29 DE JUNIO DE 2021/[#PYTHON](https://www.freecodecamp.org/espanol/news/tag/python/)

**El Manual de Python**

[**Cristian Sulbaran**](https://www.freecodecamp.org/espanol/news/author/cristian/)



El Manual de Python sigue la regla 80/20: aprende el 80% del tema en el 20% del tiempo.

Encuentro que este enfoque ofrece una descripción bastante completa.

Este libro no intenta cubrir todo lo relacionado con Python. Se centra en el núcleo del lenguaje, tratando de simplificar los temas más complejos.

Esperamos que el contenido de este libro te ayude a lograr lo que quieres: **aprender los conceptos básicos de Python.**

Nota:[puedes obtener una versión PDF, ePub y Mobi **en inglés**de este manual de Python](https://flaviocopes.com/page/python-handbook/)

¡Que lo disfrutes!

**Resumen**

* [Introducción a Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#pythonintro)
* [Cómo instalar Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#howtoinstallpython)
* [Cómo ejecutar programas en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#howtoruninpython)
* [Python 2 vs Python 3](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#pythonvs)
* [Básicos en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#pythonbasics)
* [Tipos de datos en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#datatypespython)
* [Operadores en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#operatorspython)
* [El operador ternario en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#ternaryoperatorinpython)
* [Cadenas en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#stringsinpython)
* [Booleanos en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#booleansinpython)
* [Números en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#numbersinpython)
* [Constantes en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#constantsinpython)
* [Enums en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#enumsinpython)
* [Entradas de usuario en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#userinputinpython)
* [Declaraciones de control en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#controlstatementsinpython)
* [Listas en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#listsinpython)
* [Tuplas en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#tuplesinpython)
* [Diccionarios en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#dictionairesinpython)
* [Conjuntos en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#setsinpython)
* [Funciones en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#functionsinpython)
* [Objetos en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#objectsinpython)
* [Bucles en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#loopsinpython)
* [Clases en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#classesinpython)
* [Módulos en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#modulesinpython)
* [Librería estándar de Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#pythonstandardlibrary)
* [La guía de estilo PEP8 en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#pep8pythonstyleguide)
* [Depurando en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#debugginginpython)
* [Ámbito de variable en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#variablescopeinpython)
* [Cómo aceptar argumentos desde la línea de comandos en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#howtoacceptargumentsfromcliinpython)
* [Funciones Lambda en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#lambdafunctionsinpython)
* [Recursión en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#recursioninpython)
* [Funciones anidadas en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#nestedfunctionsinpython)
* [Closures en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#closuresinpython)
* [Decoradores en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#decoratorsinpython)
* [Docstrings en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#docstringsinpython)
* [Introspección en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#introspectioninpython)
* [Anotaciones en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#annotationsinpython)
* [Excepciones en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#exceptionsinpython)
* [Declaración *with*en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#withdeclarationinpython)
* [Cómo instalar paquetes de terceros en Python usando pip](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#howtoinstall3rdpartypackageswithpipinpython)
* [Comprensiones de lista en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#listcomprehensionsinpython)
* [Polimorfismo en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#polymorphisminpython)
* [Sobrecarga de operador en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#operatoroverloadinginpython)
* [Entornos virtuales en Python](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#virtualenvironmentsinpython)
* [Conclusión](https://www.freecodecamp.org/espanol/news/el-manual-de-python/#conclusioninpython)

**Recomendación**: cuando estamos programando, integrando palabras en nuestro idioma, suele ser mejor evitar el uso de caracteres extendidos como las letras con [tildes](https://es.wikipedia.org/wiki/Acento_gr%C3%A1fico)(*á,é,ñ,ô,ü...*), para que Python, tú o la persona usando tu código no genere errores inesperados y lograr la menor confusión posible.

Por este motivo, dentro de los bloques especiales en este libro que representen código, los nombres que le demos a nuestras variables y demás objetos, **NO llevarán caracteres extendidos**.

**Introducción a Python**

Python literalmente se está comiendo el mundo de la programación. Está creciendo en popularidad y uso de formas que no tienen precedentes en la historia de las computadoras.

Python sobresale en una amplia variedad de escenarios:**las secuencias de comandos Shell, la automatización de tareas y el desarrollo web** son solo algunos ejemplos básicos.

Python es el lenguaje preferido para **data analysis y machine learning**, pero también puede adaptarse para crear juegos y trabajar con dispositivos integrados.

Más importante aún, es el idioma elegido para las carreras de**ciencias de computación** en universidades de todo el mundo.

Muchos estudiantes aprenden Python como su primer lenguaje de programación. Muchos lo están aprendiendo ahora mismo y muchos más lo aprenderán en el futuro. Y para muchos de ellos, Python será el único lenguaje de programación que necesitarán.

Gracias a esta posición única, es probable que Python crezca aún más en el futuro.

El lenguaje es simple, expresivo y bastante sencillo.

El ecosistema es enorme. Parece haber una biblioteca para todo lo que puedas imaginar.

Python es un lenguaje de programación de alto nivel adecuado para principiantes gracias a su sintaxis intuitiva, su enorme comunidad y su ecosistema vibrante.

También es apreciado por profesionales de muchos campos diferentes.

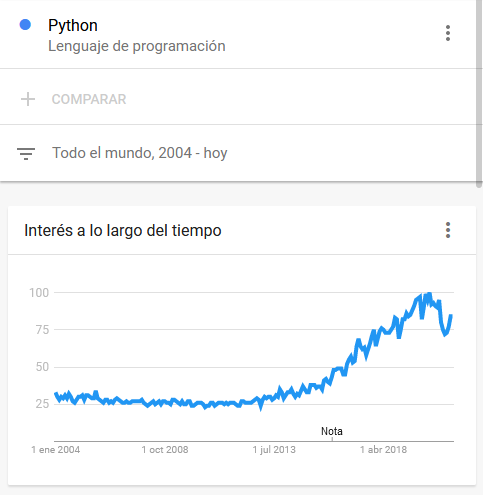
Técnicamente hablando, Python es un lenguaje interpretado que no tiene una fase de compilación intermedia como un lenguaje compilado, por ejemplo C o Java.

Y como muchos lenguajes interpretados, se escribe dinámicamente. Esto significa que no tienes que indicar los tipos de variables que usas y las variables no están vinculadas a un tipo específico.

Esto tiene pros y contras. En particular, escribes programas más rápido, pero por otro lado tienes menos ayuda de las herramientas para prevenir posibles errores. Esto significa que conocerás ciertos problemas solo ejecutando el programa en tiempo de ejecución.

Python admite una amplia variedad de paradigmas de programación diferentes, incluida la programación procedimental, la programación orientada a objetos y la programación funcional. Es lo suficientemente flexible como para adaptarse a muchas necesidades diferentes.

Creado en 1991 por Guido van Rossum, su popularidad ha ido aumentando, especialmente en los últimos 5 años, como muestra esta infografía de Tendencias de Google:

Lenguaje de programación Python en Google Trends 2004 - Actualidad(Abr,2021)

Comenzar con Python es muy fácil. Todo lo que necesitas es instalar el paquete oficial de [*python.org*](https://www.python.org/), para Windows, macOS o Linux, y estarás listo para comenzar.

Si eres nuevo en la programación, en las siguientes publicaciones te guiaremos para pasar de cero a convertirte en programador Python.

E incluso si actualmente eres un programador que se especializa en otro idioma, Python es un lenguaje que vale la pena conocer porque creemos que solo seguirá creciendo a partir de aquí.

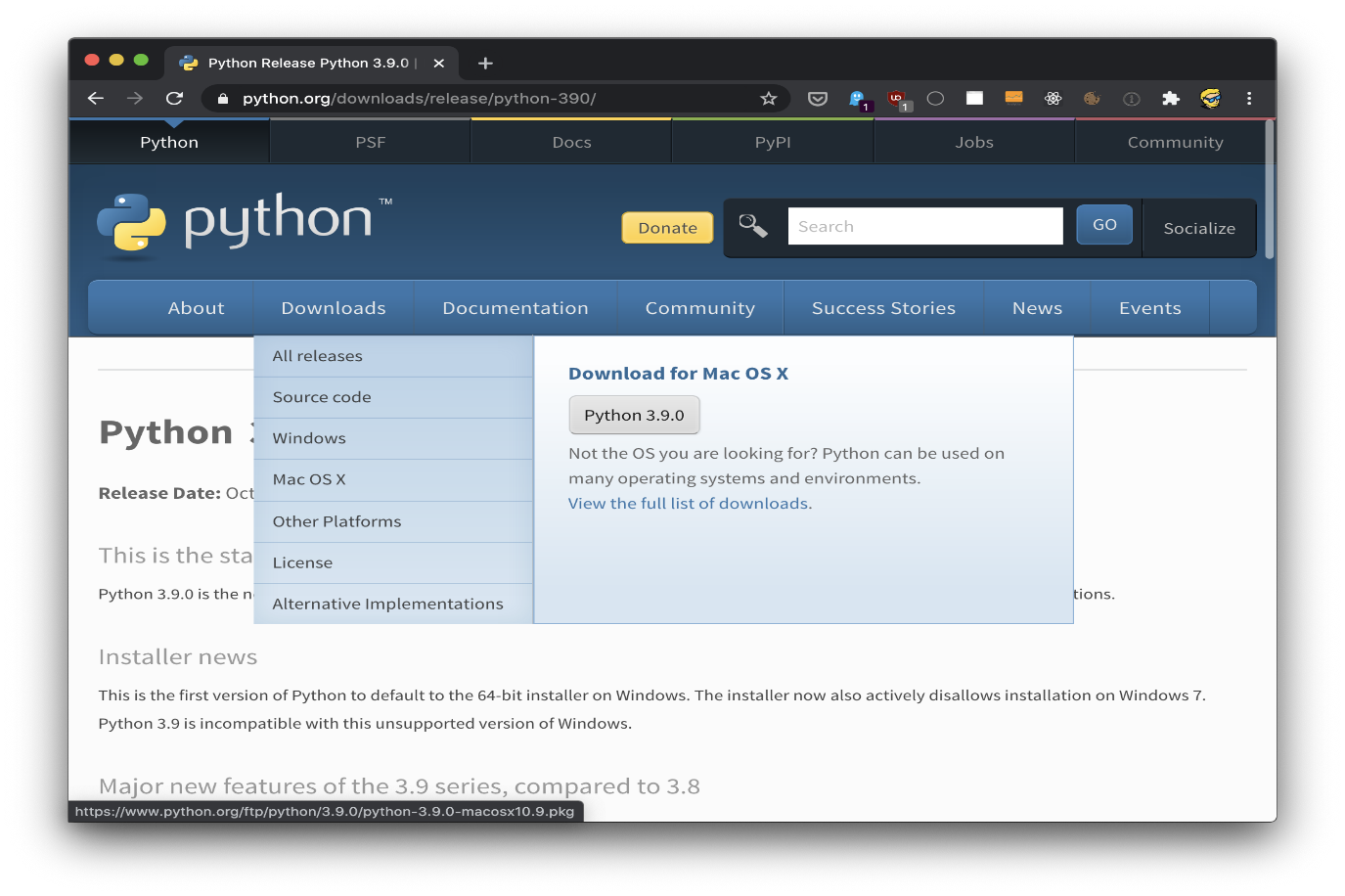
Los lenguajes de nivel inferior como C++ y Rust pueden ser excelentes para programadores expertos, pero son desalentadores para empezar y requieren mucho tiempo para dominarlos.

Python, por otro lado, es un lenguaje de programación para todos: estudiantes, personas que hacen su trabajo diario con Excel, científicos y más.

**Es el idioma que todos los interesados en programar deberían aprender primero**.

**Cómo instalar Python**

Ve a [https://www.python.org](https://www.python.org/), elije el menú Downloads (descargas), elije tu sistema operativo y aparecerá un panel con un enlace para descargar el paquete oficial:



Asegúrate de seguir las instrucciones específicas para tu sistema operativo. En macOS puedes encontrar una guía detallada(*inglés*) en <https://flaviocopes.com/python-installation-macos/>.

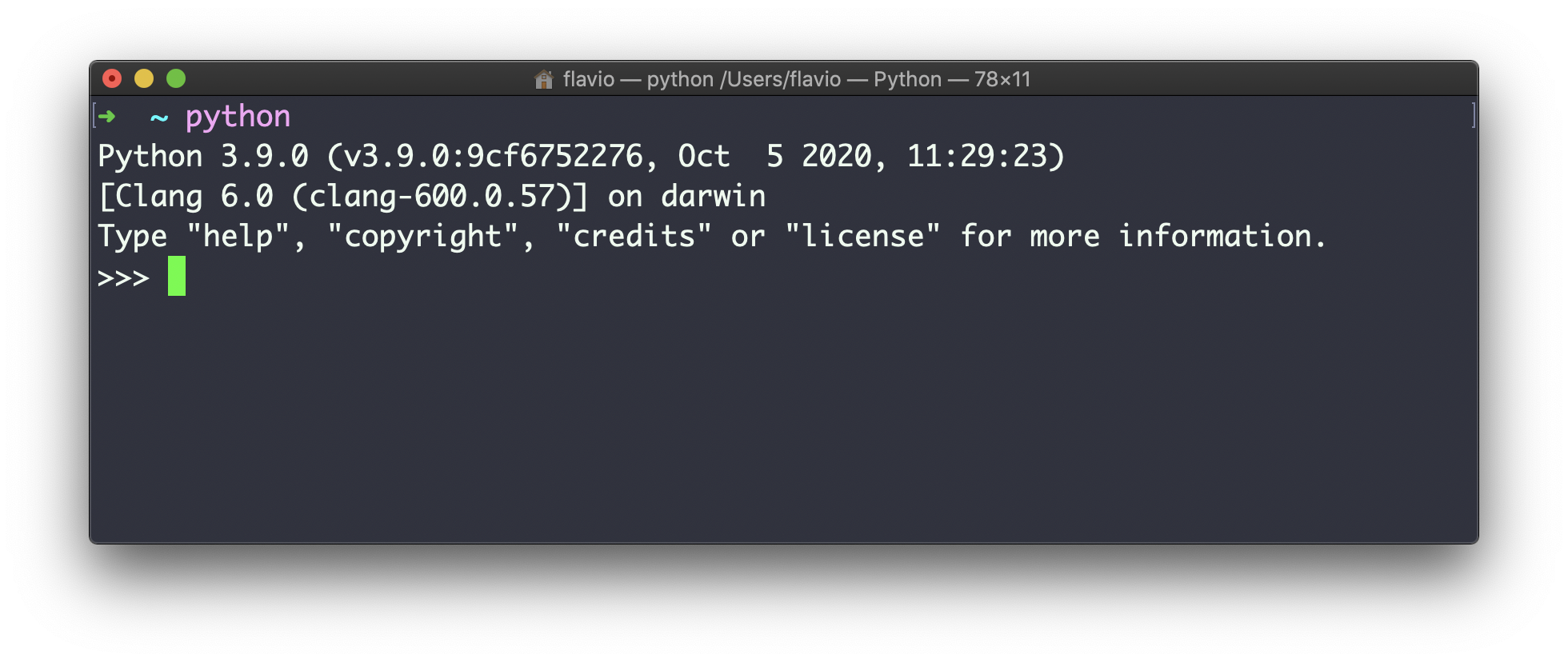
**Cómo ejecutar programas en Python**

Hay algunas formas diferentes de ejecutar programas Python.

En particular, hay una distinción entre el uso de mensajes interactivos, donde se escribe código Python y se ejecuta inmediatamente; y guardar un programa Python en un archivo y ejecutarlo.

Comencemos con indicaciones interactivas.

Si abres la terminal y escribes python, verás una pantalla parecida a esta:



Este es el Python REPL(Read-Evaluate-Print-Loop).

[REPL](https://es.wikipedia.org/wiki/REPL) es un entorno de [programación computacional](https://es.wikipedia.org/wiki/Programaci%C3%B3n) simple e interactivo que toma las entradas individuales del usuario, las evalúa y devuelve el resultado al usuario; un programa escrito en un entorno REPL es ejecutado por partes.

