Base de Datos

Programación Orientada a Objetos pt.2

Subsecretaría de Empleo
Chaco Gobierno de todos

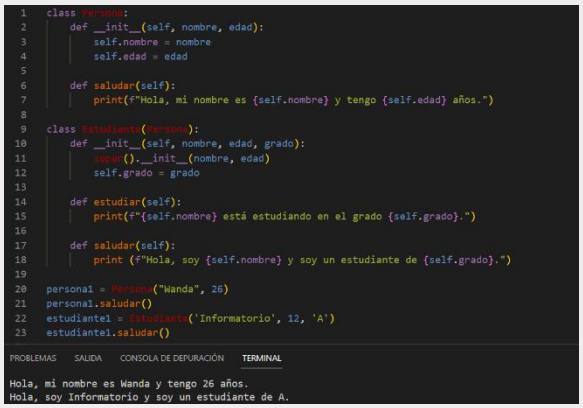
Ministerio de Producción, Industria y Empleo
Chaco Gobierno de todos

CHACO
Gobierno de todos

INFORMATORIO

Continuación

Usando herencia múltiple



```
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.
```

En este esquema de la primer parte del apunte pudimos observar varias cosas al realizar herencias.

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.

¿Lo vemos en código?

```
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 caso implementamos la herencia múltiple al crear una clase (profesor en este caso) que heredó más de una clase padre. Con esto, la clase hija pudo heredar atributos y métodos de las clases padre que especificamos.
En este caso en particular, tenemos tres clases: "Persona", "Empleado" y "Profesor", donde la clase "Profesor" hereda tanto de "Persona" como así también de "Empleado".
La clase "Profesor" tiene un método saludar propio, pero también estamos heredando los métodos saludar de las clases padre, los cuales nosotros incluimos:

```
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.')
```

obteniendo así la salida de los 3 saludos heredados; sin embargo, ¿qué pasa si no incluimos estos métodos heredados en nuestro método propio de la clase "Profesor"? Si, acertaste, el comportamiento va a ser distinto. Vamos a verlo y lo explicamos:

```
1  class Persona():
2      def __init__(self, nombre):
3          self.nombre = nombre
4
5      def saludar(self):
6          print(f"Hola, mi nombre es {self.nombre}.")
7
8  class Empleado():
9      def __init__(self, nombre, puesto):
10         self.nombre =nombre
11         self.puesto = puesto
12
13     def saludar(self):
14         print(f"El puesto en el que me desempeño es el de {self.puesto}.")
15
16 class Profesor(Persona, Empleado):
17     def __init__(self, nombre, puesto, antiguedad):
18         Persona.__init__(self, nombre)
19         Empleado.__init__(self, nombre, puesto)
20         self.antiguedad = antiguedad
21
22     def saludar(self):
23         print(f'Estoy en este puesto hace {self.antiguedad} años.')
24
25
26 profesor1 = Profesor('Informatorio', 'profesor', 11)
27 profesor1.saludar()

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Estoy en este puesto hace 11 años.
```

Seguimos con el mismo esquema pero ahora le "sacamos la herencia", o, en tal caso, otra forma de verlo sería que "no agregamos la herencia" a la clase Profesor del método "saludar()".

Lo que ocurre cuando realizamos herencia (sea simple o múltiple) y tenemos un método, o varios, con el mismo nombre, es que el método de la clase hija va a sobrescribir a los métodos heredados del mismo nombre.
Es decir, por un orden (MRO, del cual hablaremos más adeltante) establecido se buscará primero cualquier atributo o método en la clase actual, para este caso, la clase "Profesor" que tiene el método "saludar()" (al igual que las clases padre de la que hereda), si no especificamos, como lo hicimos en el ejemplo anterior, que incluimos los métodos saludar() de las clases anteriores y por ende, en el mensaje

mostrado en pantalla, fue de los 3 métodos saludar(), pasa lo que vemos en este caso actual; lo único que se muestra cuando llamamos al método saludar(), es el mensaje que existe en la clase hija "Profesor", pero no los mensajes que existen en las clases padres con el mismo nombre del método.

Ya sabemos, marea un poco, pero vamos a seguir dándote ejemplos para que puedas comprenderlo mejor. Una vez que lo comiences a relacionar con cosas cotidianas, vas a comprenderlo bien, y por eso vamos con un ejemplo que no puede fallar.

Si, efectivamente, vamos a ver el ejemplo que vas a entender si o si, y para eso vamos a pasar a la familia a código 😎

Comencemos del principio, visualiza la cantidad de cosas que hay por herencia de sangre solamente, ni hablar de cuando uno conoce a sus antecesores y hasta tiene relación con ellos (hablamos del hecho de su pudiste conocer a bisabuelos, abuelos, padres).

Bien, ahora que lo visualizaste, vamos a comenzar a pasar esto a código para poder entenderlo más rápido. Vamos a comenzar con una clase abuelo, a la cual no vamos a poner atributos, por el momento solo vamos a trabajar con métodos para simplificar un poco, y luego vamos a ver atributos. Supongamos que "Abuelo" tiene la habilidad construir casas, lo vemos en código:

