# **¿Cómo funciona Python?**

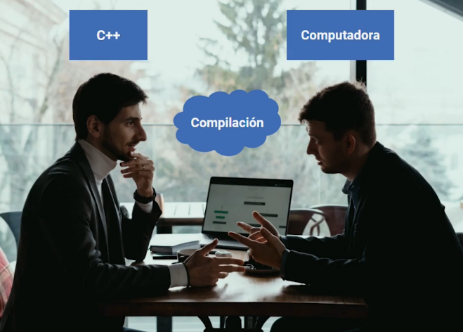
## **Python es un lenguaje interpretado**

Lo que significa que tu código es transformado por el intérprete (máquina virtual de Python) a **bytecode** antes de ser ejecutado por un ordenador con **x** sistema operativo. El **bytecode** es un lenguaje de programación de más bajo nivel, esto es básicamente desde que corres tu programa hasta que la PC lo ejecuta hay una carrera de relevos de lenguajes o protocolos hasta llegar al transistor y la señal eléctrica.

## **Garbage collector**

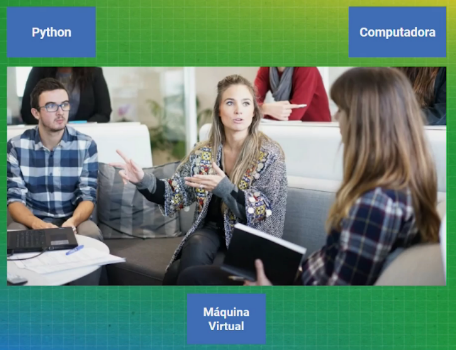
Recuerda que el garbage collector toma los objetos y variables que no están en uso y los elimina en la memoria.

## **pycache**

**\_pycache\_** es el directorio (carpeta) que contiene el **bytecode** (el código intermedio) que crea Python para que lo pueda leer la máquina virtual.

Diferencias entre Compilado e Interpretado

👨‍💻 **Compilado**: Un código en un lenguaje compilado transforma, mediante el compilador, las instrucciones directamente a 1 y 0, que son los pulsos de energía. Ejemplo de esto tenemos en lenguaje C++.

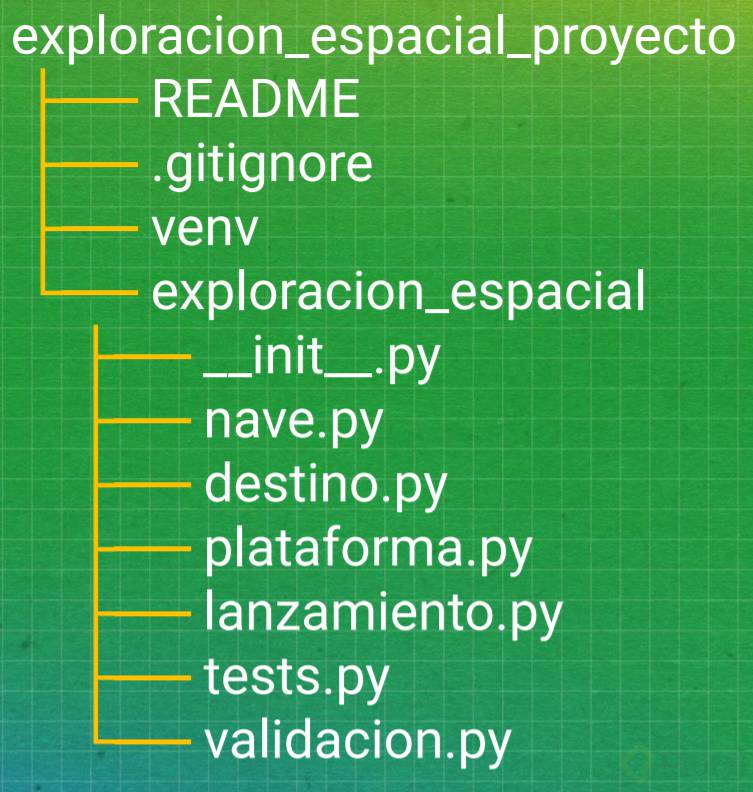
**🐍Interpretado**: Este tipo de lenguajes tiene una máquina virtual, el intérprete. Tenemos el caso de Python, quién crea el bytecode que lee la máquina virtual para traducirlo para que lo entienda la computadora.