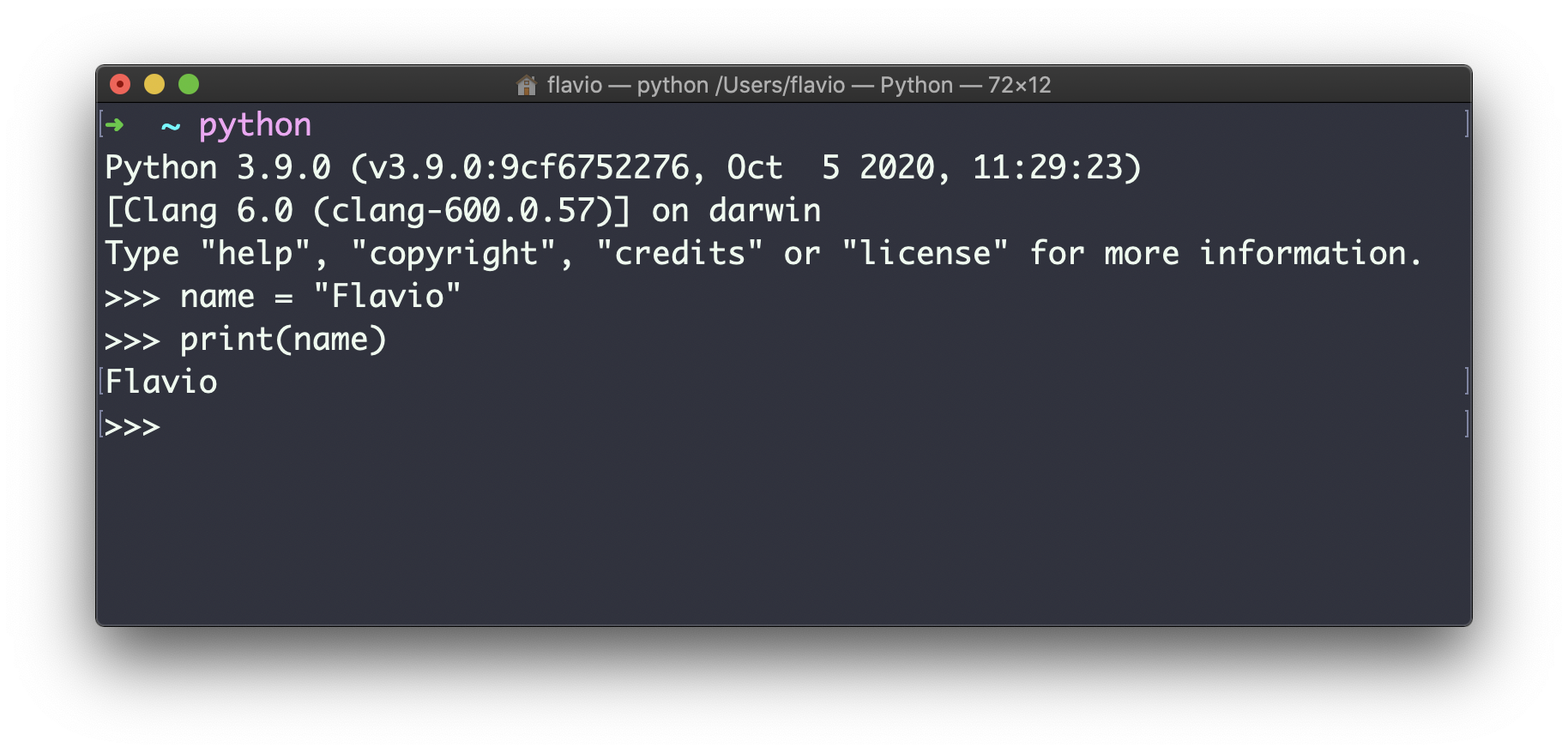
Observa el símbolo >>>  y el cursor después de eso. Puedes escribir cualquier código Python ahí y presionar la tecla enter para ejecutarlo.

Por ejemplo, intenta definir una nueva variable usando

nombre = "Flavio"

y luego imprime su valor, usando print():

print(nombre)

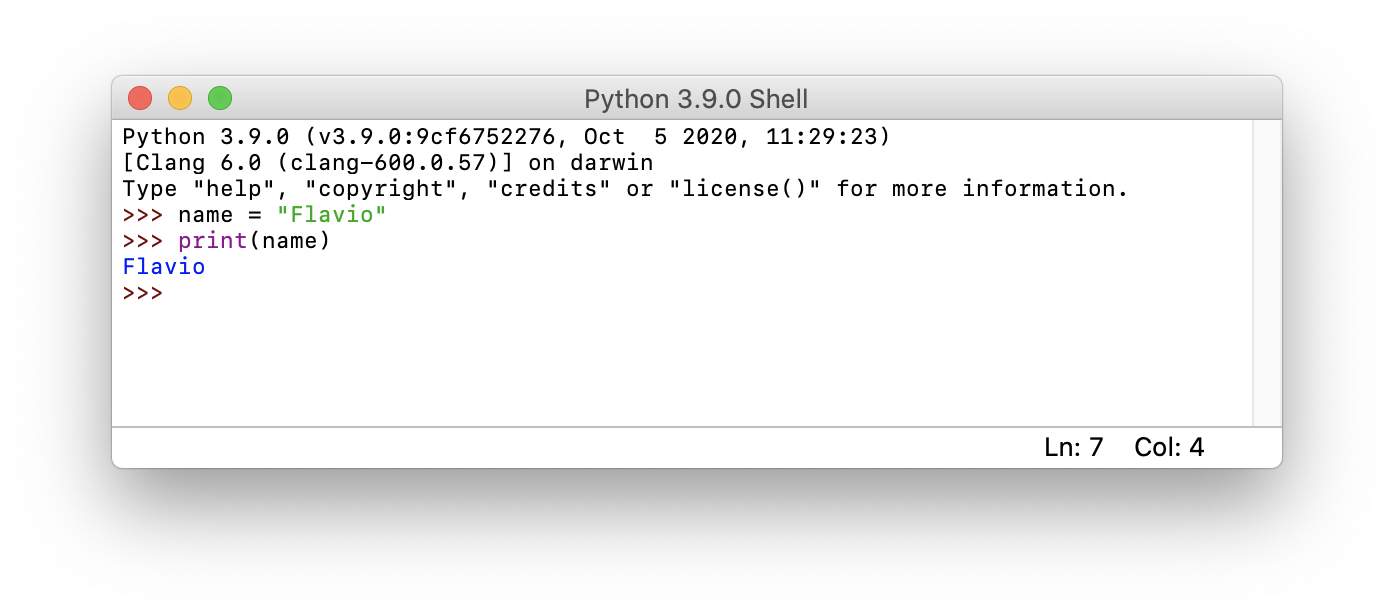


**Nota:** en el REPL, también puedes escribir el nombre, presionar la tecla enter y obtendrás el valor de vuelta. Pero en un programa, no verás ningún resultado si lo haces; deberás usar print() en su lugar.

Cualquier línea de Python que escribas ahí se ejecutará inmediatamente.

Escribe quit() para salir de este REPL de Python.

Puedes acceder al mismo mensaje interactivo utilizando la aplicación IDLE que Python instala automáticamente:



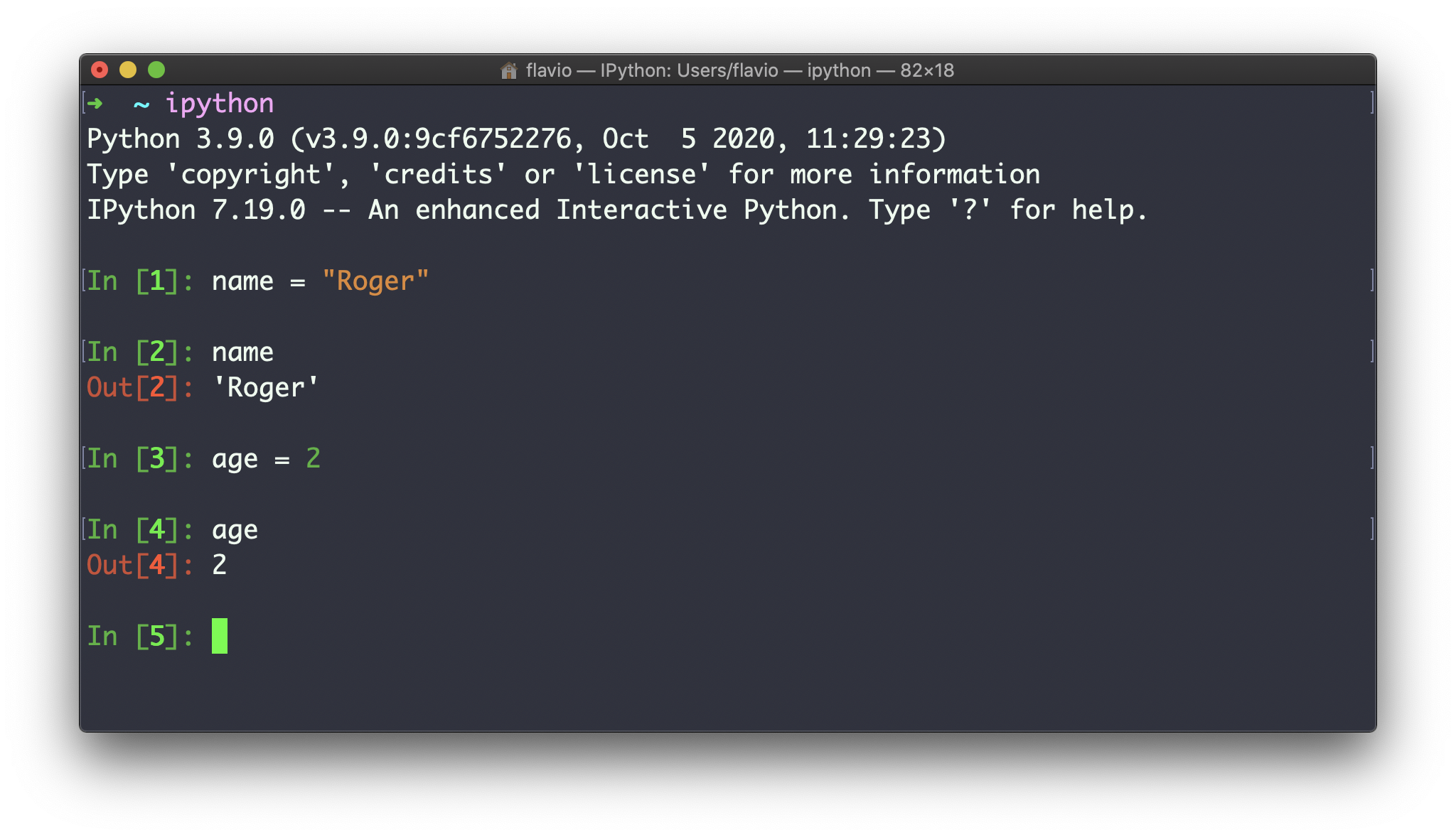
Esto te podría ser más conveniente porque con el mouse puedes moverte y copiar/pegar más fácilmente que con el terminal.

Esos son los básicos que vienen con Python de forma predeterminada. Sin embargo, recomiendo que instales [IPython](https://ipython.org/), probablemente la mejor aplicación REPL de línea de comandos que puedas encontrar.

Instálalo con

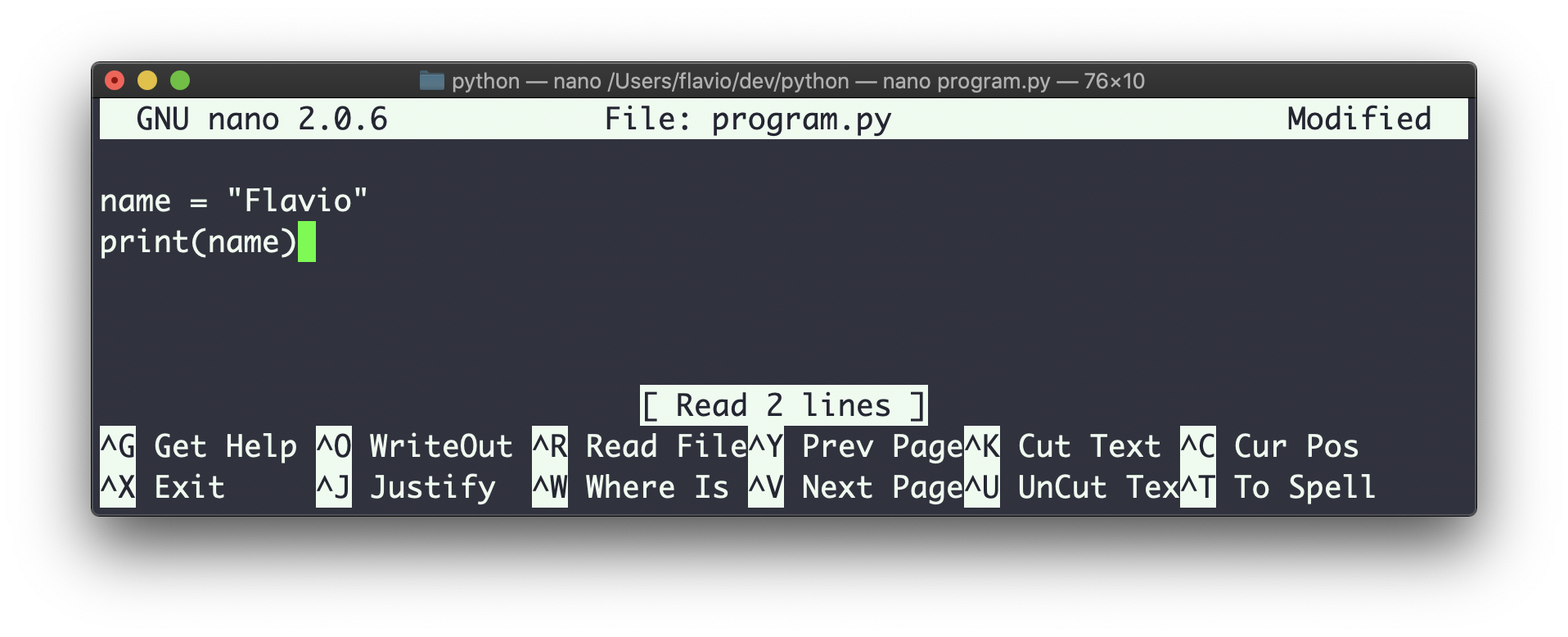
pip install ipython

Asegúrate de que los binarios pip estén en tu *path*(ruta), y ejecuta ipython:

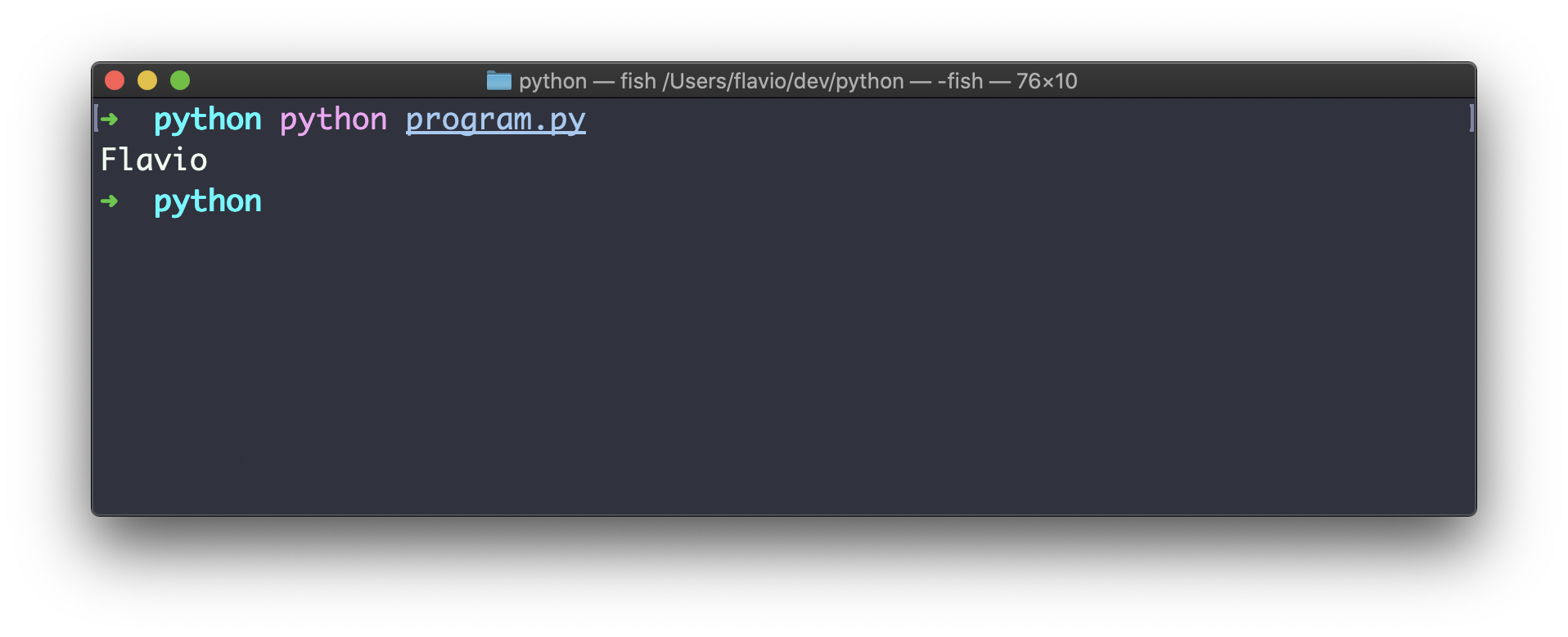


ipython es otra interfaz que te permite trabajar con un REPL de Python y proporciona algunas características interesantes como resaltado de sintaxis, finalización de código y mucho más.

