

Python / Guía 8

Programación Orientada a Objetos I

## 

## 

## OBJETIVOS DE LA GUÍA

En esta guía aprenderemos a:

* Implementar la programación orientada a objetos

creando clases y objetos

* Crear un constructor para inicializar un objeto Python
* Crear variables de instancia, modificarlas y acceder a ellas
* Crear y llamar métodos de instancia
* Implementar herencia de clases
* Implementar la técnica de agregación y composición

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

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

# 

# ¿Qué es la programación orientada a objetos?

La programación orientada a objetos (POO) es un paradigma de programación basado en el concepto de "objetos". El objeto contiene datos y código: datos en forma de propiedades, conocidos como **atributos**; y código, en forma de acciones que el objeto puede realizar, conocidas como **métodos.**

Un aspecto importante de la POO en Python es crear **código reutilizable** utilizando el concepto de herencia. Este concepto también se conoce como DRY (Don't Repeat Yourself).

| En tu proyecto, puedes mezclar POO y programación funcional. ¡Funciones y clases no se oponen! |
| --- |

## ¿Qué son las clases y objetos?

En Python, todo es un objeto. Una **clase** es como un "plantilla" para el objeto. Para crear un **objeto** vamos a requerir una plantilla, es decir, una clase.

Cada clase y cada objeto pueden tener atributos (propiedades) y métodos (acciones) diferentes. Un ejemplo de la vida real puede ser el siguiente:

Clase Persona:  
 Atributos: nombre, sexo, profesión  
 Métodos: trabajo, estudio

Objeto 1: persona\_1

Atributos:  
 Nombre: Carmen  
 Sexo: mujer  
 Profesión: ingeniera de software  
 Métodos:  
 Trabajo: está trabajando como desarrolladora de software en ABC  
 Estudio: estudia dos horas al día

Objeto 2: persona\_2

Atributos:  
 Nombre: Juan  
 Sexo: hombre  
 Profesión: médico  
 Métodos:  
 Trabajo: está trabajando como médico  
 Estudio: estudia 5 horas al día

### ¿Cuándo usar clases y objetos?

Generalmente cuando representas:

* Una estructura de datos
* Comportamientos

Iremos viendo de a poco sus usos...

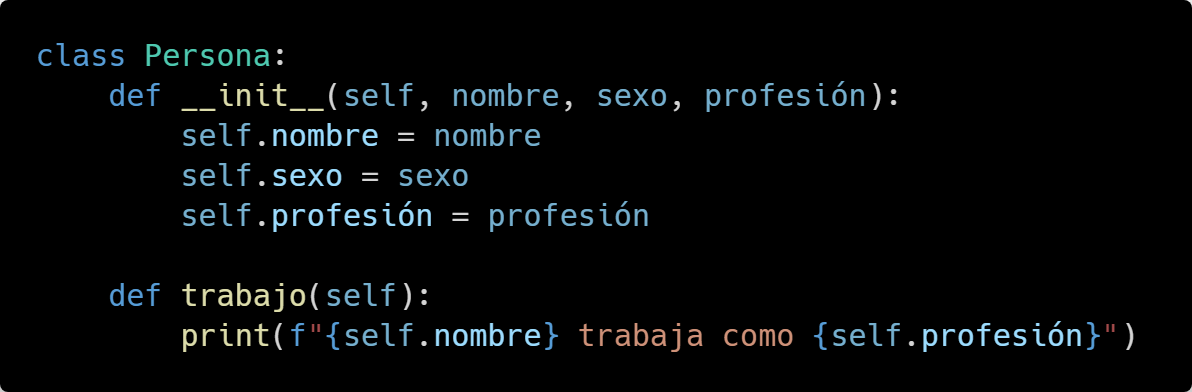
## 

## ¿Cómo creo clases y objetos en Python?

class NombreDeClase:  
 """ Docstrings """  
 <atributos y métodos>

| La comunidad de Python recomiendo el uso de CamelCase para el nombre de las clases, es decir, la primera letra de cada palabra en mayúsculas sin guiones. |
| --- |

**Ejecuta el siguiente código:**



Verás que hay dos funciones dentro de la clase: son los métodos. Está el método "trabajo", y está el método "\_\_init\_\_", que es un método especial de Python, llamado **constructor**. Lo que hace es inicializar un objeto de una clase.

Las variables que están dentro del método \_\_init\_\_ se llaman **variables de instancia** (también atributos de instancia).

La función "trabajo" es llamada **método de instancia,** o simplemente método.

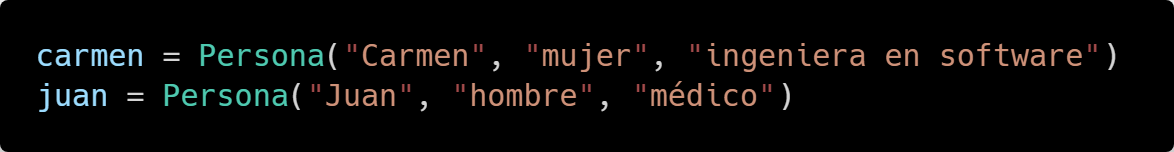
También verás la palabra **"self"**. Por convención se llama así, pero puede tomar cualquier nombre, aunque no se aconseja. Así, como cada clase es como el plano de una casa, **self** es como la casa misma, es decir, el objeto.

El objeto se crea asignando el nombre de la clase a una variable. Cuando creamos un objeto de la clase, se llama **instanciación.** Por lo tanto, el objeto se denomina **instancia de clase.**

<nombre\_del\_objeto> = <NombreDeLaClase>(<argumentos>)

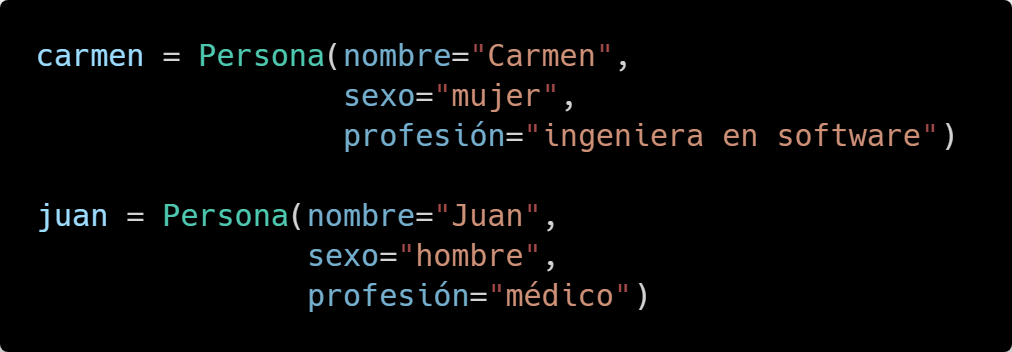
Puedes crear objetos pasando argumentos por posición:

