

Python / Guía 9

Programación Orientada a Objetos II

## 

## 

## OBJETIVOS DE LA GUÍA

En esta guía aprenderemos a:

* Implementar métodos de clase
* Implementar métodos estáticos
* Mejorar la funcionalidad de las clases con "métodos especiales"
* Entender la encapsulación en Python
* Crear atributos y métodos privados
* Entender el concepto de getters y setters
* Mejorar la funcionalidad de las clases con "propiedades"
* Crear y mantener clases con mayor facilidad gracias al módulo dataclasses

**En esta guía no te ates a la consigna. Tienes que probar y jugar con**

**todo lo que vas aprendiendo...**

# 

# Más sobre métodos

En la programación orientada a objetos, en una clase podemos definir 3 tipos de métodos:

### Métodos de instancia

Se usan para acceder o modificar el estado de un objeto. Si usamos las variables de la instancia dentro de un método, tales métodos son llamados métodos de instancia. Es necesario que tengan como parámetro a "self". Tienen acceso a las variables de clase y a las variables de instancia.

**Son los métodos que hemos venido ejercitando.**

### Métodos de clase

Se usan para acceder o modificar el estado de la clase. Si usamos solo variables de la clase dentro de un método, tales métodos deben ser declarados como "métodos de clase". En efecto, no es necesario pasar parámetro "self". Solamente tienen acceso a las variables de clase.

### Métodos estáticos

Se usan para realizar tareas de forma aislada. Dentro del método no se usan ni variables de la instancia (por lo tanto, no le hace falta el parámetro "self"), ni variables de la clase. No tienen acceso a las variables de clase ni a las variables de instancia.

## 

## Métodos de clase

Los métodos de clase son métodos que se invocan en la propia clase, no en una instancia de un objeto específico. Por lo tanto, pertenece a un nivel de clase y todas las instancias de clase comparten un método de clase.

Puede modificar el estado de la clase cambiando el valor de una variable de clase que se aplicaría a todos los objetos de clase.

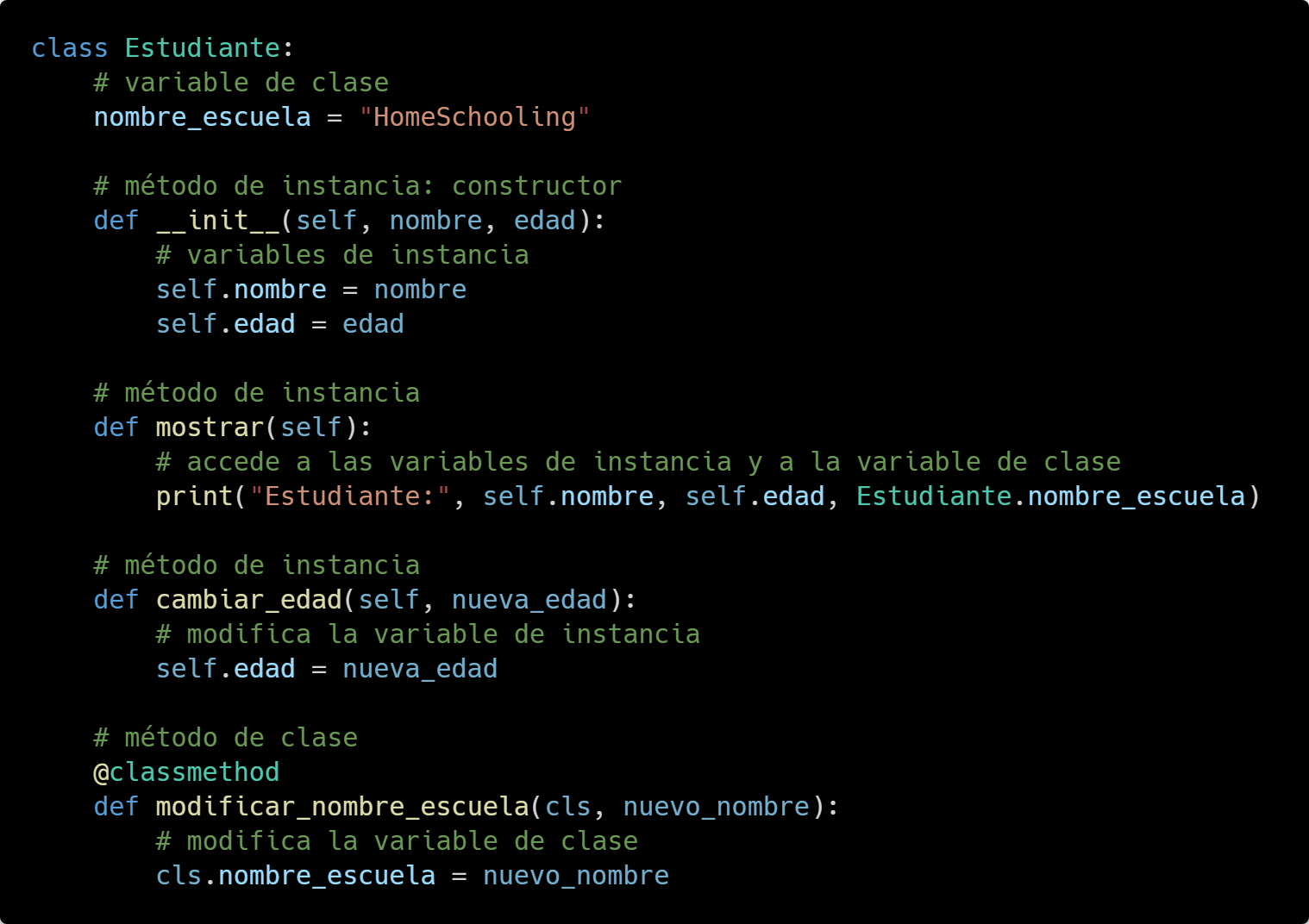
El método de clase tiene como primer parámetro a "cls", que hace referencia a la clase (parecido a "self" que hace referencia al objeto).

Para declarar un método de clase podemos usar el "decorador" @classmethod

@classmethod  
 def método\_de\_clase(cls, nuevo\_valor):  
 cls.variable\_de\_clase = nuevo\_valor

**¡MANOS A LA OBRA!**

**Ejercicio 1**



1. Crea una instancia de Estudiante.
2. Invoca mostrar()
3. Modifica la edad
4. Modifica el nombre de la escuela
5. Invoca mostrar()