La segunda forma de ejecutar un programa Python es escribir el código de tu programa Python en un archivo, por ejemplo program.py:



y luego ejecútalo con python program.py:



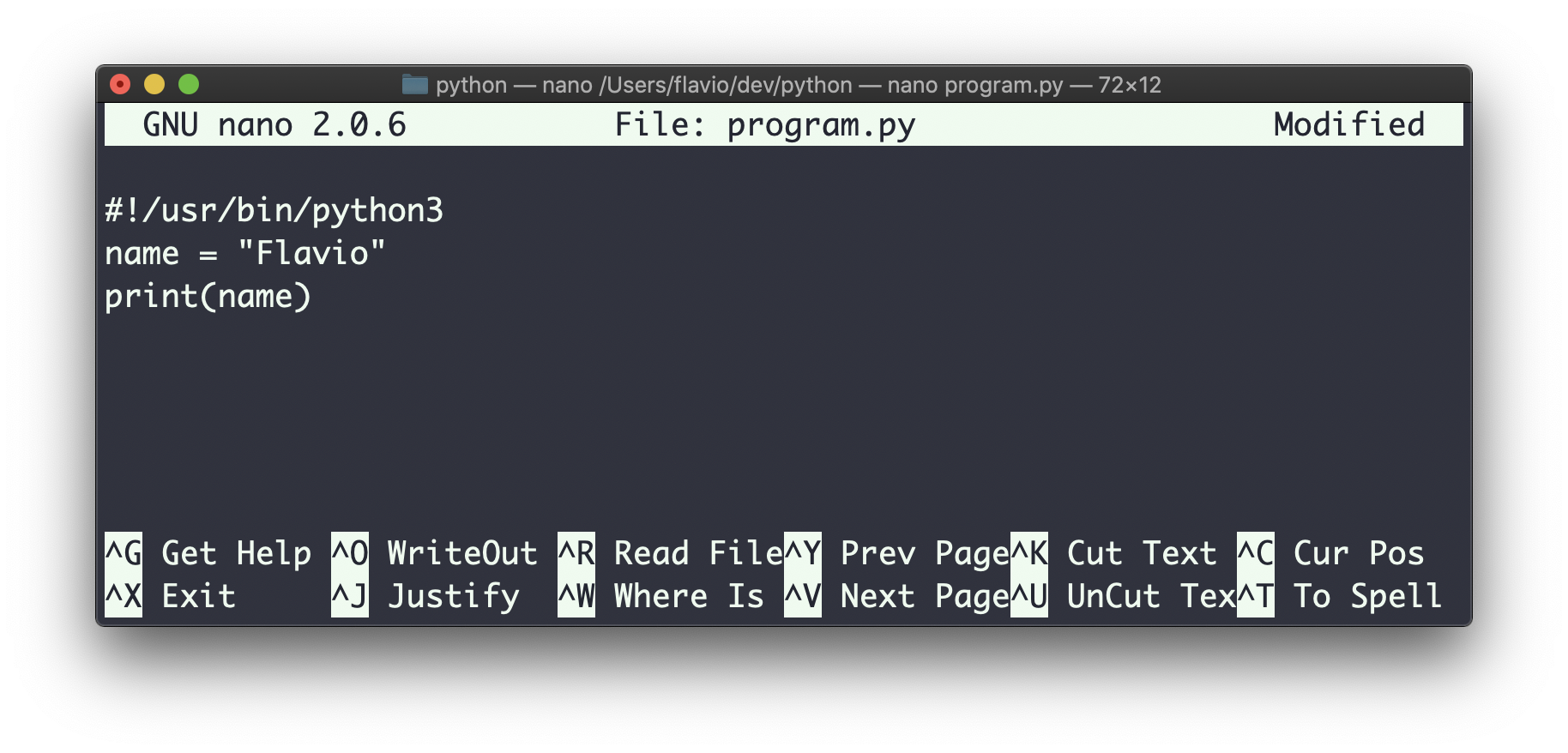
Ten en cuenta que guardamos los programas de Python con la extensión .py, eso es una convención.

En este caso, el programa se ejecuta como un todo, no una línea a la vez. Y así es como normalmente ejecutamos los programas.

Usamos REPL para la creación rápida de prototipos y para el aprendizaje.

En Linux y macOS, un programa Python también se puede transformar en un script de shell, anteponiendo todo su contenido con una línea especial que indica qué ejecutable usar para ejecutarlo.

En mi sistema, el ejecutable de Python se encuentra en /usr/bin/python3, así que escribo #!/usr/bin/python3 en la primera línea:

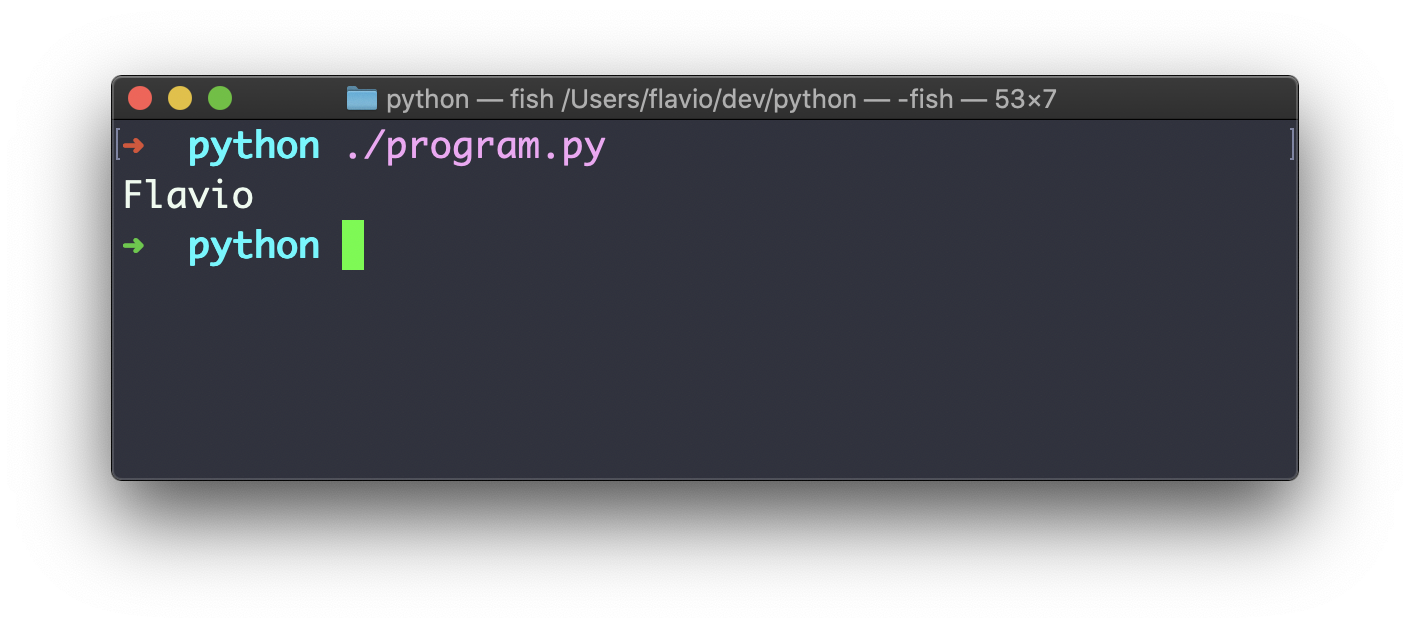


Entonces puedo establecer el permiso de ejecución en el archivo:

chmod u+x program.py

y puedo ejecutar el programa con

./program.py



Esto es especialmente útil cuando escribes scripts que interactúan con el terminal.

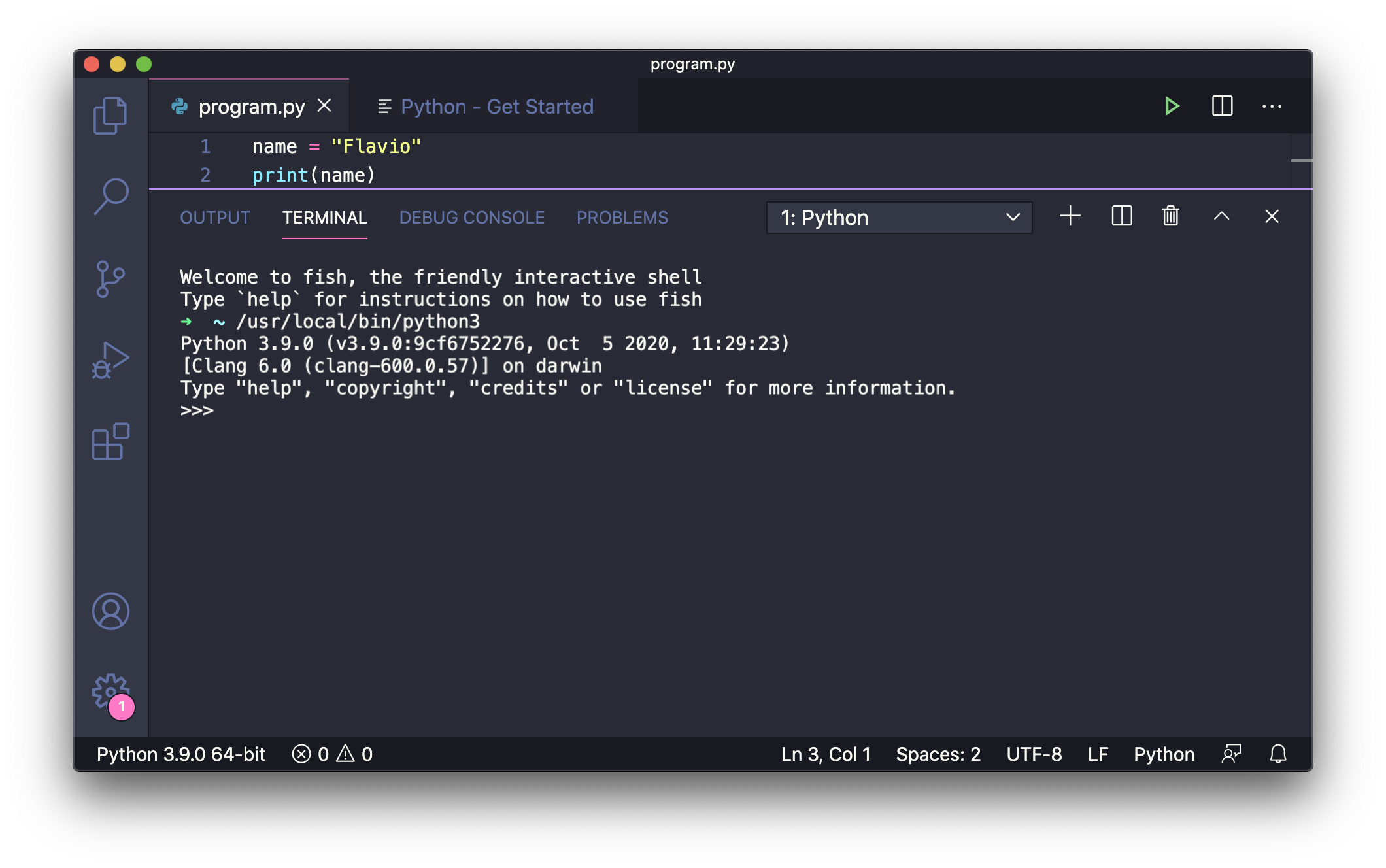
Tenemos muchas otras formas de ejecutar programas Python.

Una de ellas es usando VS Code, y en particular la extensión oficial de Python de Microsoft:

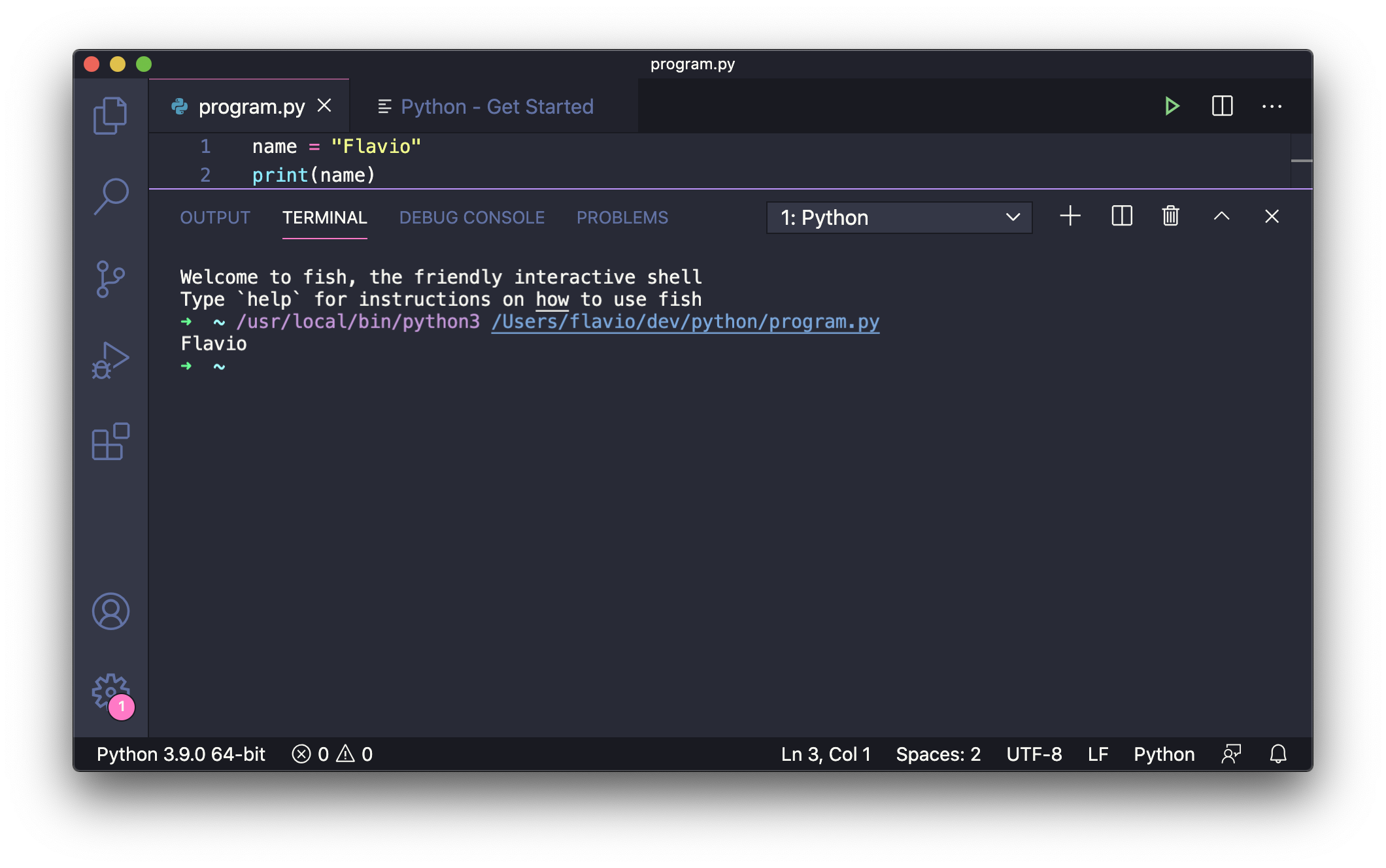


Después de instalar esta extensión, tendrás autocompletado de código Python y verificación de errores, formateo automático y verificación de código con pylint, y algunos comandos especiales, que incluyen:

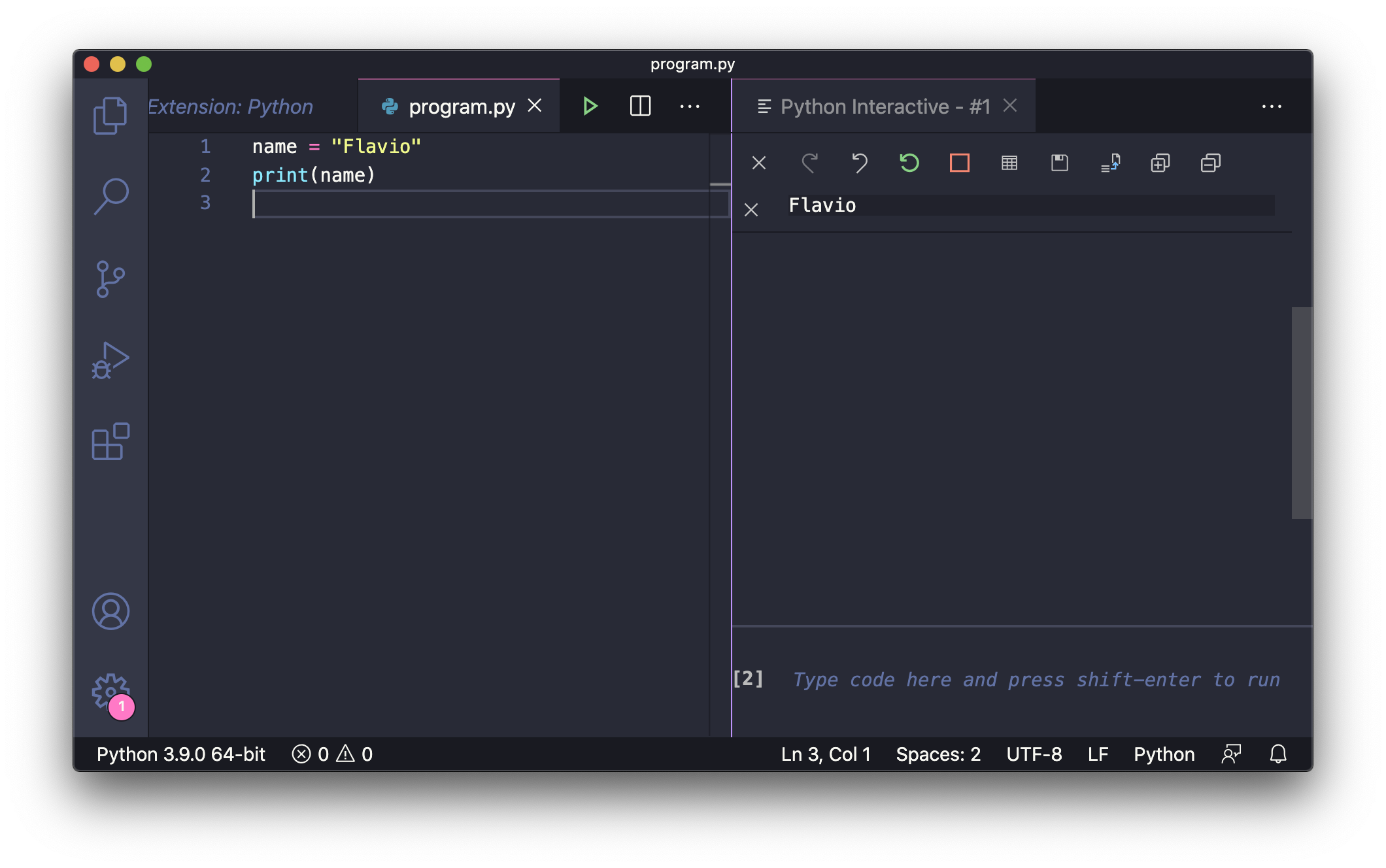
***Python: start REPL*** para ejecutar REPL en la terminal integrada:



**Python: Run Python File in Terminal** para ejecutar el archivo actual en la terminal:

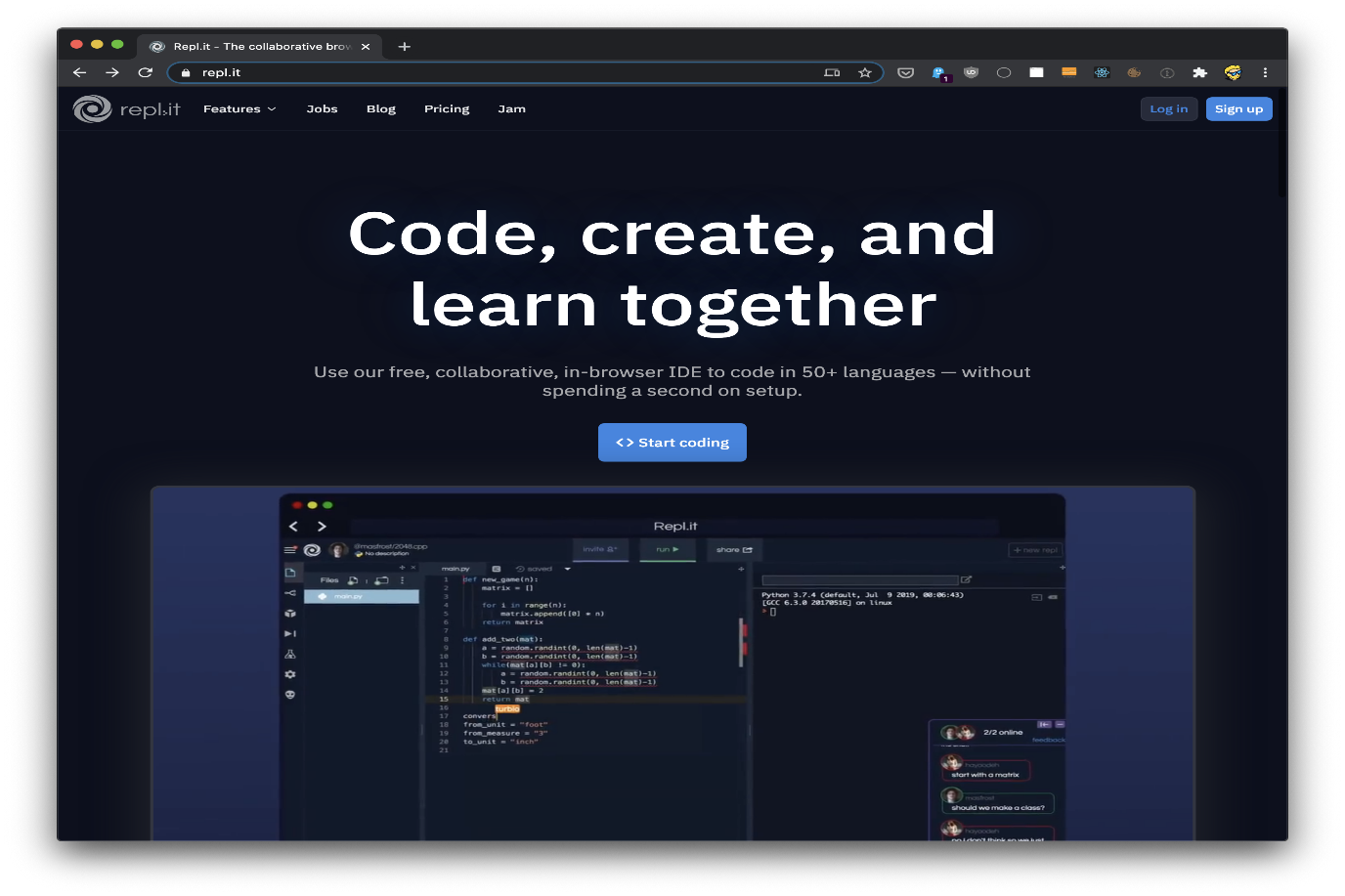


**Python: Run Current File in Python Interactive Window***(Ejecutar archivo actual en Ventana Interactiva Python)*

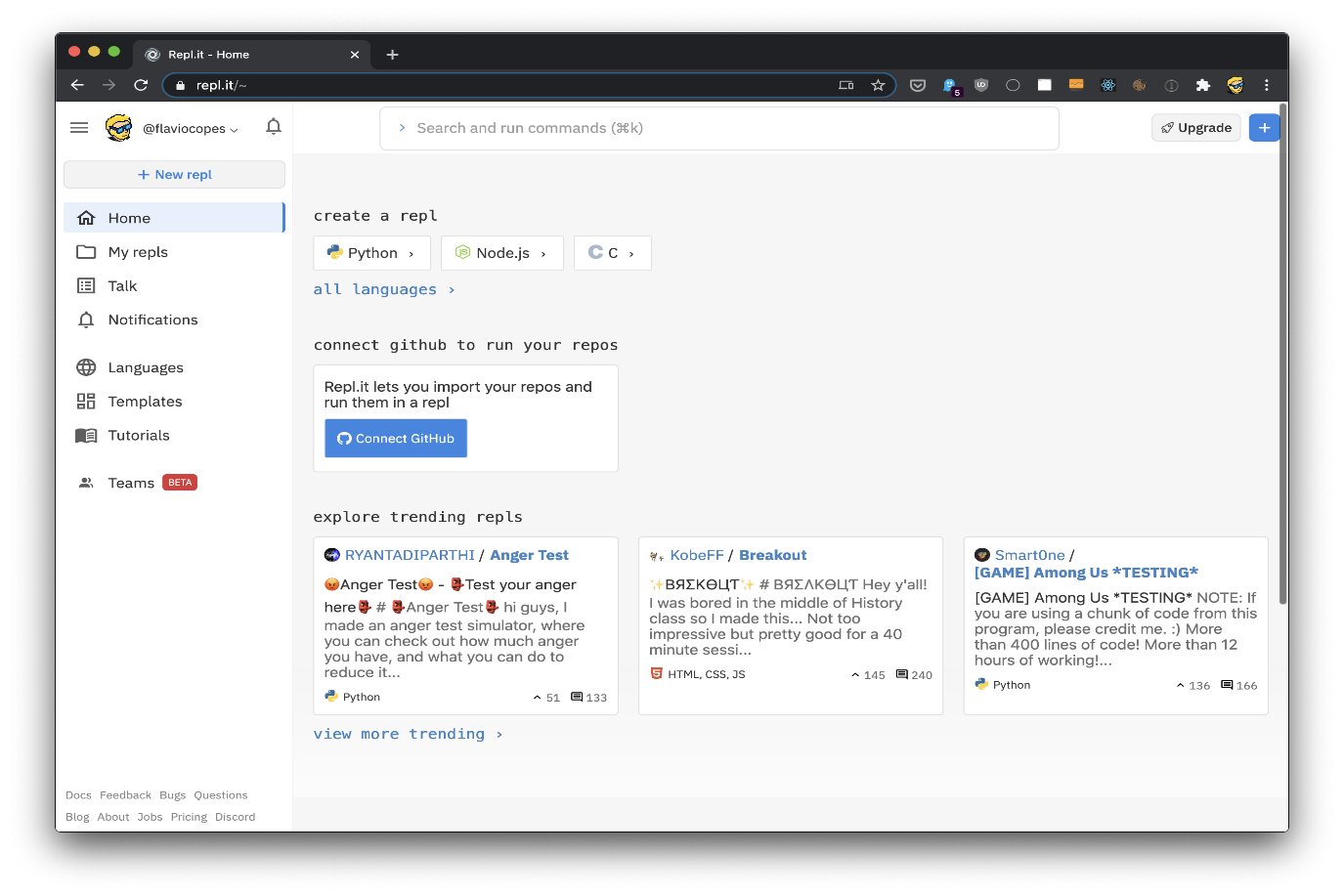


y muchos más. Simplemente abre la paleta de comandos (View -> Command palette, o Cmd/Ctrl+Shift+P ) y escribe python para ver todos los comandos relacionados con Python:

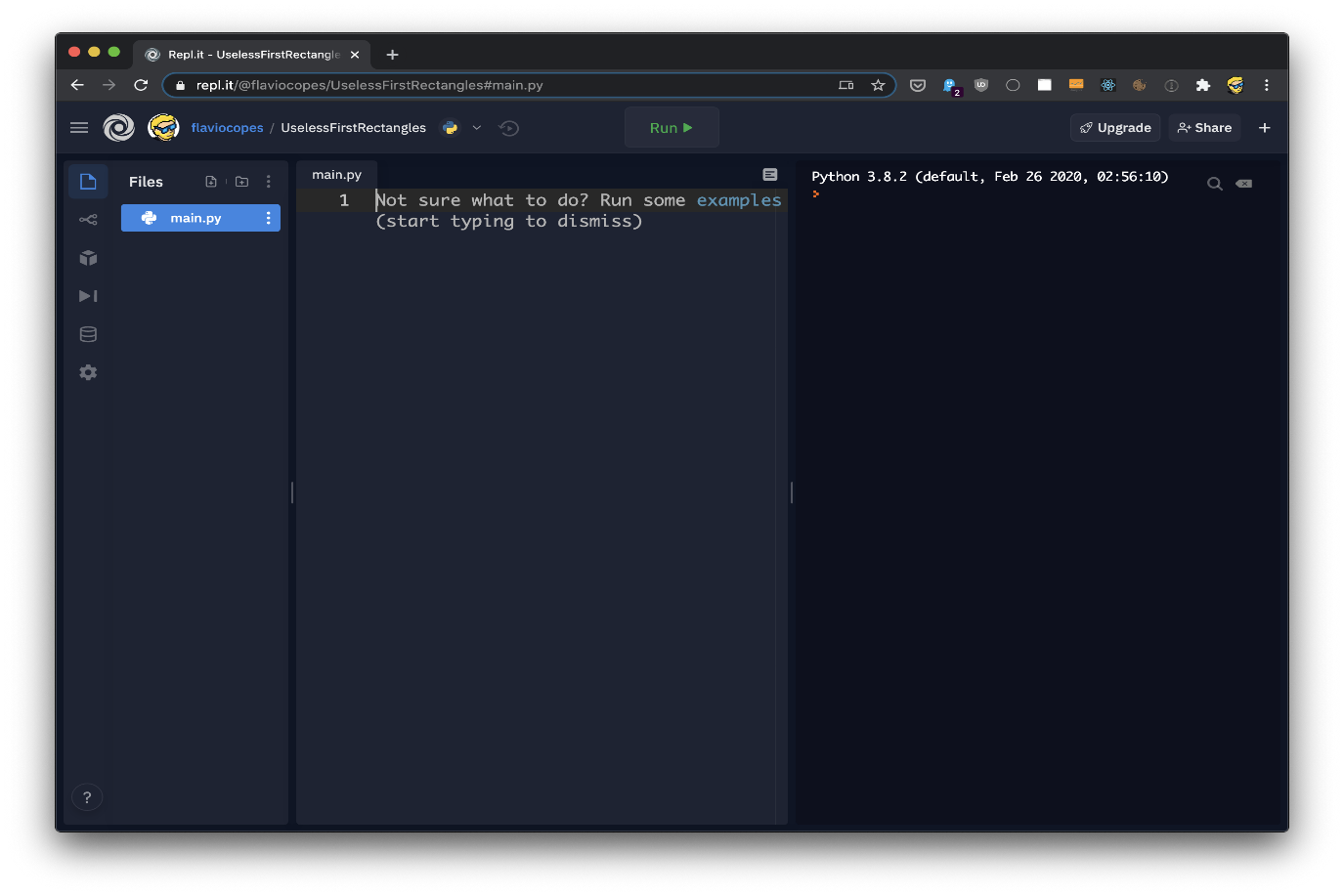
Otra forma de ejecutar fácilmente el código Python es usar [*repl.it*](https://replit.com/), un sitio web muy agradable que proporciona un entorno de programación en el que puedes crear y ejecutar tus aplicaciones, en cualquier lenguaje, incluido Python:



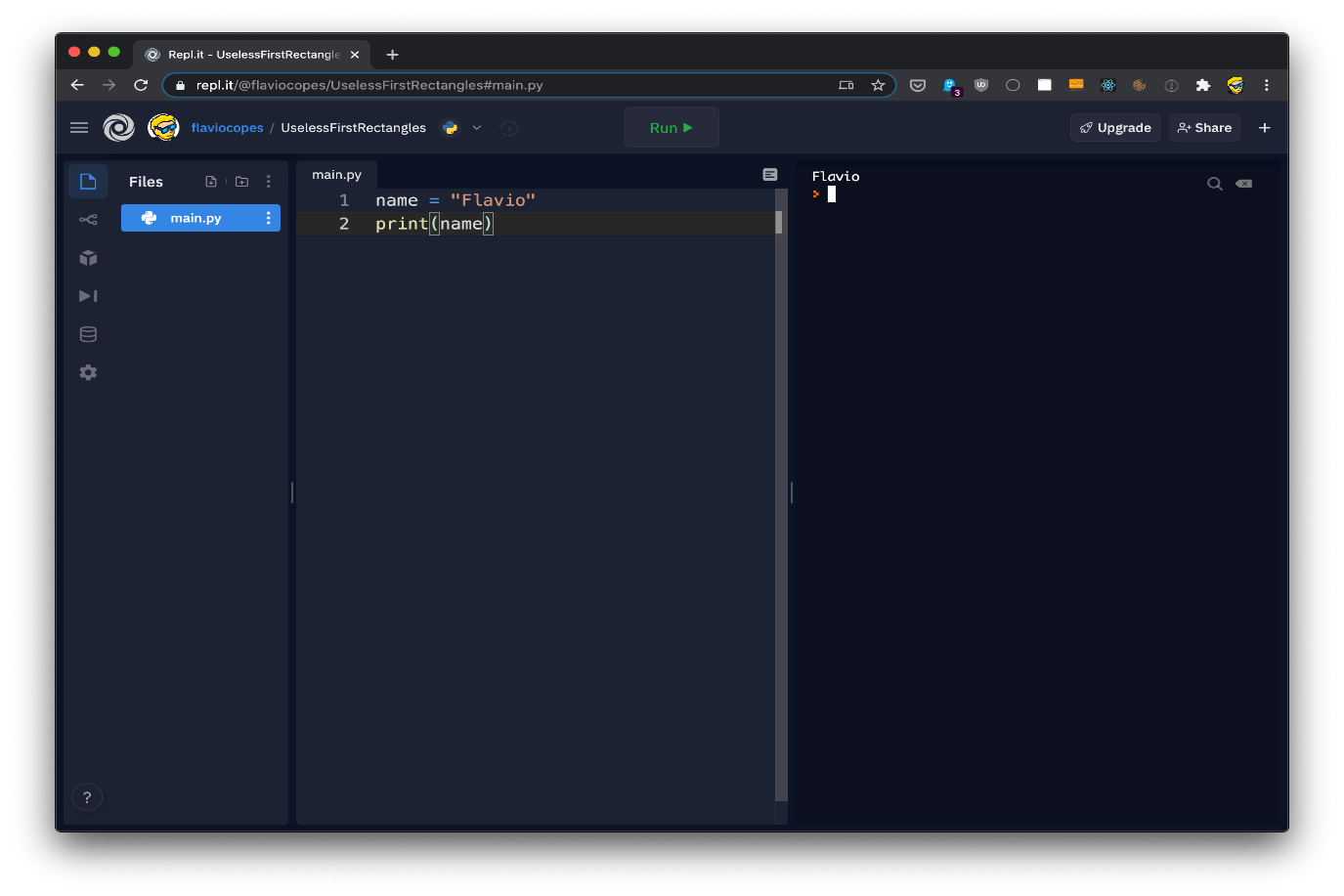
Regístrate (es gratis), luego, en ***create a repl***, haz clic en Python:



e inmediatamente se mostrará un editor con un archivo main.py, listo para ser llenado con una gran cantidad de código Python:



Una vez que tengas algo de código, haz clic en ***Run***para ejecutarlo en el lado derecho de la ventana:



Creemos que *repl.it* es útil porque:

* puedes compartir código fácilmente con solo compartir el enlace
* varias personas pueden trabajar en el mismo código
* puede albergar grandes programas de ejecución
* puedes instalar paquetes
* proporciona una base de datos de clave-valor para aplicaciones más complejas

**Python 2 vs Python 3**

Un tema clave que debemos abordar, desde el principio, es la discusión de Python 2 vs Python 3.