**Continúa con lo siguiente:**



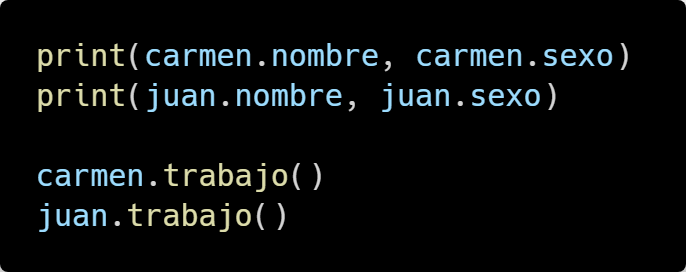
O puedes crear objetos pasando argumentos por nombre:

**Puedes probar con esta forma:**



Invoca el objeto y sus atributos y métodos de la siguiente forma:

**Continúa y ejecuta:**

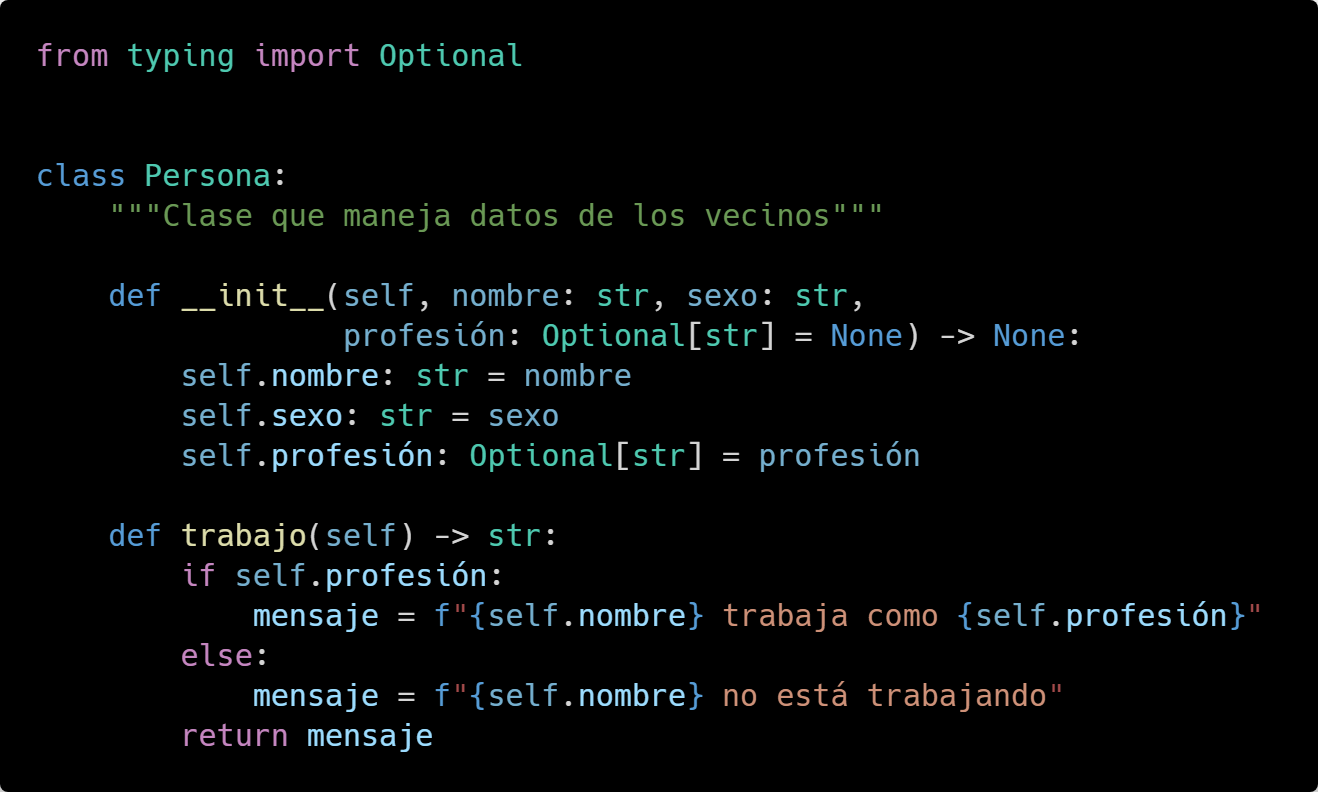


### 

### 

**¡MANOS A LA OBRA!**

**Ejercicio 1**



* Crea el objeto persona\_1 como instancia de Persona, con un nombre, sexo y una profesión.
* Crea el objeto persona\_2 como instancia de Persona, con un nombre, sexo, pero sin una profesión.
* Imprime todas las variables de instancia de cada objeto.
* Llama al método trabajo, tanto para el objeto persona\_1 como para persona\_2, guardando su devolución en una variable, y muestra el mensaje.
* Agrega al final del código las siguientes líneas y trata de explicar lo que hacen:

print(isinstance(12, int))  
print(isinstance(12, float))  
print(isinstance(persona\_1, Persona))