```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 ya 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.

```
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.')
```

```
class Hijo(Abuelo, 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.
```

¿Comenzás a visualizar mejor como va comportándose la herencia múltiple?

Fijate como pudimos crear una clase hija, prácticamente sin nada de código heredando de otras clases padre. Eso es una de las cosas más potentes en la POO, ya que la reutilización de código y el ahorro de líneas de código, se vé desde el primer instante.

Sin embargo, como recordarás, pusimos en ambas clases padres el método saludar(), así que nos podemos preguntar ¿qué pasaría si uso el método saludar() en hijo?

```
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.')
```

```
class Hijo(Abuelo, 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.
```

Como verás, nos arroja “la forma” de saludar del abuelo, ¿y por qué pasa esto?

En páginas anteriores te mencionamos el MRO, del cual te dijimos íbamos a hablar más adelante, así que vamos a tocar ese punto antes de continuar con el ejemplo de la familia.

INFORMATORIO

MRO - Método de resolución de orden en Python

Cada clase en Python se deriva de la clase **object**.

Es el tipo más básico en Python. Así que técnicamente, todas las demás clases, ya sean integradas o definidas por el usuario, son clases derivadas y todos los objetos son 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 dijimos 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:
   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 violar 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 de resolución, se utiliza ese orden para buscar y resolver los atributos y métodos heredados en caso de conflictos.

```
class A():
    pass

class B():
    pass

class C(A,B):
    pass

class D(C):
    pass

print(D.mro())
```

```
[<class '__main__.D'>, <class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]


  <class '__main__.D'>,
    <class '__main__.C'>,
      <class '__main__.A'>,
        <class '__main__.B'>,
          <class 'object'>
```

Como podés ver, usando la función de .mro(), se puede visualizar la herencia que se viene dando y como MRO lo resuelve.

Así que, ahora que vimos 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()
```

Como podés apreciar, cuando llamamos al método saludar(), lo que verificamos en pantalla fue que el saludo pertencia al abuelo, por lo que si miramos el orden en nuestra herencia, la clase Abuelo, es la primera que se hereda y luego la clase Abuela. Recordá que en cada clase de "abuelos", estaba el método saludar(), sin embargo, en cada clase se comportaba distinto, ya que el "saludo" de cada abuelo, era distinto. Entonces, teniendo en cuenta toda esta información, ¿qué pasa si cambiamos el orden de la herencia en la clase Hijo?

