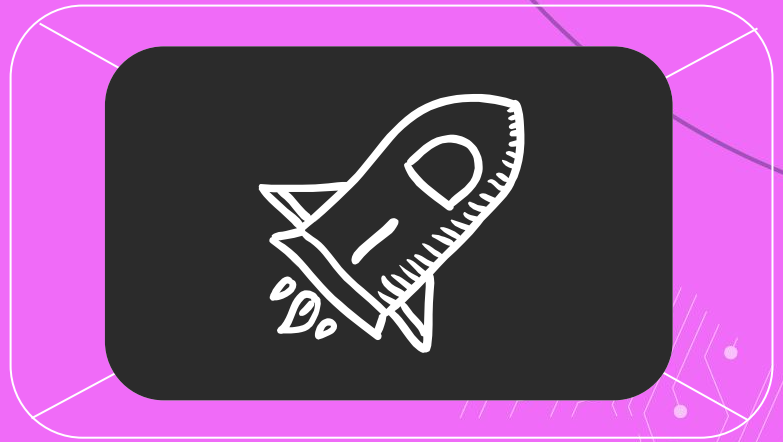
Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

Semana 6
¡Bienvenidos!



Temas de hoy

1. Herencia
2. Polimorfismo
3. Encapsulamiento
4. Métodos mágicos (dunder methods)

1

Herencia

Aprendimos en la clase anterior que con la palabra reservada ***class*** podemos crear clases, que son moldes o plantillas, y, al momento de instanciar, le damos valores a sus atributos como en el ejemplo del *coche* de la clase anterior.

Pero nos surge un problema, ¿Y si quisiéramos hacer una clase de bicicleta? ¿o una de moto? ¿o de cualquier otro vehículo?

Entonces deberíamos repetir la misma clase una y otra vez repitiendo código con los mismos atributos cambiando solamente su atributo de clase (la cantidad de ruedas).

No suena muy bien, así que veamos cómo solucionarlo 😉

Herencia en Python

Clase padre → Clase hija

La herencia es la práctica por la cual se puede crear una **clase *hija*** que herede de una **clase *padre*** todos sus *métodos* y *atributos*.
Además, la *clase hija* tiene el poder de sobreescribir los métodos y atributos que hereda, y de agregar unos nuevos específicos para esta clase.
Esto nos puede resultar muy útil cuando tengamos clases que se parecen entre sí pero tienen ciertas particularidades.
El realizar esta abstracción y buscar un común denominador para definir una *clase padre* para que hereden los demás, es una tarea compleja pero necesaria dentro del mundo de la programación.

```python
class Vehiculo:
    def __init__(self, ruedas, marca, modelo, color):
        self.ruedas = ruedas
        self.marca = marca
        self.modelo = modelo
        self.color = color

    def acelerar(self):
        print(f"El {self.marca} {self.modelo} está acelerando.")

    def frenar(self):
        print(f"El {self.marca} {self.modelo} está frenando.")
```

Volviendo al ejemplo del coche ya dado anteriormente, crearemos una *clase Vehiculo* la cual será nuestra **clase padre**.
El código es el mismo que vimos con la *clase Coche*, lo único que cambia es que tiene un nuevo atributo llamado *ruedas* que dejó de ser un atributo de clase para ser un atributo de instancia.

Herencia en Python

Ahora crearemos dos *clases hijas*: **Coche** y **Bicicleta.** Para que las mismas hereden todos los atributos y métodos de **Vehiculo** lo que debemos hacer es agregar el nombre de la clase de la cual se quiere heredar dentro de los paréntesis seguidos del nombre de la clase hija.
Como podemos observar, las clases se encuentran vacías, por lo que usamos la palabra reservada *pass*.

```
class Coche(Vehiculo):
    pass

class Bicicleta(Vehiculo):
    pass
```

```
mi_coche = Coche(4, "Toyota", "Corolla", "Rojo")

mi_bici = Bicicleta(2, "Vairo", "CRX", "Blanco")

print(f"Mi auto es un {mi_coche.marca} {mi_coche.modelo} de color {mi_coche.color} con {mi_coche.ruedas} ruedas.")

print(f"Mi bicicleta es una {mi_bici.marca} {mi_bici.modelo} de color {mi_bici.color} con {mi_bici.ruedas} ruedas.")
```

```
Mi auto es un Toyota Corolla de color Rojo con 4 ruedas.
Mi bicicleta es una Vairo CRX de color Blanco con 2 ruedas.
```