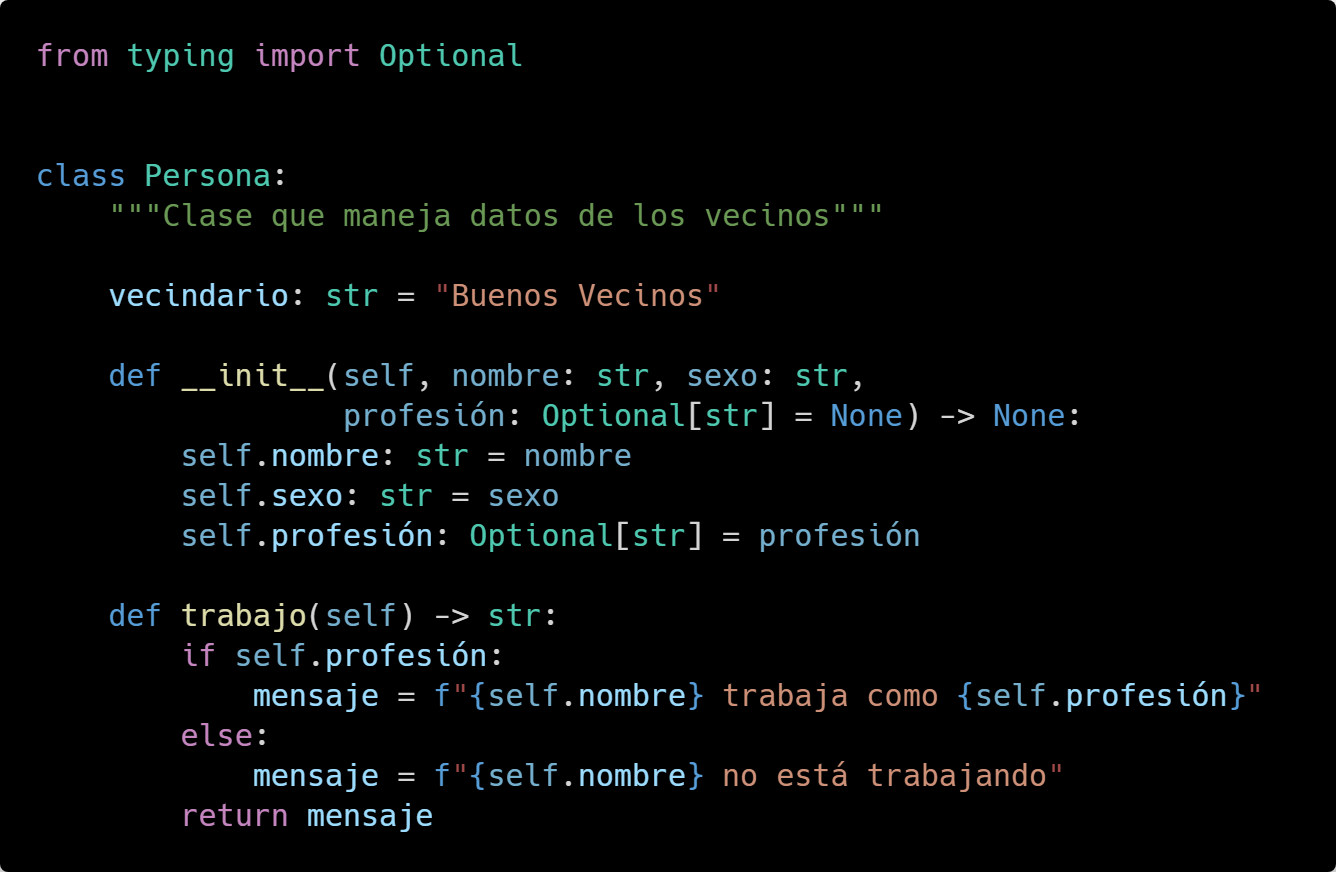
|  | **Ver video “**[**Crear objetos**](https://youtu.be/kfk3roHSfj0)**”** |
| --- | --- |

## ¿Puedo crear variables que sean compartidas?

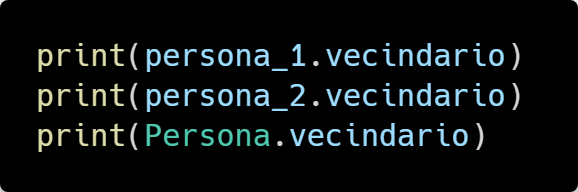
Se pueden crear **variables de clase.** Son compartidas por todos las instancias de una clase. En otras palabras, todos los objetos de una clase pueden compartir variables cuando éstas son declaradas fuera de \_\_init\_\_.

Agrega la siguiente línea, como se mostrará en la próxima captura de pantalla:

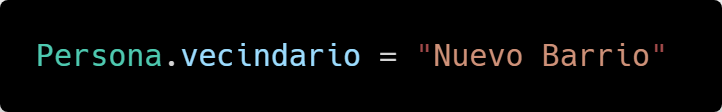
vecindario: str = "Buenos Vecinos"



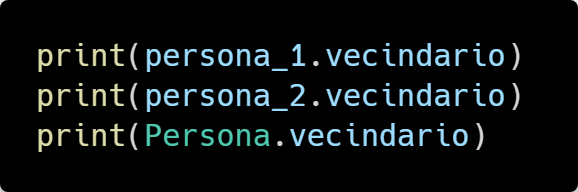
Entiendo que ya tienes creadas dos instancias de Persona. Agrega lo siguiente al final del código y prueba:



Luego, agrega la siguiente línea para cambiar la variable de clase:

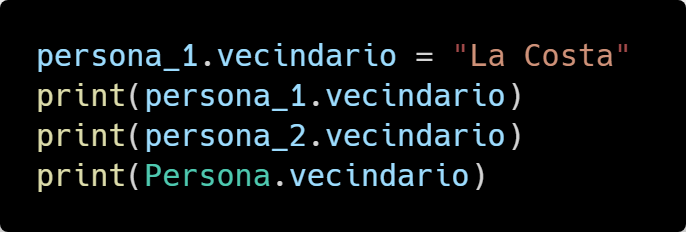


Y agrega lo siguiente y ejecuta:



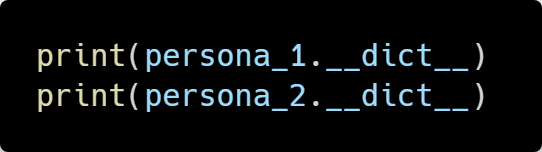
**¿Qué es lo que sucedió?**

Intentemos cambiar la variable desde la instancia:



**¿Por qué cambió persona\_1 y no se modificó el resto? ¿Acaso no se comparten?**

Para ver qué variables de instancia tiene cada objeto, podemos utilizar el método especial \_\_dict\_\_. Agrega lo siguiente y ejecuta:



¿Has visto las diferencias? Creaste de forma "dinámica" una nueva variable de instancia.

| Conclusión: para cambiar una **variable de clase** no uses el nombre del objeto, sino el nombre de la clase, ya que podrías crear **variables de instancia** y generar inconsistencias.. |
| --- |

**¡MANOS A LA OBRA!**

**Ejercicio 2**

Basándote en la clase Persona:

* Agrega una variable de instancia requerida:

edad: int

| Si ves que Python te marca un error, significa que no puedes agregar **"parámetros requeridos"** (u obligatorios) luego de los **"parámetros predeterminados"**. Se arregla poniendo los parámetros requeridos antes de los predeterminados. |
| --- |

* Crea 3 instancias de Persona.
* Modifica la edad en una de las instancias.
* Agrega una variable de clase:

localidad: str

* Modifica la variable de clase.
* Desde una instancia, muestra "vecindario" y "localidad".
* Guarda los objetos en una lista llamada "personas", itérala y muestra todas las edades.

|  | **Ver video “**[**Variable de Instancia Requerida**](https://youtu.be/YtME6oYnPt4)**”** |
| --- | --- |

## 

## Revisemos lo aprendido hasta aquí

* Entender el concepto de programación orientada a objetos
* Diferenciar clases de instancias de clase
* Crear instancias de clase por medio del método constructor
* Pasar argumentos en la creación de instancias
* Acceder y modificar variables de instancias
* Crear variables de clase, modificarlas y acceder a ellas
* Usar la función isinstance para ver si una instancia pertenece a una clase
* Usar el método especial \_\_dict\_\_ para ver los atributos de una instancia

# 

# Herencia

En un lenguaje de programación orientado a objetos, la herencia es un aspecto importante. En Python, la herencia es el proceso de heredar las propiedades de la clase padre en una clase hija. El propósito principal de la herencia es la reutilización del código. Usando la herencia, podemos usar la clase existente para crear una nueva clase en lugar de recrearla desde cero.