# **Cómo organizar las carpetas de tus proyectos**

**Módulo** es cualquier archivo de Python. Generalmente, contiene código que puedes reutilizar.

**Paquete** es un conjunto de módulos. Siempre posee el archivo **\_\_init\_\_.py**.

Un ejemplo de organizar los archivos de Python es de la siguiente manera.

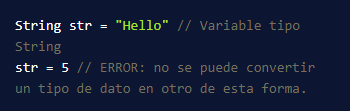


# **¿Qué son los tipados?**

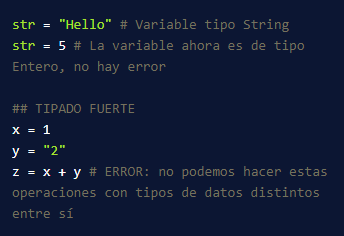
Los tipados es una clasificación de los lenguajes de programación, tenemos cuatro tipos:

* Estático.
* Dinámico.
* Débil.
* Fuerte.
* El tipado del lenguaje depende de cómo trata a los tipos de datos.

El tipado estático es el que levanta un error en el tiempo de compilación, ejemplo en JAVA:



El tipado dinámico levanta el error en tiempo de ejecución, ejemplo en Python:

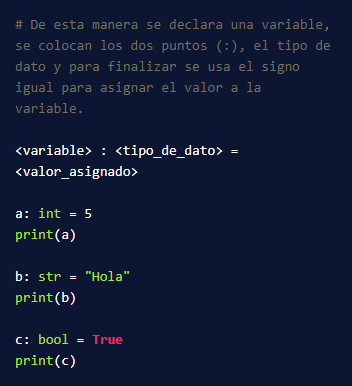


El tipado débil es el que hace un cambio en un tipo de dato para poder operar con él, como lo hace JavaScript y PHP.

Python es un lenguaje de tipado Dinámico y Fuerte.

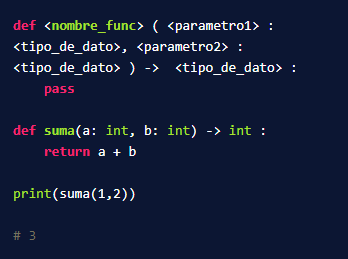
# **Tipado estático en Python**

Para hacer que Python sea de tipado estático es necesario agregar algo de sintaxis adicional a lo aprendido, además, esta característica solo se puede aplicar a partir de la versión 3.6.

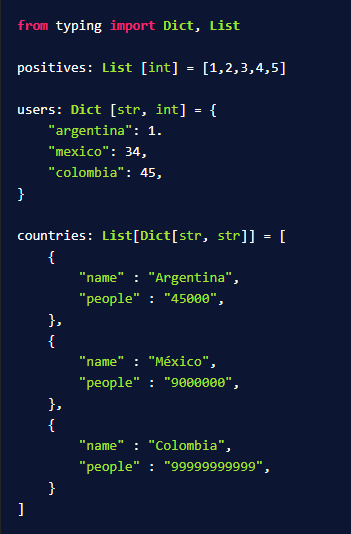


Del mismo modo se puede usar esta metodología de tipado en Python a funciones adicionando el signo menos a continuación del signo mayor que para determinar el tipo de dato.

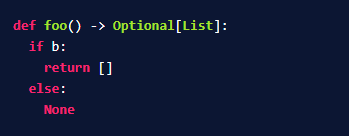
Ejemplo:



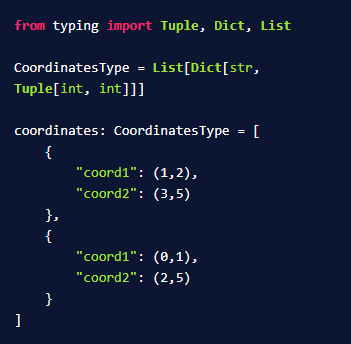
Existe una librería de fabrica que viene preinstalada con Python que se llama **typing**, que es de gran utilidad para trabajar con tipado con estructuras de datos entre la versión 3.6 y 3.9



Usar **Optional[]**, de la librería **typing**, es para cuando ocupamos retornar **None** u otra variable.