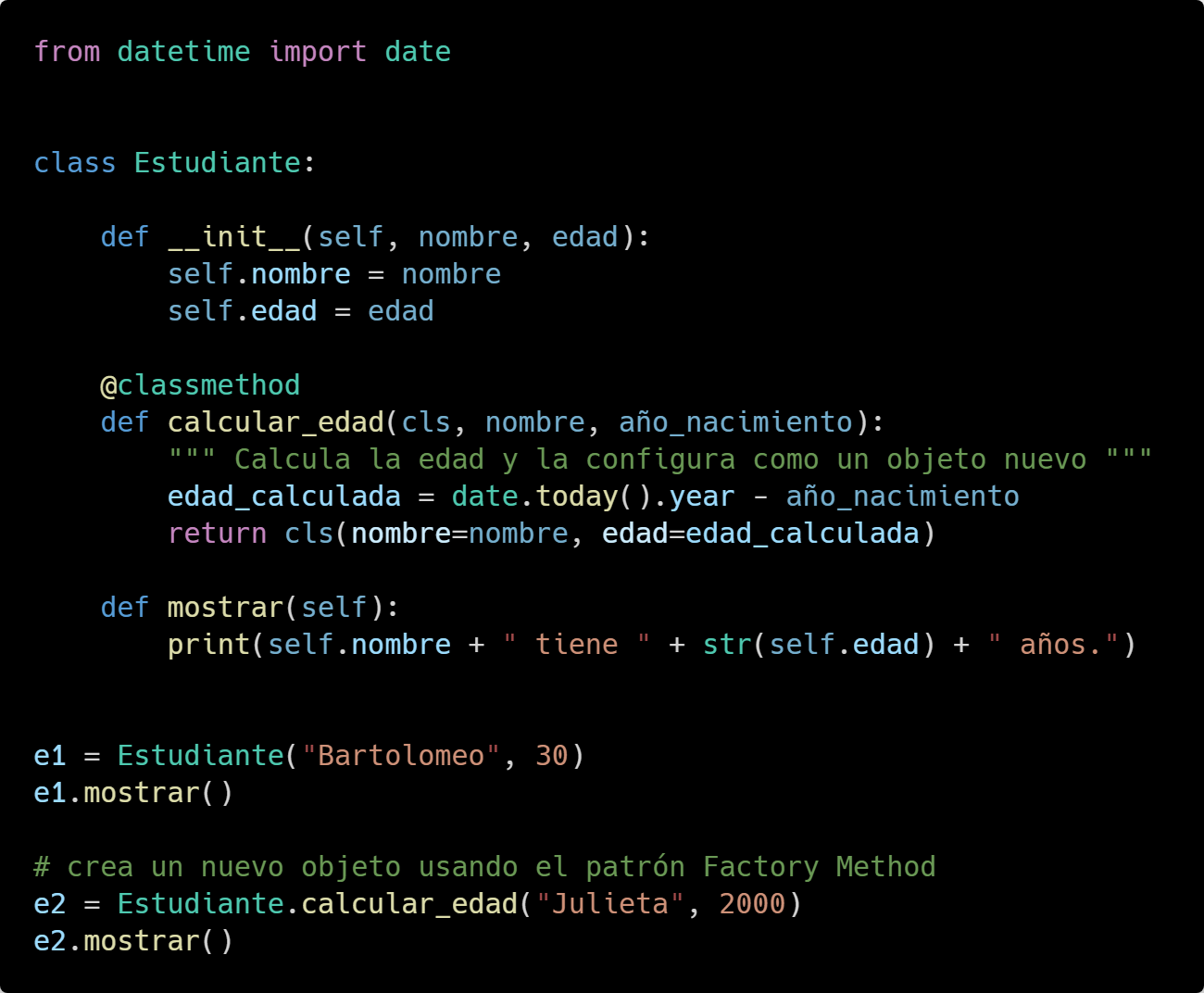
### Factory Method

Factoy Method es un patrón de diseño que crea objetos sin especificar sus clases concretas. Es un método que se utiliza para crear objetos en lugar de una llamada al constructor. Los métodos de clase sirven para tal propósito.

| Los patrones de diseño son técnicas para resolver problemas comunes en el desarrollo de software. Es importante que profundices en esto cuando domines Python. |
| --- |

**¡MANOS A LA OBRA!**

**Ejercicio 2**



1. Ejecuta el código de arriba.
2. Debate en el equipo qué hace el método today() del módulo "date" del paquete "datetime" que viene incorporado en Python.
3. Debate en el equipo cómo trabaja el Factory Method en este ejercicio.

## 

## Métodos estáticos

Un método estático está enlazado a la clase y no al objeto de la clase. Por lo tanto, podemos llamarlo usando el nombre de la clase.

Un método estático no tiene acceso a las variables de clase e instancia porque no recibe el argumento "self" o "cls". En consecuencia, no puede modificar el estado del objeto o clase.

Debemos decirle explícitamente a Python que es un método estático utilizando el decorador @staticmethod

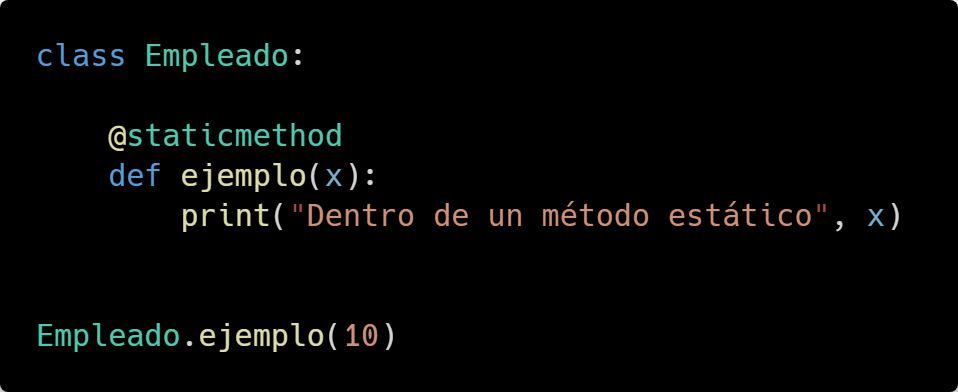
@staticmethod  
 def método\_estático():  
 ...

**Los métodos estáticos consumen menos memoria:** los métodos de instancia son objetos y crearlos tiene un costo. Tener un método estático evita eso. Supongamos que tienes diez objetos de "empleado", y si creas un método de instancia, Python tiene que crear diez copias de este método (separadas para cada objeto) que consumirán más memoria. Los métodos estáticos tienen solo una copia.

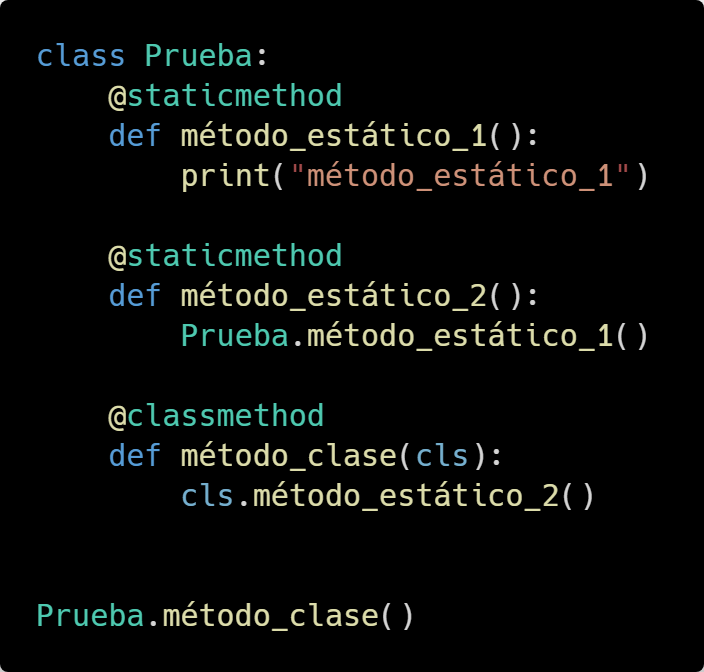
**Sirven para escribir funciones de utilidad:** los métodos estáticos tienen un uso limitado porque no tienen acceso a las variables de instancia y a las variables de clase. Sin embargo, pueden ser útiles como en la conversión de un tipo de datos a otro.

**Son legibles:** Al ver la parte superior del método, el decorador, sabremos que el método no depende del estado del objeto o del estado de clase.

**Ejecuta el siguiente código:**

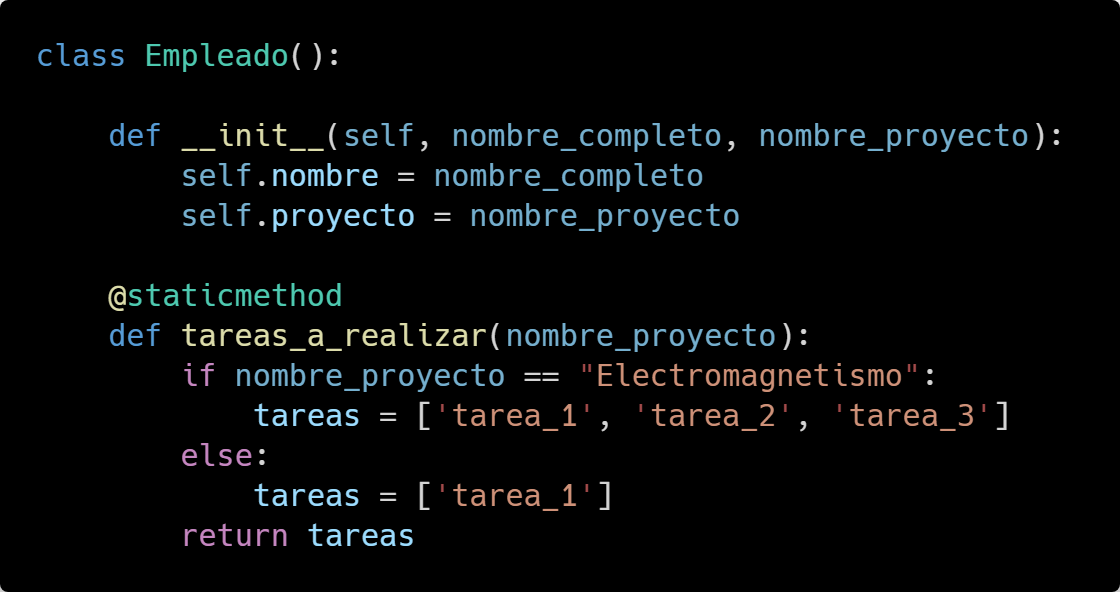


**Ejecuta el siguiente código:**



**¡MANOS A LA OBRA!**

#### Ejercicio 3



A partir del código de arriba:

* Crea un método de instancia "trabajar", que guardará en la variable "tareas" la devolución del método estático las tareas correspondientes, según el nombre del proyecto. Iterar la variable "tareas" e imprimir cada tarea: "Tarea completada: {tarea}"
* Crea un empleado y llámalo a trabajar en el proyecto "Electromagnetismo"

## Revisemos lo aprendido hasta aquí

* Crear y llamar métodos de clase
* Implementar el patrón de diseño Factory Method
* Crear y llamar métodos estáticos

# 

# Métodos especiales

Los "métodos especiales" son también llamados "métodos mágicos" o "dunders" (de doble underscores) porque tienen doble guion bajo al principio y al final de sus nombres. Son utilizados para crear nuevas funcionalidades.

