprendido otros conceptos importantes, como los operadores y decoradores, que nos permiten personalizar el comportamiento de nuestras clases y funciones. Hemos visto ejemplos prácticos de cómo utilizarlos para extender y modificar el comportamiento predeterminado.

Es importante destacar que, aunque hemos cubierto una amplia gama de conceptos, la comprensión profunda y la maestría de la POO requieren práctica y dedicación. A medida que te sumerjas en proyectos y desafíos de programación, te encontrarás con escenarios más complejos donde podrás aplicar estos conceptos de manera más avanzada.

Por lo tanto, te animamos a practicar y explorar diferentes ejemplos y casos de uso. Crea tus propias clases, experimenta con las relaciones entre ellas, implementa herencia y polimorfismo en tus programas. A medida que adquieras más experiencia, vas a descubrir cómo la POO te permite construir programas más estructurados, reutilizables y fáciles de mantener.

Recuerda que la documentación oficial de Python y otros recursos en línea son valiosos para ampliar tu conocimiento y enfrentar desafíos más complejos. Además, aprovecha la comunidad de programadores del info, como así también de otros lugares y participa en proyectos colaborativos para obtener retroalimentación y aprender de otros.

En síntesis, la POO es una herramienta poderosa para el desarrollo de software, y con dedicación y práctica, podrás comprender y dominar sus conceptos. ¡Sigue explorando el apasionante mundo de la programación orientada a objetos en Python!