Más ejemplos de typing



## **Modulo mypy**

El módulo **mypy** se complementa con el módulo **typing** ya que permitirá mostrar los errores de tipado débil en Python.

## **Ventajas de usar tipado estático**

* Nos da claridad a nuestro código.
* Nos devuelve los errores **antes** que el programa se ejecute.

# **Scope: alcance de las variables**

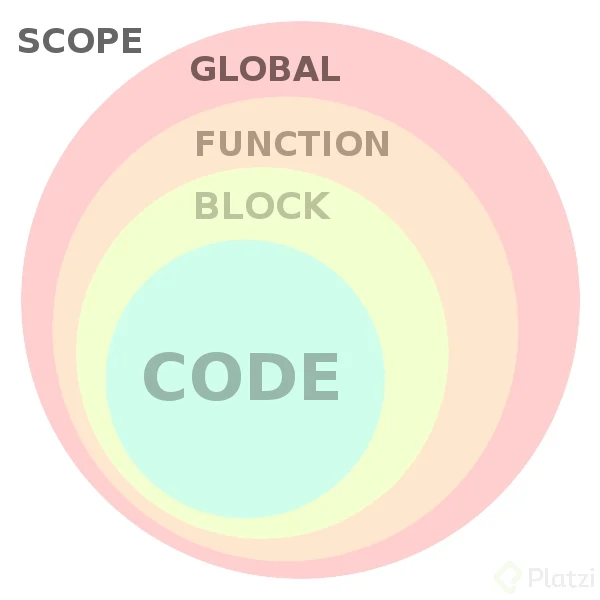
El scope es el alcance que tienen las variables. Depende de donde declares o inicialices una variable para saber si tienes acceso. Regla de oro: **Una variable solo está disponible dentro de la región donde fue creada**.

## **Local Scope**

Es la región que corresponde el ámbito de una función, donde podremos tener una o más variables, las variables van a ser accesibles únicamente en esta región y no serán visibles para otras regiones

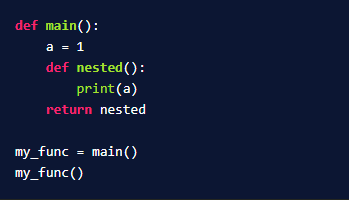
## **Global Scope**

Al escribir una o más variables en esta región, estas podrán ser accesibles desde cualquier parte del código.



# **Closures**

Es una forma de acceder a variables de otros scopes superior a través de una **Nested Function**. Se retorna la Nested Function y esta recuerda el valor que imprime, aunque a la hora de ejecutarla no esté dentro de su alcance.



**Nested functions**: Son simplemente funciones creadas dentro de otra función. Podemos hacer **return** de una función creada dentro de otra función y luego guardar esas funciones en variables que podemos utilizar.