Ya hemos usado \_\_init\_\_ que es llamado "constructor". Veremos algunos métodos más.

## \_\_str\_\_

Devuelve una representación string.

**Ejecuta el siguiente código:**

class Persona:  
 def \_\_init\_\_(self, nombre, apellido):  
 self.nombre = nombre  
 self.apellido = apellido

persona = Persona("Michael", "Faraday")  
 print(persona)

**Ejecuta el siguiente código:**

class Persona:  
 def \_\_init\_\_(self, nombre, apellido):  
 self.nombre = nombre  
 self.apellido = apellido

def \_\_str\_\_(self) -> str:  
 return f"{self.nombre} {self.apellido}"

persona = Persona("Michael", "Faraday")  
 print(persona)

¿Qué ha sucedido? Intenta cambiar la devolución de \_\_str\_**\_**

## \_\_repr\_\_

**Ejecuta el siguiente código:**

class Persona:  
 lista\_personas: list["Persona"] = []

def \_\_init\_\_(self, nombre, apellido):  
 self.nombre = nombre  
 self.apellido = apellido  
 Persona.lista\_personas.append(self)

def \_\_str\_\_(self) -> str:  
 return f"{self.nombre} {self.apellido}"

persona\_1 = Persona("Michael", "Faraday")  
persona\_2 = Persona("André", "Ampère")  
persona\_3 = Persona("Alessandro", "Volta")

print(Persona.lista\_personas)

¿Por qué no funciona \_\_str\_\_? Agrega el siguiente método y vuelve a probar:

def \_\_repr\_\_(self) -> str:  
 return f"{self.nombre} {self.apellido}"

\_\_repr\_\_ es la representación del objeto en formato string y sirve para el programador, para que pueda depurar sin ambigüedades, ya que \_\_str\_\_ podría presentarlas. En este sentido, conviene agregar el nombre de la clase y crear algo más reutilizable como lo siguiente:

def \_\_repr\_\_(self) -> str:  
 return f"{self.\_\_class\_\_.\_\_name\_\_}{self.\_\_dict\_\_}"

## \_\_add\_\_

**Ejecuta el siguiente código:**

class Producto:  
 def \_\_init\_\_(self, nombre: str, cantidad: int):  
 self.nombre: str = nombre  
 self.cantidad: int = cantidad

p1 = Producto(nombre="Teclado", cantidad=3)  
p2 = Producto(nombre="Mouse", cantidad=4)  
total\_cantidad\_productos = p1 + p2  
print(total\_cantidad\_productos)

**¿TypeError?**

**Ejecuta el siguiente código:**

class Producto:  
 def \_\_init\_\_(self, nombre: str, cantidad: int):  
 self.nombre: str = nombre  
 self.cantidad: int = cantidad

def \_\_add\_\_(self, producto: "Producto"):  
 suma\_cantidades = self.cantidad + producto.cantidad  
 return suma\_cantidades

p1 = Producto("Teclado", 3)  
p2 = Producto("Mouse", 4)  
total\_cantidad\_productos = p1 + p2  
print(total\_cantidad\_productos)  
print(p1.cantidad)  
print(p2.cantidad)

**\_\_add\_\_** permite que usemos el operador + en nuestra clase.

| Si quieres hacer lo siguiente:  lista\_productos = [p1, p2]  total\_cantidad\_productos = sum(lista\_productos)  Entonces debes implementar \_\_radd\_\_ en la clase de la siguiente forma:  def \_\_radd\_\_(self, otro):  if otro == 0:  return self  else:  return self.\_\_add\_\_(otro) |
| --- |

## Más métodos especiales

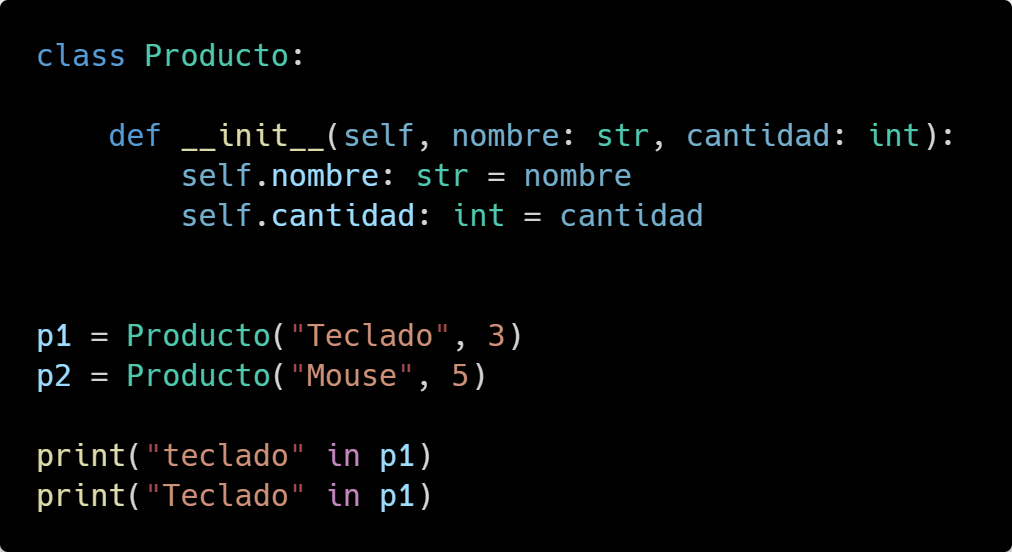
Visita: <https://docs.python.org/es/3/reference/datamodel.html#special-method-names>

Algunos métodos especiales:

| \_\_sub\_\_ | - |
| --- | --- |
| \_\_mul\_\_ | \* |
| \_\_truediv\_\_ | / |
| \_\_floordiv\_\_ | // |
| \_\_mod\_\_ | % |
| \_\_pow\_\_ | \*\* |
| \_\_and\_\_ | & |
| \_\_or\_\_ | | |
| \_\_it\_\_ | < |
| \_\_le\_\_ | <= |
| \_\_eq\_\_ | == |
| \_\_ne\_\_ | != |
| \_\_gt\_\_ | > |
| \_\_ge\_\_ | >= |
| \_\_len\_\_ | len() |
| \_\_getitem\_\_ | para indexar |
| \_\_setitem\_\_ | para asignar valores indexados |
| \_\_delitem\_\_ | para borrar valores indexados |
| \_\_iter\_\_ | para iteración sobre objetos |
| \_\_contains\_\_ | para implementar operadores de prueba de pertenencia |

**¡MANOS A LA OBRA!**

**Ejercicio 4**



* Define el método especial \_\_contains\_\_ para que el primer print muestre False, y el segundo print muestre True. (Recuerda que el método se declara dentro de la clase, recibe como parámetro self, y devuelve True o False)

## Revisemos lo aprendido hasta aquí

* Implementar los métodos especiales para mejorar el comportamiento de nuestras clases

# 

# Encapsulación

### ¿Qué es la encapsulación en Python?

La encapsulación en Python consiste en agrupar datos y métodos dentro de una sola unidad. Entonces, cuando creas una clase, significa que estás implementando la encapsulación. Una clase es un ejemplo de encapsulación, ya que enlaza todos los atributos y métodos en una sola unidad.

Usando la encapsulación, podemos ocultar la representación interna de un objeto desde el exterior, en otras palabras, es ocultación de información.

Puedes restringir el acceso a variables y métodos y así evitar la modificación accidental de datos. Lo puede hacer mediante la creación de **variables y métodos privados** dentro de una clase.

En Python, no tenemos, como en otros lenguajes, modificadores de acceso directo como "público", "privado" y "protegido". Podemos lograr esto mediante el uso de guiones bajos simples y guiones bajos dobles.

| En Python las propiedades y métodos privados no existen como existen en otros lenguajes, por lo que son fácilmente sobreescribibles, sin embargo, hay formas de proteger los datos. |
| --- |

**Miembro público:** Accesible en cualquier lugar, incluso fuera de la clase.

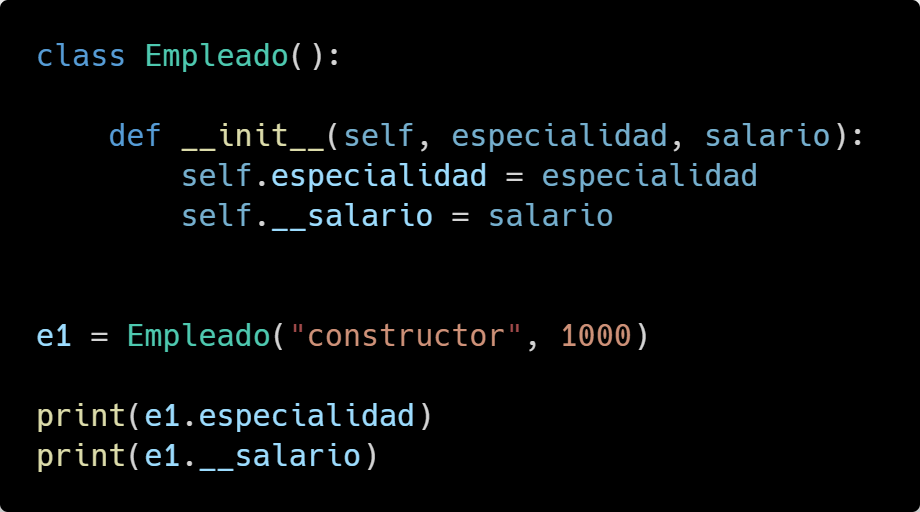
**Miembro privado:** Accesible solamente dentro de la clase: \_\_privado