Aún así, aunque las clases estén vacías, al momento de instanciar ambas clases, nos permiten pasarles valores a los mismos atributos que su *clase padre* (Vehiculo) y funcionar correctamente como si hubiese repetido el código.

Herencia en Python

Como mencionamos anteriormente, las *clases hijas* tienen el poder de reemplazar los métodos heredados o agregar nuevos métodos específicos para la *clase hija*. Veamos ambos ejemplos:

- **Ejemplo 1**:

```
class Coche(Vehiculo):
    def bocina(self):
        print("¡PIIIIIIIII! 🔊")

mi_coche = Coche(4, "Toyota", "Corolla", "Rojo")
mi_coche.bocina()
```

```
¡PIIIIIIIII! 🔊
```

En el caso de la *clase Coche,* creamos un nuevo *método bocina* el cual, hará la onomatopeya del sonido.
Ni la *clase Vehiculo*, ni la *clase Bicicleta* serán capaces de acceder a este método.

- **Ejemplo 2**:

```
class Bicicleta(Vehiculo):
    def acelerar(self):
        print(f"Estoy empezando a pedalear 🚴")

mi_bici = Bicicleta(2, "Vairo", "CRX", "Blanco")
mi_bici.acelerar()
```

```
Estoy empezando a pedalear 🚴
```

En el caso de la *clase Bicicleta* estamos modificando un método ya heredado como lo es *acelerar*, en el cual, modificamos el mensaje por uno más acertado al vehículo en cuestión.
Por lo tanto, al momento de utilizar el método, no mostrará el mensaje heredado de *Vehiculo*, si no, el modificado.

Herencia en Python: super()

super()

Puede suceder que necesitemos que nuestra *clase hija Coche* tenga un parámetro extra al momento de instanciar, como podría ser el nombre del dueño.
Para hacerlo tenemos dos opciones:

→ 1er opción:

```python
class Coche(Vehiculo):

    def __init__(self, ruedas, marca, modelo, color, duenio):
        self.ruedas = ruedas
        self.marca = marca
        self.modelo = modelo
        self.color = color
        self.duenio = duenio

    def bocina(self):
        print("¡PIIIIIIIII! 🔊")


mi_coche = Coche(4, "Toyota", "Corolla", "Rojo", "Franco")
print(f"El dueño del auto es {mi_coche.duenio}")
```

```
El dueño del auto es Franco
```

Donde volvemos a iniciar un *constructor* en la *clase hija,* y, guardamos nuevamente cada uno de los atributos uno por uno agregando el nuevo atributo como veníamos haciendo.

→ 2da opción:

```python
class Coche(Vehiculo):

    def __init__(self, ruedas, marca, modelo, color, duenio):
        super().__init__(ruedas, marca, modelo, color)
        self.duenio = duenio

    def bocina(self):
        print("¡PIIIIIIIII! 🔊")


mi_coche = Coche(4, "Toyota", "Corolla", "Rojo", "Franco")
print(f"El dueño del auto es {mi_coche.duenio}")
```

Podemos iniciar un *constructor* que reciba los argumentos que definimos al momento de instanciar la clase, y, luego ocupamos ***super()*** para hacer un llamado al método *__init__* de la *clase padre* para pasarle los parámetros que ya aceptaba anteriormente, y asignamos solamente la variable de duenio manualmente.
Por lo que, la función ***super()*** nos permite llamar métodos de la *clase padre* desde dentro de una *clase hija* o *subclase*.

