

# Programa del Curso

## **Módulo 1 - Introducción a Python y a la programación de computadoras**

- Python: una herramienta, no un reptil.
- Hay mas de un Python.
- Comencemos nuestra aventura en Python.

## **Módulo 2 - Tipos de datos, variables, operaciones básicas de entrada y salida, operadores básicos**

- Tu primer programa.
- Literales de Python.
- Operadores: herramientas de manipulación de datos.
- Variables - cuadros en forma de datos.
- ¿Cómo hablar con la computadora?

## **Módulo 3: valores booleanos, ejecución condicional, bucles, procesamiento de listas y listas, operaciones lógicas y bit a bit**

- Tomando decisiones en Python.
- Bucles en Python.
- Operaciones lógicas y de bits en Python.
- Listas - colecciones de datos.
- Ordenar listas simples: el algoritmo de clasificación de burbuja.
- Listas: algunos detalles más.
- Listas en aplicaciones avanzadas.

## **Módulo 4 - Funciones, tuplas, diccionarios y procesamiento de datos**

- Escribir funciones en Python.
- ¿Cómo se comunican las funciones con su entorno?
- Devolver un resultado de una función.
- Scopes en Python.
- Funciones.
- Tuplas y diccionarios.

## **Módulo 5 - Módulos, paquetes, cadenas y métodos de lista y excepciones**

- El uso de módulos.
- Algunos módulos útiles.
- ¿Qué es un paquete?
- Errores: el pan de cada día del programador.
- La anatomía de la excepción.

- Algunas de las excepciones más útiles.
- Caracteres y cadenas.
- La naturaleza de las cadenas en Python.
- Métodos de cadenas.
- Cadenas en acción.
- Cuatro programas simples.

## **Módulo 6 - El enfoque orientado a objetos: clases, métodos, objetos y las características estándar; Manejo de excepciones y manejo de archivos**

- Conceptos básicos de programación de orientada a objetos.
- Un corto viaje desde el enfoque procedimental al enfoque orientado a objetos.
- Propiedades.
- Métodos.
- Herencia: uno de los fundamentos de la programación de objetos.
- Excepciones una vez más.
- Generadores y cierres.
- Procesando archivos de texto.
- Trabajando con archivos reales.

## **Módulo 1**

Introducción a Python y a la programación.

### **En este módulo, aprenderás sobre:**

- Fundamentos de programación.
- Establecimiento de tu entorno de programación.
- Compilación vs. interpretación.
- Introducción a Python.

## ¿Cómo funciona un programa de computadora?

Este curso tiene como objetivo explicar el lenguaje Python y para que se utiliza. Vamos a comenzar desde los fundamentos básicos.

Un programa hace que una computadora sea utilizable. Sin un programa, una computadora, incluso la más poderosa, no es más que un objeto. Del mismo modo, sin un pianista, un piano no es más que una caja de madera.

Las computadoras pueden realizar tareas muy complejas, pero esta habilidad no es innata. La naturaleza de una computadora es bastante diferente.

Una computadora puede ejecutar solo operaciones extremadamente simples, por ejemplo, una computadora no puede evaluar el valor de una función matemática complicada por sí misma, aunque esto no está más allá de los límites posibles en un futuro próximo.

Las computadoras contemporáneas solo pueden evaluar los resultados de operaciones muy fundamentales, como sumar o dividir, pero pueden hacerlo muy rápido y pueden repetir estas acciones prácticamente cualquier cantidad de veces.

Imagina que quieres saber la velocidad promedio que has alcanzado durante un largo viaje. Sabes la distancia, sabes el tiempo, necesitas la velocidad.

Naturalmente, la computadora podrá calcular esto, pero la computadora no es consciente de cosas como la distancia, la velocidad o el tiempo. Por lo tanto, es necesario instruir a la computadora para que:

- Acepte un número que represente la distancia.
- Acepte un número que represente el tiempo de viaje.
- Divida el valor anterior por el segundo y almacene el resultado en la memoria.
- Muestre el resultado (representando la velocidad promedio) en un formato legible.

Estas cuatro acciones simples forman un **programa**. Por supuesto, estos ejemplos no están formalizados, y están muy lejos de lo que la computadora puede entender, pero son lo suficientemente buenos como para traducirlos a un idioma que la computadora pueda aceptar.

La palabra clave es el **lenguaje**.

## Lenguajes naturales vs. Lenguajes de programación

Un lenguaje es un medio (y una herramienta) para expresar y registrar pensamientos. Hay muchos lenguajes a nuestro alrededor. Algunos de ellos no requieren hablar ni escribir, como el lenguaje corporal. Es posible expresar tus sentimientos más profundos de manera muy precisa sin decir una palabra.

Otro lenguaje que empleas cada día es tu lengua materna, que utilizas para manifestar tu voluntad y para pensar en la realidad. Las computadoras también tienen su propio lenguaje, llamado lenguaje **máquina**, el cual es muy rudimentario.

Una computadora, incluso la más técnicamente sofisticada, carece incluso de un rastro de inteligencia. Se podría decir que es como un perro bien entrenado, responde solo a un conjunto predeterminado de comandos conocidos. Los comandos que reconoce son muy simples. Podemos imaginar que la computadora responde a órdenes como "Toma ese número, divídelo por otro y guarda el resultado".

Un conjunto completo de comandos conocidos se llama **lista de instrucciones**, a veces abreviada **IL** (por sus siglas en inglés de Instruction List). Los diferentes tipos de computadoras pueden variar según el tamaño de sus IL y las instrucciones pueden ser completamente diferentes en diferentes modelos.

Nota: los lenguajes máquina son desarrollados por humanos.

Ninguna computadora es actualmente capaz de crear un nuevo idioma. Sin embargo, eso puede cambiar pronto.

Por otro lado, las personas también usan varios idiomas muy diferentes, pero estos idiomas se crearon ellos mismos. Además, todavía están evolucionando.

Cada día se crean nuevas palabras y desaparecen las viejas. Estos lenguajes se llaman **lenguajes naturales**.

## ¿Qué hace a un lenguaje?

Podemos decir que cada idioma (máquina o natural, no importa) consta de los siguientes elementos:

Un conjunto de símbolos utilizados para formar palabras de un determinado idioma (por ejemplo, el alfabeto latino para el inglés, el alfabeto cirílico para el ruso, el kanji para el japonés, etc.).

(También conocido como diccionario) un conjunto de palabras que el idioma ofrece a sus usuarios (por ejemplo, la palabra "computadora" proviene del diccionario en inglés, mientras que "abcde" no; la palabra "chat" está presente en los diccionarios de inglés y francés, pero sus significados son diferentes.

Un conjunto de reglas (formales o informales, escritas o interpretadas intuitivamente) utilizadas para precisar si una determinada cadena de palabras forma una oración válida (por ejemplo, "Soy una serpiente" es una frase sintácticamente correcta, mientras que "Yo serpiente soy una" no lo es).

Un conjunto de reglas que determinan si una frase tiene sentido (por ejemplo, "Me comí una dona" tiene sentido, pero "Una dona me comió" no lo tiene).

La IL es, de hecho, **el alfabeto de un lenguaje máquina**. Este es el conjunto de símbolos más simple y principal que podemos usar para dar comandos a una computadora. Es la lengua materna de la computadora.

Desafortunadamente, esta lengua está muy lejos de ser una lengua materna humana. Todos (tanto las computadoras como los humanos) necesitamos algo más, un lenguaje común para las computadoras y los seres humanos, o un puente entre los dos mundos diferentes.

Necesitamos un lenguaje en el que los humanos puedan escribir sus programas y un lenguaje que las computadoras puedan usar para ejecutar los programas, que es mucho más complejo que el lenguaje máquina y más sencillo que el lenguaje natural.

Tales lenguajes son a menudo llamados lenguajes de programación de alto nivel. Son algo similares a los naturales en que usan símbolos, palabras y convenciones legibles para los humanos. Estos lenguajes permiten a los humanos expresar comandos a computadoras que son mucho más complejas que las ofrecidas por las IL.

Un programa escrito en un lenguaje de programación de alto nivel se llama **código fuente** (en contraste con el código de máquina ejecutado por las computadoras). Del mismo modo, el archivo que contiene el código fuente se llama **archivo fuente**.