**Miembro protegido:** Accesible dentro de la clase y sus subclases: \_protegido

| Es probable que los veas utilizar en programas que recurren a muchas clases. De lo contrario, resulta, en realidad, innecesario. |
| --- |

## Atributos y métodos privados

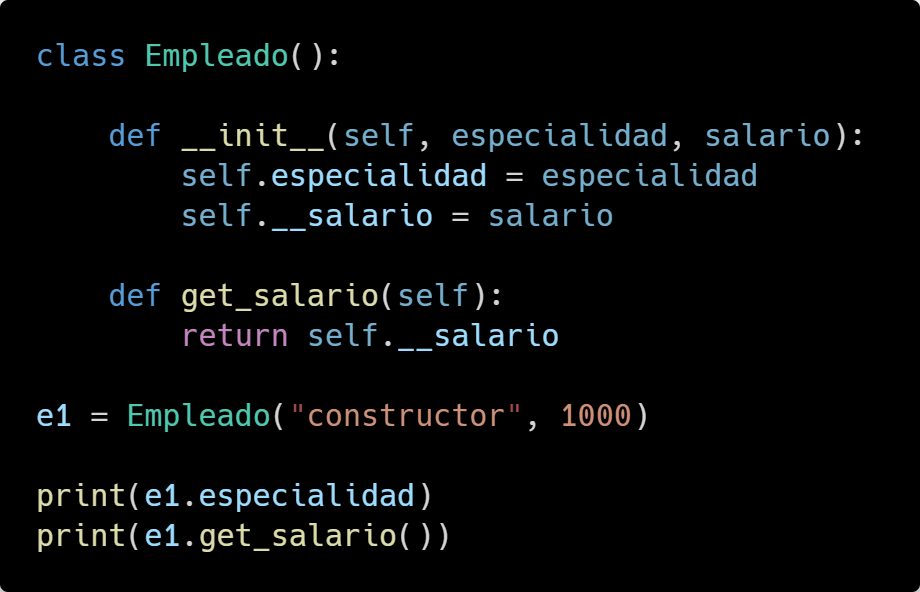
**Ejecuta el siguiente código:**



**¿AttributeError?**

Necesitas un método para **obtener** el valor del atributo. Puedes llamar al método como quieras, pero, por costumbre, suele comenzar con la palabra inglesa **get:**

**Ejecuta el siguiente código:**



Ahora prueba con cambiar el salario y mostrar su valor ¿Veremos un AttributeError?

e1.\_\_salario = 2000  
 print(e1.\_\_salario)

¿Algo anda mal? Deberías ver qué hay dentro de e1...

print(e1.\_\_dict\_\_)

Python **ha creado un nuevo atributo público** en forma dinámica... Eso lo has hecho con la instrucción e1.\_\_salario = 2000. Pero **ha conservado el atributo privado** de una manera especial: ha creado un identificador con:  
 - un guión bajo,  
 - el nombre de la clase,  
 - y luego el nombre del atributo privado.

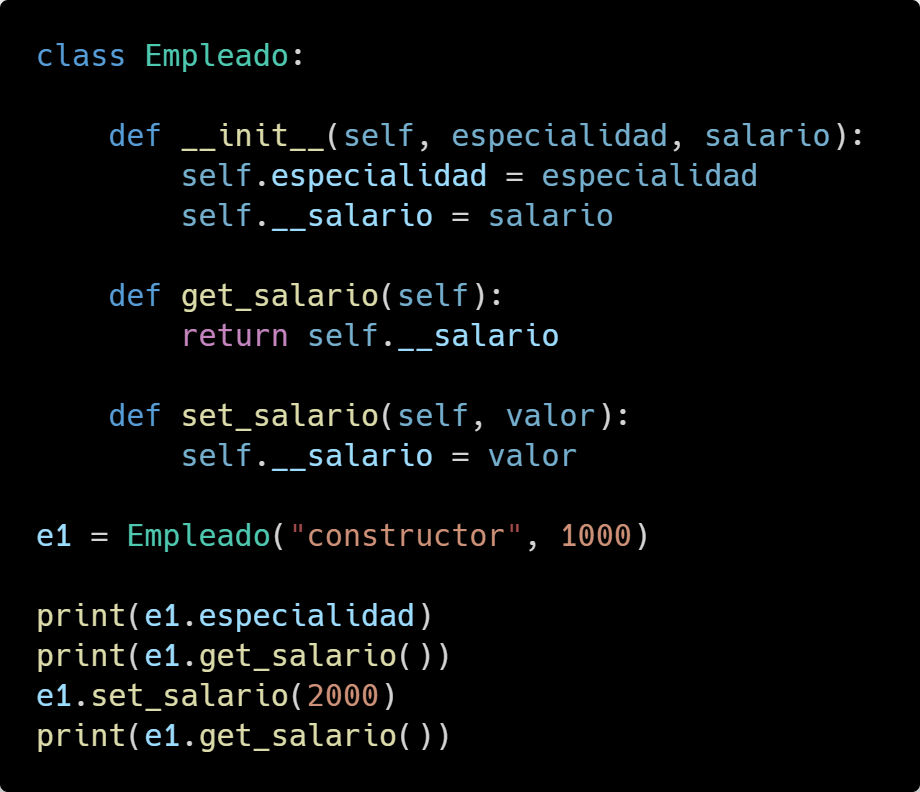
Puedes acceder a él de la misma forma:

print(e1.\_Empleado\_\_salario)

| El editor de código te mostrará un error, pero aun así puedes ejecutar el archivo. |
| --- |

Esto significa que deberías crear un método para **modificar,** para poder cambiar un valor en el atributo privado. Puedes llamar al método como quieras, pero, por costumbre, suele comenzar con la palabra inglesa **set:**

**Ejecuta el siguiente código:**



### Getters y Setters en Python

Para implementar una encapsulación adecuada en Python, necesitamos usar setters y getters. El propósito principal del uso de getters y setters en programas orientados a objetos es garantizar la encapsulación de datos. Utiliza métodos getters para **obtener** acceso a los miembros de datos y los métodos setter para **modificar** los miembros de datos.

En Python, las variables privadas no son campos ocultos como en otros lenguajes de programación. Los métodos getters y setters se utilizan a menudo cuando:

* Cuando queremos evitar el acceso directo a variables privadas.
* Para agregar lógica de validación cuando establecemos un valor.