Herencia en Python: Herencia múltiple

Heredando de más de una clase

La herencia múltiple es una característica que permite a una clase heredar más de una *clase padre*. Esto significa que la *clase hija* va a poder acceder a los atributos y métodos de múltiples clases, lo que tiene un potencial muy grande cuando necesitamos combinar funcionalidades de diferentes clases en una sola.

Vamos a ver en un ejemplo:

```python
class A:
    def __init__(self):
        print("Soy la clase A")

class B:
    def __init__(self):
        print("Soy la clase B")

class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print("Soy la clase C")

prueba = C()
```

```
Soy la clase A
Soy la clase B
Soy la clase C
```

En este ejemplo, tenemos una *clase A*, una *clase B* y una *clase C* con sus respectivos constructores.
La *clase C* hereda de las dos primeras, lo que nos permite ocupar las funciones que tienen, desde esta *clase hija* (C), como por ejemplo, acceder a sus constructores.
Por lo tanto, al instanciar la *clase C* y ejecutarse su *constructor*, lo primero que hace es ejecutar el *constructor* de la *clase A*, luego el de la *clase B* y, por último, el *print* de la *clase C*.

Herencia en Python: MRO

Method Resolution Orden

La herencia múltiple trae una cierta complejidad en el diseño de nuestro código, ya que puede suceder que nuestras *clases padres* tengan métodos o atributos con el mismo nombre 😢.
Python maneja estos conflictos utilizando un **orden de resolución de métodos** - **Method Resolution Orden** o por sus siglas en inglés, **MRO**. Esto determina un orden de prioridad entre los métodos y atributos de las clases padres.

Sigamos con el ejemplo de las *clases A*, *B* y *C* con unas pequeñas modificaciones:

```python
class A:
    def __init__(self):
        super().__init__()
        print("Soy la clase A")

class B:
    def __init__(self):
        super().__init__()
        print("Soy la clase B")

class C(A, B):
    def __init__(self):
        super().__init__()
        print("Soy la clase C")

prueba = C()
```

Como podemos ver, ahora todas las clases dentro de sus constructores utilizan *super()* para llamar a sus clases padre, pero la *clase C* tiene dos ¿cómo se resolverá esto?

Herencia en Python: MRO

Para eso utilizaremos la función **__mro__** en nuestra clase y nos mostrará cual es el orden en el que los métodos serán llamados.

```
print(f"MRO: {C.__mro__}")
```

```
MRO: (<class '__main__.C'>, <class '__main__.A'>,
<class '__main__.B'>, <class 'object'>)
```

Por lo tanto, según el MRO:

```
class C(A, B):
    def __init__(self):
        super().__init__()
        print("Soy la clase C")
```

El *super()* de la *clase C* llamará primero al constructor de la *clase A*.

```
class A:
    def __init__(self):
        super().__init__()
        print("Soy la clase A")
```

Luego, el constructor llamará al constructor de su *clase padre* que según, el **MRO** es la *clase B*.

```
class B:
    def __init__(self):
        super().__init__()
        print("Soy la clase B")
```

Y por último, la *clase B* llamará a su *clase padre*, que es la *clase base* final (*object*) de la cual, todas las clases siempre heredan de ella.

Una vez que la *clase B* termina de llamar a la *clase object*, ejecuta su segunda línea (*Soy la clase B*), por lo tanto, una vez finalizado todo el bloque constructor de la *clase B*, se da por terminado lo que fue la primera línea del constructor de la *clase A* así que, vuelve a la misma ejecutando su segunda línea también (Soy la clase A). Por último, al terminar de completar todo el constructor de dicha clase, termina en la *clase C* mostrando en pantalla su mensaje (Soy la clase C).

```
Soy la clase B
Soy la clase A
Soy la clase C
```

2

Polimorfismo

El polimorfismo es otra característica fundamental que nos permitirá definir métodos en la *clase padre* que puedan ser redefinidos en las *clases hijas*, pero más importante, es que podremos manipular a las instancias de diferentes clases de la misma manera incluso si se comportan de forma diferente.