## Compilación vs. Interpretación

La programación de computadora es el acto de establecer una secuencia de instrucciones con la cual se causará el efecto deseado. El efecto podría ser diferente en cada caso específico: depende de la imaginación, el conocimiento y la experiencia del programador.

Por supuesto, tal composición tiene que ser correcta en muchos sentidos, tales como:

- **Alfabéticamente:** Un programa debe escribirse en una secuencia de comandos reconocible, por ejemplo, el Romano, Cirílico, etc.
- **Léxicamente:** Cada lenguaje de programación tiene su diccionario y necesitas dominarlo; afortunadamente, es mucho más simple y más pequeño que el diccionario de cualquier lenguaje natural.
- **Sintácticamente:** Cada idioma tiene sus reglas y deben ser obedecidas.
- **Semánticamente:** El programa tiene que tener sentido.

Desafortunadamente, un programador también puede cometer errores en cada uno de los cuatro sentidos anteriores. Cada uno de ellos puede hacer que el programa se vuelva completamente inútil.

Supongamos que ha escrito correctamente un programa. ¿Cómo persuadimos a la computadora para que la ejecute? Tienes que convertir tu programa en lenguaje máquina. Afortunadamente, la traducción puede ser realizada por una computadora, haciendo que todo el proceso sea rápido y eficiente.

Hay dos formas diferentes de **transformar un programa de un lenguaje de programación de alto nivel a un lenguaje de máquina**:

- El programa fuente se traduce una vez (sin embargo, esta ley debe repetirse cada vez que se modifique el código fuente) obteniendo un archivo (por ejemplo, un archivo .exe si el código está diseñado para ejecutarse en MS Windows) que contiene el código de la máquina; ahora puedes distribuir el archivo en todo el mundo; el programa que realiza esta traducción se llama compilador o traductor.

- Tú (o cualquier usuario del código) puedes traducir el programa fuente cada vez que se ejecute; el programa que realiza este tipo de transformación se denomina intérprete, ya que interpreta el código cada vez que está destinado a ejecutarse; también significa que no puede distribuir el código fuente tal como está, porque el usuario final también necesita que el intérprete lo ejecute.

Debido a algunas razones muy fundamentales, un lenguaje de programación de alto nivel particular está diseñado para caer en una de estas dos categorías.

Hay muy pocos idiomas que se pueden compilar e interpretar. Por lo general, un lenguaje de programación se proyecta con este factor en la mente de sus constructores: ¿Se compilará o interpretará?

## ¿Qué hace realmente el intérprete?

Supongamos una vez más que has escrito un programa. Ahora, existe como un **archivo de computadora**: un programa de computadora es en realidad una pieza de texto, por lo que el código fuente generalmente se coloca en **archivos de texto**. Nota: debe ser **texto puro**, sin ninguna decoración, como diferentes fuentes, colores, imágenes incrustadas u otros medios. Ahora tienes que invocar al intérprete y dejar que lea el archivo fuente.

El intérprete lee el código fuente de una manera que es común en la cultura occidental: de arriba hacia abajo y de izquierda a derecha. Hay algunas excepciones: se cubrirán más adelante en el curso.

En primer lugar, el intérprete verifica si todas las líneas subsiguientes son correctas (utilizando los cuatro aspectos tratados anteriormente).

Si el compilador encuentra un error, termina su trabajo inmediatamente. El único resultado en este caso es un **mensaje de error**. El intérprete le informará dónde se encuentra el error y qué lo causó. Sin embargo, estos mensajes pueden ser engañosos, ya que el intérprete no puede seguir tus intenciones exactas y puede detectar errores a cierta distancia de tus causas reales.

Por ejemplo, si intentas usar una entidad de un nombre desconocido, causará un error, pero el error se descubrirá en el lugar donde se intenta usar la entidad, no donde se introdujo el nombre de la nueva entidad.

En otras palabras, la razón real generalmente se ubica un poco antes en el código, por ejemplo, en el lugar donde se tuvo que informar al intérprete de que usaría la entidad del nombre.

Si la línea se ve bien, el intérprete intenta ejecutarla (nota: cada línea generalmente se ejecuta por separado, por lo que el trío "Lectura - Verificación - Ejecución", pueden repetirse muchas veces, más veces que el número real de líneas en el archivo fuente, como algunas partes del código pueden ejecutarse más de una vez).

También es posible que una parte significativa del código se ejecute con éxito antes de que el intérprete encuentre un error. Este es el comportamiento normal en este modelo de ejecución.

Puedes preguntar ahora: ¿Cuál es mejor? ¿El modelo de "compilación" o el modelo de "interpretación"? No hay una respuesta obvia. Si hubiera habido, uno de estos modelos habría dejado de existir hace mucho tiempo. Ambos tienen sus ventajas y sus desventajas.

## Compilación vs. Interpretación - Ventajas y Desventajas

	COMPILACIÓN	INTERPRETACIÓN
VENTAJAS	<ul style="list-style-type: none"><li>• La ejecución del código traducido suele ser más rápida.</li><li>• Solo el usuario debe tener el compilador; el usuario final puede usar el código sin él.</li><li>• El código traducido se almacena en lenguaje máquina, ya que es muy difícil de entender, es probable que tus propios inventos y trucos de programación sigan siendo secreto.</li></ul>	<ul style="list-style-type: none"><li>• Puede ejecutar el código en cuanto lo complete; no hay fases adicionales de traducción.</li><li>• El código se almacena utilizando el lenguaje de programación, no el de la máquina; esto significa que puede ejecutarse en computadoras que utilizan diferentes lenguajes máquina; no compila el código por separado para cada arquitectura diferente.</li></ul>
DESVENTAJAS	<ul style="list-style-type: none"><li>• La compilación en sí misma puede llevar mucho tiempo; es posible que no puedas ejecutar tu código inmediatamente después de cualquier modificación.</li><li>• Tienes que tener tantos compiladores como plataformas de hardware en los que deseas que se ejecute su código.</li></ul>	<ul style="list-style-type: none"><li>• No esperes que la interpretación incremente tu código a alta velocidad: tu código compartirá la potencia de la computadora con el intérprete, por lo que no puede ser realmente rápido.</li><li>• Tanto tú como el usuario final deben tener el intérprete para ejecutar su código.</li></ul>

## ¿Qué significa todo esto para ti?

- Python es un **lenguaje interpretado**. Esto significa que hereda todas las ventajas y desventajas descritas. Por supuesto, agrega algunas de sus características únicas a ambos conjuntos.
- Si deseas programar en Python, necesitarás el **intérprete de Python**. No podrás ejecutar tu código sin él. Afortunadamente, **Python es gratis**. Esta es una de sus ventajas más importantes.

Debido a razones históricas, los lenguajes diseñados para ser utilizados en la manera de interpretación a menudo se llaman **lenguajes de programación**, mientras que los programas fuente codificados que los usan se llaman **scripts**.

## ¿Qué es Python?

Python es un lenguaje de programación de alto nivel, interpretado, orientado a objetos y de uso generalizado con semántica dinámica, que se utiliza para la programación de propósito general.

Y aunque puede que conozcas a la pitón como una gran serpiente, el nombre del lenguaje de programación Python proviene de una vieja serie de comedia de la BBC llamada **Monty Python's Flying Circus**.

En el apogeo de su éxito, el equipo de Monty Python estaba realizando sus escenas para audiencias en vivo en todo el mundo, incluso en el Hollywood Bowl.

Dado que Monty Python es considerado uno de los dos nutrientes fundamentales para un programador (el otro es la pizza), el creador de Python nombró el lenguaje en honor del programa de televisión.

## ¿Quién creó Python?

Una de las características sorprendentes de Python es el hecho de que en realidad es el trabajo de una persona. Por lo general, los grandes lenguajes de programación son desarrollados y publicados por grandes compañías que emplean a muchos profesionales, y debido a las normas de derechos de autor, es muy difícil nombrar a cualquiera de las personas involucradas en el proyecto. Python es una excepción.

No hay muchos idiomas cuyos autores son conocidos por su nombre. Python fue creado por **Guido van Rossum**, nacido en 1956 en Haarlem, Países Bajos. Por supuesto, Guido van Rossum no desarrolló y evolucionó todos los componentes de Python.