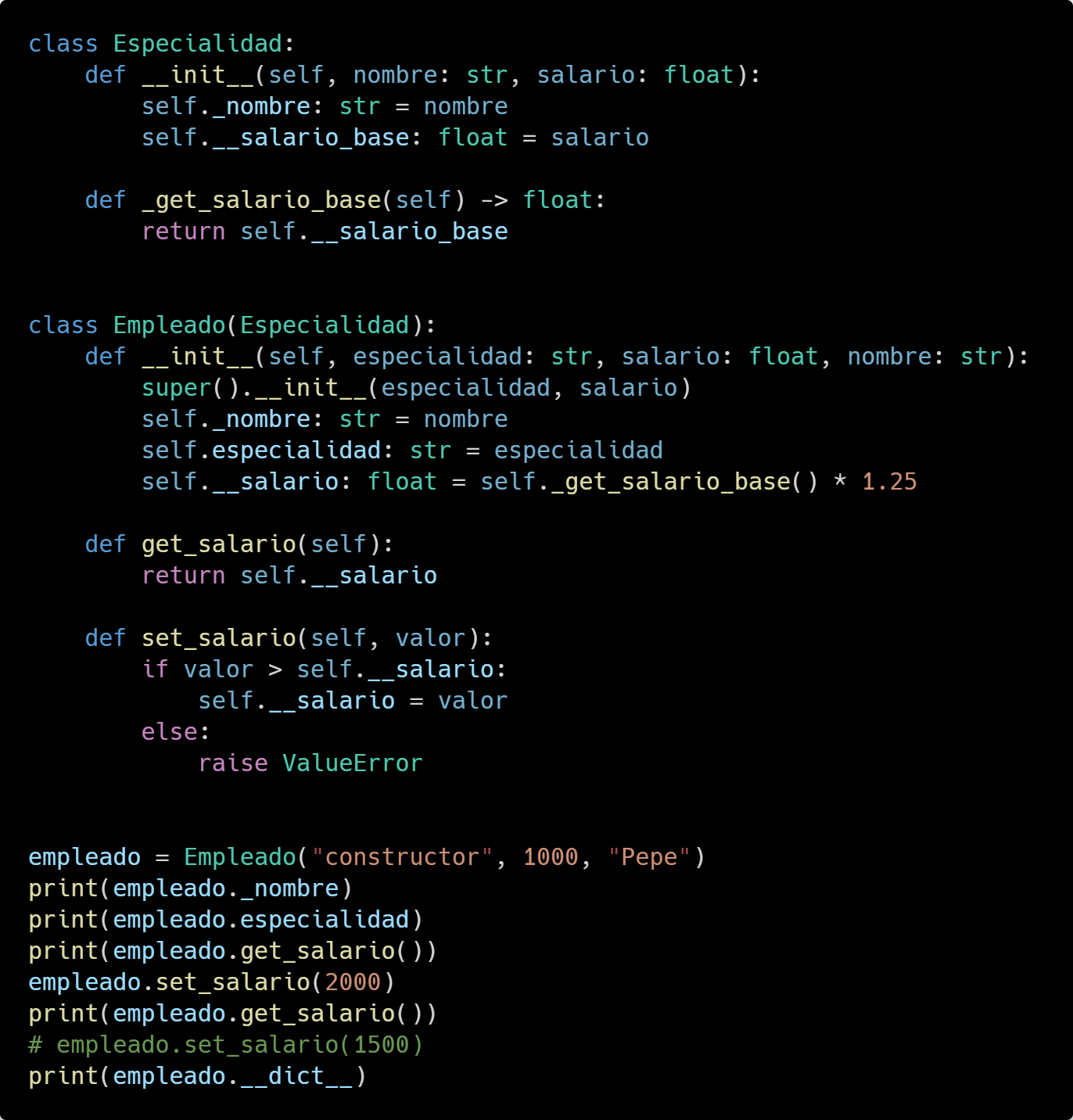
### 

### 

**¡MANOS A LA OBRA!**

**Ejercicio 5**

Examina y explica qué hace el siguiente código, y prueba, con la ayuda del editor, ir cambiando los atributos y métodos públicos a privados o protegidos, y viceversa. Verifica sus efectos en la herencia, y sus efectos cuando quieres acceder a ellos fuera de la clase y dentro de la clase.



### Ventajas de la encapsulación

**Seguridad:** La principal ventaja de usar la encapsulación es la seguridad de los datos. La encapsulación protege un objeto del acceso no autorizado. Permite niveles de acceso privados y protegidos para evitar la modificación accidental de datos.

**Ocultamiento de datos:** El usuario no debería saber lo que está sucediendo detrás de escena. Solo debería saber que, para modificar un atributo, debe llamar al método setter; y, para leer un atributo de datos, llamar al método getter. Lo que estos métodos setter y getter están haciendo, está oculto para el usuario.

**Simplicidad:** Simplifica el mantenimiento de la aplicación al mantener las clases separadas y evita que se sobrescriban atributos entre sí.

## Revisemos lo aprendido hasta aquí

* Encapsular usando atributos y métodos protegidos y privados
* Acceder y modificar atributos protegidos y privados
* Entender la importancia de la encapsulación en clases que pueden ser reutilizadas

## 

## Propiedades

Las "propiedades" son atributos que manejamos mediante getters, setters y deleters.

Las propiedades nos permiten llamar código personalizado cuando una variable o método es leído, escrito, o eliminado.

La función integrada property() nos permitirá interceptar la escritura, lectura, eliminación. Además, nos permitirá incorporar documentación sobre los mismos. La sintaxis para invocarla es la siguiente:

@property

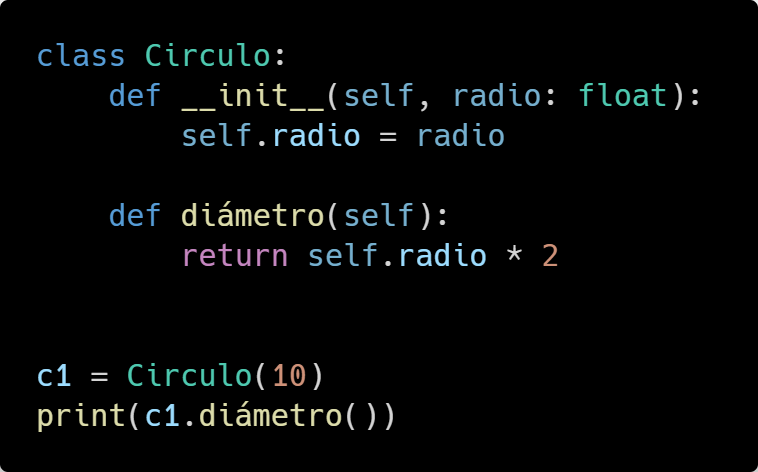
Una vez creada la propiedad, puedes utilizar los siguientes métodos:

.getter: Interceptará la lectura

.setter: Interceptará la escritura

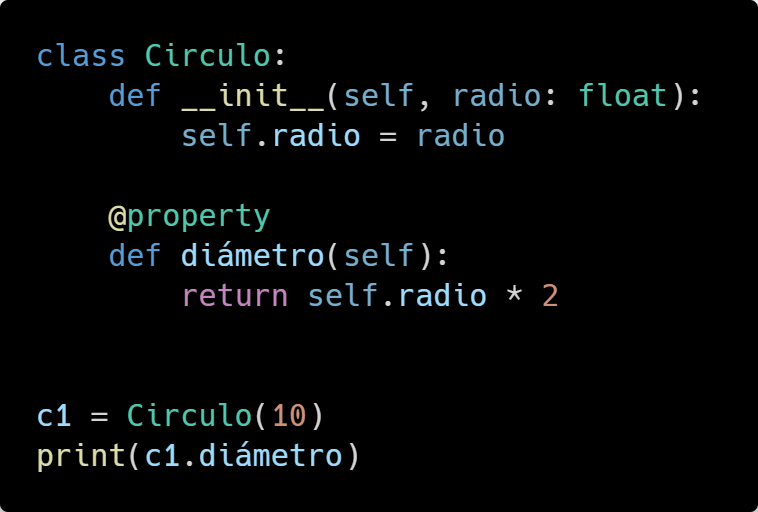
.deleter: Interceptará la eliminación

**Ejecuta el siguiente código:**



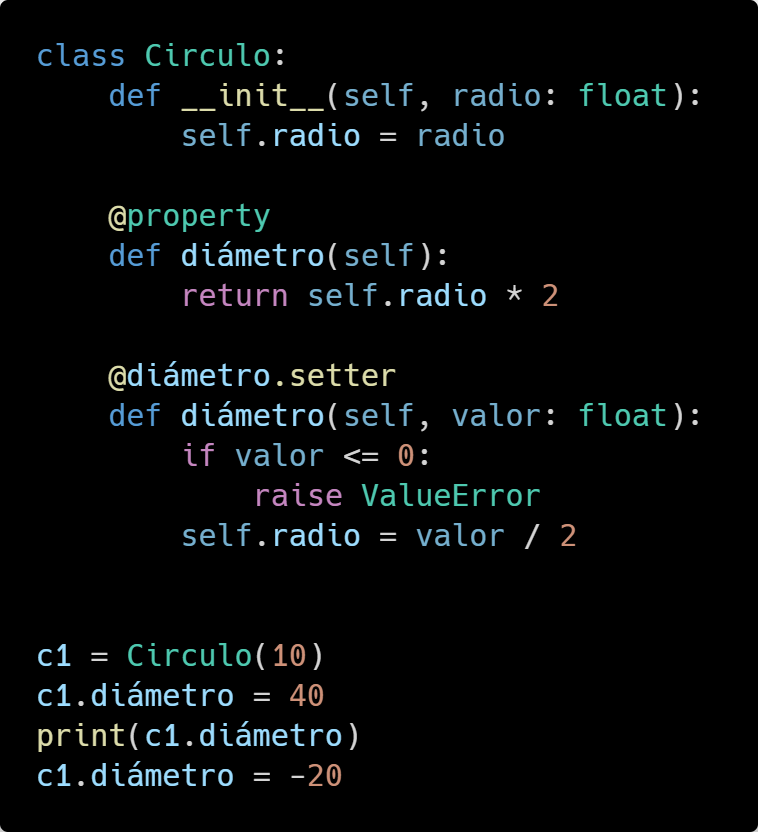
En este ejemplo "común", observa cómo llamas a diámetro.

**Ejecuta el siguiente código:**



En este ejemplo, con una "propiedad", observa cómo llamas a diámetro.

**Ejecuta el siguiente código:**



Observa cómo con la propiedad "diámetro" y su método setter, ahora puedes agregar lógica de forma muy simple.