De ahí yace su nombre que deriva de dos términos distintos: **poli**, que significa numerosos, y **morfos**, que significa formas.

Polimorfismo en Python

Ejemplo

→ *Problema*: Calcularemos el sueldo anual de los empleados de una empresa según su rol en la misma.

```python
class Empleado:
    def __init__(self, sueldo):
        self.sueldo = sueldo

    def calcularSueldoAnual(self):
        sueldoAnual = 12 * self.sueldo * (1 + 1/100)
#                     12 meses * sueldo del empleado * 1% de bonus al ser empleado.
        print(f"El sueldo anual del empleado NORMAL es de {sueldoAnual}")
```

Creamos nuestra *clase padre* con el cálculo base y un bonus mínimo para los empleados generales.

```python
class Programador(Empleado):

    def calcularSueldoAnual(self):
        sueldoAnual = 12 * self.sueldo * (1 + 4/100) # 4% bonus
        print(f"El sueldo anual del empleado PROGRAMADOR es de ARS${sueldoAnual}")
```

Luego creamos las clases que heredan de *Empleado*. Nótese que empezamos a utilizar polimorfismo al utilizar el mismo método heredado pero modificamos su contenido; en el caso de *Programador* tiene un bonus del 4% y el *diseñador* un 5%, además de que, el mensaje también cambia según el rol.

```python
class Diseniador(Empleado):

    def calcularSueldoAnual(self):
        sueldoAnual = 12 * self.sueldo * (1 + 5/100) # 5% bonus
        print(f"El sueldo anual del empleado DISEÑADOR es de ARS${sueldoAnual}")
```

Pero esto ya habíamos visto anteriormente con herencia, entonces ¿Cuál es la diferencia?

Polimorfismo en Python

Siguiendo con el ejemplo

La particularidad del polimorfismo es que podemos utilizar una misma interfaz general para poder manipular cualquier instancia de clase que se le pase por parámetro.
Sigamos con el ejemplo:

```
empleados = [
    Empleado("Franco", 1000),
    Programador("Nelson", 1850),
    Empleado("Susana", 2500),
    Diseniador("Dario", 1054),
    Diseniador("Jazmin", 2202),
    Programador("Luciana", 5070)
]

def calculoEmpleadosEmpresa():
    for empleado in empleados:
        empleado.calcularSueldoAnual()

calculoEmpleadosEmpresa()
```

Supongamos que tenemos una lista con todos los empleados de la empresa con sus respectivos nombres y sueldos mensuales en sus roles correspondientes como podemos ver en la imagen.

Gracias al polimorfismo, podemos nombrar los métodos con el mismo nombre, lo que nos permite crear una interface (función calculoEmpleadosEmpresa) que manipule todas las instancias creadas por igual aunque sean diferentes mediante un bucle.

```
El sueldo anual de Franco (EMPLEADO), es de ARS$12120.0
El sueldo anual de Nelson (PROGRAMADOR), es de ARS$23088.0
El sueldo anual de Franco (EMPLEADO), es de ARS$12120.0
El sueldo anual de Nelson (PROGRAMADOR), es de ARS$23088.0
El sueldo anual de Nelson (PROGRAMADOR), es de ARS$23088.0
El sueldo anual de Dario (DISEÑADOR), es de ARS$13280.400000000001
El sueldo anual de Jazmin (DISEÑADOR), es de ARS$27745.2
El sueldo anual de Luciana (PROGRAMADOR), es de ARS$63273.6
```

Para mejor entendimiento del polimorfismo en Python en específico se recomienda leer sobre duck typing ya que tiene diferencia con otros lenguajes debido a su tipado dinámico característico de este lenguaje.

3

Encapsulamiento

Aprendimos a crear clases, atributos, métodos y que compartan entre sí sus funcionalidades con la herencia.