La velocidad con la que Python se ha extendido por todo el mundo es el resultado del trabajo continuo de miles de (muy a menudo anónimos) programadores, evaluadores, usuarios (muchos de ellos no son especialistas en TI) y entusiastas, pero hay que decir que la primera idea (la semilla de la que brotó Python) llegó a una cabeza: la de Guido.

## Un proyecto de programación por hobby

Las circunstancias en las que se creó Python son un poco desconcertantes. Según Guido van Rossum:



En diciembre de 1989, estaba buscando un proyecto de programación de "pasatiempo" que me mantendría ocupado durante la semana de Navidad. Mi oficina (...) estaría cerrada, pero tenía una computadora en casa y no mucho más en mis manos. Decidí escribir un intérprete para el nuevo lenguaje de scripting en el que había estado pensando últimamente: un descendiente de ABC que atraería a los hackers de Unix / C. Elegí Python como un título de trabajo para el proyecto, estando en un estado de ánimo ligeramente irreverente (y un gran fanático de Monty Python's Flying Circus). *Guido van Rossum*

## Los objetivos de Python

En 1999, Guido van Rossum definió sus objetivos para Python:

- Un lenguaje **fácil e intuitivo** tan poderoso como los de los principales competidores.
- De **código abierto**, para que cualquiera pueda contribuir a su desarrollo.
- El código que es tan **comprensible** como el inglés simple.
- **Adecuado para tareas cotidianas**, permitiendo tiempos de desarrollo cortos.

Unos 20 años después, está claro que todas estas intenciones se han cumplido. Algunas fuentes dicen que Python es el lenguaje de programación más popular del mundo, mientras que otros afirman que es el tercero o el quinto.

De cualquier manera, todavía ocupa un alto rango en el top ten de la [PYPL Popularity of Programming Language](#) y la [TIOBE Programming Community Index](#).

Python no es una lengua joven. **Es maduro y digno de confianza**. No es una maravilla de un solo golpe. Es una estrella brillante en el firmamento de programación, y el tiempo dedicado a aprender Python es una muy buena inversión.

## ¿Qué hace especial a Python?

¿Por qué los programadores, jóvenes y viejos, experimentados y novatos, quieren usarlo? ¿Cómo fue que las grandes empresas adoptaron Python e implementaron sus productos estrella al usarlo?

Hay muchas razones. Ya hemos enumerado algunas de ellas, pero vamos a enumerarlas de una manera más práctica:

- Es **fácil de aprender** - El tiempo necesario para aprender Python es más corto que en muchos otros lenguajes; esto significa que es posible comenzar la programación real más rápido.
- Es **fácil de enseñar** - La carga de trabajo de enseñanza es menor que la que necesitan otros lenguajes; esto significa que el profesor puede poner más énfasis en las técnicas de programación generales (independientes del lenguaje), no gastando energía en trucos exóticos, extrañas excepciones y reglas incomprensibles.
- Es **fácil de utilizar** - Para escribir software nuevo; a menudo es posible escribir código más rápido cuando se usa Python.
- Es **fácil de entender** - A menudo, también es más fácil entender el código de otra persona más rápido si está escrito en Python.
- Es **fácil de obtener, instalar y desplegar** - Python es gratuito, abierto y multiplataforma; No todos los lenguajes pueden presumir de eso.

Por supuesto, Python también tiene sus inconvenientes:

- No es un demonio de la velocidad; Python no ofrece un rendimiento excepcional.
- En algunos casos puede ser resistente a algunas técnicas de prueba más simples, lo que puede significar que la depuración del código de Python puede ser más difícil que con otros lenguajes. Afortunadamente, cometer errores siempre es más difícil en Python.
-



También debe señalarse que Python no es la única solución de este tipo disponible en el mercado de TI.

Tiene muchos seguidores, pero hay muchos que prefieren otros lenguajes y ni siquiera consideran Python para sus proyectos.

## Rivales de Python

Python tiene dos competidores directos, con propiedades y predisposiciones comparables. Estos son:

- **Perl** - un lenguaje de scripting originalmente escrito por Larry Wall.
- **Ruby** - un lenguaje de scripting originalmente escrito por Yukihiro Matsumoto.

El primero es más tradicional, más conservador que Python, y se parece a algunos de los buenos lenguajes antiguos derivados del lenguaje de programación C clásico.

En contraste, este último es más innovador y está más lleno de ideas nuevas. Python se encuentra en algún lugar entre estas dos creaciones.

Internet está lleno de foros con discusiones infinitas sobre la superioridad de uno de estos tres sobre los otros, si deseas obtener más información sobre cada uno de ellos.

## ¿Dónde podemos ver a Python en acción?

Lo vemos todos los días y en casi todas partes. Se utiliza ampliamente para implementar complejos **servicios de Internet** como motores de búsqueda, almacenamiento en la nube y herramientas, redes sociales, etc. Cuando utilizas cualquiera de estos servicios, en realidad estás muy cerca de Python.

Muchas **herramientas de desarrollo** se implementan en Python. Cada vez se escriben mas **aplicaciones de uso diario** en Python. Muchos **científicos** han abandonado las costosas herramientas patentadas y se han cambiado a Python. Muchos **evaluadores** de proyectos de TI han comenzado a usar Python para llevar a cabo procedimientos de prueba repetibles. La lista es larga.



## ¿Por qué no Python?

A pesar de la creciente popularidad de Python, todavía hay algunos nichos en los que Python está ausente o rara vez se ve:

- **Programación de bajo nivel** (a veces llamada programación "cercana al metal"): si deseas implementar un controlador o motor gráfico extremadamente efectivo, no se usaría Python
- **Aplicaciones para dispositivos móviles**: este territorio aún está a la espera de ser conquistado por Python, lo más probable es que suceda algún día.

## Hay más de un Python

Hay dos tipos principales de Python, llamados Python 2 y Python 3.

Python 2 es una versión anterior del Python original. Su desarrollo se ha estancado intencionalmente, aunque eso no significa que no haya actualizaciones. Por el contrario, las actualizaciones se emiten de forma regular, pero no pretenden modificar el idioma de manera significativa. Prefieren arreglar cualquier error recién descubierto y agujeros de seguridad. La ruta de desarrollo de Python 2 ya ha llegado a un callejón sin salida, pero Python 2 en sí todavía está muy vivo.

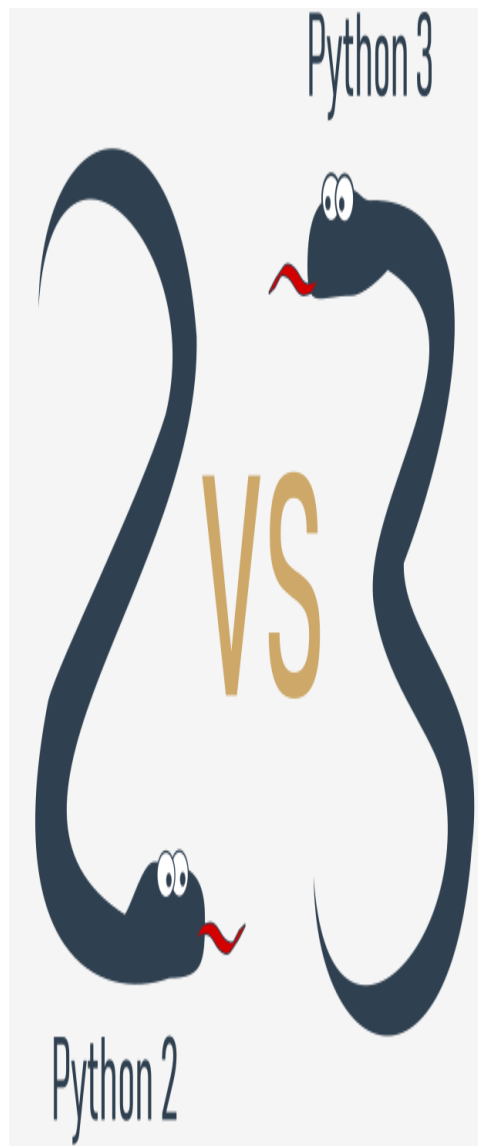
**Python 3 es la versión más nueva (para ser precisos, la actual) del lenguaje. Está atravesando su propio camino de evolución, creando sus propios estándares y hábitos.**

El primero es más tradicional, más conservador que Python, y se parece a algunos de los buenos lenguajes antiguos derivados del lenguaje de programación C clásico.