#### 

**¡MANOS A LA OBRA!**

**Ejercicio 6**

### 

Ejecuta el programa y sigue las siguientes instrucciones y observa el comportamiento:

1. Crea una instancia de Persona llamada p1 con un nombre cualquiera
2. p1.nombre
3. p1.nombre = "Otro Nombre"
4. p1.nombre
5. del p1.nombre
6. p1.nombre
7. p1.nombre = " "
8. Posiciona el cursor del mouse sobre "nombre" en la definición del método, justo debajo del decorador @property, presiona F2 y cambia el nombre del método a otro y ejecuta.

## Revisemos lo aprendido hasta aquí

* Crear propiedades en Python
* Usar el método getter, setter y deleter de las propiedades
* Crear métodos personalizados a través de las propiedades

# 

# Dataclasses

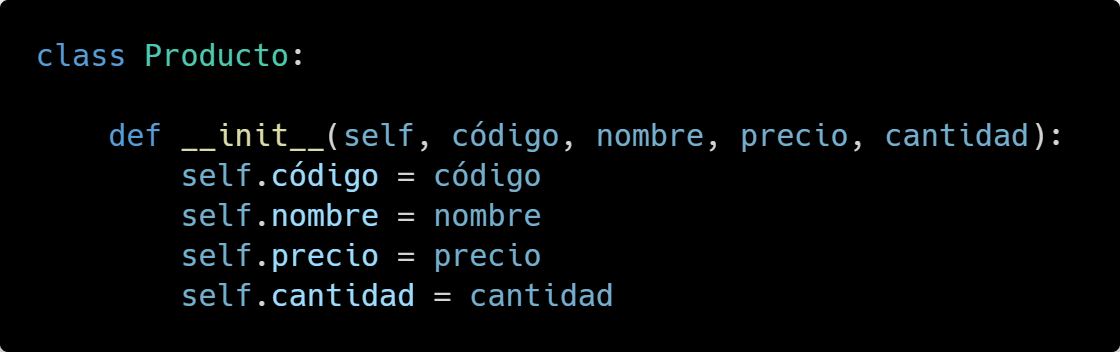
A partir de la versión 3.7, Python introdujo el módulo dataclasses, una nueva característica para definir clases. Tiene estos beneficios, entre otros:

* Menos código repetitivo
* Mejor legibilidad del código
* Mejor mantenibilidad del código

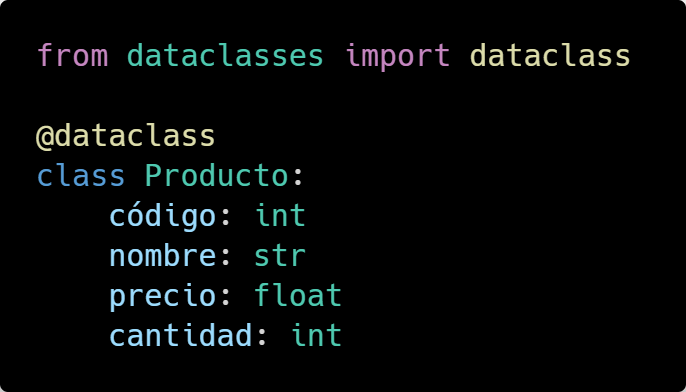
A veces definimos clases que solo actúan como contenedores de datos, y cuando lo hacemos, pasamos mucho tiempo escribiendo código repetitivo con muchos argumentos. Dataclasses alivia este problema, al tiempo que proporciona métodos útiles adicionales.

## @dataclasses

**Ejecuta el siguiente código:**



**Ejecuta el siguiente código:**



Algunas cosas a tener en cuenta sobre esta sintaxis:

Hay menos código repetido: definimos cada atributo una vez y "no nos repetimos".

Utilizamos anotaciones de tipo para cada atributo. Aunque esto no impone la validación de tipos, ayuda a tu editor de texto a proporcionarte una mejor ayuda, o si usas un verificador de tipos, como MyPy. Tu código seguiría funcionando si no respetas los tipos de datos, pero tu editor de código señalará las inconsistencias.

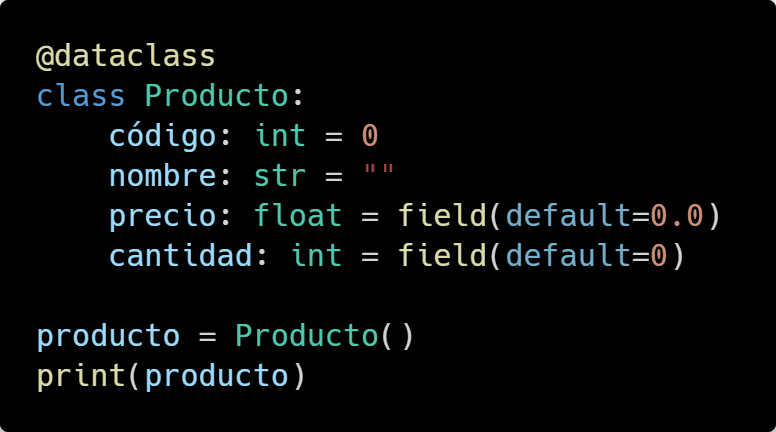
Prueba crear una instancia "producto" de la clase Producto, y ejecutando lo siguiente, verás que ya creó para ti el método \_\_repr\_\_:

print(producto)

## Valores predeterminados

Puedes agregar valores predeterminados a cada atributo mientras conservas la anotación. Puedes hacerlo de dos forma, la tradicional, o agregando field(default= )

**Ejecuta el siguiente código:**



Ten en cuenta que, si no usas **field()**, los campos sin valores predeterminados no pueden estar después de los campos con valores predeterminados.

En las herencias, si en la clase padre hay valores predeterminados (opcionales) y en la clase hija hay valores no predeterminados (obligatorios) . Cuando se mezclan, puede aparece este error. Se puede solucionar agregando lo siguiente al decorador:

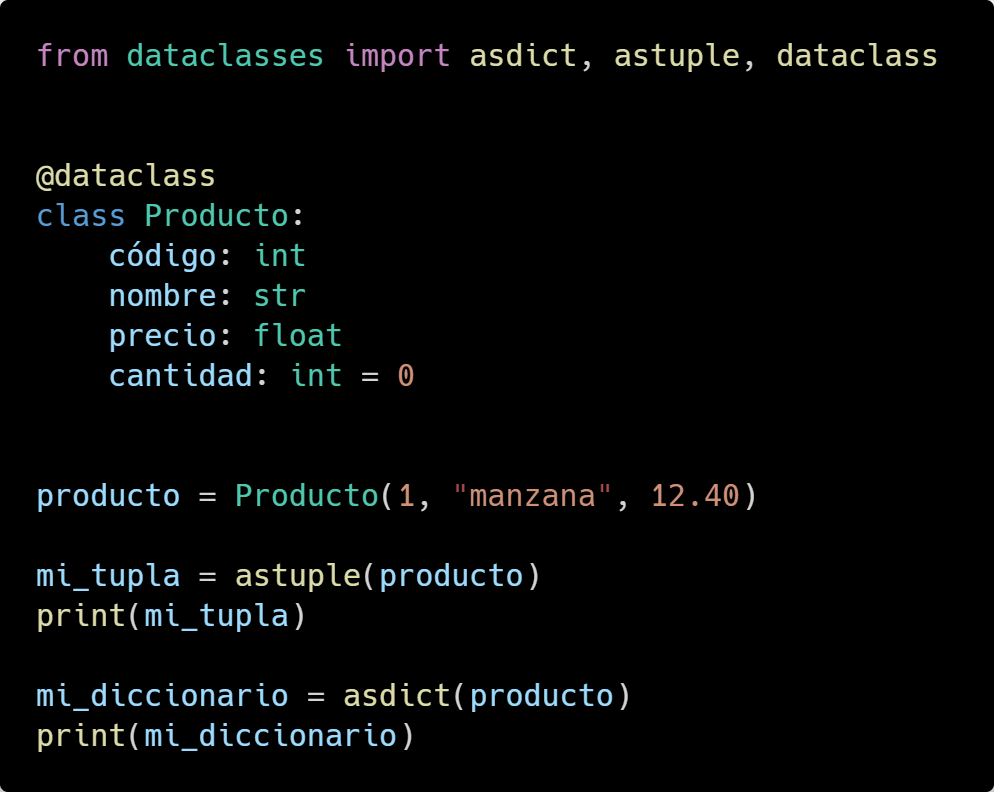
@dataclass(kw\_only=True)

También puedes crear objetos de "solo lectura". Tienes que hacer lo siguiente:  
@dataclass(frozen=True)

## Serialización a tuplas o diccionarios

Las instancias se pueden "serializar", es decir, convertir los atributos de un objeto a un tipo de datos de Python fácilmente, a un diccionario o a una tupla.