¿Pero qué pasaría si creamos una *clase Persona* en el que se requiera su DNI por parámetro?

y si por ejemplo, hiciéramos: **persona.DNI = “Martin”**, modificaríamos el valor sin ningún problema, lo que podía generar conflictos e incoherencias innecesarias.

Para cuidar la integridad de los datos de los atributos y restringir o regular los accesos a sus métodos es que veremos encapsulamiento 🤭

Encapsulamiento: Modificadores de Acceso

Atributos y Métodos

Públicos	Protegidos	Privados
Se pueden acceder desde **cualquier** parte del código, sea **dentro** o **fuera** de la clase o clases hijas, etc.	Pueden ser accedidos por la misma clase o clases hijas	Son accesibles **únicamente** dentro de la misma clase.

Encapsulamiento: Utilidades

- Nos permite ocultar métodos y atributos fuera de la propia clase.
- Nos permite regular la modificación de los atributos (privados) al evitar su acceso. Se acostumbra hacer **métodos públicos** dentro de la misma clase para modificar estos atributos (*getters* y *setters*)
- Se “enmascara” la complejidad de algunos métodos al ser **privados** y los reutilizamos dentro de **métodos públicos.**

Si bien aprenderemos a utilizar correctamente el encapsulamiento y definir nuestros atributos y métodos de forma privada, protegida y pública, en Python es simplemente una convención ya que no cumple a “raja tabla” su función porque existen prácticas (no recomendadas) de acceder igualmente a nuestros componentes privados 🤫.
De igual manera, aplicar el concepto en Python es fundamental para indicar, entre programadores, cómo se espera que sean utilizados los atributos y métodos en el algoritmo.
En otros lenguajes como JAVA o C++ sí cumplen estrictamente las reglas.

Encapsulamiento: Modificadores

Modificador Público

Trabajaremos todos los conceptos basándonos en un mismo ejemplo, pero modificando sus atributos y métodos.
Hay que tener en cuenta que, si bien ejemplificaremos con los atributos, todos los conceptos aplican igualmente en los métodos.

Entonces, para el ejemplo, crearemos una *clase Circulo* con el atributo *radio* que recibirá un valor por parámetro y un atributo *pi* con su valor constante. Además, contiene dos métodos con los cuales calculamos el área y perímetro del círculo según el radio ingresado.

```python
class Circulo:
    def __init__(self, radio):
        self.radio = radio
        self.pi = 3.1415

    def calcularPerimetro(self):
        return 2 * self.pi * self.radio

    def calcularArea(self):
        return self.pi * self.radio ** 2


circulo_uno = Circulo(3.5)

print(circulo_uno.calcularArea())
print(circulo_uno.calcularPerimetro())
print(f"El valor de PI en la clase es {circulo_uno.pi}")
```

Dentro de la clase

Fuera de la clase

```
38.483375
21.9905
El valor de PI en la clase es 3.1415
```

Hasta el momento, veníamos haciendo nuestras clases de esta manera, donde todos los componentes son públicos y lo podemos corroborar porque fuera de la clase lo instanciamos y, podemos acceder a ambos métodos y podemos hacer un *print* del atributo *pi*.

Encapsulamiento: Modificadores

Modificador Privado

Como cualquier persona que ocupe esta *clase* puede hacer, por ejemplo, *Circulo.pi = "pizza"* y, corromper el código, por lo que, si necesitamos que no se pueda modificar libremente desde fuera de la clase, lo definiremos como atributo privado agregándole dos guiones bajos (**__atributo**) al comienzo del nombre, como lo podemos ver en el constructor.

```python
class Circulo:
    def __init__(self, radio):
        self.radio = radio
        self.__pi = 3.1415

    def calcularPerimetro(self):
        return 2 * self.__pi * self.radio

    def calcularArea(self):
        return self.__pi * self.radio ** 2


circulo_uno = Circulo(3.5)

print(circulo_uno.calcularArea())
print(circulo_uno.calcularPerimetro())

print(f"El valor de PI en la clase es {circulo_uno.pi}")

print(f"El valor de PI en la clase es {circulo_uno.__pi}")
```

Podés ver que, dentro de la clase, en **todos los lugares** donde fue ocupado el atributo también debemos agregarle su nueva sintaxis.