Python 3 se introdujo en 2008 y ha estado en desarrollo como la versión principal de Python, mientras que Python 2 se mantuvo con correcciones de errores y parches de seguridad hasta principios de 2020.

En esa fecha, se descontinuó el soporte a Python 2.

Muchos programas todavía se escriben con Python 2, y las organizaciones aún trabajan activamente en ellos, porque la migración a Python 3 no es trivial y requeriría mucho trabajo actualizar esos programas. Y las migraciones grandes e importantes siempre introducen nuevos errores.

Pero el código nuevo, a menos que tengas que cumplir con las reglas establecidas por tu organización que obliguen a usar Python 2, siempre debe escribirse en Python 3.

Este libro se enfoca en Python 3.

**Básicos en Python**

**Variables en Python**

Podemos crear una nueva variable de Python asignando un valor a una etiqueta, usando el operador de asignación =.

En este ejemplo asignamos una cadena con el valor "Roger" a nombre:

nombre = "Roger"

Aquí tienes un ejemplo con un número:

edad = 8

Un nombre de variable puede estar compuesto por caracteres, números y el carácter de subrayado \_. No puede empezar con un número. Todos estos son nombres de variables **válidos**:

nombre1

EDAD

eDAD

a11111

mi\_nombre

\_nombre

Estos son nombres de variables **inválidos**:

123

prueba!

nombre%

Aparte de eso, cualquier cosa es válida a menos que sea una **palabra clave de Python.** Hay algunas palabras clave como for, if, while, import y más.

No es necesario memorizarlos, ya que Python te alertará si usas uno de ellos como variable, y los reconocerás gradualmente como parte de la sintaxis del lenguaje de programación Python.

**Expresiones y declaraciones en Python**

Podemos *expresar*cualquier tipo de código que devuelva un valor. Por ejemplo

1 + 1

"Roger"

Una declaración, por otro lado, es una operación sobre un valor. Por ejemplo, estas son 2 declaraciones:

nombre = "Roger"

print(nombre)

Un programa está formado por una serie de declaraciones. Cada declaración se coloca en su propia línea, pero puedes usar un punto y coma para tener más de una declaración en una sola línea:

nombre = "Roger"; print(nombre)

**Comentar**

En un programa de Python, todo lo que aparece después de una marca # se ignora y se considera un comentario.

#esto es una linea comentada

nombre = "Roger" #esto es un comentario en-linea

**Indentación en Python**

La indentación en Python es significativa.

No puede indentar aleatoriamente así:

nombre = "Flavio"

print(nombre)

Algunos otros lenguajes no tienen espacios en blanco significativos, pero en Python, el indentado es importante.

En este caso, si intentas ejecutar este programa, obtendrás un error IndentationError: unexpected indent, porque el indentado tiene un significado especial.

Todo lo indentado pertenece a un bloque, como una declaración de control o un bloque condicional, o una función o cuerpo de clase. Veremos más sobre ellos más adelante.

**Tipos de datos en Python**

Python tiene varios tipos integrados.

Si creas la variable nombre asignándole el valor "Roger", automáticamente esta variable ahora representa un tipo de datos **String**(cadena)

nombre = "Roger"

Puedes verificar el tipo de una variable usando la función type(), pasando la variable como un argumento y luego comparando el resultado con str

nombre = "Roger"

type(nombre) == str #True

O usando isinstance():

nombre = "Roger"

isinstance(nombre, str) #True

Ten en cuenta que para ver el valor True (verdadero) en Python, fuera de un REPL, debes envolver este código dentro de print(), pero para mayor claridad, evitamos usarlo

Usamos la clase str aquí, pero lo mismo funciona para otros tipos de datos.

Primero, tenemos números. Los números enteros se representan mediante la clase int. Los números flotantes,decimales o fracciones, son de tipo float:

edad = 1

type(edad) == int #True

fraccion = 0.1

type(fraccion) == float #True

Viste cómo crear un tipo a partir de un valor literal, como este:

nombre = "Flavio"

edad = 20

Python detecta automáticamente el tipo a partir del tipo de valor.

También puedes crear una variable de un tipo específico utilizando el constructor de clase, pasando un valor literal o un nombre de variable:

nombre = str("Flavio")

otroNombre = str(nombre)

También puedes convertir de un tipo a otro utilizando el constructor de clases. Python intentará determinar el valor correcto, por ejemplo, extrayendo un número de una cadena:

edad = int("20")

print(edad) #20

fraccion = 0.1

fraccionEntera = int(fraccion)

print(fraccionEntera) #0

A esto se le llama [**casting**](https://es.wikipedia.org/wiki/Conversi%C3%B3n_de_tipos). Por supuesto, es posible que esta conversión no siempre funcione según el valor pasado. Si escribes prueba en lugar de 20 en la cadena anterior, obtendrás unValueError: invalid literal for int() with base 10: 'prueba'.  
Esos son solo los conceptos básicos de los tipos. Tenemos muchos más tipos en Python:

* complex para números complejos
* bool para booleanos(verdadero/falso)
* list para listas
* tuple para tuplas
* range para rangos
* dict para diccionarios
* set para conjuntos(*sets*)

¡y más!

Los exploraremos todos pronto.

**Operadores en Python**

Los operadores de Python son símbolos que usamos para ejecutar operaciones sobre valores y variables.

Podemos dividir los operadores en función del tipo de operación que realizan:

* Operador de asignación
* Operadores aritméticos
* Operadores de comparación
* Operadores lógicos
* Operadores bit a bit

además de algunos interesantes como is e in.

**Operador de asignación en Python**

El operador de asignación se utiliza para asignar un valor a una variable:

edad = 8

O para asignar un valor de variable a otra variable:

edad = 8

otraVariable = edad

Desde Python 3.8, el **operador walrus**(*morsa*) :=  se usa para asignar un valor a una variable como parte de otra operación. Por ejemplo, dentro de un if o en la parte condicional de un bucle. Más sobre eso luego.

**Operadores aritméticos en Python**

Python tiene varios operadores aritméticos: +,-, *, / (división), %(resto),*\*\*(exponenciación) y // (división de piso):

1 + 1 #2

2 - 1 #1

2 \* 2 #4

4 / 2 #2

4 % 3 #1

4 \*\* 2 #16

4 // 2 #2

Ten en cuenta que no necesitas un espacio entre los operandos, pero es bueno para la legibilidad.

- también funciona como un operador unario negativo:

print(-4) #-4

+ también se usa para concatenar valores de cadena:

"Roger" + " es un buen perro"

#Roger es un buen perro

Podemos combinar el operador de asignación con operadores aritméticos:

* +=
* -=
* \*=
* /=
* %=
* ...y así

Ejemplo:

edad = 8

edad += 1

#edad es ahora 9

**Operadores de comparación en Python**

Python define algunos operadores de comparación:

* ==
* !=
* >
* <
* >=
* <=

Puedes usar esos operadores para obtener un valor booleano (True o False) según el resultado:

a = 1

b = 2

a == b #False

a != b #True

a > b #False

a <= b #True

**Operadores booleanos en Python**

Python nos da los siguientes operadores booleanos:

* not
* and
* or

Cuando se trabaja con atributos True o False, estos funcionan como "Y", "O" y "NO" lógicos, y a menudo se usan en la evaluación de la expresión condicional if:

condicion1 = True

condicion2 = False

not condicion1 #False

condicion1 and condicion2 #False

condicion1 or condicion2 #True

De lo contrario, presta atención a una posible fuente de confusión:

or usado en una expresión devuelve el valor del primer operando que no es un valor falso (False, 0, '', []...). De lo contrario, devuelve el último operando.

print(0 or 1) ## 1

print(False or 'hey') ## 'hey'

print('hola' or 'hey') ## 'hola'

print([] or False) ## 'False'

print(False or []) ## '[]'

Los documentos de Python lo describen como :

si "x" es falso, entonces "y", si no "x".

and solo evalúa el segundo argumento si el primero es verdadero. Entonces, si el primer argumento es falso (False, 0, '', []...), devuelve ese argumento. De lo contrario, evalúa el segundo argumento:

print(0 and 1) ## 0

print(1 and 0) ## 0

print(False and 'hey') ## False

print('hola' and 'hey') ## 'hey'

print([] and False ) ## []

print(False and [] ) ## False

Los documentos de Python lo describen como:

si "x" es falso, entonces "x", si no "y"

**Operadores bit a bit en Python**

Algunos operadores se utilizan para trabajar con bits y números binarios:

* & realiza el binario AND
* | realiza el binario OR
* ^ realiza una operación binaria XOR
* ~ realiza una operación binaria NOT
* << [operación shift left](https://es.wikipedia.org/wiki/Operador_a_nivel_de_bits#Operaciones_de_desplazamiento_y_rotaci%C3%B3n)
* >> [operación shift right](https://es.wikipedia.org/wiki/Operador_a_nivel_de_bits#Operaciones_de_desplazamiento_y_rotaci%C3%B3n)

Los operadores bit a bit se utilizan raramente, solo en situaciones muy específicas, pero vale la pena mencionarlos.

**is e in en Python**

is se llama**el operador de identidad**. Se utiliza para comparar dos objetos y devuelve verdadero si ambos son el mismo objeto. Más sobre objetos luego.

in se llama **el operador de membresía**. Se utiliza para saber si un valor está contenido en una lista o en otra secuencia. Más sobre listas y otras secuencias luego.

**El operador ternario en Python**

El operador ternario en Python permite definir rápidamente un condicional.

Digamos que tienes una función que compara una variable de edad con el valor 18 y devuelve True o False según el resultado.

En lugar de escribir:

def es\_adulto(edad):

if edad > 18:

return True

else:

return False

Puedes implementarlo con el operador ternario de esta manera:

def es\_adulto(edad):

return True if edad > 18 else False

Primero defines el resultado si la condición es verdadera, luego evalúas la condición, luego defines el resultado si la condición es falsa:

<resultado\_si\_es\_verdadero> if <condicion> else <resultado\_si\_es\_falso>

**Cadenas en Python**

Una cadena en Python es una serie de caracteres entre comillas o comillas dobles:

"Roger"

'Roger'

Puedes asignar un valor de cadena a una variable:

nombre = "Roger"

Puedes concatenar dos cadenas usando el operador +:

frase = "Roger" + " es un buen perro"

Puedes agregar a una cadena usando +=:

nombre = "Roger"

nombre += " es un buen perro"

print(nombre) #Roger es un buen perro

Puedes convertir un número en una cadena usando el constructor de clase str:

str(8) #"8"

Esto es esencial para concatenar un número a una cadena:

print("Roger tiene " + str(8) + " a\_os de edad")

#Roger tiene 8 a\_os de edad

Una cadena puede ser de varias líneas cuando se define con una sintaxis especial, encerrando la cadena en un conjunto de 3 comillas:

print("""Roger tiene

8

a\_os de edad

""")

#comillas dobles o simples

print('''

Roger tiene

8

a\_os de edad

''')

Una cadena tiene un conjunto de métodos integrados, como:

* isalpha() para comprobar si una cadena contiene solo caracteres y no está vacía
* isalnum() para comprobar si una cadena contiene caracteres o dígitos y no está vacía
* isdecimal() para comprobar si una cadena contiene dígitos y no está vacía
* lower() para obtener una versión en minúsculas de una cadena
* islower() para comprobar si una cadena está en minúsculas
* upper() para obtener una versión en mayúsculas de una cadena
* isupper() para comprobar si una cadena está en mayúsculas
* title() para obtener una versión capitalizada de una cadena
* startsswith() para comprobar si la cadena comienza con una subcadena específica
* endswith() para comprobar si la cadena termina con una subcadena específica
* replace() para reemplazar una parte de una cadena
* split() para dividir una cadena en un separador de caracteres específico
* strip() para recortar el espacio en blanco de una cadena
* join() para agregar nuevas letras a una cadena
* find() para encontrar la posición de una subcadena

Y muchas mas.

Ninguno de esos métodos altera la cadena original. En su lugar, devuelven una cadena nueva y modificada. Por ejemplo:

nombre = "Roger"

print(nombre.lower()) #"roger"

print(nombre) #"Roger"

También puedes usar algunas funciones globales para trabajar con cadenas.

En particular, pienso en len(), que te da la longitud de una cadena:

nombre = "Roger"

print(len(nombre)) #5

El operador in te permite verificar si una cadena contiene una subcadena:

nombre = "Roger"

print("ger" in nombre) #True

Escapar es una forma de agregar caracteres especiales a una cadena.

Por ejemplo, ¿cómo agregas una comilla doble en una cadena que está envuelta en comillas dobles?

nombre = "Roger"

"Ro"Ger" no funcionará, ya que Python pensará que la cadena termina en "Ro".

El camino a seguir es hacer escapar a a las comillas dobles dentro de la cadena, con el carácter \ (barra invertida):

nombre = "Ro\"ger"

#"Ro"ger"

Esto también se aplica a las comillas simples \' y a los caracteres de formateo especial como \t para tabulación, \n para nueva línea y \\ para la barra invertida.

Dada una cadena, puedes obtener sus caracteres usando corchetes cuadrados para obtener un elemento específico, dado su índice, comenzando desde cero [0]:

nombre = "Roger"

nombre[0] #'R'

nombre[1] #'o'

nombre[2] #'g'

El uso de un número negativo comenzará a contar desde el final:

nombre = "Roger"

nombre[-1] #"r"

También puedes usar un rango, usando lo que llamamos **rebanar**(slicing):

nombre = "Roger"

nombre[0:2] #"Ro"

nombre[:2] #"Ro"

nombre[2:] #"ger"

**Booleanos en Python**

Python proporciona el tipo bool , que puede tener dos valores: True (Verdadero) y False (Falso) (*capitalizado*).

hecho = False

hecho = True

Los booleanos son especialmente útiles con estructuras de control condicionales como declaraciones if :

hecho = True

if hecho:

# Ejecuta algun codigo

else:

# Ejecuta algun otro codigo

Al evaluar un valor para True o False, si el valor no es un bool , tenemos algunas reglas dependiendo del tipo que estemos verificando:

* los números son siempre True excepto el número 0,
* las cadenas son False solo cuando están vacías '',
* las listas, tuplas, conjuntos y diccionarios son False solo cuando están vacíos.

Puedes verificar si un valor es booleano de esta manera:

hecho = True

type(hecho) == bool #True

O usando isinstance(), pasando 2 argumentos: la variable y la clase bool :

hecho = True

isinstance(hecho, bool) #True

La función global any() también es muy útil cuando se trabaja con valores booleanos, ya que devuelve True si **alguno de los valores** del iterable (lista, por ejemplo) pasado como argumento es True :

libro\_1\_leido = True

libro\_2\_leido = False

algun\_libro\_leido = any([libro\_1\_leido, libro\_2\_leido])

#True

La función global all() es la misma, pero devuelve True si**todos los valores** que se le pasan son True:

ingredientes\_comprados = True

comida\_preparada = False

list\_para\_servir = all([ingredientes\_comprados, comida\_preparada])

#False

**Números en Python**

Los números en Python pueden ser de 3 tipos: int, float y complex.

**Números enteros en Python**

Los números enteros se representan mediante la clase int . Puedes definir un entero mediante un valor literal:

edad = 8

También puedes definir un número entero usando el constructor int():

edad = int(8)

Para comprobar si una variable es de tipo int, puedes utilizar la función global type():

type(edad) == int #True

**Números flotantes en Python**

Los números flotantes (fracciones, decimales) son de tipo float. Puedes definir un entero usando un valor literal:

fraccion = 0.1

O usando el constructor float():

fraccion = float(0.1)

Para comprobar si una variable es de tipo float, puede utilizar la función global type() :

type(fraccion) == float #True

**Números complejos en Python**

Los números complejos son de tipo complex.