```
class Abuelo():
    pass

    def saludar(self):
        print('Cuando saludo, pas

    def construir_casas(self):
        print('Puedo construir ca


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

Efectivamente, como lo estabas pensando, **Hijo** primero hereda de **Abuela** y luego de **Abuelo** el método con el mismo nombre, en este caso "saludar()", que sería el que entra en conflicto ya que en ambas clases tiene el mismo nombre de método. Entonces, el MRO va a resolver que el método que se va a heredar primero, va a ser el de la clase que esté primera, o más a la izquierda, si lo vemos 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.

```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 podés observar, al instanciar un objeto de clase **Hijo** y llamar a los métodos heredados, llama 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**).

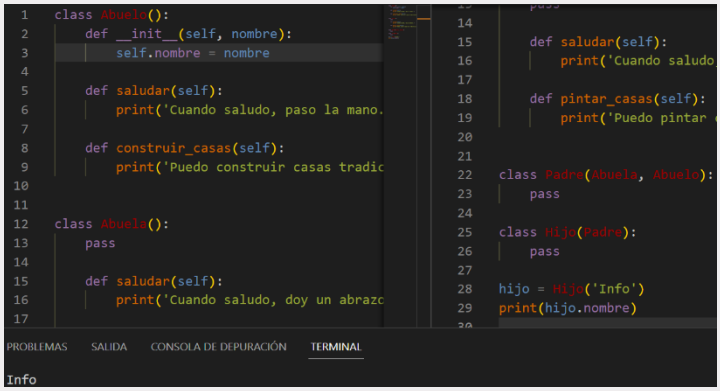
MRO fue resolviendo a medida que se realizaron las herencias.

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.

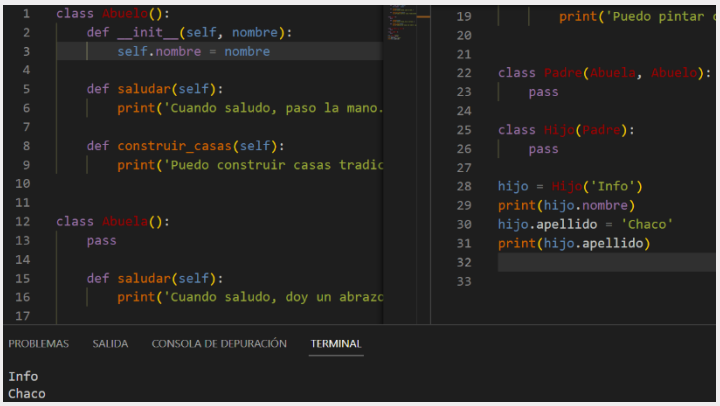
INFORMATORIO

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

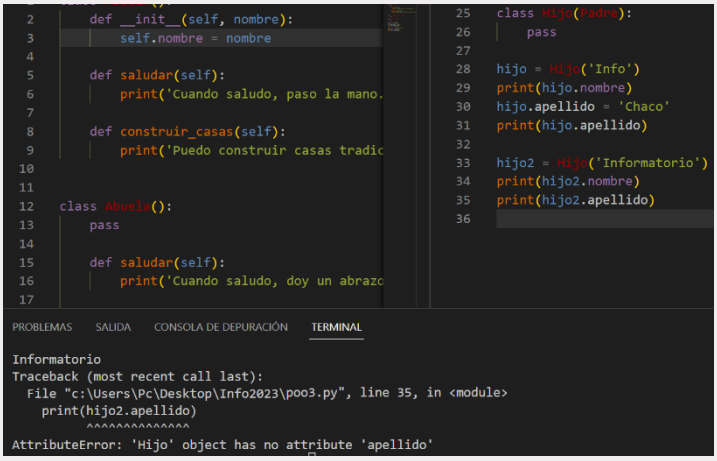


Como verás agregamos un atributo en una de las primeras clases (Abuelo) y lo estamos accediendo desde la última clase creada (hijo), para que puedas seguir visualizando como se van heredando atributos y métodos desde clases anteriores.

Ahora vamos a agregar un atributo dinámicamente, es decir, fuera de las clases, al objeto instanciado.



Lo que hicimos fue agregar directamente un atributo al objeto, por lo que como te darás cuenta, esto nos sirve para agregar a una instancia, algo que solo se va a usar en esa instancia, es decir en ese objeto, pero no se va a poder usar en otros objetos, así sean de la misma clase.



Y como vés, si queremos usar ese atributo agregado dinámicamente en otra instancia, no indica un error donde directamente nos 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 te mostramos que por más herencia que haya desde clases anteriores, tenemos algunos recursos más para poder usar sobre un objeto instanciado, sin necesidad de que estén en los demás objetos de la misma u otra clase.

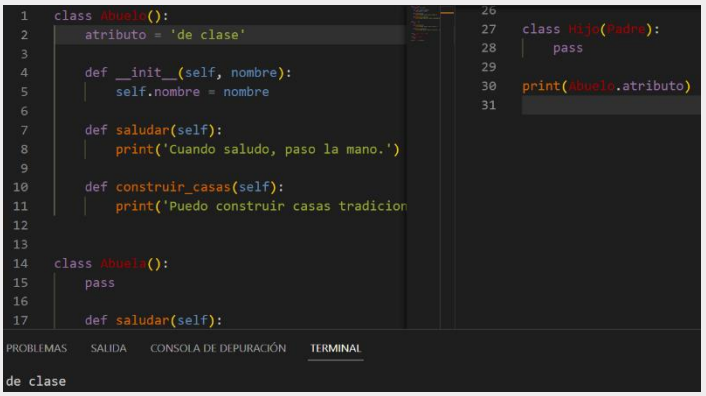
Entonces, siguiendo con los atributos, tenemos otro tipo de atributos de los que estamos manejando hasta el momento. Recientemente vimos un **atributo creado dinámicamente**, el cual era accesible, solo desde el objeto instanciado.

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
```