Estas dos versiones de Python no son compatibles entre sí. Las secuencias de comandos de Python 2 no se ejecutarán en un entorno de Python 3 y viceversa, por lo que si deseas que un intérprete de Python 3 ejecute el código Python 2 anterior, la única solución posible es volver a escribirlo, no desde cero, por supuesto. Como grandes partes del código pueden permanecer intactas, pero tienes que revisar todo el código para encontrar todas las incompatibilidades posibles. Desafortunadamente, este proceso no puede ser completamente automatizado.

Es demasiado difícil, consume mucho tiempo, es demasiado caro y es demasiado arriesgado migrar una aplicación Python 2 antigua a una nueva plataforma. Es posible que reescribir el código le introduzca nuevos errores. Es más fácil y mas sensato dejar estos sistemas solos y mejorar el intérprete existente, en lugar de intentar trabajar dentro del código fuente que ya funciona.

Python 3 no es solo una versión mejorada de Python 2, es un lenguaje completamente diferente, aunque es muy similar a su predecesor. Cuando se miran a distancia, parecen ser los mismos, pero cuando se observan de cerca, se notan muchas diferencias.



Si estás modificando una solución Python existente, entonces es muy probable que esté codificada en Python 2. Esta es la razón por la que Python 2 todavía está en uso. Hay demasiadas aplicaciones de Python 2 existentes para descartarlo por completo.

Si se va a comenzar un nuevo proyecto de Python, **deberías usar Python 3, esta es la versión de Python que se usará durante este curso.**

Es importante recordar que puede haber diferencias mayores o menores entre las siguientes versiones de Python 3 (p. Ej., Python 3.6 introdujo claves de diccionario ordenadas de forma predeterminada en la implementación de CPython). La buena noticia es que todas las versiones más nuevas de Python 3 son **compatibles** con las versiones anteriores de Python 3. Siempre que sea significativo e importante, siempre intentaremos resaltar esas diferencias en el curso.

Todos los ejemplos de código que encontrarás durante el curso se han probado con Python 3.4, Python 3.6 y Python 3.7.

## Python alias CPython

Además de Python 2 y Python 3, hay más de una versión de cada uno.

En primer lugar, están los Pythons que mantienen las personas reunidas en torno a PSF ([Python Software Foundation](#)), una comunidad que tiene como objetivo desarrollar, mejorar, expandir y popularizar Python y su entorno. El presidente del PSF es el propio Guido van Rossum, y por esta razón, estos Pythons se llaman **canónicos**. También se consideran **Pythons de referencia**, ya que cualquier otra implementación del lenguaje debe seguir todos los estándares establecidos por el PSF.



Guido van Rossum utilizó el lenguaje de programación "C" para implementar la primera versión de su lenguaje y esta decisión aún está vigente. Todos los Pythons que vienen del PSF están escritos en el lenguaje "C". Hay muchas razones para este enfoque y tiene muchas consecuencias. Una de ellos (probablemente la más importante) es que gracias a él, Python puede ser portado y migrado fácilmente a todas las plataformas con la capacidad de compilar y ejecutar programas en lenguaje "C" (virtualmente todas las plataformas tienen esta característica, lo que abre muchas expansiones y oportunidades para Python).

Esta es la razón por la que la implementación de PSF a menudo se denomina **CPython**. Este es el Python más influyente entre todos los Pythons del mundo.

## Cython

Otro miembro de la familia Python es **Cython**.

Cython es una de las posibles soluciones al rasgo de Python más doloroso: la falta de eficiencia. Los cálculos matemáticos grandes y complejos pueden ser fácilmente codificados en Python (mucho más fácil que en "C" o en cualquier otro lenguaje tradicional), pero la ejecución del código resultante puede requerir mucho tiempo.

¿Cómo se reconcilian estas dos contradicciones? Una solución es escribir tus ideas matemáticas usando Python, y cuando estés absolutamente seguro de que tu código es correcto y produce resultados válidos, puedes traducirlo a "C". Ciertamente, "C" se ejecutará mucho más rápido que Python puro.

Esto es lo que pretende hacer Cython: traducir automáticamente el código de Python (limpio y claro, pero no demasiado rápido) al código "C" (complicado y hablador, pero ágil).

## Jython

Otra versión de Python se llama **Jython**.

"J" es para "Java". Imagina un Python escrito en Java en lugar de C. Esto es útil, por ejemplo, si desarrollas sistemas grandes y complejos escritos completamente en Java y deseas agregarles cierta flexibilidad de Python. El tradicional CPython puede ser difícil de integrar en un entorno de este tipo, ya que C y Java viven en mundos completamente diferentes y no comparten muchas ideas comunes.

Jython puede comunicarse con la infraestructura Java existente de manera más efectiva. Es por esto que algunos proyectos lo encuentran útil y necesario.

Nota: la implementación actual de Jython sigue los estándares de Python 2. Hasta ahora, no hay Jython conforme a Python 3.



## PyPy y RPython

Echa un vistazo al logo de abajo. Es un rebus. ¿Puedes resolverlo?





Es un logotipo de **PyPy** - un Python dentro de un Python. En otras palabras, representa un entorno de Python escrito en un lenguaje similar a Python llamado **RPython** (Restricted Python). En realidad es un subconjunto de Python. El código fuente de PyPy no se ejecuta de manera interpretativa, sino que se traduce al lenguaje de programación C y luego se ejecuta por separado.

Esto es útil porque si deseas probar cualquier característica nueva que pueda ser o no introducida en la implementación de Python, es más fácil verificarla con PyPy que con CPython. Esta es la razón por la que PyPy es más una herramienta para las personas que desarrollan Python que para el resto de los usuarios.

Esto no hace que PyPy sea menos importante o menos serio que CPython.

Además, PyPy es compatible con el lenguaje Python 3.

Hay muchos más Pythons diferentes en el mundo. Los encontrarás si los buscas, pero **este curso se centrará en CPython**.

## ¿Cómo obtener Python y cómo usarlo?

Hay varias formas de obtener tu propia copia de Python 3, dependiendo del sistema operativo que utilices.

**Es probable que los usuarios de Linux tengan Python ya instalado** - este es el escenario más probable, ya que la infraestructura de Python se usa de forma intensiva en muchos componentes del sistema operativo Linux.

Por ejemplo, algunas distribuciones pueden unir sus herramientas específicas con el sistema y muchas de estas herramientas, como los administradores de paquetes, a menudo están escritas en Python. Algunas partes de los entornos gráficos disponibles en el mundo de Linux también pueden usar Python.

Si eres un usuario de Linux, abre la terminal/consola y escribe:

```
python3
```

En el indicador de shell, presiona Enter y espera.

Si ves algo como esto:

```
Python 3.4.5 (default, Jan 12 2017, 02:28:40) [GCC 4.2.1 Compatible Clang 3.7.1 (tags/RELEASE_371/final)] on linux
Type "help", "copyright", "credits" or "license" for more information. >>>
```

Entonces no tienes que hacer nada más.

Si Python 3 está ausente, consulta la documentación de Linux para saber cómo usar tu administrador de paquetes para descargar e instalar un paquete nuevo: el que necesitas se llama `python3` o su nombre comienza con eso.

Todos los usuarios que no sean Linux pueden descargar una copia en <https://www.python.org/downloads/>.

## Descargando e instalando Python

Debido a que el navegador le dice al sitio web que se ingresó, el sistema operativo que se utiliza, el único paso que se debe seguir es hacer clic en la versión de Python que se desee.

En este caso, selecciona Python 3. El sitio siempre te ofrece la última versión.

Si eres un **usuario de Windows**, utiliza el archivo `.exe` descargado y sigue todos los pasos.

Deja las configuraciones predeterminadas que el instalador sugiere por ahora, con una excepción: mira la casilla de verificación denominada **Agregar Python 3.x a PATH** y selecciónala.

Esto hará las cosas más fáciles.

Si eres un usuario de **macOS**, es posible que ya se haya preinstalado una versión de Python 2 en tu computadora, pero como estaremos trabajando con Python 3, aún deberás descargar e instalar el archivo .pkg correspondiente desde el sitio de Python.