## 

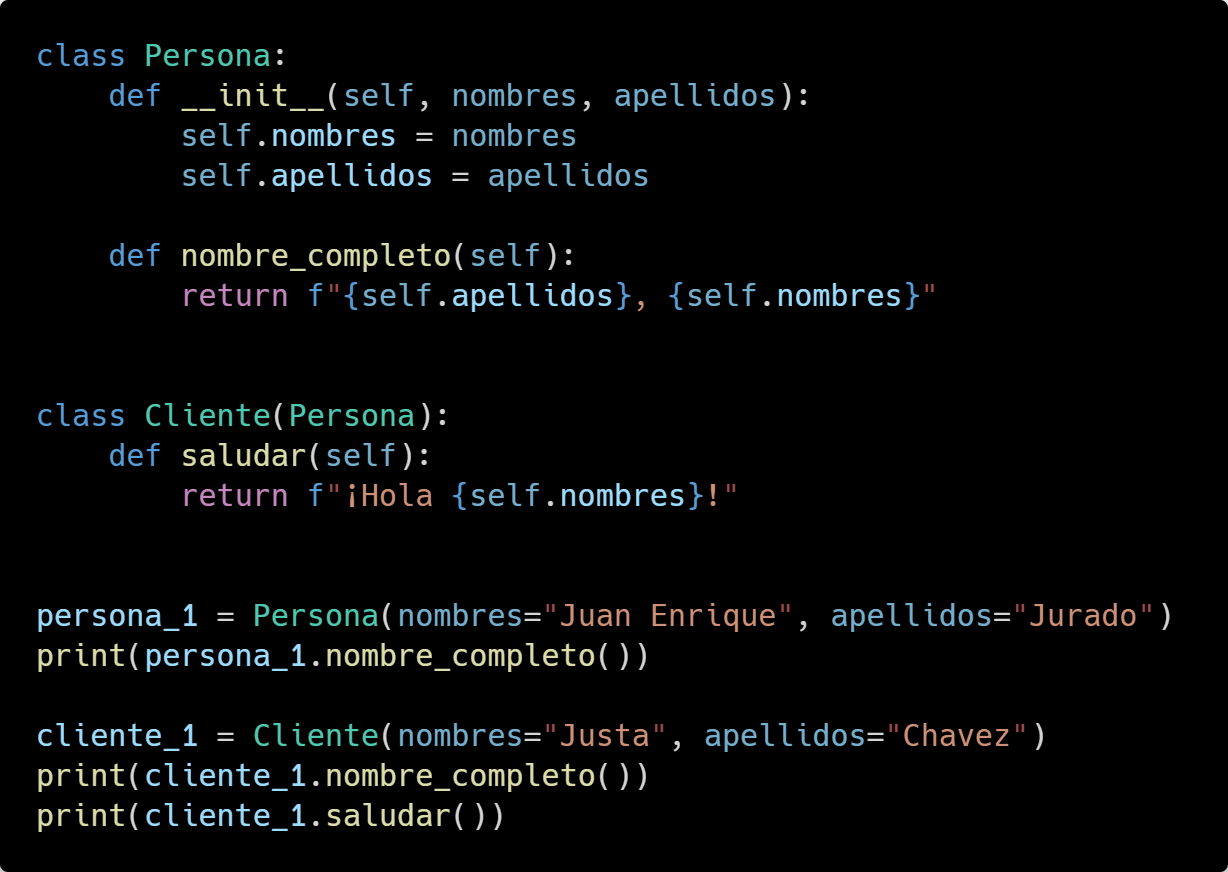
## ¿Cómo heredar clases?

Simplemente, al lado del nombre de la clase, envolver entre paréntesis a la clase padre.

class Hija(Padre):  
 ...

Tienes que tener en cuenta que la clase hija hereda todos los atributos (variables) de la clase padre y todos sus métodos (funciones).