Puedes definirlos usando un valor literal:

numeroComplejo = 2+3j

o usando el constructor complex():

numeroComplejo = complex(2, 3)

Una vez que tengas un número complejo, puedes obtener su parte real e imaginaria:

numeroComplejo.real #2.0

numeroComplejo.imag #3.0

Nuevamente, para verificar si una variable es de tipo complex, puedes usar la función global type():

type(numeroComplejo) == complex #True

**Operaciones aritméticas con números en Python**

Puedes realizar operaciones aritméticas con números, utilizando los operadores aritméticos: +, -, \*, /(división),% (resto), \*\* (exponenciación) y // (división de piso):

1 + 1 #2

2 - 1 #1

2 \* 2 #4

4 / 2 #2

4 % 3 #1

4 \*\* 2 #16

4 // 2 #2

y puedes utilizar los operadores de asignación compuesta

* +=
* -=
* \*=
* /=
* %=
* y así...

para realizar rápidamente operaciones en variables, también:

edad = 8

edad += 1

**Funciones integradas en Python**

Hay 2 funciones integradas que ayudan con los números:

abs() devuelve el valor absoluto de un número.

round() dado un número, devuelve su valor redondeado al entero más cercano:

round(0.12) #0

Puedes especificar un segundo parámetro para establecer la precisión del punto decimal:

round(0.12, 1) #0.1

Varias otras funciones de utilidad matemática y constantes son proporcionadas por la biblioteca estándar de Python:

* el paquete math proporciona funciones y constantes matemáticas generales.
* el paquete cmath proporciona utilidades para trabajar con números complejos.
* el paquete decimal proporciona utilidades para trabajar con decimales y números decimales.
* el paquete fractions proporciona utilidades para trabajar con números racionales.

Exploraremos algunos de ellos por separado más adelante.

**Constantes en Python**

Python no tiene forma de exigir que una variable sea una constante.

Lo más cercano que puedes obtener es usar una enumeración:

class Constantes(Enum):

ANCHURA = 1024

ALTURA = 256

Y obtener cada valor usando, por ejemplo, Constantes.ANCHURA.value.

Nadie puede reasignar ese valor.

De lo contrario, si deseas confiar en las convenciones de nomenclatura, puedes adherirte a esta: declara las variables que nunca deben cambiar, **en mayúsculas**:

ANCHURA = 1024

Nadie evitará que sobrescribas este valor, y Python no lo detendrá.

Eso es lo que hace la mayoría del código Python que verás.

**Enums en Python**

Los Enums son nombres legibles que están vinculados a un valor constante.

Para usar Enums, importa Enum desde el módulo de biblioteca estándar enum:

from enum import Enum

Luego puedes inicializar un nuevo Enum de esta manera:

class Estado(Enum):

INACTIVO = 0

ACTIVO = 1

Una vez que lo hagas, puedes hacer referencia a Estado.INACTIVO y Estado.ACTIVO, y sirven como constantes.

Ahora, si intentas imprimir Estado.ACTIVO, por ejemplo:

print(Estado.ACTIVO)

no devolverá 1, sino Estado.ACTIVO.

El mismo valor puede ser alcanzado por el número asignado en el enum: print(Estado(1)) devolverá Estado.ACTIVO. Lo mismo para usar la notación de corchetes Estado['ACTIVO'].

Sin embargo, puedes obtener el valor mediante  Estado.ACTIVO.value.

Puedes enlistar todos los valores posibles de un enum:

list(Estado) # [<Estado.INACTIVO: 0>, <Estado.ACTIVO: 1>]

Puedes contarlos:

len(Estado) # 2

**Entradas de usuario en Python**

En una aplicación de línea de comandos de Python, puedes mostrar información al usuario mediante la función print():

nombre = "Roger"

print(nombre)

También podemos aceptar la entrada del usuario, usando input():

print('¿Cual es tu edad?')

edad = input()

print('Tu edad es ' + edad)

Este enfoque recibe entradas en tiempo de ejecución, lo que significa que el programa detendrá la ejecución y esperará hasta que el usuario escriba algo y presione la tecla Enter.

También puedes realizar un procesamiento de entradas más complejo y aceptar la entrada en el momento de la invocación del programa, y ​​veremos cómo hacerlo más adelante.

Esto funciona para aplicaciones de línea de comandos. Otros tipos de aplicaciones necesitarán una forma diferente de aceptar la entrada.

**Declaraciones de control en Python**

Cuando se trata de valores booleanos y expresiones que devuelven un valor booleano en particular, podemos tomar decisiones y diferentes caminos en función de sus valores True o False.

En Python lo hacemos usando la declaración if:

condicion = True

if condicion == True:

# Hace algo

Cuando la prueba de condición se resuelve en True, como en el caso anterior, el bloque se ejecuta.

¿Qué es un bloque? Un bloque es la parte que tiene una indentación de un nivel (normalmente 4 espacios) a la derecha.

condicion = True

if condicion == True:

print("La condicion")

print("era VERDADERA")

El bloque puede estar formado por una sola línea, o también por varias líneas, y termina cuando regresa al nivel de indentación anterior:

condicion = True

if condicion == True:

print("La condicion")

print("era VERDADERA")

print("Fuera del IF")

En combinación con if, puedes tener un bloque else que se ejecute si la prueba de condición de if da como resultado False:

condicion = True

if condicion == True:

print("La condicion")

print("era VERDADERA")

else:

print("La condicion")

print("era FALSA")

Y puedes tener diferentes verificaciones vinculadas if con elif que se ejecutan si la verificación anterior fue falsa:

condicion = True

nombre = "Roger"

if condicion == True:

print("La condicion")

print("era VERDADERA")

elif nombre == "Roger":

print("Hola Roger")

else:

print("La condicion")

print("era FALSA")

En este caso, el segundo bloque se ejecuta si condicion es False y el valor de la variable nombre es "Roger".

En una declaración if, puedes tener solo una verificación if y else, pero varias series de verificaciones elif:

condicion = True

nombre = "Roger"

if condicion == True:

print("La condicion")

print("era VERDADERA")

elif nombre == "Roger":

print("Hola Roger")

elif nombre == "Syd":

print("Hola Syd")

elif nombre == "Flavio":

print("Hola Flavio")

else:

print("La condicion")

print("era FALSA")

if y else también se pueden usar en un formato en-línea, lo que nos permite devolver un valor u otro en función de una condición.

Ejemplo:

a = 2

resultado = 2 if a == 0 else 3

print(resultado) # 3

**Listas en Python**

Las listas son una estructura de datos esencial de Python.

Te permiten agrupar varios valores y hacer referencia a todos ellos con un nombre común.

Por ejemplo:

perros = ["Roger", "Syd"]

Una lista puede contener valores de diferentes tipos:

items = ["Roger", 1, "Syd", True]

Puedes verificar si un elemento está contenido en una lista con el operador in:

print("Roger" in items) # True

Una lista también se puede definir como vacía:

items = []

Puedes hacer referencia a los elementos en una lista por su índice, comenzando desde cero:

items[0] # "Roger"

items[1] # 1

items[3] # True

Usando la misma notación, puedes cambiar el valor almacenado en un índice específico:

items[0] = "Roger"

También puedes utilizar el método index():

items.index(0) # "Roger"

items.index(1) # 1

Al igual que con las cadenas, el uso de un índice negativo comenzará a buscar desde el final:

items[-1] # True

También puedes extraer una parte de una lista, utilizando porciones(*slices*):

items[0:2] # ["Roger", 1]

items[2:] # ["Syd", True]

Obtén el número de elementos contenidos en una lista usando la función global len(), la misma que usamos para obtener la longitud de una cadena:

len(items) #4

Puedes agregar elementos a la lista mediante el método append():

items.append("Prueba")

o el método extend()

items.extend(["Prueba"])

También puedes utilizar el operador +=:

items += ["Prueba"]

# los items son ['Roger', 1, 'Syd', True, 'Prueba']

Consejo: con extend() o += no olvides los corchetes. No hagas items += "Prueba" or items.extend("Prueba") o Python agregará 6 caracteres individuales a la lista, lo que dará como resultado ['Roger', 1, 'Syd', True, 'P', 'r', 'u', 'e', 'b', 'a']

Elimina un elemento usando el método remove():

items.remove("Prueba")

Puedes agregar varios elementos usando

items += ["Prueba 1", "Prueba 2"]

#o

items.extend(["Prueba 1", "Prueba 2"])

Estos anexan el elemento al final de la lista.

Para agregar un elemento en el medio de una lista, en un índice específico, usa el método insert():

items.insert("Prueba", 1)

# agrega "Prueba" en el indice 1

Para agregar varios elementos en un índice específico, debes usar porciones(*slices*):

items[1:1] = ["Prueba 1", "Prueba 2"]

Ordena una lista usando el método sort():

items.sort()

Consejo: sort() solo funcionará si la lista contiene valores que se pueden comparar. Las cadenas y los enteros, por ejemplo, no se pueden comparar, y obtendrás un error como TypeError: '<' not supported between instances of 'int' and 'str' si lo intentas.

El método  sort() ordena primero las letras mayúsculas y luego las minúsculas. Para solucionar este problema, utiliza:

items.sort(key=str.lower)

El orden(*sort*) modifica el contenido de la lista original. Para evitarlo, puedes copiar el contenido de la lista utilizando

copiaitems = items[:]

o usa la función global sorted():

print(sorted(items, key=str.lower))

que devolverá una nueva lista, ordenada, en lugar de modificar la lista original.

**Tuplas en Python**

Las tuplas son otra estructura de datos fundamental de Python.

Te permiten crear grupos inmutables de objetos. Esto significa que una vez que se crea una tupla, no se puede modificar. No puedes agregar ni quitar elementos.

Se crean de forma similar a las listas, pero utilizando paréntesis en lugar de corchetes:

nombres = ("Roger", "Syd")

Una tupla está ordenada, como una lista, por lo que puedes obtener sus valores haciendo referencia a un valor de índice:

nombres[0] # "Roger"

nombres[1] # "Syd"

También puedes usar el método index() :

nombres.index('Roger') # 0

nombres.index('Syd') # 1

Al igual que con las cadenas y las listas, el uso de un índice negativo comenzará a buscar desde el final:

nombres[-1] # True

Puedes contar los elementos en una tupla con la función len():

len(nombres) # 2

Puedes verificar si un elemento está contenido en una tupla con el operador in:

print("Roger" in nombres) # True

También puedes extraer una parte de una tupla, utilizando porciones(*slices*):

nombres[0:2] # ('Roger', 'Syd')

nombres[1:] # ('Syd',)

Puedes crear una versión ordenada de una tupla usando la función global sorted():

sorted(nombres)

Puedes crear una nueva tupla a partir de tuplas utilizando el operador +:

nuevaTupla = nombres + ("Vanille", "Tina")

**Diccionarios en Python**

Los diccionarios son una estructura de datos de Python muy importante.

Mientras que las listas te permiten crear colecciones de valores, los diccionarios permiten crear colecciones de **pares clave/valor**.

Aquí hay un ejemplo de diccionario con un par clave/valor:

perro = { 'nombre': 'Roger' }

La clave puede ser cualquier valor inmutable como una cadena, un número o una tupla. El valor puede ser el que desees.

Un diccionario puede contener varios pares clave/valor:

perro = { 'nombre': 'Roger', 'edad': 8 }

Puedes acceder a valores de *claves*individuales utilizando esta notación:

perro['nombre'] # 'Roger'

perro['edad'] # 8

Usando la misma notación, puedes cambiar el valor almacenado en un índice específico:

perro['nombre'] = 'Syd'

Y otra forma es usar el método get(), que tiene una opción para agregar un valor predeterminado:

perro.get('nombre') # 'Roger'

perro.get('prueba', 'predeterminado') # 'predeterminado'

El método pop() recupera el valor de una clave y, posteriormente, elimina el elemento del diccionario.

perro.pop('nombre') # 'Roger'

El método popitem() recupera y elimina el último par clave/valor insertado en el diccionario:

perro.popitem()

Puedes verificar si una clave está contenida en un diccionario con el operador in:

'nombre' in perro # True

Obtén una lista con las claves en un diccionario usando el método keys(), pasando su resultado al constructor list():

list(perro.keys()) # ['nombre', 'edad']

Obtén los valores usando el método values() y las tuplas de pares clave/valor usando el método items():

print(list(perro.values()))

# ['Roger', 8]

print(list(perro.items()))

# [('nombre', 'Roger'), ('edad', 8)]

Obtén la longitud de un diccionario usando la función global len(), la misma que usamos para obtener la longitud de una cadena o los elementos de una lista:

len(perro) #2

Puedes agregar un nuevo par clave/valor al diccionario de esta manera:

perro['comida favorita'] = 'Carne'

Puedes eliminar un par clave/valor de un diccionario usando la declaración del:

del perro['comida favorita']

Para copiar un diccionario, use el método copy():

perroCopia = perro.copy()

**Conjuntos en Python**

Los conjuntos(*sets*) son otra estructura de datos importante de Python.

Podemos decir que funcionan como tuplas, pero no están ordenadas y son **mutables**.

O podemos decir que funcionan como diccionarios, pero no tienen claves.

También tienen una versión inmutable, llamada frozenset.

Puedes crear un conjunto usando esta sintaxis:

nombres = {"Roger", "Syd"}

Los conjuntos funcionan bien cuando se los considera conjuntos matemáticos.

Puedes intersecar dos conjuntos:

conjunto1 = {"Roger", "Syd"}

conjunto2 = {"Roger"}

interseccion = conjunto1 & conjunto2 #{'Roger'}

Puedes crear una unión de dos conjuntos:

conjunto1 = {"Roger", "Syd"}

conjunto2 = {"Luna"}

union = conjunto1 | conjunto2

#{'Syd', 'Luna', 'Roger'}

Puedes obtener la diferencia entre dos conjuntos:

conjunto1 = {"Roger", "Syd"}

conjunto2 = {"Roger"}

diferencia = conjunto1 - conjunto2 #{'Syd'}

Puedes comprobar si un conjunto es un superconjunto de otro (y, por supuesto, si un conjunto es un subconjunto de otro):

conjunto1 = {"Roger", "Syd"}

conjunto2 = {"Roger"}

esSuperconjunto = conjunto1 > conjunto2 # True

Puedes contar los elementos de un conjunto con la función global len():

nombres = {"Roger", "Syd"}

len(nombres) # 2

Puedes obtener una lista de los elementos de un conjunto pasando el conjunto al constructor list():

nombres = {"Roger", "Syd"}

list(nombres) #['Syd', 'Roger']

Puedes verificar si un elemento está contenido en un conjunto con el operador in:

print("Roger" in nombres) # True

**Funciones en Python**

Una función nos permite crear un conjunto de instrucciones que podemos ejecutar cuando sea necesario.

Las funciones son esenciales en Python y en muchos otros lenguajes de programación. Nos ayudan a crear programas significativos, porque nos permiten descomponer un programa en partes manejables y promueven la legibilidad y la reutilización del código.

Aquí hay una función de ejemplo llamada hola que imprime "¡Hola!":

def hola():

print('Hola!')

Esta es la **definición**de la función. Hay un nombre hola y un cuerpo, el conjunto de instrucciones, que es la parte que sigue a los dos puntos. Tiene una indentación de un nivel a la derecha.  
Para ejecutar esta función, debemos llamarla. Esta es la sintaxis para llamar a la función:

hola()

Podemos ejecutar esta función una o varias veces.

El nombre de la función, hola, es muy importante. Debe ser descriptivo, para que cualquiera que lo llame pueda imaginar lo que hace la función.

Una función puede aceptar uno o más parámetros:

def hola(nombre):

print('Hola ' + nombre + '!')

En este caso llamamos a la función pasando el argumento

hola('Roger')

Llamamos parámetros a los valores aceptados por la función dentro de la definición de la función, y argumentos a los valores que pasamos a la función cuando la llamamos. Es común confundirse con esta distinción.

Un argumento puede tener un valor predeterminado que se aplica si no se especifica el argumento:

def hola(nombre='mi amigo'):

print('Hola ' + nombre + '!')

hola()

# Hola mi amigo!

Así es como podemos aceptar múltiples parámetros:

def hola(nombre, edad):

print('Hola ' + nombre + ', tienes ' + str(edad) + ' a\_os de edad!')

En este caso llamamos a la función pasando un conjunto de argumentos:

hola('Roger', 8)

Los parámetros se pasan por referencia. Todos los tipos en Python son objetos, pero algunos de ellos son inmutables, incluidos enteros, booleanos, flotantes, cadenas y tuplas. Esto significa que si los pasas como parámetros y modificas su valor dentro de la función, el nuevo valor no se refleja fuera de la función:

def cambio(valor):

valor = 2

val = 1

cambio(val)

print(val) #1

Si pasas un objeto que no es inmutable y cambia una de sus propiedades, el cambio se reflejará en el exterior.

Una función puede devolver un valor mediante la declaración de return. Por ejemplo, en este caso devolvemos el nombre del parámetro nombre:

def hola(nombre):

print('Hola ' + nombre + '!')

return nombre

Cuando la función cumple con la declaración return, la función finaliza.

Podemos omitir el valor:

def hola(nombre):

print('Hola ' + nombre + '!')

return

Podemos tener la declaración de retorno dentro de un condicional, que es una forma común de finalizar una función si no se cumple una condición inicial:

def hola(nombre):

if not nombre:

return

print('Hola ' + nombre + '!')

Si llamamos a la función pasando un valor que se evalúa como False, como una cadena vacía, la función finaliza antes de llegar a la declaración print().

Puedes devolver varios valores utilizando valores separados por comas:

def hola(nombre):

print('Hola ' + nombre + '!')

return nombre, 'Roger', 8

En este caso, al llamar a hola('Syd'), el valor de retorno es una tupla que contiene esos 3 valores: ('Syd', 'Roger', 8).

**Objetos en Python**

Todo en Python es un objeto.