## Comenzando tu trabajo con Python

Ahora que tienes Python 3 instalado, es hora de verificar si funciona y de hacer el primer uso.

Este será un procedimiento muy simple, pero debería ser suficiente para convencerte de que el entorno de Python es completo y funcional.

Hay muchas formas de utilizar Python, especialmente si vas a ser un desarrollador de Python.

Para comenzar tu trabajo, necesitas las siguientes herramientas:

- Un **editor** que te ayudará a escribir el código (debes tener algunas características especiales, no disponibles en herramientas simples); este editor dedicado te dará más que el equipo estándar del sistema operativo.
- Una **consola** en la que puedes iniciar tu código recién escrito y detenerlo por la fuerza cuando se sale de control.
- Una herramienta llamada **depurador**, capaz de ejecutar tu código paso a paso y te permite inspeccionarlo en cada momento de su ejecución.

Además de sus muchos componentes útiles, la instalación estándar de Python 3 contiene una aplicación muy simple pero extremadamente útil llamada IDLE.

**IDLE** es un acrónimo de: Integrated Development and Learning Environment (Desarrollo Integrado y Entorno de Aprendizaje).

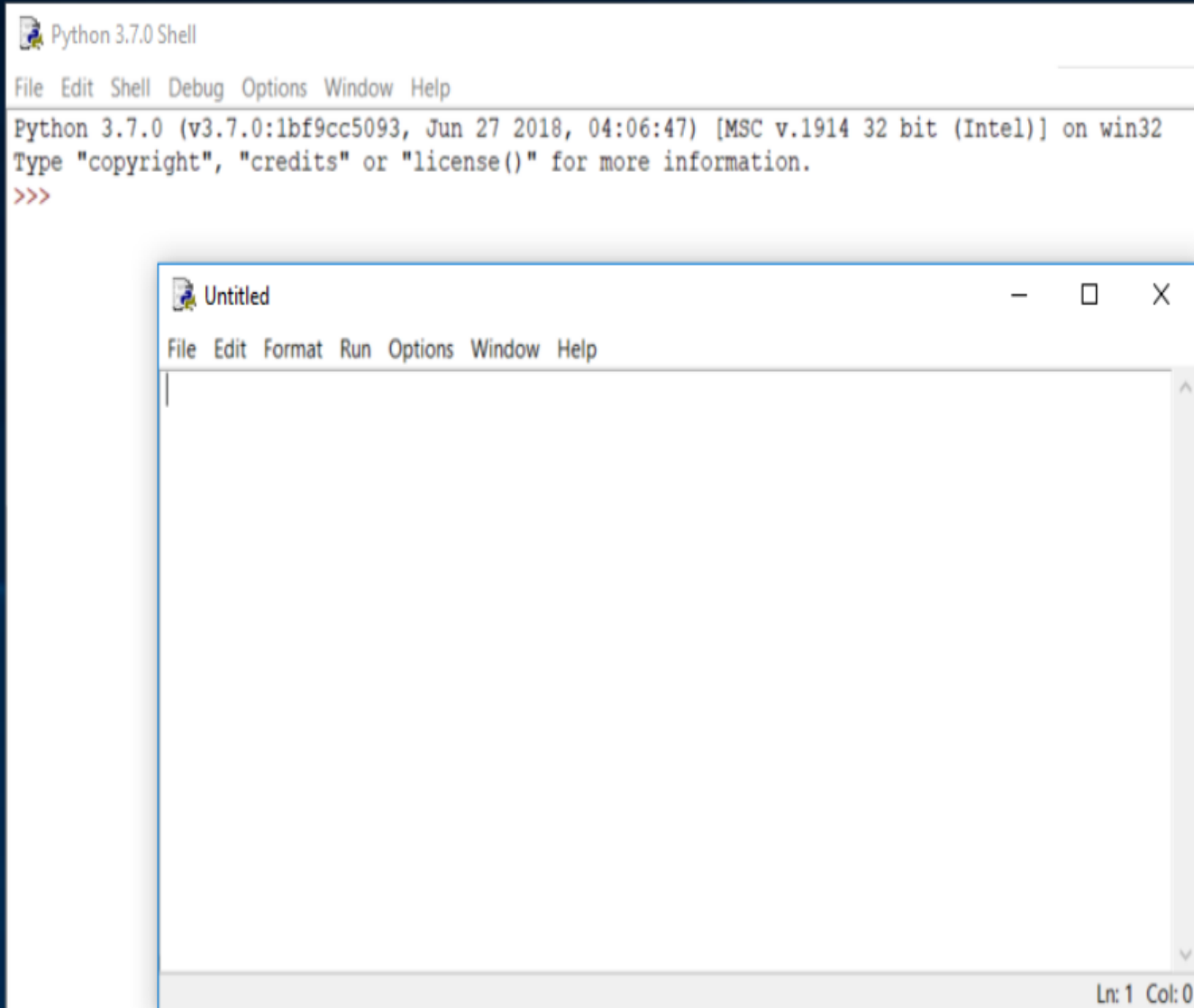
Navega por los menús de tu sistema operativo, encuentra IDLE en algún lugar debajo de Python 3.x y ejecútalo.

Esto es lo que deberías ver:

## ¿Cómo escribir y ejecutar tu primer programa?

Ahora es el momento de escribir y ejecutar tu primer programa en Python 3. Por ahora, será muy simple.

El primer paso es crear un nuevo archivo fuente y llenarlo con el código. Haz clic en *File* en el menú del IDLE y elige *New File*.

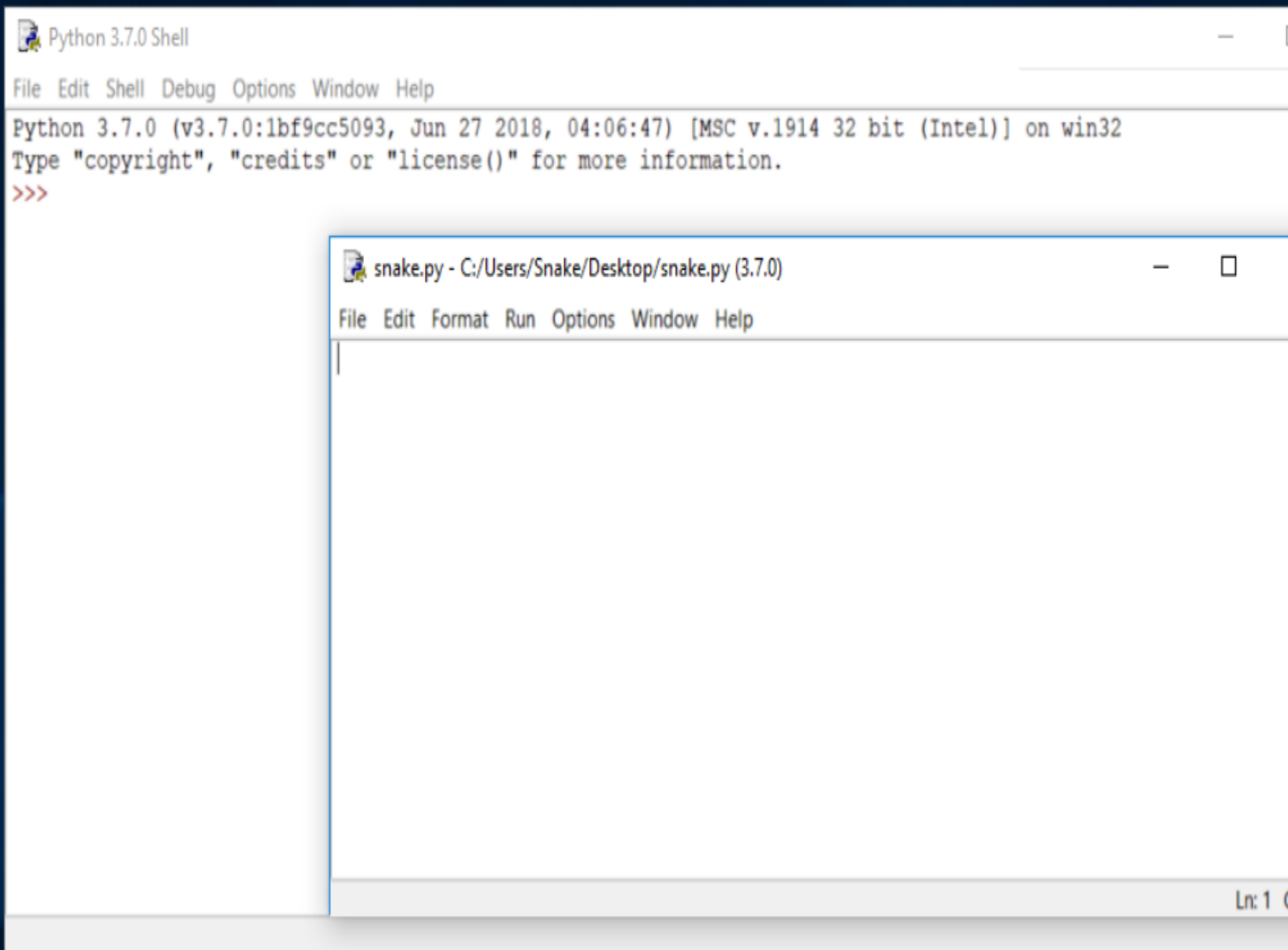


Como puedes ver, IDLE abre una nueva ventana para ti. Puedes usarla para escribir y modificar tu código.