Pero al momento de ejecutar este código, devuelve lo siguiente:

```
Traceback (most recent call last):
  File "c:\Informatorio\herencia.py", line 183, in <module>
    print(f"El valor de PI en la clase es {circulo_uno.pi}")
                                           ^^^^^^^^^^^^^^^
AttributeError: 'Circulo' object has no attribute 'pi'
```

Según lo visto sobre los modificadores privados, no debemos ser capaces de acceder a dicho atributo desde fuera de la clase, por eso mismo nos devuelve un **error** aclarando que "El objeto Circulo no tiene un atributo 'pi' ".

Es decir que, desde fuera de la clase, cuando instanciamos la *clase Circulo* y queremos imprimir en pantalla el valor de *pi*, Python es incapaz de encontrar dicho atributo, pero sí podemos hacer uso de este dentro de la clase Circulo.

Encapsulamiento: Propiedades de atributos - Getter, Setter

Get

Como comentamos entre las utilidades que tiene el encapsulamiento, si bien nombramos atributos de forma privada, se acostumbra a crear métodos públicos para poder acceder a ellas o modificarlas. Pero ¿por qué? Si la razón principal de hacerlas privadas era que no puedan hacerlo.

El encapsulamiento regula **el acceso sin restricciones**, pero podemos permitir deliberadamente y de forma controlada el acceso al atributo en caso de ser necesario usando **getters** y **setters**.

Se llaman de esa manera al ser una simple convención de los verbos **get** (obtener) y **set** (asignar o establecer).

```python
class Circulo:
    def __init__(self, radio):
        self.radio = radio
        self.__pi = 3.1415

    def calcularPerimetro(self):
        return 2 * self.__pi * self.radio

    def calcularArea(self):
        return self.__pi * self.radio ** 2

    def getPi(self):
        return f"El valor de PI es{self.__pi}"
```

```
El valor de PI es 3.1415
```

Definimos entonces el método ***getPi*** en nuestra *clase Circulo* y si el atributo fuese público, al momento de llamarlo obtendríamos el valor sin más.

Pero al crear un *getter* que nos devuelve la información, podemos controlar el formato en que devolveremos la información de la manera que nos sea más conveniente, como en este caso que lo devolvemos con un mensaje.

Encapsulamiento: Propiedades de atributos - Getter, Setter

Set

El mismo concepto estaríamos aplicando con el ***setter***.
Si bien al hacer privado el atributo restringimos su modificación, podemos hacer un método público que tenga permitido hacerlo, pero con la ventaja de que podemos hacer verificaciones del valor por el cual será alterado.

En este caso primero corroboramos que el nuevo valor de *PI* sea un número, ya sea entero o decimal, y en caso de serlo, verificamos que sea positivo.

Si cumple ambas condiciones, entonces realizamos la modificación, pero si alguna no se cumple, devolvemos un mensaje de error.

```
    def setPi(self, nuevoValor):
        if type(nuevoValor) == int or type(nuevoValor) == float:
            if nuevoValor > 0:
                self.__pi = nuevoValor
                print(f"El nuevo valor de PI es {self.__pi}")
            else:
                print("PI no puede ser negativo")
        else:
            print("El valor debe ser un número positivo")


circulo_uno = Circulo(3.5)

circulo_uno.setPi(4.6)
```

```
circulo_uno = Circulo(3.5)

circulo_uno.setPi(4.6)
circulo_uno.setPi(-5)
circulo_uno.setPi("Hola")
```

```
El nuevo valor de PI es 4.6
PI no puede ser negativo
El valor debe ser un número positivo
```

4

Métodos mágicos
(Dunder Methods)

Existe una característica muy valiosa en Python que es la capacidad de darle funcionalidades a nuestros objetos para hacer aún más limpio e intuitivo nuestro código.