## **Reglas para encontrar un Closure**

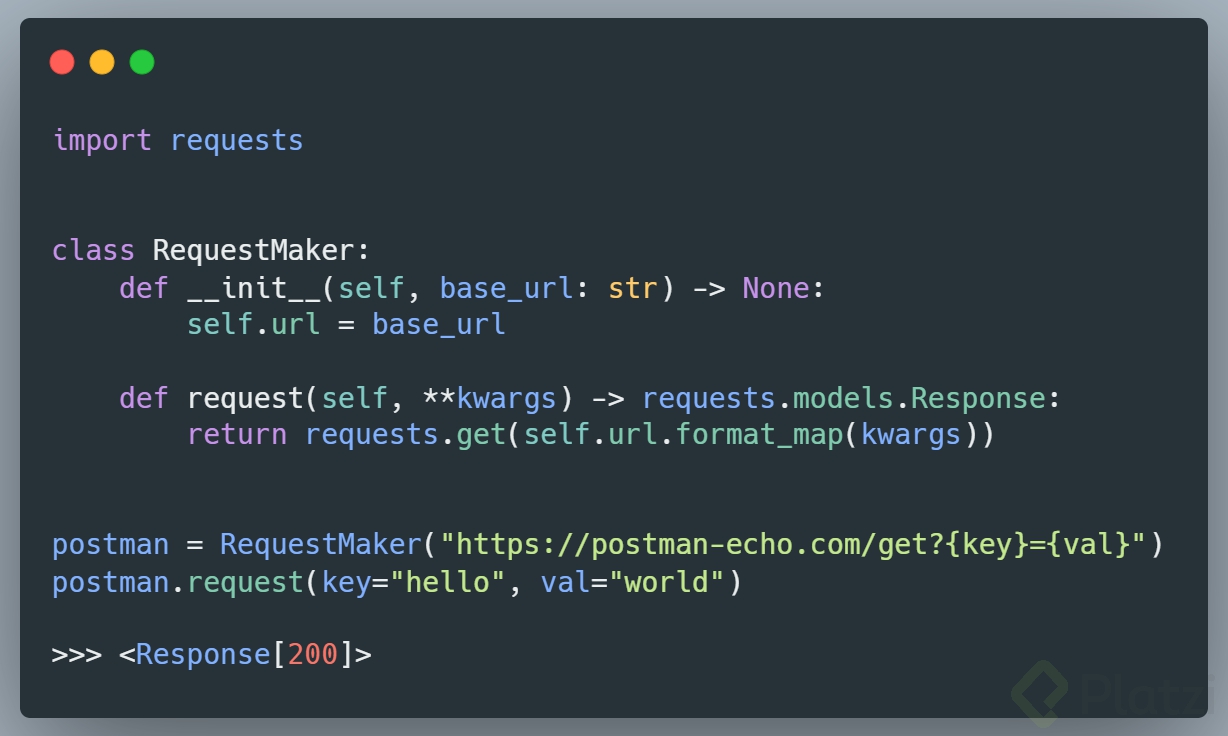
* Debemos tener una **nested function**.
* La **nested function** debe referenciar un valor de un **scope** **superior**.
* La función que envuelve a la **nested function** debe retornarla también.

## **¿Dónde aparecen los Closure?**

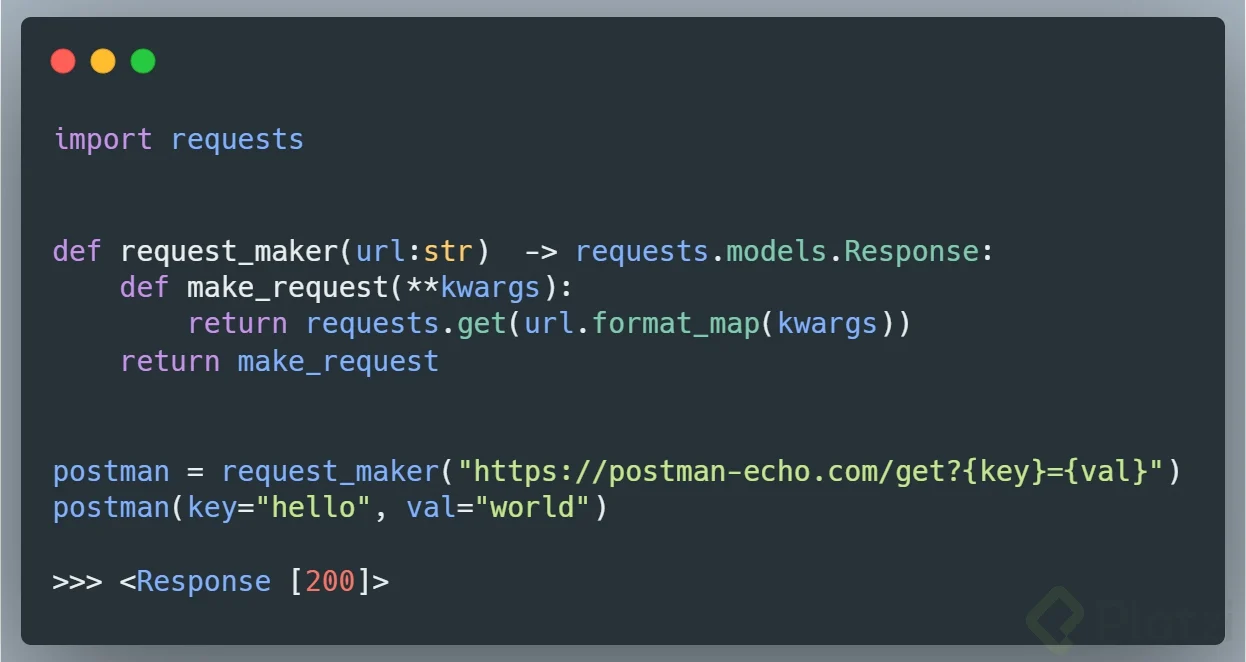
* Cuando se tiene una clase corta, es decir, pequeña que tenga sólo un método.
* Cuando se trabaja con decoradores.