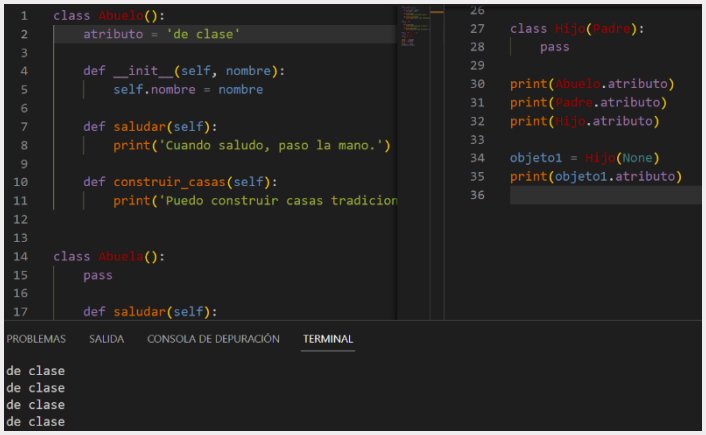
Al cual acceden todos los objetos instanciados de esa clase, y si es por herencia, también los heredan las subclases, o clases hijas.

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.



Así podemos ver en el ejemplo que, 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. -

INFORMATORIO

Aprendiendo a ocultar - Encapsulación

Como vemos 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? Vamos a verlo:

```
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 tradicion


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 necesitaramos 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
```

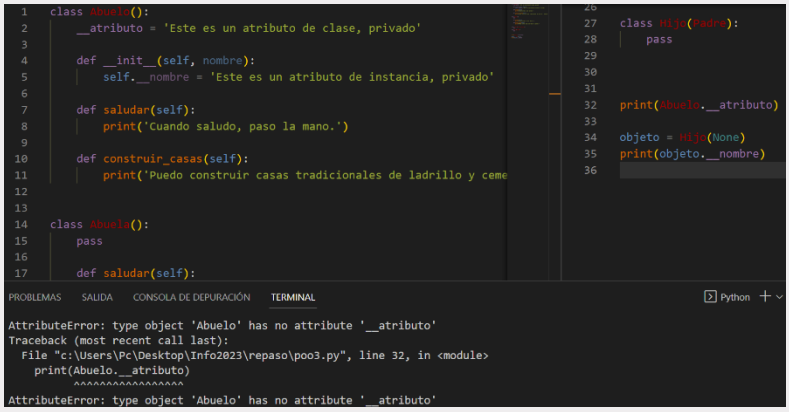
Efectivamente esta forma la podemos aplicar tanto a atributos de clases, atributos de instancia e incluso a métodos.

¿Te animás a probarlo en métodos?

Sin embargo, este se puede modificar por fuera de la clase siguiendo la misma línea de modificación del ejemplo anterior.