Esta es la **ventana del editor**. Su único propósito es ser un lugar de trabajo en el que se trate tu código fuente. No confundas la ventana del editor con la ventana de shell. Realizan diferentes funciones.

La ventana del editor actualmente no tiene título, pero es una buena práctica comenzar a trabajar nombrando el archivo de origen.

Haz clic en *File* (en la nueva ventana), luego haz clic en *Save as ...* , selecciona una carpeta para el nuevo archivo (el escritorio es un buen lugar para tus primeros intentos de programación) y elige un nombre para el nuevo archivo.



Nota: no establezcas ninguna extensión para el nombre de archivo que vas a utilizar. Python necesita que sus archivos tengan la extensión `.py`, por lo que debes confiar en los valores predeterminados de la ventana de diálogo. El uso de la extensión `.py` estándar permite que el sistema operativo abra estos archivos correctamente.

## ¿Cómo escribir y ejecutar tu primer programa?

Ahora pon solo una línea en tu ventana de editor recién abierta y con nombre.

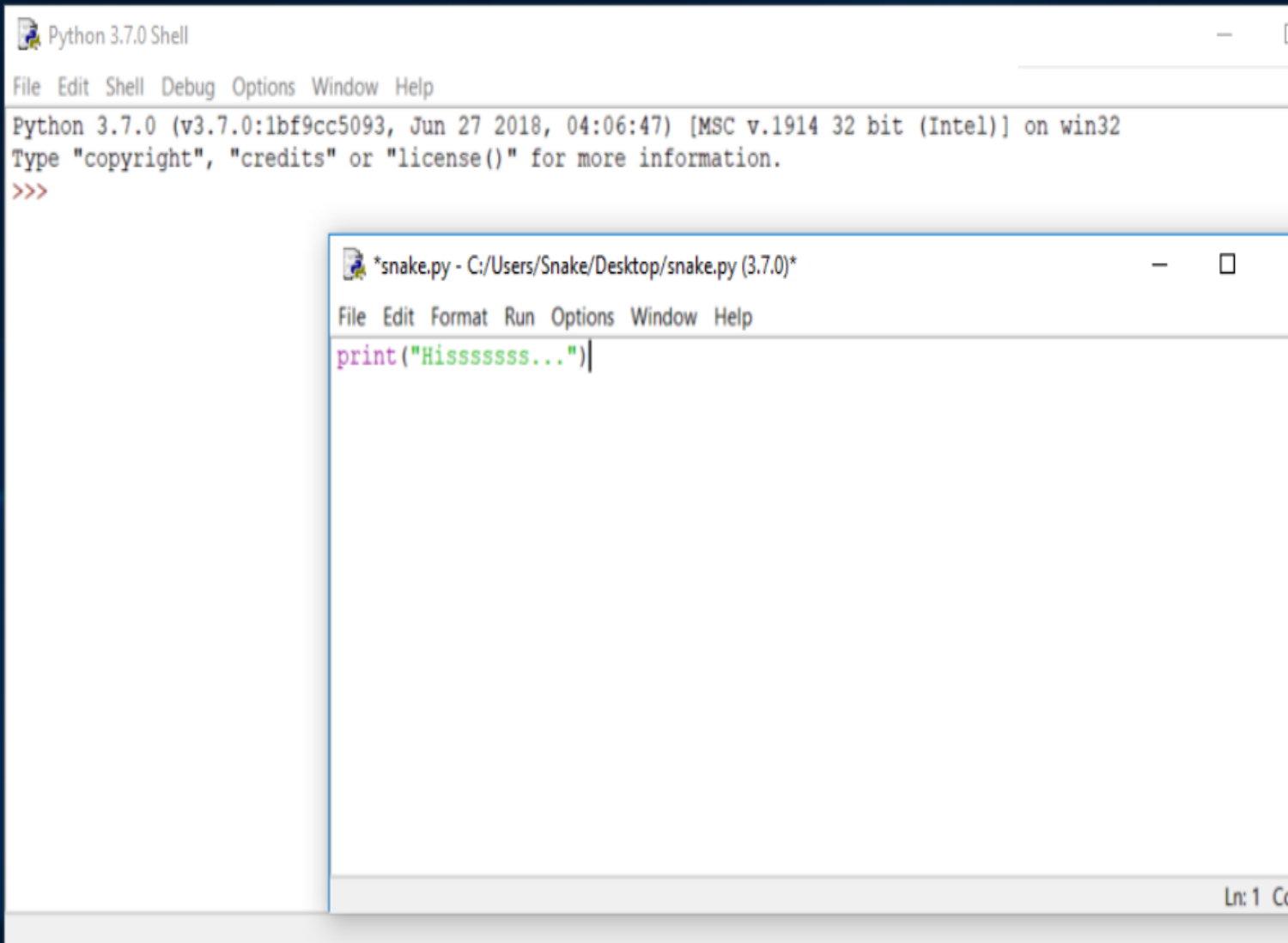
La línea se ve así:

```
print("Hissssss...")
```

Puedes utilizar el portapapeles para copiar el texto en el archivo.

No vamos a explicar el significado del programa en este momento. Encontrarás una discusión detallada en el siguiente capítulo.

Echa un vistazo más de cerca a las comillas. Estas son la forma más simple de comillas (neutral, recta, etc.) que se usan comúnmente en los archivos de origen. No intentes utilizar citas tipográficas (curvadas, rizadas, etc.), utilizadas por los procesadores de texto avanzados, ya que Python no las acepta.



Si todo va bien y no hay errores en el código, la ventana de la consola mostrará los efectos causados por la ejecución del programa.

En este caso, el programa se ejecutara de manera correcta.

Intenta ejecutarlo una vez más. Y una vez más.

Ahora cierra ambas ventanas ahora y vuelve al escritorio.

Python 3.7.0 Shell

File Edit Shell Debug Options Window Help

Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32  
Type "copyright", "credits" or "license()" for more information.

>>>

===== RESTART: C:/Users/Snake/Desktop/snake.py =====

Hisssssss...

>>>

snake.py - C:/Users/Snake/Desktop/snake.py (3.7.0)

File Edit Format Run Options Window Help

```
print("Hisssssss...")
```

```
|
```

Ln: 1 C

## ¿Cómo estropear y arreglar tu código?

Ahora ejecuta IDLE otra vez.

Haz clic en *File* , *Open* , señala el archivo que guardaste anteriormente y deja que IDLE lo lea.

Intenta ejecutarlo de nuevo presionando *F5* cuando la ventana del editor está activa.