¿A qué nos referimos con funcionalidades?

Nos referimos a que podemos definir **cómo** se comportan nuestras clases cuando hacemos operaciones comunes sobre ellos y para eso ocuparemos **Métodos mágicos** ✨

Dunder Methods

Los **dunder methods** o **métodos mágicos** son los métodos encargados de especificar el comportamiento que tendrá nuestra instancia de una clase cuando es utilizada en alguna operación común.
Normalmente, los métodos mágicos no están pensados para ser llamados directamente en el código; más bien, serán llamados por Python mientras se ejecuta el programa.

Estas operaciones comunes incluyen:

- **Operaciones aritméticas**
- **Operaciones de comparación**
- **Operaciones de ciclo de vida**
- **Operaciones de representación**

Ya estuvieron ocupando estos métodos sin saberlo, ¿se acuerdan de cómo creamos un *constructor* en una clase?.
Exactamente, con el **__init__()**. La sintaxis de estos métodos se caracterizan por llevar doble guión bajo al principio y al final del método.
Este método *__init__()* definía una funcionalidad a nuestras clases que era la de un constructor, es decir, que al momento de crear una instancia de nuestra clase, este método se ejecutaba automáticamente por sí mismo y, a todo el bloque de código que lo conformaba sin tener que hacer una invocación previa.
Así como existe el *__init__()* también existen otros métodos que nutrirán nuestro código.

Dunder Methods: __str__

Probando sin __str__

```python
class Rectangulo:
    def __init__(self, anchura, altura):
        self.anchura = anchura
        self.altura = altura

rect_uno = Rectangulo(10, 3)
print(rect_uno)
```

Si hacemos un *print* de nuestro objeto (instancia) lo que obtendremos es la dirección de memoria en donde está alojada la clase (información que no es pertinente para nosotros)

```
<__main__.Rectangulo object at 0x00000162D43D92D0>
```

Probando con __str__

```python
class Rectangulo:
    def __init__(self, anchura, altura):
        self.anchura = anchura
        self.altura = altura

    def __str__(self):
        return f"El ancho del rectangulo es {self.anchura}, y su altura={self.altura}"

rect_uno = Rectangulo(10, 3)
print(rect_uno)
```

El método **__str__()** lo ocuparemos para generar un mensaje que represente a nuestro objeto.
El método se ejecutará cuando pasemos la instancia de nuestra clase en una función *print()* por ejemplo.
Entonces en este caso si hacemos el mismo *print* que en el ejemplo, la clase nos devuelve un mensaje con la información almacenada en la clase.

```
El ancho del rectangulo es 10, y su altura es 3
```

Dunder Methods: __add__

Probando sin __add__

```
class Rectangulo:
    def __init__(self, anchura, altura):
        self.anchura = anchura
        self.altura = altura

    def __str__(self):
        return f"El ancho del rectangulo es {self.anchura}, y su altura es {self.altura}"

rect_uno = Rectangulo(10, 3)
rect_dos = Rectangulo(5, 8)

rect_tres = rect_uno + rect_dos
print(rect_tres)
```

```
Traceback (most recent call last):
  File "c:\herencia.py", line 217, in <module>
    rect_tres = rect_uno + rect_dos
                ~~~~~~~~~^~~~~~~~~~
TypeError: unsupported operand type(s) for +: 'Rectangulo' and 'Rectangulo'
```

Como podemos deducir, no existe un resultado lógico para la suma de dos instancias.

Si intentamos sumar las instancias **rect_uno** y **rect_dos** nos devolverá un error donde especifica que ambos valores que buscamos operar, no son de un tipo de dato preparado para la suma ya que, ambos son de tipo *Rectangulo*.