Incluso los valores de tipos primitivos básicos (enteros, cadenas, flotantes...) son objetos. Las listas son objetos, al igual que las tuplas, los diccionarios, todo.

Los objetos tienen **atributos**y **métodos**a los que se puede acceder mediante la sintaxis de punto.

Por ejemplo, intenta definir una nueva variable de tipo int:

edad = 8

edad ahora tiene acceso a las propiedades y métodos definidos para todos los objetos int.

Esto incluye, por ejemplo, el acceso a la parte real e imaginaria de ese número:

print(edad.real) # 8

print(edad.imag) # 0

print(edad.bit\_length()) #4

# el metodo bit\_length() devuelve el numero de bits necesarios para representar este numero en notación binaria

Una variable que contiene un valor de lista tiene acceso a un conjunto diferente de métodos:

items = [1, 2]

items.append(3)

items.pop()

Los métodos dependen del tipo de valor.

La función global id() proporcionada por Python permite inspeccionar la ubicación en la memoria de un objeto en particular

id(edad) # 140170065725376

Tu valor de memoria cambiará, solo lo muestro como un ejemplo.

Si asignas un valor diferente a la variable, su dirección cambiará, porque el contenido de la variable ha sido reemplazado por otro valor almacenado en otra ubicación de la memoria:

edad = 8

print(id(edad)) # 140535918671808

edad = 9

print(id(edad)) # 140535918671840

Pero si modificas el objeto usando sus métodos, la dirección permanece igual:

items = [1, 2]

print(id(items)) # 140093713593920

items.append(3)

print(items) # [1, 2, 3]

print(id(items)) # 140093713593920

La dirección solo cambia si reasignas una variable a otro valor.

Algunos objetos son mutables, mientras que otros son inmutables. Esto depende del objeto en sí.

Si el objeto proporciona métodos para cambiar su contenido, entonces es mutable. De lo contrario, es inmutable.

La mayoría de los tipos definidos por Python son inmutables. Por ejemplo, un int es inmutable. No hay métodos para cambiar su valor. Si incrementas el valor usando

edad = 8

edad = edad + 1

# o

edad += 1

y verificas con id(edad), encontrarás que la edad apunta a una ubicación de memoria diferente. El valor original no ha cambiado, simplemente cambiamos a otro valor.

**Bucles en Python**

Los bucles son una parte esencial de la programación.

En Python tenemos 2 tipos de bucles: bucles **while**y bucles **for**.

**Bucles *while*en Python**

Bucles while se definen utilizando la palabra clave while y repiten su bloque hasta que la condición se evalúa como False:

condicion = True

while condicion == True:

print("La condicion es verdadera")

Este es un **bucle infinito**. Nunca termina.

Detengamos el ciclo justo después de la primera iteración:

condicion = True

while condicion == True:

print("La condicion es verdadera")

condicion = False

print("Despues del bucle")

En este caso, se ejecuta la primera iteración, ya que la prueba de condición se evalúa como True. En la segunda iteración, la prueba de condición se evalúa como False, por lo que el control pasa a la siguiente instrucción después del ciclo.

Es común tener un contador para detener la iteración después de algunos ciclos:

contador = 0

while contador < 10:

print("La condicion es verdadera")

contador = contador + 1

print("Despues del bucle")

**Bucles *for*en Python**

Usando bucles for podemos decirle a Python que ejecute un bloque por una cantidad predeterminada de veces, por adelantado, y sin la necesidad de una variable separada y condicional para verificar su valor.

Por ejemplo, podemos iterar los elementos de una lista:

items = [1, 2, 3, 4]

for item in items:

print(item)

O puedes iterar una cantidad específica de veces usando la función range():

for item in range(04):

print(item)

range(4)crea una secuencia que comienza desde 0 y contiene 4 elementos: [0, 1, 2, 3].

Para obtener el índice, debes ajustar la secuencia en la función enumerate():

items = [1, 2, 3, 4]

for indice, item in enumerate(items):

print(indice, item)

***Break* y *continue* en Python**

Tanto los bucles while como for pueden interrumpirse dentro del bloque, utilizando dos palabras clave especiales:  break y continue.

continue detiene la iteración actual y le dice a Python que ejecute la siguiente.

break detiene el bucle por completo y continúa con la siguiente instrucción después que termina el bucle.

El primer ejemplo aquí imprime 1, 3, 4. El segundo ejemplo imprime 1:

items = [1, 2, 3, 4]

for item in items:

if item == 2:

continue

print(item)

items = [1, 2, 3, 4]

for item in items:

if item == 2:

break

print(item)

**Clases en Python**

Además de usar los tipos proporcionados por Python, podemos declarar nuestras propias clases y, a partir de las clases, podemos crear instancias de objetos.

Un objeto es una instancia de una clase. Una clase es el tipo de objeto.

Podemos definir una clase de esta manera:

class <nombre\_clase>:

# mi clase

Por ejemplo, definamos una clase Perro:

class Perro:

# la clase Perro

Una clase puede definir métodos:

class Perro:

# la clase Perro

def ladrido(self):

print('Guau!')

self ya que el argumento del método apunta a la instancia del objeto actual y debe especificarse al definir un método.

Creamos una instancia de una clase, **un objeto**, usando esta sintaxis:

roger = Perro()

Ahora roger es un nuevo objeto de tipo Perro.

Si ejecutas:

print(type(roger))

Obtendrás <class '\_\_main\_\_.Perro'>

Un tipo especial de método, \_\_init\_\_() se llama **constructor**, y podemos usarlo para inicializar una o más propiedades cuando creamos un nuevo objeto de esa clase:

class Perro:

# la clase Perro

def \_\_init\_\_(self, nombre, edad):

self.nombre = nombre

self.edad = edad

def ladrido(self):

print('Guau!')

La usamos de esta manera:

roger = Perro('Roger', 8)

print(roger.nombre) # 'Roger'

print(roger.edad) # 8

roger.ladrido() # 'Guau!'

Una característica importante de las clases es la **herencia**.

Podemos crear una clase Animal con un método caminar():

class Animal:

def caminar(self):

print('Caminando..')

y la clase Perro puede heredar de Animal:

class Perro(Animal):

def ladrido(self):

print('Guau!')

Ahora, la creación de un nuevo objeto de la clase Perro tendrá el método caminar(), ya que se hereda de Animal:

roger = Perro()

roger.caminar() # 'Caminando..'

roger.ladrido() # 'Guau!'

**Módulos en Python**

Cada archivo de Python es un módulo.

Puedes importar un módulo desde otros archivos, y esa es la base de cualquier programa de complejidad moderada, ya que promueve una organización sensata y la reutilización del código.

En el programa típico de Python, un archivo actúa como punto de entrada. Los otros archivos son módulos y exponen funciones que podemos llamar desde otros archivos.

El archivo perro.py contiene este código:

def ladrido():

print('Guau!')

Podemos importar esta función desde otro archivo usando import. Y una vez que lo hagamos, podemos hacer referencia a la función usando la notación de puntos, perro.ladrido():

import perro

perro.ladrido()

O podemos usar la sintaxis from ... import y llamar a la función directamente:

from perro import ladrido

ladrido()

La primera estrategia nos permite cargar todo lo definido en un archivo.

La segunda estrategia nos permite elegir las cosas que necesitamos.

Esos módulos son específicos de tu programa y la importación depende de la ubicación del archivo en el sistema de archivos.

Supongamos que colocas perro.py en una subcarpeta lib.

En esa carpeta, debes crear un archivo vacío llamado \_\_init\_\_.py. Esto le dice a Python que la carpeta contiene módulos.

Ahora puedes elegir -puedes importar perro desde lib:

from lib import perro

perro.ladrido()

o puedes hacer referencia a la función específica del módulo de perro que se importa desde lib.perro:

from lib.perro import ladrido

ladrido()

**Librería estándar de Python**

Python expone una gran cantidad de funciones integradas a través de su **librería estándar**.

La librería estándar es una gran colección de todo tipo de utilidades, que van desde las matemáticas hasta la depuración y la creación de interfaces gráficas de usuario.

Puedes encontrar la lista completa *en inglés* de módulos de la biblioteca estándar aquí: <https://docs.python.org/3/library/index.html>

Algunos de los módulos importantes son:

* math para utilidades matemáticas
* re para expresiones regulares
* json para trabajar con JSON
* datetime para trabajar con fechas
* sqlite3 para usar SQLite
* os para utilidades de Sistemas Operativos
* random para la generación de números aleatorios
* statistics para utilidades estadísticas
* requests para realizar solicitudes de red HTTP
* http para crear servidores HTTP
* urllib para administrar URLs

Introduzcámos cómo usar un módulo de la biblioteca estándar. Ya sabes cómo usar los módulos que creas, importando desde otros archivos en la carpeta del programa.

Bueno, eso es lo mismo con los módulos proporcionados por la biblioteca estándar:

import math

math.sqrt(4) # 2.0

ó

from math import sqrt

sqrt(4) # 2.0

Pronto exploraremos los módulos más importantes de forma individual para comprender qué podemos hacer con ellos.

**La guía de estilo PEP8 en Python**

Cuando escribes código, debes adherirte a las convenciones del lenguaje de programación que utilizas.

Si aprendes las convenciones correctas de nomenclatura y formato desde el principio, será más fácil leer el código escrito por otras personas, y las personas encontrarán tu código más fácil de leer.

Python define sus convenciones en la guía de estilo PEP8. PEP son las siglas de ***Python Enhancement Proposals*** y es el lugar donde ocurren todas las mejoras y discusiones del lenguaje Python.

Hay muchas propuestas de PEP, todas disponibles en <https://www.python.org/dev/peps/>.

PEP8 es una de las primeras y también una de las más importantes. Define el formato y también algunas reglas sobre cómo escribir Python de forma "pitónica"(*"pythonic"*).

Puedes leer su contenido completo aquí: <https://www.python.org/dev/peps/pep-0008/> pero aquí hay un resumen rápido de los puntos importantes con los que puedes comenzar:

* Aplicar indentación mediante espacios, no tabulaciones
* Indentar usando 4 espacios.
* Los archivos de Python están codificados en UTF-8
* Usa un máximo de 80 columnas para tu código
* Escribe cada declaración en su propia línea.
* Las funciones, los nombres de las variables y los nombres de los archivos están en minúsculas, con guiones bajos entre las palabras (***snake\_case***)
* Los nombres de las clases están en mayúscula, las palabras separadas también se escriben con la letra mayúscula, (**c*amelCase***)
* Los nombres de los paquetes están en minúsculas y no tienen guiones bajos entre las palabras.
* Las variables que no deben cambiar (constantes) se escriben en mayúsculas
* Los nombres de las variables deben ser significativos
* Agrega comentarios útiles, pero evita los comentarios obvios
* Agregar espacios alrededor de los operadores
* No uses espacios en blanco innecesarios
* Agregar una línea en blanco antes de una función
* Agregar una línea en blanco entre los métodos de una clase
* Dentro de las funciones/métodos, las líneas en blanco se pueden usar para separar bloques de código relacionados para ayudar a la legibilidad

**Depurando en Python**

La depuración es una de las mejores habilidades que puedes aprender, ya que te ayudará en muchas situaciones difíciles.

Cada lenguaje tiene su depurador. Python tiene pdb, disponible a través de la biblioteca estándar.

Depura agregando un punto de interrupción en tu código:

breakpoint()

Puedes agregar más puntos de interrupción si es necesario.

Cuando el intérprete de Python llegue a un punto de interrupción en tu código, se detendrá y te dirá cuál es la siguiente instrucción que ejecutará.

Luego, puedes hacer algunas cosas.

Puedes escribir el nombre de cualquier variable para inspeccionar su valor.

Puedes presionar n para pasar a la siguiente línea en la función actual. Si el código llama a funciones, el depurador no entra en ellas y las considera "cajas negras"(*black boxes*).

Puedes presionar s para pasar a la siguiente línea en la función actual. Si la siguiente línea es una función, el depurador entra en ella y luego puedes ejecutar una instrucción de esa función a la vez.

Puedes presionar c para continuar la ejecución del programa con normalidad, sin necesidad de hacerlo paso a paso

Puedes presionar q para detener la ejecución del programa.

La depuración es útil para evaluar el resultado de una instrucción, y es especialmente bueno saber cómo usarla cuando tienes iteraciones o algoritmos complejos que deseas corregir.

**Ámbito de variable en Python**

Cuando declaras una variable, esa variable es visible en partes de tu programa, dependiendo de dónde la declares.

Si la declaras fuera de cualquier función, la variable es visible para cualquier código que se ejecute después de la declaración, incluidas las funciones:

edad = 8

def prueba():

print(edad)

print(edad) # 8

prueba() # 8

Lo llamamos **variable global**.

Si defines una variable dentro de una función, esa variable es una**variable local** y solo es visible dentro de esa función. Fuera de la función, no es accesible:

def prueba():

edad = 8

print(edad)

prueba() # 8

print(edad)

# # # NameError: name 'edad' is not defined

# Error: el nombre 'edad' no está definido

**Cómo aceptar argumentos desde la línea de comandos en Python**

Python ofrece varias formas de manejar los argumentos pasados cuando invocamos el programa desde la línea de comandos.

Hasta ahora has ejecutado programas desde un REPL o usando

python <nombre\_de\_archivo>.py

Puedes pasar argumentos y opciones adicionales cuando lo hagas, como este:

python <nombre\_de\_archivo>.py <argumento**1**>

python <nombre\_de\_archivo>.py <argumento**1**> <argumento**2**>

Una forma básica de manejar esos argumentos es usar el módulo sys de la biblioteca estándar.

Puedes obtener los argumentos pasados en la lista sys.argv:

import sys

print(len(sys.argv))

print(sys.argv)

La lista sys.argv contiene como primer elemento el nombre del archivo que se ejecutó, por ejemplo ['archivo.py'].

Es una forma sencilla, pero tienes mucho trabajo por hacer. Debes validar los argumentos, asegurarte de que su tipo sea correcto y debes imprimir comentarios al usuario si no está usando el programa correctamente.

Python proporciona otro paquete en la biblioteca estándar para ayudarlo: argparse.

Primero importa argparse y llama a argparse.ArgumentParser(), pasando la descripción de tu programa:

import argparse

parser = argparse.ArgumentParser(

description='Este programa imprime los nombres de mis perros'

)

Luego procedes a agregar los argumentos que deseas aceptar, por ejemplo en este programa aceptamos una opción -c para pasar un color, así:

python program.py -c red

import argparse

parser = argparse.ArgumentParser(

description='Este programa imprime un valor de color HEX'

)

parser.add\_argument('-c', '--color', metavar='color', required=True, help='el color que se va a buscar')

args = parser.parse\_args()

print(args.color) # 'red'(rojo)

Si no se especifica el argumento, el programa genera un error:

➜ python python programa.py

usage: programa.py [-h] -c color

programa.py: error: the following arguments are required: -c

(#programa.py: error: los siguientes argumentos son requeridos: -c)

Puedes configurar una opción para tener un conjunto específico de valores, usando choices(opciones):

parser.add\_argument('-c', '--color', metavar='color', required=True, choices={'red','yellow'}, help='el color a buscar, en ingles')

➜ python python programa.py -c blue

usage: programa.py [-h] -c color

programa.py: error: argument -c/--color: invalid choice: 'blue' (choose from 'yellow', 'red')

#programa.py: error: argumentp -c/--color: opcion invalida: 'blue' (elija entre 'yellow', 'red')

Hay más opciones, pero esas son las básicas.

Y hay paquetes comunitarios que también brindan esta funcionalidad, como[**Click**](https://click.palletsprojects.com/en/7.x/)y[**Python Prompt Toolkit**](https://python-prompt-toolkit.readthedocs.io/en/master/index.html)**.**

**Funciones Lambda en Python**

Las funciones lambda (también llamadas funciones anónimas) son funciones diminutas que no tienen nombre y solo tienen una expresión como cuerpo.

En Python se definen mediante la palabra clave lambda:

lambda <argumentos> : <expresión>

El cuerpo debe ser una sola expresión -una expresión, no una declaración.

Esta diferencia es importante. Una expresión devuelve un valor, una declaración no.

El ejemplo más simple de una función lambda es una función que duplica el valor de un número:

lambda num : num \* 2

Las funciones Lambda pueden aceptar más argumentos:

lambda a, b : a \* b

Las funciones de Lambda no se pueden invocar directamente, pero puedes asignarlas a variables:

multiplicar = lambda a, b : a \* b

print(multiplicar(2, 2)) # 4

La utilidad de las funciones lambda viene cuando se combina con otras funciones de Python, por ejemplo en combinación con map(), filter() y reduce().

**Recursión en Python**

Una función en Python puede llamarse a sí misma. Eso es la recursividad. Y puede ser bastante útil en muchos escenarios.

La forma común de explicar la recursividad es mediante el cálculo factorial.

El factorial de un número es el número n multiplicado por n-1, multiplicado por n-2 ... y así sucesivamente, hasta llegar al número 1:

3! = 3 \* 2 \* 1 = 6

4! = 4 \* 3 \* 2 \* 1 = 24

5! = 5 \* 4 \* 3 \* 2 \* 1 = 120

Usando la recursividad podemos escribir una función que calcule el factorial de cualquier número:

def factorial(n):

if n == 1: return 1

return n \* factorial(n-1)

print(factorial(3)) # 6

print(factorial(4)) # 24

print(factorial(5)) # 120

Si dentro de la función factorial() llamas a factorial(n) en lugar de factorial(n-1), vas a provocar una recursión infinita. Python, de forma predeterminada detendrá las recursiones en 1000 llamadas, y cuando se alcance este límite, obtendrás un error RecursionError.