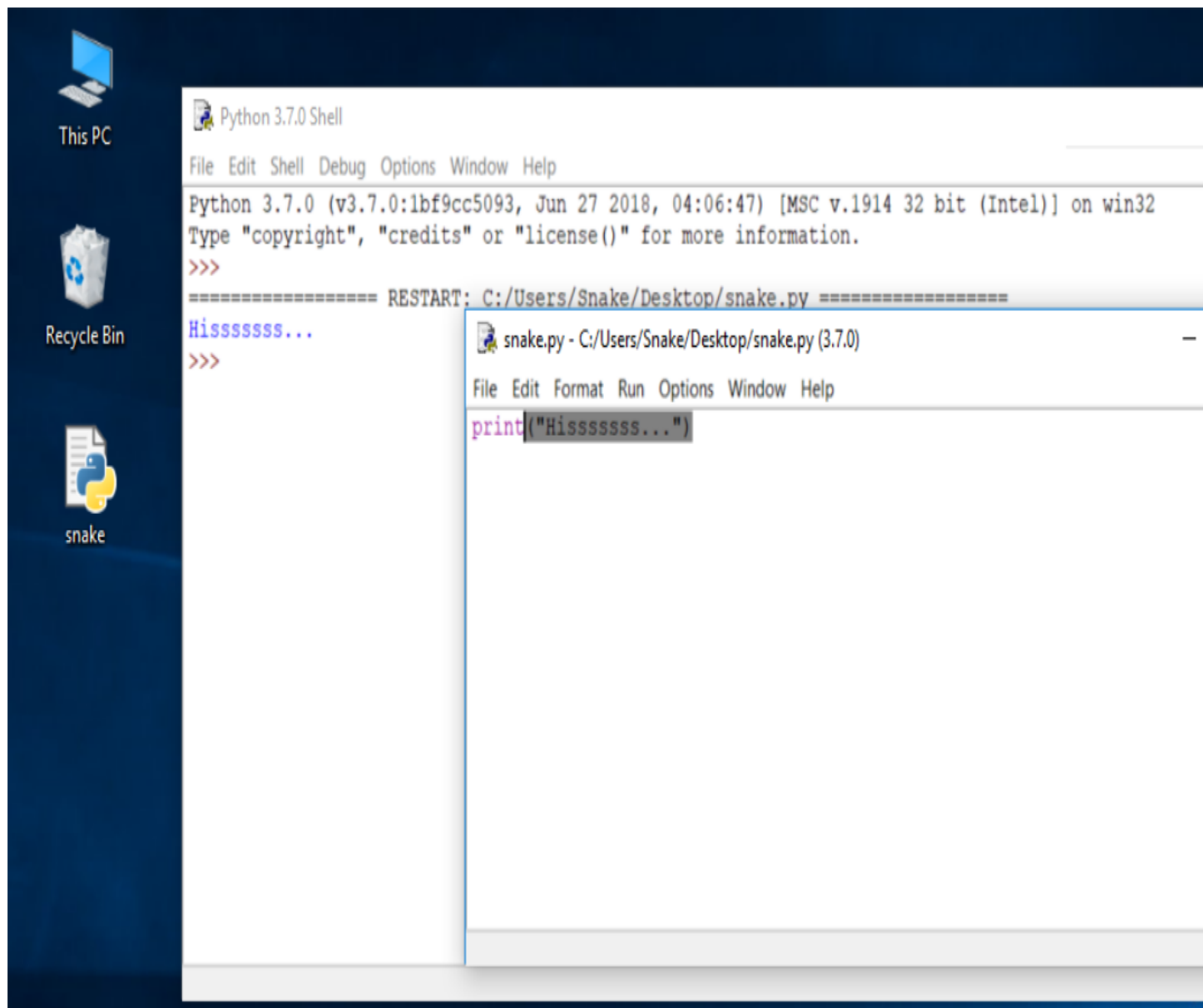
Como puedes ver, IDLE puede guardar tu código y recuperarlo cuando lo necesites de nuevo.

IDLE contiene una característica adicional y útil.

Primero, quita el paréntesis de cierre.

Luego ingresa el paréntesis nuevamente.

Tu código debería parecerse al siguiente:



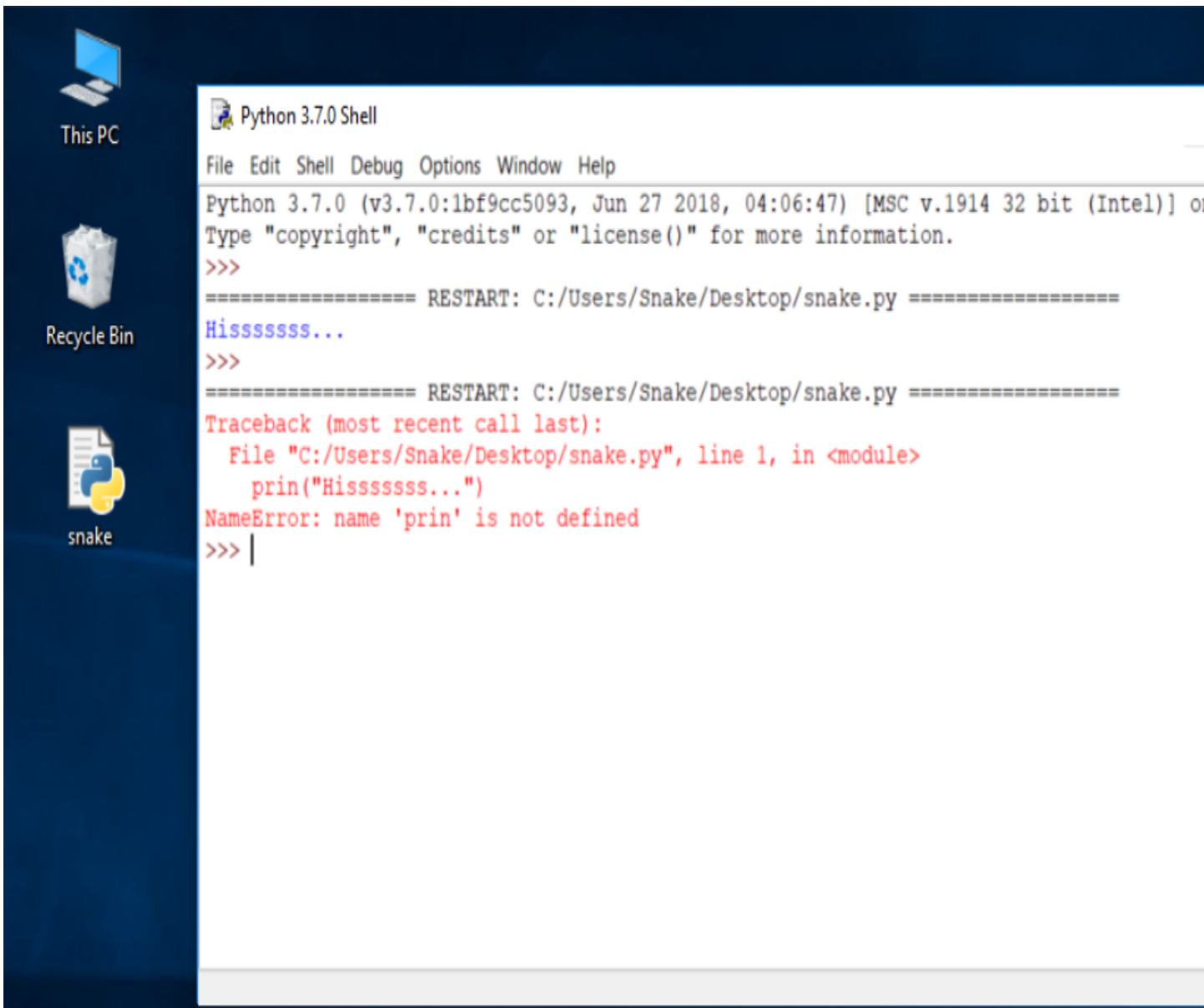
Cada vez que coloques el paréntesis de cierre en tu programa, IDLE mostrará la parte del texto limitada con un par de paréntesis correspondientes. Esto te ayuda a recordar **colocarlos en pares**.

Retira nuevamente el paréntesis de cierre. El código se vuelve erróneo. Ahora contiene un error de sintaxis. IDLE no debería dejar que lo ejecute.

Intenta ejecutar el programa de nuevo. IDLE te recordará que guardes el archivo modificado. Sigue las instrucciones.

## ¿Cómo estropear y arreglar tu código?

Es posible que hayas notado que el mensaje de error generado para el error anterior es bastante diferente del primero.