**Ejecuta el siguiente código:**

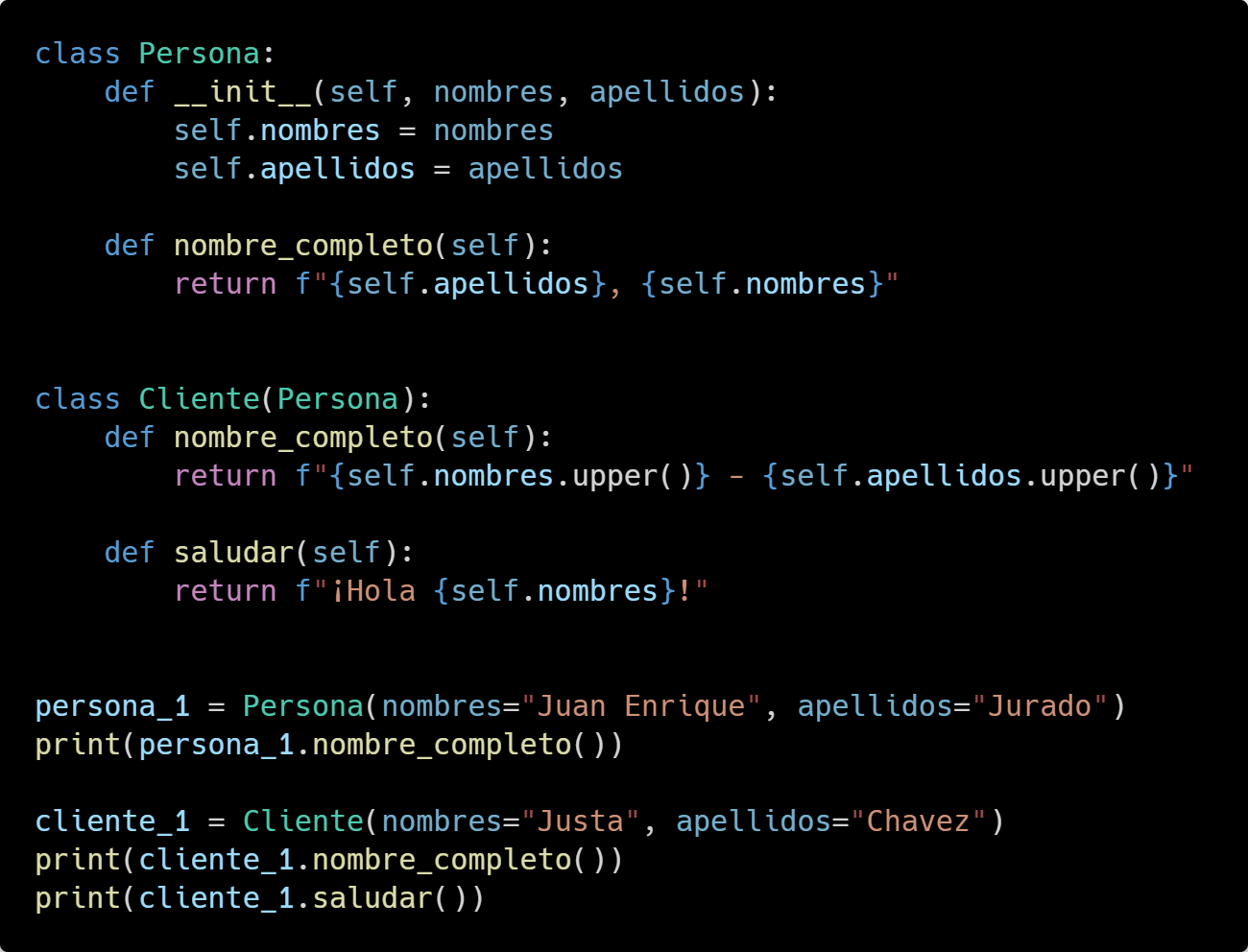


Puedes ver que la Clase Cliente tiene todos los métodos de la clase Persona, más uno nuevo que es saludar().

### 

**¿Cómo hacer para modificar algún método heredado?**

Reescribe el método en la clase hija. Por ejemplo, si quieres modificar el método nombre\_completo, debes sobreescribirlo. Ejecuta:



**¿Cómo hacer para modificar el constructor?**

Tienes que volver a escribir el método \_\_init\_\_

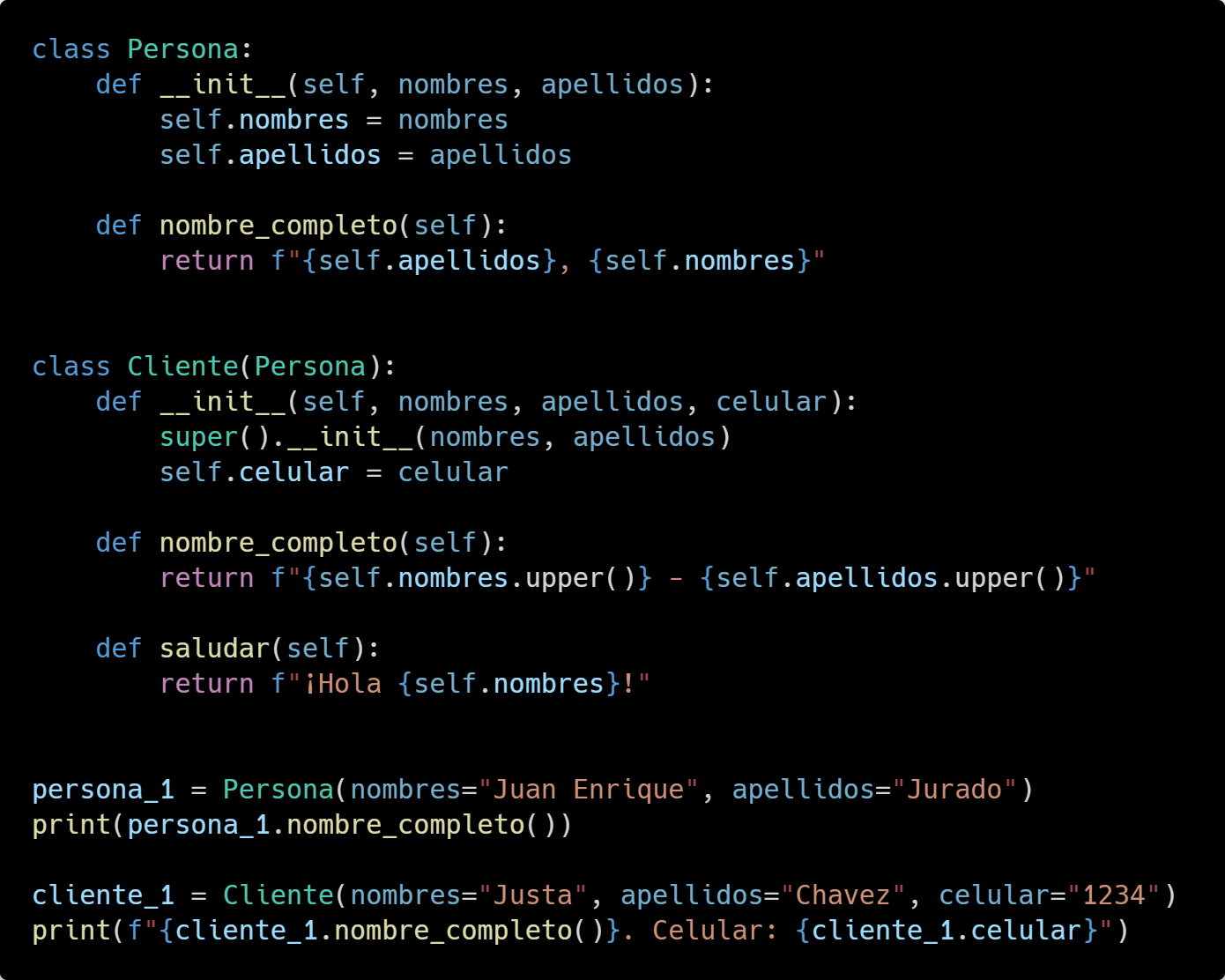
Luego, si la clase padre tiene parámetros requeridos (obligatorios), debes repetirlos, y finalmente agrega el nuevo parámetro.

def \_\_init\_\_(self, <parámetros clase padre>, <nuevo parámetro>):

Para que Python entienda que debe inicializar el objeto como lo hace en la clase padre, debes ser explícito, de lo contrario, va a sobreescribir el método \_\_init\_\_ y no tendrás el efecto deseado. Debes ser explícito, y puedes usar el método **super()**, que representa a la clase padre **(no uses self adentro):**

super().\_\_init\_\_(<parámetros clase padre>)