Probando con __add__

```
class Rectangulo:
    def __init__(self, anchura, altura):
        self.anchura = anchura
        self.altura = altura

    def __str__(self):
        return f"El ancho del rectangulo es {self.anchura}, y su altura es {self.altura}"

    def __add__(self, otro):
        nueva_anchura = self.anchura + otro.anchura
        nueva_altura = self.altura + otro.altura
        return Rectangulo(nueva_anchura, nueva_altura)

rect_uno = Rectangulo(10, 3)
rect_dos = Rectangulo(5, 8)

rect_tres = rect_uno + rect_dos
print(rect_tres)
```

```
El ancho del rectangulo es 15, y su altura es 11
```

El método **__add__()** nos permite definir el comportamiento de la suma "+" entre instancias de la clase *Rectangulo*.

Este método debe tener dos parámetros:

- el **self** obligatorio para los métodos de una clase el cual

Dunder Methods: __eq__

recibirá como argumento la primera instancia a operar (*rect_uno*)

- Un segundo parámetro con nombre a elección, en este caso "otro", en el cual guardará como argumento la instancia que se encuentra del otro lado del signo "+".

Dentro del método se efectúa la suma de los atributos de cada instancia y se retorna una nueva instancia con los datos sumados y se guarda en la variable **rect_tres**.

```
rect_tres = rect_uno + rect_dos
```

```
def __add__(self, otro):
    nueva_anchura = self.anchura + otro.anchura
    nueva_altura = self.altura + otro.altura
    return Rectangulo(nueva_anchura, nueva_altura)
```

*Así como existe **__add__** también existe el método **__mul__** para el producto, y su implementación es similar.*

Método mágico __eq__

Así como vimos que existen métodos para operaciones aritméticas, también podemos definir el comportamiento de las instancias ante operaciones de comparación como "==" , "<" y ">".

En el ejemplo de la siguiente página se ocupará **__eq__** para el operador "==" pero la implementación de **__lt__** para el operador "<" y **__gt__** para el operador ">" es la misma.

Dunder Methods: __eq__

```python
class Rectangulo:
    def __init__(self, anchura, altura):
        self.anchura = anchura
        self.altura = altura

    def __eq__(self, otro):
        return self.anchura == otro.anchura and self.altura == otro.altura


rect_uno = Rectangulo(10, 8)
rect_dos = Rectangulo(10, 8)

if rect_uno == rect_dos:
    print("El primer rectangulo es igual al segundo rectangulo")
else:
    print("Los rectangulos son desiguales")
```

```
El primer rectangulo es igual al segundo rectangulo
```

Al definir este método en una clase, tendremos la posibilidad de personalizar cómo será la igualdad entre dos instancias de la misma clase.
Al igual que los métodos anteriores, el método **__eq__** requiere un parámetro adicional al *self*, en el cual se guardará la segunda instancia a comparar y nos retornará un booleano.
En este ejemplo en particular requerimos que tanto las anchuras como las alturas de los rectángulos deben ser iguales y, gracias a ello, podemos utilizarlo en una estructura condicional.

Métodos de comparación:

- **__lt__** : Menor que (<)
- **__le__**: Menor o igual que (<=)
- **__gt__**: Mayor que (>)
- **__ge__**: Mayor o igual que (>=)
- **__eq__**: Igual a (==)

Lo que vimos

Los fundamentos que aprendimos durante esta clase no solo permiten la reutilización de código, sino también, la creación de código más robusto y mantenibles a futuro.

Aprendimos a cómo pensar y desarrollar código con más jerarquía con la herencia y polimorfismo, y, a cuidar más la integridad de nuestros datos y, proteger nuestras clases con el encapsulamiento.

Por último, vimos cómo hacer que nuestras instancias tengan una interacción más intuitiva y poderosa utilizando los métodos mágicos, cerrando así, el conjunto de fundamentos necesarios para trabajar bajo este paradigma de la **p**rogramación **o**rientada a **o**bjetos 👌

¡Nos vemos En la próxima clase!