Atributo privado

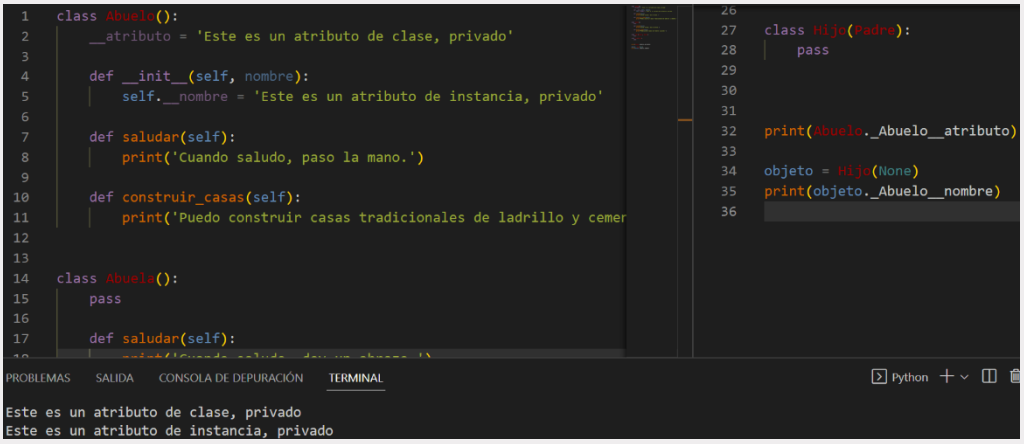
Para este caso, no se puede acceder desde afuera, sin embargo, existe una forma de hacerlo, pero vamos con el ejemplo primero. La forma de hacerlo en este caso es poner **doble guion bajo delante del nombre** del atributo:



Si intentamos acceder al atributo, no da un error, y esto ocurre porque internamente, Python ahora cambió el nombre del atributo a uno que nosotros no conocemos, por lo que no es posible acceder directamente a él. Este mecanismo se conoce como *name mangling*.

Atributos privados, realmente no existen en Python, pero usamos estas nomenclaturas para evitar conflictos sobre todo.

Y, decimos que no existen realmente atributos privados ya que conociendo la clase desde donde viene el atributo, se puede llegar a acceder al mismo. Vamos a verlo en el siguiente ejemplo.



Con esto podemos ver que realmente, no existen atributos que sean completamente inaccesibles, sin embargo, con estas nomenclaturas podemos tratar de mitigar esto para evitar posibles errores o conflictos.

Atributo mágico

Por último, tenemos otra nomenclatura que ya la vimos y la venimos viendo desde que arrancamos.

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 tocar cada uno de estos name mangling, pero si te los mostramos para que sepas que existen y se los puede 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 te puede estar resultando conocida, no? Esto es porque, básicamente, ya lo estuvimos usando cuando dimos los ejemplos de la clase **Persona, Empleado, Profesor** (e incluso, desde el apunte anterior). Vamos a recordarte el ejemplo:

```
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.')
```

Con el termino 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.

Entonces, lo pasamos en limpio con un ejemplo:

```
class A():
    def saludo(self):
        print('Mensaje desde la clase A')

class B(A):
    def saludo(self):
        print('Mensaje desde la clase B')

class C(A):
    def saludo(self):
        print('Mensaje desde la clase C')


ejemplo1 = B()
ejemplo2 = C()

ejemplo1.saludo()
ejemplo2.saludo()
```

```
Mensaje desde la clase B
Mensaje desde la clase C
```

Mismo nombre de método,

pero distinto comportamiento

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.

INFORMATORIO

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

En conclusión, 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, aprendiste 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 pudiste ver cómo se establecen las relaciones entre las clases, como la herencia, la composición, la agregación, la asociación y la dependencia.

Además, vimos 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 vas a poder aplicar estos conceptos de manera más avanzada.

Por lo tanto, te animamos a practicar mucho 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.

Recordá que la documentación oficial de Python y otros recursos en línea son valiosos para ampliar tu conocimiento y enfrentar desafíos más complejos. Además, aprovechá 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. ¡Seguí explorando el apasionante mundo de la programación orientada a objetos en Python!