Por ejemplo, si en la clase Cliente queremos conservar los atributos/variables de instancia "nombres" y "apellidos", pero queremos agregar una nueva: "celular", debemos hacer lo siguiente:

****

### 

**¡MANOS A LA OBRA!**

**Ejercicio 3**

Basándote en la captura de la pantalla anterior:

* Agrega las anotaciones de tipo.
* Crea una nueva clase Usuario que herede de Cliente
* En la clase Usuario, crea un nuevo método que se llame generar\_contraseña que devolverá la siguiente cadena: los primeros 6 caracteres de "nombres" concatenados a los 2 últimos caracteres de "celular".
* Agrega al método constructor de Usuario el atributo "contraseña".
* Asigna a "contraseña" la cadena que devuelve el método generar\_contraseña. La asignación debe ser dentro del constructor.
* Crea una instancia de Persona, muestra su nombre completo.
* Crea una instancia de Cliente, muestra su nombre completo y celular.
* Crea una instancia de Usuario, muestra su nombre completo, celular y contraseña.

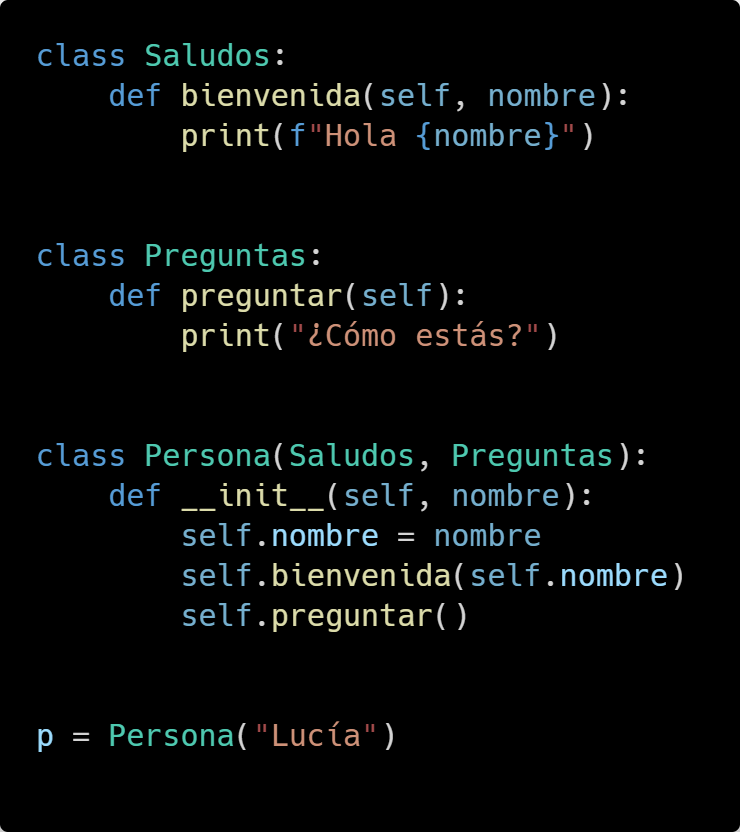
|  | **Ver video “**[**Herencia**](https://youtu.be/7AZBsqssbZA)**”** |
| --- | --- |

## ¿Puedo heredar más de una clase?

A esto se le llama herencia múltiple. Algunos lenguajes de programación no están creados para la herencia múltiple. Python sí, pero hay que usarla con cuidado, debido a que el código va perdiendo legibilidad, haciéndonos más propensos al error. La estructura es la misma, agregándose una clase más en la herencia:

class Hija(Padre1, Padre2):  
 ...

**Ejecuta el siguiente código:**



El código de arriba es un ejemplo simple, porque las dos primeras clases no tienen un constructor. Pero, si cada clase tuviese uno, no bastaría el método super, ya es un método que representa a la clase padre inmediata, y aquí hay varias clases padre. Si queremos inicializar nuestra nueva clase con los atributos de las clases padres, tendremos que llamar el método \_\_init\_\_ de cada clase padre de forma explícita.

| ¡Cada vez que invoques \_\_init\_\_ sin super(), no olvides self! |
| --- |

### Mixins

La herencia múltiple es muy usada en Python para crear un **mixin** (mezcla). Un mixin es una clase que no está destinada a ser instanciada, sino que sirve para ser utilizada por otra "clase base" a través de la herencia. El mixin debe contener métodos, para que la "clase base" los pueda usar.

Por ejemplo: Un Auto, un Avión, un Bote, pueden ser clases que heredan otra clase Vehículo, pues todos son vehículos y tienen en común la función de transportar.

class Vehículo:  
 pass

class Auto(Vehículo):  
 def viajar()

class Avión(Vehículo):  
 def viajar()

class Bote(Vehículo):  
 def viajar()

Ahora queremos agregar una funcionalidad al auto: escuchar radio. Esta funcionalidad no la tiene un avión y un bote, así que necesitaremos una clase que se mezclará con Auto:

class Radio:  
 def escuchar\_radio()