**Ejecuta el siguiente código:**

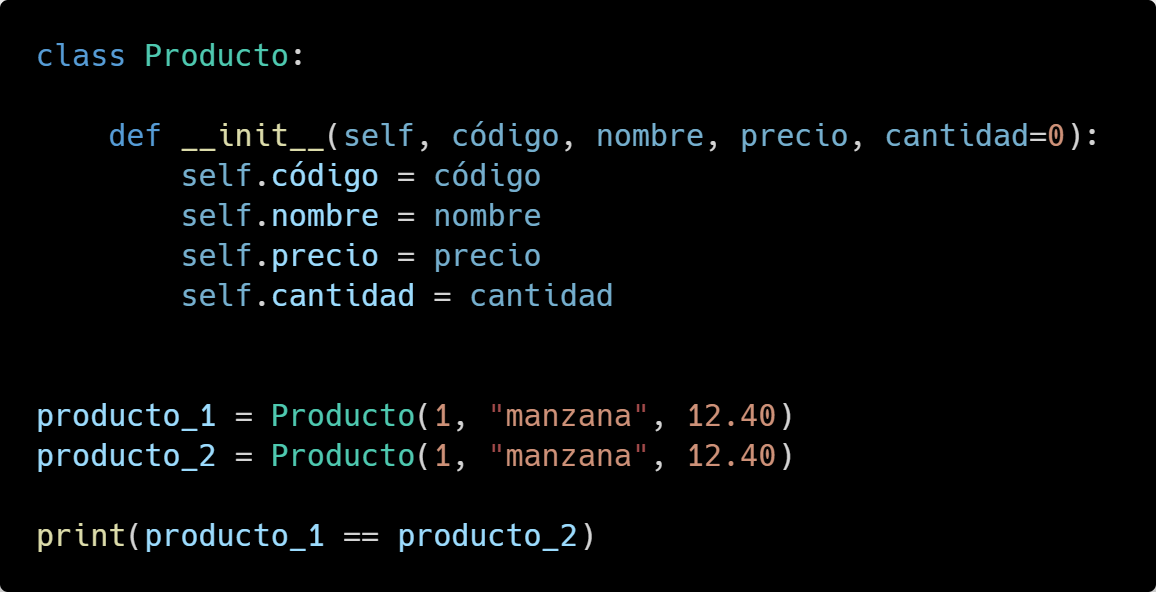


| Esto es muy útil cuando tu código interactúa con otros programas que esperan estos formatos. |
| --- |

## Comparar objetos

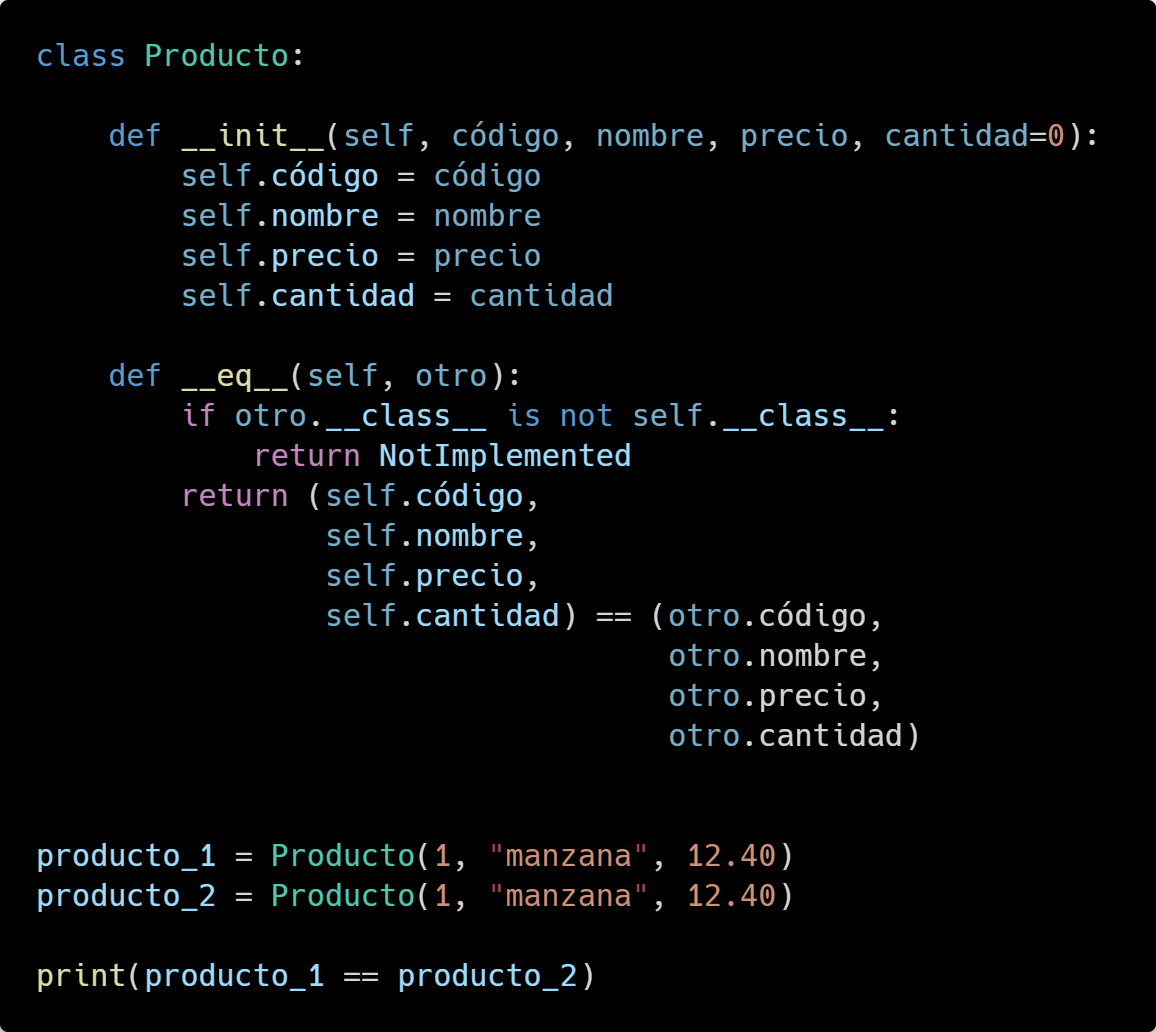
Cuando defines una clase utilizando la sintaxis estándar de Python, y pruebas la igualdad entre dos instancias que tienen los mismos valores de atributo, obtendrías en el print, False:

**Ejecuta el siguiente código:**



Estos dos objetos son iguales en su contenido, pero la comparación devuelve False porque la comparación no está implementando el método \_\_eq\_\_. Para agregarlo, tendrías que implementar el método de la siguiente manera, y se imprimirá True.

**Ejecuta el siguiente código:**



El método creado comprueba primero que los dos objetos son instancias de la misma clase y, a continuación, prueba la igualdad entre tuplas de atributos.

Ahora, si decides agregar más atributos a tu clase, tendrías que actualizar el método nuevamente. Y si quieres implementar más comparaciones, \_\_ge\_\_, \_\_gt\_\_, \_\_le\_\_, \_\_lt\_\_, deberías implementar un método para cada uno de ellos.

Parece mucho código innecesario, ¿verdad? Por suerte, dataclasses lidia con todo esto.

**Ejecuta el siguiente código**

:

## 

## \_\_post\_init\_\_

En algunas situaciones, es posible que debas crear un atributo que solo se defina internamente, y no cuando se crea una instancia de la clase. Este puede ser el caso de un atributo que tiene un valor que depende de atributos establecidos previamente.

Aquí es donde deberías usar la función field() de dataclasses. Debes importarla. Para que el atributo no se inicialice al crear una instancia, harás field(init=False)

A este campo al que le asignamos field, para asignarle un valor después de la inicialización, hay que invocar al método \_\_post\_init\_\_ Como su nombre lo indica claramente, este método se llama justo después de que se llama al método \_\_init\_\_.

**Ejecuta el siguiente código:**



Ten en cuenta que el argumento dentro de la función se ha establecido para que sea visible cuando se imprima el objeto. Si no quieres que se imprima precio\_final, modifica la línea:

precio\_final: float = field(init=False, repr=False)

## 

## Ordenar objetos

Una característica útil que debes tener cuando se trata de objetos que contienen datos, es la capacidad de compararlos y ordenarlos en el orden que desees.