Ejemplo con clases

Clase original



Con closure

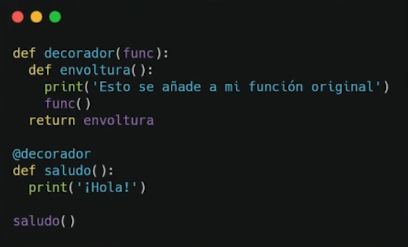


# **Decoradores**

Un **decorador** es una **función** que recibe como **parámetro** **otra función**, le añade cosas y retorna una **función diferente**. Tienen la misma estructura que los **Closures**, pero en vez de variables lo que se envía es una función. Ejemplo:



Con **sugar syntax**

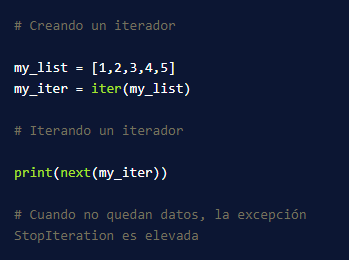


# **Iteradores**

Antes de entender qué son los **iteradores**, primero debemos entender a los **iterables**.

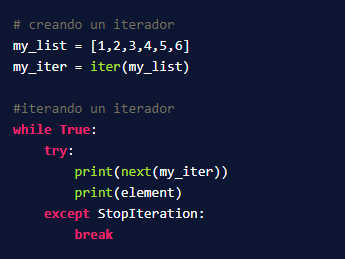
Son todos aquellos objetos que podemos recorrer en un ciclo. Son aquellas estructuras de datos divisibles en elementos únicos que yo puedo recorrer en un **ciclo**.

Pero en Python las cosas no son así. Los iterables se convierten en **iteradores**.



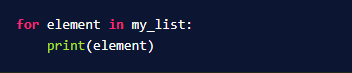
Cuando hacemos un ciclo, python no está iterando(internamente). Lo que sucede **es que ese iterable se convierte en un objeto especial el cual se llama iterador**.

1. **iter** convierte un iterable en un iterador.
2. Una vez que se hace, se usa al **next()** para ir al siguiente elemento del iterador
3. Cuando ya no quede ningún elemento python imprimirá un error.



## **Ciclo for (azúcar sintáctica)**

El ciclo **for** hace lo mismo que el ciclo **while true**, en vez de escribir todo ese código se puede resumir en:

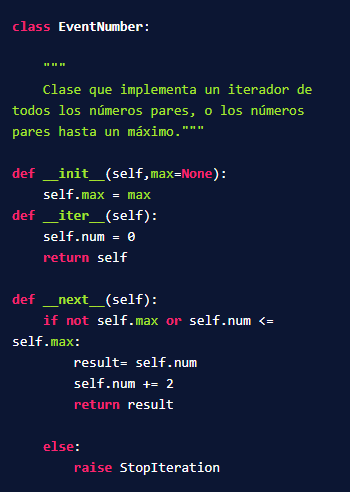


El ciclo **for** no existe en python, lo que hace python es convertir una **iterable** en un **iterador** y con un ciclo **while** recorrerlo hasta encontrar el **StopIteration**.

## **¿Cómo construyo un iterador?**

El protocolo de los iteradores: Para construir un **iterador**, hay que construir una clase que contenga dos métodos importantes:





* **\_\_iter\_\_()**: Retorna al objeto al en sí mismo. Convierte un iterable en un iterador.
* **\_\_next\_\_()**: Convierte a la función next. Nos permite extraer cada uno de los elementos del iterador.

Con los iteradores trabajamos más rápido y ahorramos memoria.

## **Ventajas de usar iteradores**

Es una expresión que explica como obtener el resultado. También ahorra memoria y es más rápido.