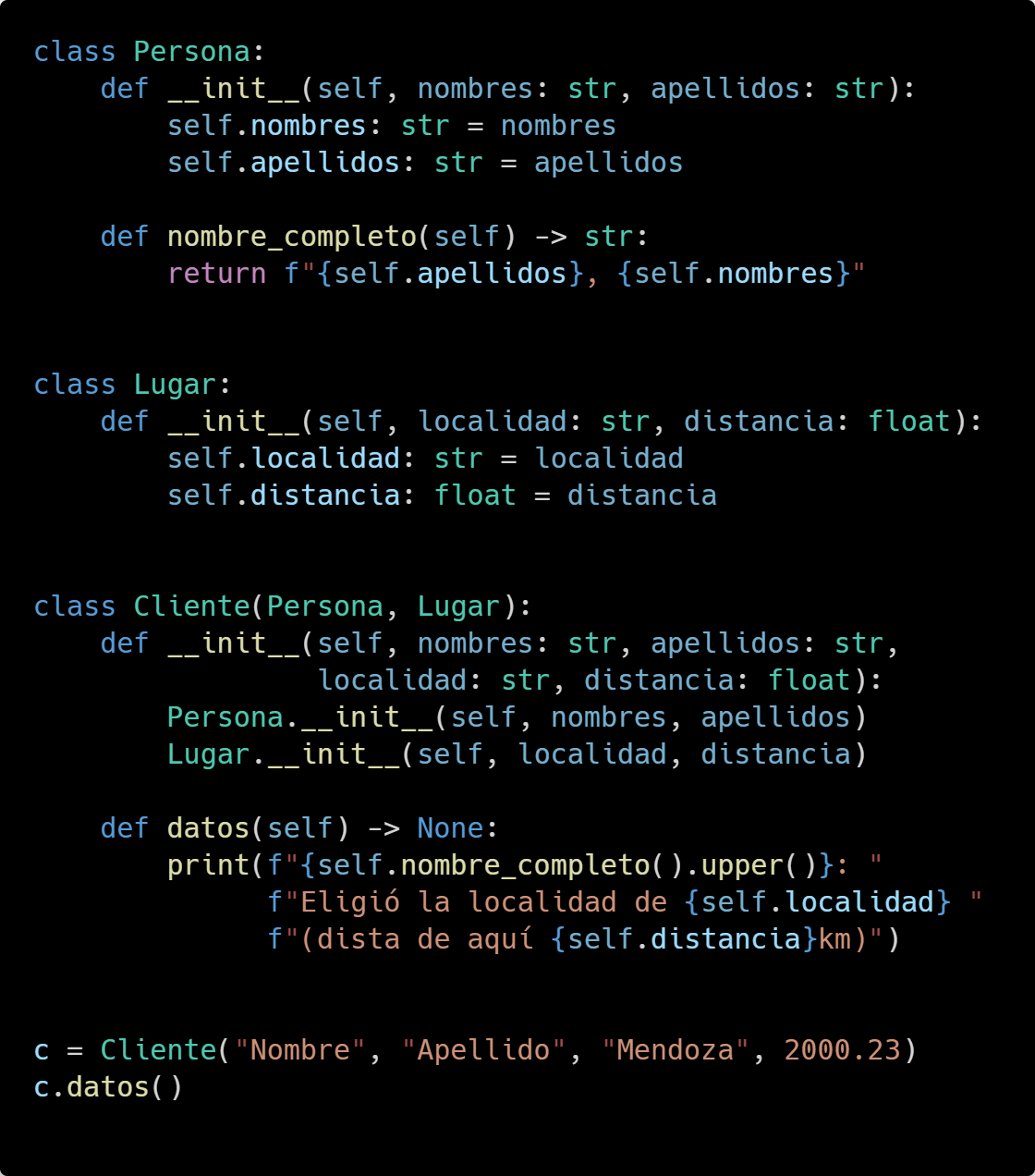
La herencia múltiple para mezclar:

class Auto(Vehículo, Radio):  
 def viajar()

Si instanciamos auto, ahora podemos escuchar música:

auto = Auto  
 auto.viajar()  
 auto.escuchar\_radio()

**Ejecuta el siguiente código:**



**¡MANOS A LA OBRA!**

**Ejercicio 4**

Basándote en el ejercicio 3, en el que tienes tres clases: Persona, Cliente, Usuario.

* Crea una nueva clase Lugar, que tenga como atributo de clase "estado" que será de tipo bool, tendrá por valor predeterminado True. Además, crea atributos de instancia (requeridos) "localidad": str y "distancia": float.
* Crea una nueva clase llamada Saldo, que tenga como atributo de instancia (requerido) "balance": float, cuyo valor predeterminado sea 0.
* Cliente heredará de Persona, Lugar y Saldo.
* Crea una instancia de cliente y muestra todos sus datos mediante la llamada de un método.

| Generalmente no es una buena práctica la herencia múltiple de más de 2 clases (ahora estamos ejercitando) Tampoco lo es cuando creas varios niveles de herencia. Se aconseja simplificar el código siempre y cuando se pueda. |
| --- |

## Revisemos lo aprendido hasta aquí

* Implementar la herencia simple
* Implementar la herencia múltiple
* Conocer el patrón de diseño llamado "mixin" a través de la herencia múltiple
* Usar el método super