De forma predeterminada, dataclasses lo implementa . Para permitir los otros tipos de comparación (menor que, menor o igual, mayor que y mayor o igual), tenemos que establecer el argumento en el decorador de la siguiente forma:

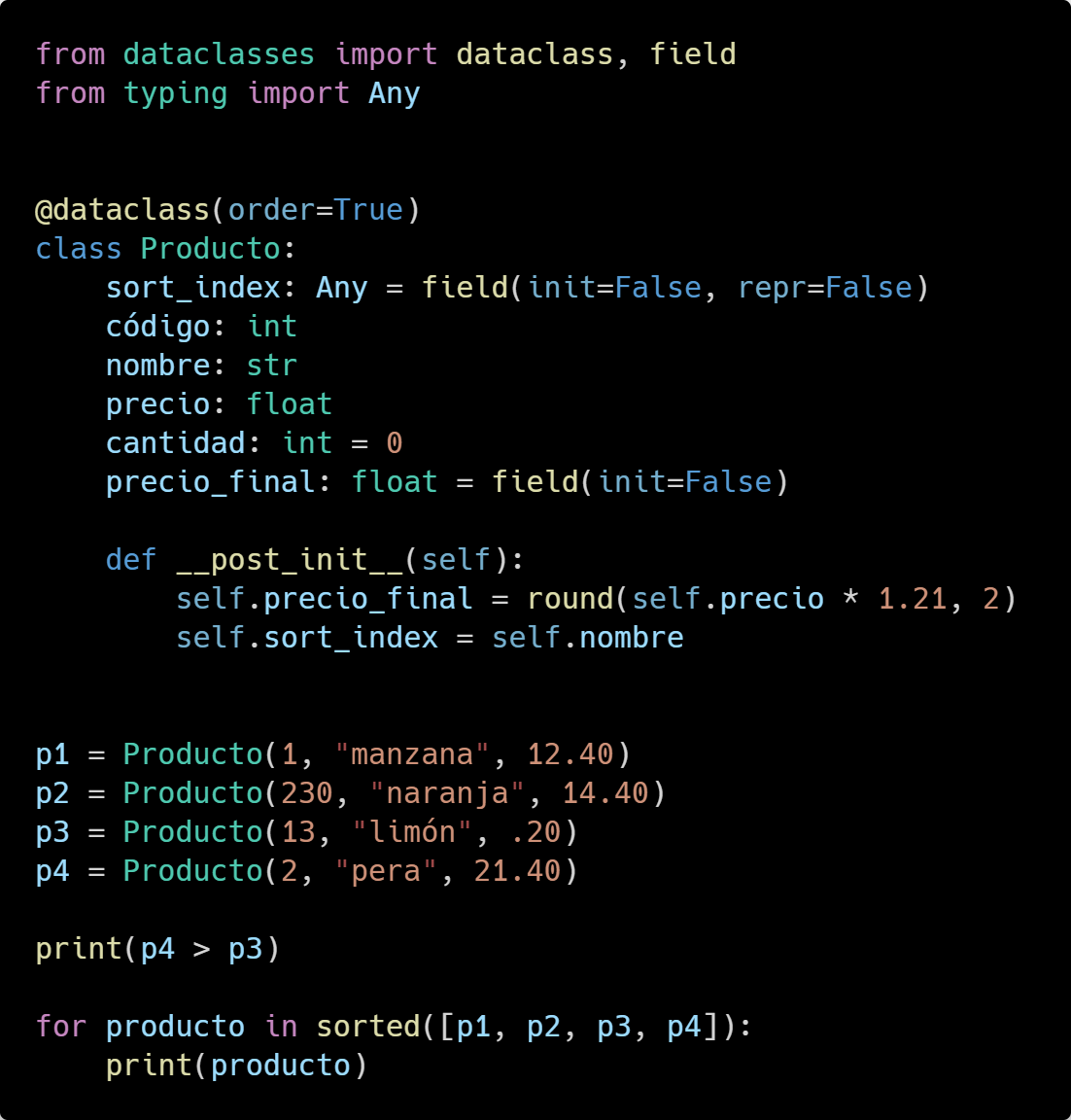
@dataclasses(order=True)

La forma en que se implementan estos métodos de comparación consiste en tomar cada campo definido y los compara en el orden en que se definen hasta que haya un valor que no sea igual.

Comparar las instancias, no campo por campo, sino por un atributo específico, tiene más sentido. Para esto, añade como **primer campo** el atributo de dataclasses llamado: **sort\_index**

Luego, en \_\_post\_init\_\_ debes asignarle el atributo por el cual vas a ordenar la clase.

**Ejecuta el siguiente código:**

****

| No olvides que sort\_index debe ir al principio. Prueba ir cambiando  self.sort\_index = self.nombre por self.sort\_index = precio\_final  u otros atributos. |
| --- |

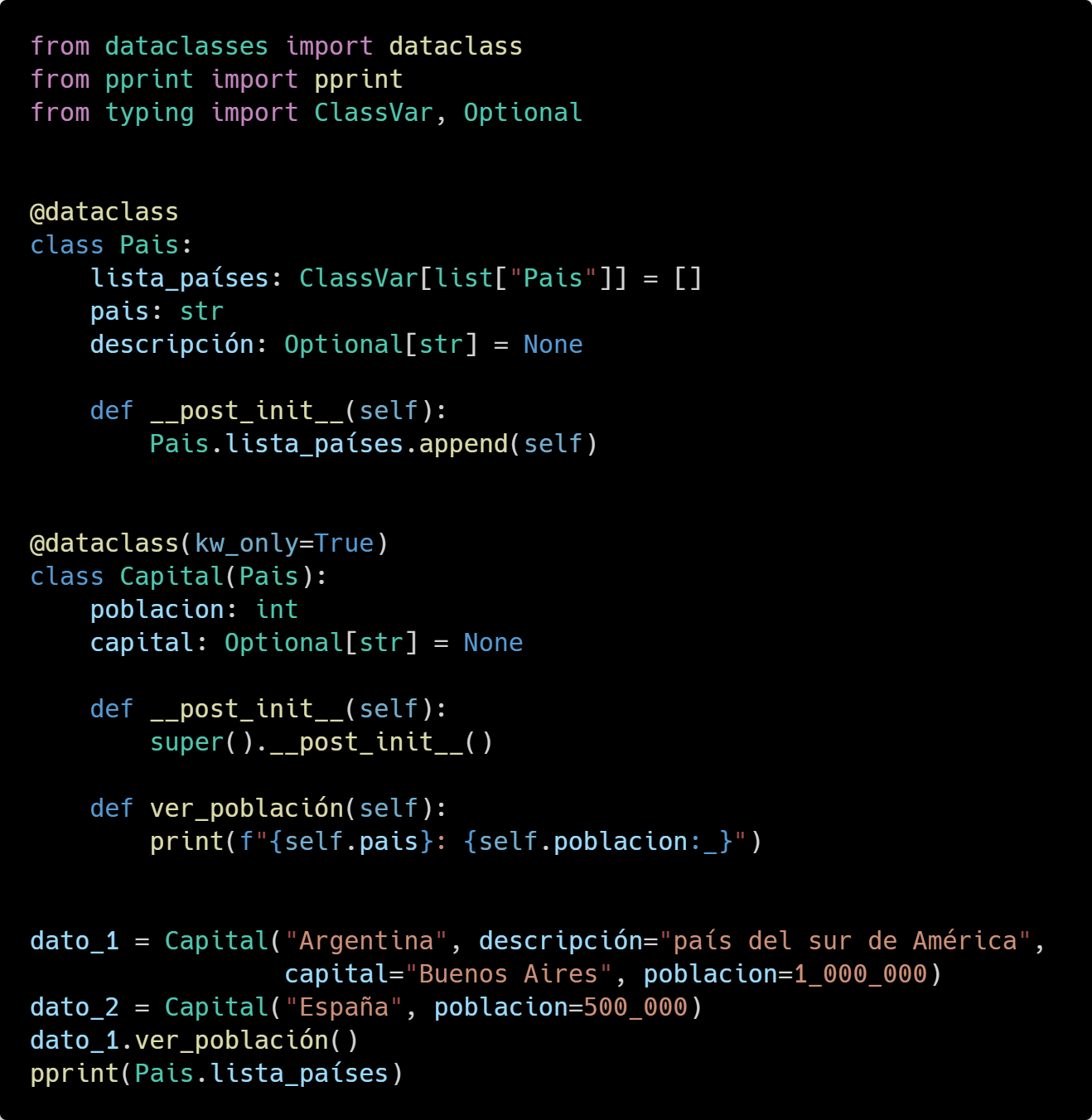
## Herencia y variables de clase

Para crear variables de clase, utiliza la anotación de tipo **ClassVar** del módulo typing.

Para heredar sin el problema de sobreponer atributos predefinidos y no predefinidos, modifica el decorador: @dataclass(kw\_only=True)

Y, para heredar el comportamiento de \_\_post\_init\_\_ de la clase padre, utiliza el método: super().\_\_post\_init\_\_()

**Ejecuta el siguiente código:**



**¡MANOS A LA OBRA!**

**Ejercicio**

Usando dataclasses, crea una clase Estudiante que tenga al menos 4 atributos, además, la clase Profesor con al menos 4 atributos, que herede de Persona, que también tenga al menos 4 atributos. Crea 4 instancias de cada estudiante y profesor. Ordénalos por su edad. Serializa sus datos a diccionarios. Muestra sus diccionarios

## Revisemos lo aprendido hasta aquí

* Facilitar la creación de clases con dataclasses
* Crear clases congeladas con @dataclass(frozen=True)
* Serializar objetos con astuple() y asdict()
* Usar field, \_\_post\_init\_\_, super().\_\_post\_init\_\_()
* Variables de clase con la anotación ClassVar
* Heredar fácilmente con @dataclass(kw\_only=True)
* Ordenar objetos en la definición de la clase con @dataclasses(order=True) y sort\_index