La recursividad es útil en muchos lugares y nos ayuda a simplificar nuestro código cuando no hay otra forma óptima de hacerlo, por lo que es bueno conocer esta técnica.

**Funciones anidadas en Python**

Las funciones en Python se pueden anidar dentro de otras funciones.

Una función definida dentro de una función es visible solo dentro de esa función.

Esto es conveniente para crear utilidades que sean útiles para una función, pero no útiles fuera de ella.

Podrías preguntar: ¿por qué debería "ocultar" esta función, si no hace daño?

Uno, porque siempre es mejor ocultar la funcionalidad que es local a una función y no es útil en otros lugares.

Además, porque podemos hacer uso de ***closures***(sobre esto más adelante).

Aquí hay un ejemplo:

def hablar(frase):

def decir(palabra):

print(palabra)

palabras = frase.split(' ')

for palabra in palabras:

decir(palabra)

hablar('Voy a comprar la leche')

Si deseas acceder a una variable definida en la función externa desde la función interna, primero debes declararla como nonlocal(*no local*):

def conteo():

conteo = 0

def incrementar():

nonlocal conteo

conteo = conteo + 1

print(conteo)

incrementar()

conteo()

Esto es útil especialmente con ***closures***, como veremos a continuación.

**Closures en Python**

Si devuelves una función anidada de una función, esa función anidada tiene acceso a las variables definidas en esa función, incluso si esa función ya no está activa.

Aquí hay un ejemplo de contador simple.

def contador():

conteo = 0

def incrementar():

nonlocal conteo

conteo = conteo + 1

return conteo

return incrementar

incrementar = contador()

print(incrementar()) # 1

print(incrementar()) # 2

print(incrementar()) # 3

Devolvemos la función interna incrementar(), que todavía tiene acceso al estado de la variable conteo aunque la función contador() haya finalizado.

**Decoradores en Python**

Los decoradores son una forma de cambiar, mejorar o alterar de cualquier forma el funcionamiento de una función.

Los decoradores se definen con el símbolo @ seguido del nombre del decorador, justo antes de la definición de la función.

Ejemplo:

@registro

def hola():

print('¡hola!')

Esta función hola tiene asignado el decorador de registro.

Siempre que llamemos a hola(), se llamará al decorador.

Un decorador es una función que toma una función como parámetro, envuelve la función en una función interna que realiza el trabajo que tiene que hacer y devuelve esa función interna. En otras palabras:

def registro(func):

def envoltura():

# hace algo antes

val = func()

# hace algo después

return val

return envoltura

**Docstrings en Python**

La documentación es muy importante, no solo para comunicar a otras personas cuál es el objetivo de una función / clase / método / módulo, sino que también te lo comunicas a ti mismo.

Cuando regreses a tu código dentro de 6 o 12 meses, es posible que no recuerdes todo el conocimiento que tienes en la cabeza. En ese momento, leer tu código y comprender lo que se supone que debe hacer será mucho más difícil.

Los comentarios son una forma de ayudarte a tí mismo (y a los demás):

# esto es un comentario

num = 1 # esto es otro comentario

Otra forma es utilizar **docstrings**(cadenas de documentación).

La utilidad de las **docstrings** es que siguen convenciones. Como tales, pueden procesarse automáticamente.

Así es como se define una **docstring** para una función:

def incrementar(n):

"""Incrementa un numero"""

return n + 1

Así es como se define una docstring para una clase y un método:

class Perro:

"""Una clase que representa un perro"""

def \_\_init\_\_(self, nombre, edad):

"""Inicializar un nuevo perro"""

self.nombre = nombre

self.edad = edad

def ladrido(self):

"""Permite ladrar al perro"""

print('WOF!')

Documenta un módulo colocando una cadena de documentos en la parte superior del archivo, por ejemplo, suponiendo que se trata de perro.py:

"""Modulo Perro

Este modulo hace ... bla bla bla y provee las siguientes clases:

- Perro

...

"""

class Perro:

"""Una clase que representa un perro"""

def \_\_init\_\_(self, nombre, edad):

"""Inicializar un nuevo perro"""

self.nombre = nombre

self.edad = edad

def ladrido(self):

"""Permite ladrar al perro"""

print('WOF!')

Docstrings pueden abarcar varias líneas:

def incrementar(n):

"""Incrementa

un numero

"""

return n + 1

Python los procesará, y puedes usar la función global help() para obtener la documentación de una clase/método/función/módulo.

Por ejemplo, llamar a help(incrementar) te dará esto:

#Help on function incrementar in module \_\_main\_\_:

#Ayuda en función incrementar en el modulo \_\_main\_\_:

incrementar(n)

Incrementa

un número

Hay muchos estándares diferentes para formatear docstrings, y puedes elegir adherirte a tu favorito.

Me gusta el estándar de Google(*inglés*): <https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings>

Los estándares permiten tener herramientas para extraer docstrings y generar automáticamente documentación para tu código.

**Introspección en Python**

Las funciones, variables y objetos se pueden analizar mediante la introspección.

Primero, usando la función global help() podemos obtener la documentación si se proporciona en forma de docstrings.

Luego, puede usar print() para obtener información sobre una función:

def incrementar(n):

return n + 1

print(incrementar)

# <function incrementar at 0x7f420e2973a0>

o un objeto:

class Perro():

def ladrido(self):

print('WOF!')

roger = Perro()

print(roger)

# <\_\_main\_\_.Perro object at 0x7f42099d3340>

La función type() nos da el tipo de objeto:

print(type(incrementar))

# <class 'function'>

print(type(roger))

# <class '\_\_main\_\_.Perro'>

print(type(1))

# <class 'int'>

print(type('test'))

# <class 'str'>

La función global dir() nos permite conocer todos los métodos y atributos de un objeto:

print(dir(roger))

# ['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'ladrido']

La función global id() nos muestra la ubicación en la memoria de cualquier objeto:

print(id(roger)) # 140227518093024

print(id(1)) # 140227521172384

Puede resultar útil comprobar si dos variables apuntan al mismo objeto.

El módulo de biblioteca estándar para inspección inspect nos brinda más herramientas para obtener información sobre objetos, y puedes consultarlo aquí: <https://docs.python.org/3/library/inspect.html>(*inglés*)

**Anotaciones en Python**

Python se escribe dinámicamente. No tenemos que especificar el tipo de una variable o parámetro de función, o un valor de retorno de función.

Las anotaciones nos permiten (opcionalmente) hacer eso.

Esta es una función sin anotaciones:

def incrementar(n):

return n + 1

Esta es la misma función con anotaciones:

def incrementar(n: int) -> int:

return n + 1

También puedes anotar variables:

conteo: int = 0

Python ignorará esas anotaciones. Una herramienta separada llamada mypy se puede ejecutar de forma independiente o integrada por IDE como VS Code o PyCharm, para verificar automáticamente los errores de ***tipo***de forma estática, mientras estás programando. También te ayudará a detectar errores de discrepancia de tipos incluso antes de ejecutar el código.

Una gran ayuda, especialmente cuando tu software se vuelve grande y necesitas refactorizar el código.

**Excepciones en Python**

Es importante tener una forma de manejar los errores, y Python nos brinda un manejo de excepciones para hacerlo.

Si envuelven líneas de código en un bloque try:

try:

# algunas lineas de codigo

Si ocurre un error, Python te alertará y podrás determinar qué tipo de error ocurrió usando bloques except:

try:

# algunas lineas de codigo

except <ERROR1>:

# manejo de <ERROR1>

except <ERROR2>:

# manejo de <ERROR2>

Para atrapar a todas las excepciones, puedes usar except sin ningún tipo de error:

try:

# algunas lineas de codigo

except <ERROR1>:

# manejo de <ERROR1>

except:

# atrapar todas las demas excepciones

El bloque else se ejecuta si no se encontraron excepciones:

try:

# algunas lineas de codigo

except <ERROR1>:

# manejo de <ERROR1>

except <ERROR2>:

# manejo de <ERROR2>

else:

# no surgieron excepciones, el codigo se ejecuto correctamente

Un bloque finally permite realizar alguna operación en cualquier caso, independientemente de si se produjo un error o no:

try:

# algunas lineas de codigo

except <ERROR1>:

# manejo de <ERROR1>

except <ERROR2>:

# manejo de <ERROR2>

else:

# no surgieron excepciones, el codigo se ejecuto correctamente

finally:

# hacer algo de todos modos

El error específico que va a ocurrir depende de la operación que estés realizando.

Por ejemplo, si estás leyendo un archivo, es posible que obtengas un EOFError. Si divides un número por cero obtendrás un ZeroDivisionError. Si tienes un problema de conversión de tipos, es posible que obtengas un TypeError.

Prueba este código:

resultado = 2 / 0

print(resultado)

El programa terminará con un error:

Traceback (most recent call last):

File "main.py", line 1, in <module>

resultado = 2 / 0

ZeroDivisionError: division by zero

#Rastreo (la mas reciente última llamada):

# Archivo "main.py", linea 1, en <modulo>

# resultado = 2 / 0

#ZeroDivisionError: division por cero

Mensaje en idioma original (traducción cercana señalada con #)

y las líneas de código después del error no se ejecutarán.

Agregar esa operación en un bloque try: nos permite recuperarnos con gracia y seguir adelante con el programa:

try:

resultado = 2 / 0

except ZeroDivisionError:

print('No se puede dividir por cero!')

finally:

resultado = 1

print(resultado) # 1

También puedes generar excepciones en tu propio código, utilizando la declaración raise:

raise Exception('Ha ocurrido un error!')

Esto genera una excepción general y puedes interceptarla usando:

try:

raise Exception('Ha ocurrido un error!')

except Exception as error:

print(error)

También puedes definir tu propia clase de excepción, desde Exception:

class PerroNoEncontradoExcepcion(Exception):

pass

pass aquí significa "nada" y debemos usarlo cuando definimos una clase sin métodos, o una función sin código también.

try:

raise PerroNoEncontradoExcepcion()

except PerroNoEncontradoExcepcion:

print('Perro no encontrado!')

**Declaración *with* en Python**

La declaración with es muy útil para simplificar el trabajo con el manejo de excepciones.

Por ejemplo, cuando trabajamos con archivos, cada vez que abrimos un archivo, debemos recordar cerrarlo.

with hace que este proceso sea transparente.

En lugar de escribir:

nombre = '/Users/flavio/prueba.txt'

try:

archivo = open(nombre, 'r')

contenido = archivo.read()

print(contenido)

finally:

archivo.close()

Puedes escribir:

nombre = '/Users/flavio/prueba.txt'

with open(nombre, 'r') as archivo:

contenido = archivo.read()

print(contenido)

En otras palabras, tenemos un manejo de excepciones implícito incorporado, ya que close() se llamará automáticamente para nosotros.

with no solo es útil para trabajar con archivos. El ejemplo anterior solo pretende presentar sus capacidades.

**Cómo instalar paquetes de terceros en Python usando pip**

La biblioteca estándar de Python contiene una gran cantidad de utilidades que simplifican nuestras necesidades de desarrollo de Python, pero nada puede satisfacerlo *todo.*

Es por eso que las personas y las empresas crean paquetes y los ponen a disposición como software de código abierto para toda la comunidad.

Todos esos módulos se recopilan en un solo lugar, **Python Package Index** (*índice de paquetes de Python*); disponible en [https://pypi.org](https://pypi.org/), y se pueden instalar en tu sistema usando pip.

Hay más de 270.000 paquetes disponibles gratuitamente al momento de escribir este artículo.

Deberías tener pip ya instalado si seguiste las instrucciones de instalación de Python.

Instala cualquier paquete usando el comando pip install:

pip install <paquete>

o, si tienes problemas, también puedes ejecutarlo a través de python -m:

python -m pip install <paquete>

Por ejemplo, puedes instalar el paquete [requests](https://pypi.org/project/requests/), una biblioteca HTTP popular:

pip install requests

y una vez que lo hagas, estará disponible para todos tus scripts de Python, porque los paquetes se instalan globalmente.

La ubicación exacta depende de tu sistema operativo.

En macOS, con Python 3.9, la ubicación es /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages.

Actualiza un paquete a su última versión usando:

pip install –U <paquete>

Instale una versión específica de un paquete usando:

pip install <paquete>==<version>

Desinstala un paquete usando:

pip uninstall <paquete>

Muestra los detalles de un paquete instalado, incluida la versión, el sitio web de documentación y la información del autor mediante:

pip show <paquete>

**Comprensiones de lista en Python**

Las comprensiones de lista son una forma de crear listas de una manera muy concisa.

Supongamos que tienes una lista:

numeros = [1, 2, 3, 4, 5]

Puedes crear una nueva lista usando una comprensión de lista, compuesta por los elementos de la lista de números, elevados a 2:

numeros\_elevados\_a\_2 = [n\*\*2 for n in numeros]

# [1, 4, 9, 16, 25]

Las comprensiones de lista son una sintaxis que a veces se prefiere a los bucles, ya que es más legible cuando la operación se puede escribir en una sola línea:

numeros\_elevados\_a\_2 = []

for n in numeros:

numeros\_elevados\_a\_2.append(n\*\*2)

y sobre map():

numeros\_elevados\_a\_2 = list(map(lambda n : n\*\*2, numeros))

**Polimorfismo en Python**

El polimorfismo generaliza una funcionalidad para que pueda funcionar en diferentes tipos. Es un concepto importante en la programación orientada a objetos.

Podemos definir el mismo método en diferentes clases:

class Perro:

def comer():

print('Comiendo comida de perro')

class Gato:

def comer():

print('Comiendo comida de gato')

Luego podemos generar objetos y podemos llamar al método comer() independientemente de la clase a la que pertenezca el objeto, y obtendremos resultados diferentes:

animal1 = Perro()

animal2 = Gato()

animal1.comer()

animal2.comer()

Creamos una interfaz generalizada y ahora no necesitamos saber que un animal es un Gato o un Perro.

**Sobrecarga de operador en Python**

La sobrecarga de operadores es una técnica avanzada que podemos utilizar para hacer que las clases sean comparables y que funcionen con los operadores de Python.

Tomemos una clase Perro:

class Perro:

# la clase Perro

def \_\_init\_\_(self, nombre, edad):

self.nombre = nombre

self.edad = edad

Creemos 2 objetos Perro:

roger = Perro('Roger', 8)

syd = Perro('Syd', 7)

Podemos usar la sobrecarga de operadores para agregar una forma de comparar esos 2 objetos, según la propiedad edad:

class Perro:

# la clase Perro

def \_\_init\_\_(self, nombre, edad):

self.nombre = nombre

self.edad = edad

def \_\_gt\_\_(self, otro):

return True if self.edad > otro.age else False

Ahora, si intentas ejecutar print(roger > syd) obtendrás el resultado True.

De la misma manera que definimos \_\_gt\_\_() (que significa ***mayor que***), podemos definir los siguientes métodos:

* \_\_eq\_\_() para comprobar la **igualdad**
* \_\_lt\_\_() para comprobar si un objeto debe considerarse ***menor que*** otro con el operador <
* \_\_le\_\_() para menor o igual (<=)
* \_\_ge\_\_() para mayor o igual (>=)
* \_\_ne\_\_() para **no** igual (!=)

Entonces tienes métodos para interoperar con operaciones aritméticas:

* \_\_add\_\_() responde al operador +
* \_\_sub\_\_()responde al operador -
* \_\_mul\_\_() responde al operador \*
* \_\_truediv\_\_() responde al operador /
* \_\_floordiv\_\_() responde al operador //
* \_\_mod\_\_() responde al operador %
* \_\_pow\_\_() responde al operador \*\*
* \_\_rshift\_\_() responde al operador >>
* \_\_lshift\_\_() responde al operador <<
* \_\_and\_\_() responde al operador &
* \_\_or\_\_() responde al operador |
* \_\_xor\_\_() responde al operador ^

Hay algunos métodos más para trabajar con otros operadores, pero entiendes la idea.

**Entornos virtuales en Python**

Es común tener varias aplicaciones de Python ejecutándose en tu sistema.

Cuando las aplicaciones requieren el mismo módulo, en algún momento llegarás a una situación complicada en la que una aplicación necesita una versión de un módulo y otra aplicación una versión diferente de ese mismo módulo.

Para solucionar esto, utiliza***entornos virtuales***.

Usaremos venv. Otras herramientas funcionan de manera similar, como pipenv.

Crea un entorno virtual usando

python -m venv .venv

en la carpeta donde deseas iniciar el proyecto, o donde ya tienes un proyecto existente.

Entonces ejecuta

source .venv/bin/activate

Usa source .venv/bin/activate.fish en Fish shell

La ejecución del programa activará el entorno virtual de Python. Dependiendo de tu configuración, es posible que también veas un cambio en el indicador de tu terminal.

El mío cambió de

➜ carpeta

a

(.venv) ➜ carpeta

Ahora, la ejecución de pip utilizará este entorno virtual en lugar del entorno global.