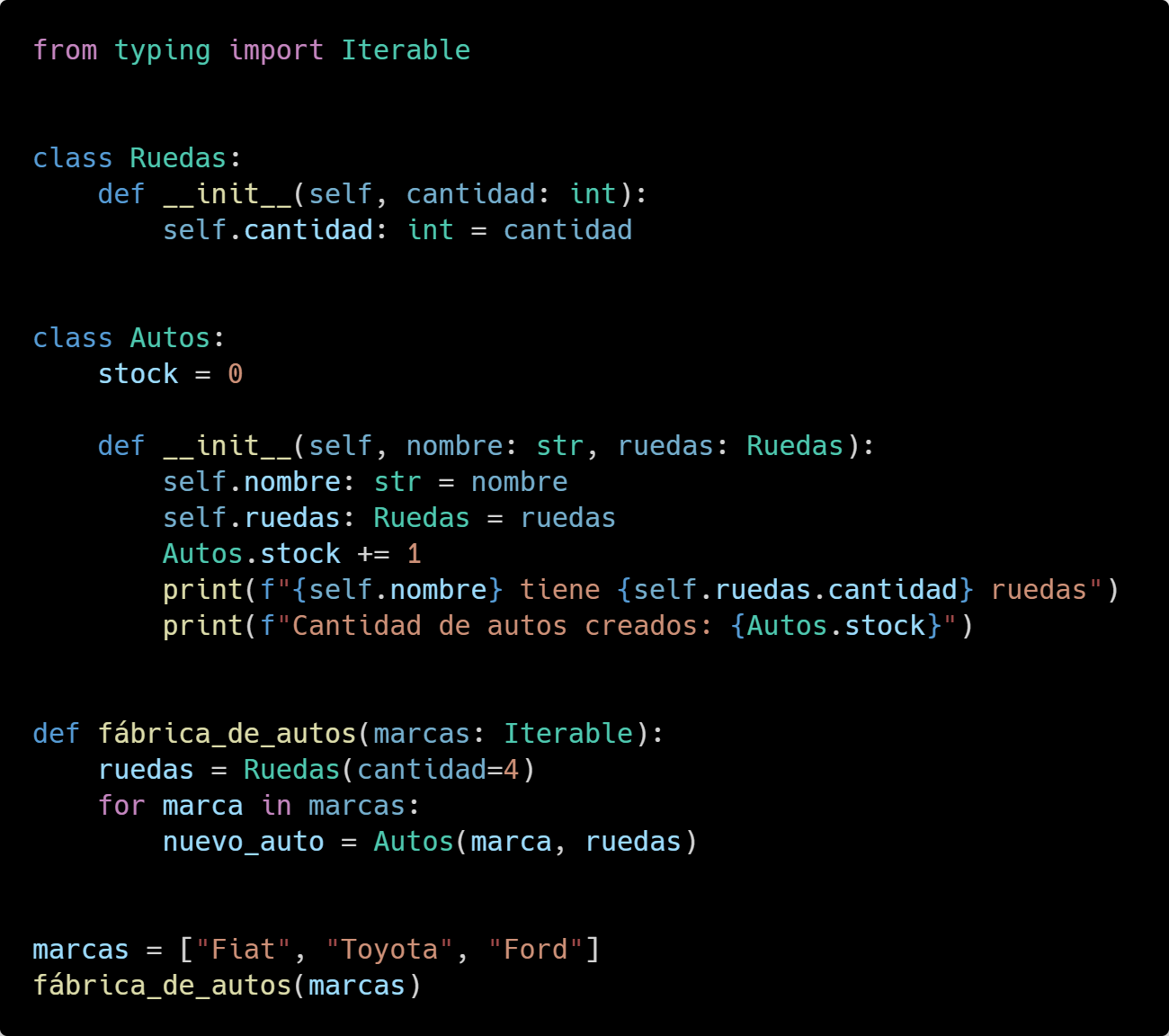
# 

# Relaciones entre clases

La herencia de clases nos permite abordar casos de uso en términos de "es un". Usuario es una Persona. Una Persona es un SerVivo, y así... Pero existen otras situaciones en que la herencia no es la mejor opción. Podemos relacionar clases en los términos de "tiene un". Las relaciones más usadas son la de **agregación** y **composición**. La composición implica que el objeto utilizado no puede «funcionar» sin la presencia de su propietario. La agregación implica que el objeto utilizado puede funcionar por sí mismo.

## Agregación

La agregación es una relación entre dos clases, cuando **inicializa un atributo de su instancia con una instancia de otra clase.** Ambas instancias son independientes entre sí, de tal forma que si se destruye alguna instancia, no modifica a la otra.



En el caso de que destruyamos la instancia "ruedas" con el siguiente comando:

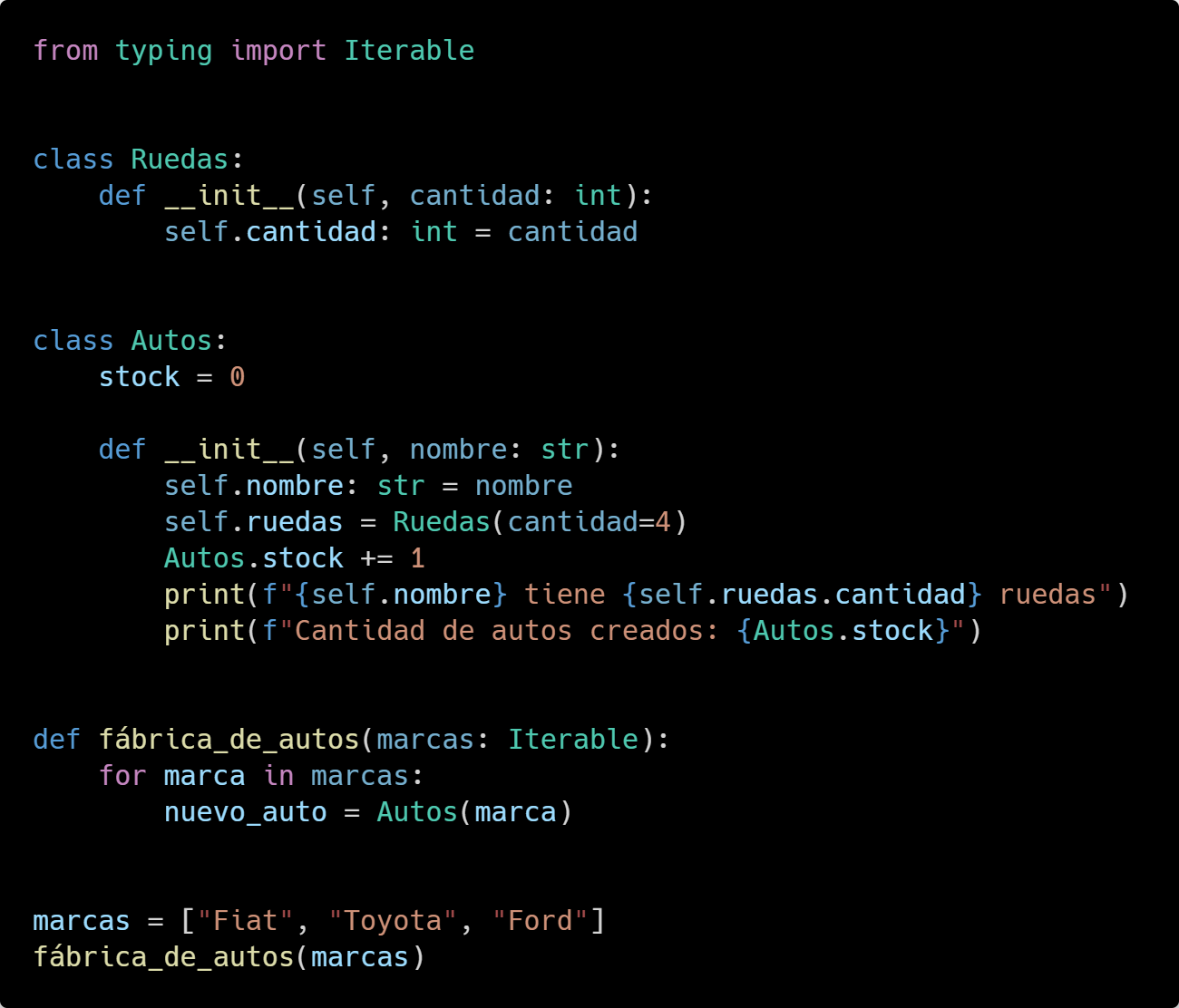
del ruedas

instancia\_auto seguirá existiendo, y, al revés, sucederá lo mismo. Es decir, las instancias son independientes.

## Composición

La composición es una relación entre dos clases, cuando una de ellas **inicializa un atributo de su instancia, siendo esta una instancia de otra clase.**

Observa el cambio sutil, pero importante respecto a la agregación, y trata de explicar cómo funciona:



**¡MANOS A LA OBRA!**

**Ejercicio 5**

Basándote en el ejercicio 3, en el que tienes tres clases: Persona, Cliente, Usuario.

* No utilices la herencia
* Cliente tendrá una relación de composición respecto a Persona
* Usuario tendrá una relación de agregación respecto a Cliente
* Haz las pruebas unitarias sobre las clases (otro archivo con el sufijo \_test)

|  | **Ver video “**[**Relaciones**](https://youtu.be/xluOzvxzaGE)**”** |
| --- | --- |

## Revisemos lo aprendido hasta aquí

* Implementar la técnica de agregación
* Implementar la técnica de composición