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/Snake/Desktop/snake.py =====
Hisssssss...
>>>
===== RESTART: C:/Users/Snake/Desktop/snake.py =====
Traceback (most recent call last):
  File "C:/Users/Snake/Desktop/snake.py", line 1, in <module>
    prin("Hisssssss...")
NameError: name 'prin' is not defined
>>> |
```

Esto se debe a que la naturaleza del error es **diferente** y el error se descubre en una **etapa diferente** de la interpretación.



La ventana del editor no proporcionará ninguna información útil sobre el error, pero es posible que las ventanas de la consola sí.

El mensaje (en rojo) muestra (en las siguientes líneas):

- El **rastreo** (que es la ruta que el código atraviesa a través de diferentes partes del programa, puedes ignorarlo por ahora, ya que está vacío en un código tan simple).
- La **ubicación del error** (el nombre del archivo que contiene el error, el número de línea y el nombre del módulo); nota: el número puede ser engañoso, ya que Python generalmente muestra el lugar donde se da cuenta por primera vez de los efectos del error, no necesariamente del error en sí.
- El **contenido de la línea errónea**: nota: la ventana del editor de IDLE no muestra números de línea, pero muestra la ubicación actual del cursor en la esquina inferior derecha; utilízalo para ubicar la línea errónea en un código fuente largo.
- El **nombre del error** y una breve explicación.
- 

Experimenta creando nuevos archivos y ejecutando tu código. Intenta enviar un mensaje diferente a la pantalla, por ejemplo, ¡rawr!, miau, o incluso tal vez un ¡oink! Intenta estropear y arreglar tu código, observa que sucede.

## Interfaz de práctica

Este curso contiene cuatro tipos diferentes de interfaces.

Hasta ahora, haz visto la **Interfaz de estudio** (una o dos ventanas con texto e imágenes/animación) y la **Interfaz de Sandbox**, que puedes usar para probar tu propio código (haz clic en *Sandbox* para cambiar a la Interfaz de Sandbox).

Lo que ves ahora es la **Interfaz de práctica**, que te permite estudiar cosas nuevas y realizar tareas de codificación al mismo tiempo. Utilizarás este tipo de interfaz la mayor parte del tiempo durante el curso.

La Interfaz de práctica consiste en un área de texto a la izquierda y las ventanas del Editor/Consola a la derecha.

Otro tipo de interfaz que verás en el futuro es la Interfaz de prueba/examen, que te permitirá verificar tus conocimientos y habilidades para ver que tan bien has dominado el material de estudio.

## ¡Felicidades! Has completado el Módulo 1

¡Bien hecho! Has llegado al final del Módulo 1 y has completado una meta importante en tu educación de programación en Python. Aquí hay un breve resumen de los objetivos que has cubierto y con los que te has familiarizado en el Módulo 1:

- Los fundamentos de la programación de computadoras, es decir, como funciona la computadora, como se ejecuta el programa, como se define y construye el lenguaje de programación.
- La diferencia entre compilación e interpretación.
- La información básica sobre Python y cómo se posiciona entre otros lenguajes de programación, y qué distingue a sus diferentes versiones.
- Los recursos de estudio y los diferentes tipos de interfaces que utilizarás en el curso.
- 

Ahora estás listo para tomar el cuestionario del módulo, que te ayudará a evaluar lo que has aprendido hasta ahora.

## Módulo 2

Tipos de datos, variables, operaciones básicas de entrada y salida, operadores básicos.

## ¡Hola, Mundo!

Es hora de comenzar a escribir **código real y funcional en Python**. Por el momento será muy sencillo.

Como se muestran algunos conceptos y términos fundamentales, estos fragmentos de código no serán complejos ni difíciles.

Ejecuta el código en la ventana del editor a la derecha. Si todo sale bien, veras la **línea de texto** en la ventana de consola.

Como alternativa, inicia IDLE, crea un nuevo archivo fuente de Python, coloca este código, nombra el archivo y guárdalo. Ahora ejecútalo. Si todo sale bien, verás una línea en la ventana de la consola IDLE. El código que has ejecutado debería parecerte familiar. Viste algo muy similar cuando te guiamos a través de la configuración del entorno IDLE.

Ahora dedicaremos un poco de tiempo para mostrarte y explicarte lo que estás viendo y por que se ve así.

Como puedes ver, el primer programa consta de las siguientes partes:

- La palabra print.
- Un paréntesis de apertura.
- Una comilla.
- Una línea de texto: ¡Hola, Mundo!.
- Otra comilla.
- Un paréntesis de cierre.

Cada uno de los elementos anteriores juega un papel muy importante en el código.

## La función `print()`

Mira la línea de código a continuación:

```
print(";Hola, Mundo!")
```

La palabra **print** que puedes ver aquí es el **nombre de una función**. Eso no significa que dondequiera que aparezca esta palabra, será siempre el nombre de una función. El significado de la palabra proviene del contexto en el cual se haya utilizado la palabra.

Probablemente hayas encontrado el término función muchas veces antes, durante las clases de matemáticas. Probablemente también puedes recordar varios nombres de funciones matemáticas, como seno o logaritmo.

Las funciones de Python, sin embargo, son más flexibles y pueden contener más contenido que sus parientes matemáticos.

Una función (en este contexto) es una parte separada del código de computadora el cual es capaz de:

- **Causar algún efecto** (por ejemplo, enviar texto a la terminal, crear un archivo, dibujar una imagen, reproducir un sonido, etc.); esto es algo completamente inaudito en el mundo de las matemáticas.
- **Evaluar un valor o algunos valores** (por ejemplo, la raíz cuadrada de un valor o la longitud de un texto dado); esto es lo que hace que las funciones de Python sean parientes de los conceptos matemáticos.

Además, muchas de las funciones de Python pueden hacer las dos cosas anteriores juntas.

¿De dónde provienen las funciones?

- Pueden venir **de Python mismo**. La función `print` es una de este tipo; dicha función es un valor agregado de Python junto con su entorno (está **integrada**); no tienes que hacer nada especial (por ejemplo, pedirle a alguien algo) si quieres usarla.
- Pueden provenir de uno o varios de los **módulos** de Python llamados complementos; algunos de los módulos vienen con Python, otros pueden requerir una instalación por separado, cual sea el caso, todos deben estar conectados explícitamente con el código (te mostraremos cómo hacer esto pronto).
- Puedes **escribirlas tú mismo**, colocando tantas funciones como desees y necesites dentro de su programa para hacerlo más simple, claro y elegante.

El nombre de la función debe ser **significativo** (el nombre de la función `print` es evidente), imprime en la terminal. Si vas a utilizar alguna función ya existente, no podrás modificar su nombre, pero cuando comiences a escribir tus propias funciones, debes considerar cuidadosamente la elección de nombres.

## La función `print()`

Como se dijo anteriormente, una función puede tener:

- Un **efecto**.
- Un **resultado**.

También hay un tercer componente de la función, muy importante, el o los **argumento(s)**.

Las funciones matemáticas usualmente toman un argumento, por ejemplo,  $\sin(x)$  toma una  $x$ , que es la medida de un ángulo.

Las funciones de Python, por otro lado, son más versátiles. Dependiendo de las necesidades individuales, pueden aceptar cualquier número de argumentos, tantos como sea necesario para realizar sus tareas. Nota: algunas funciones de Python no necesitan ningún argumento.

```
print("¡Hola, Mundo!")
```

A pesar del número de argumentos necesarios o proporcionados, las funciones de Python demandan fuertemente la presencia de **un par de paréntesis** - el de apertura y de cierre, respectivamente.

Si deseas entregar uno o más argumentos a una función, colócalos **dentro de los paréntesis**. Si vas a utilizar una función que no tiene ningún argumento, aún tiene que tener los paréntesis.

Nota: para distinguir las palabras comunes de los nombres de funciones, coloca un **par de paréntesis vacíos** después de sus nombres, incluso si la función correspondiente requiere uno o más argumentos. Esta es una medida estándar.

La función de la que estamos hablando aquí es `print()`.

¿La función `print()` en nuestro ejemplo tiene algún argumento?

Por supuesto que sí, pero ¿Qué son los argumentos?

## La función `print()`

El único argumento entregado a la función `print()` en este ejemplo es una **cadena**:

```
print("¡Hola, Mundo!")
```

Como se puede ver, la **cadena está delimitada por comillas** - de hecho, las comillas forman la cadena, recortan una parte del código y le asignan un significado diferente.

Podemos imaginar que las comillas significan algo así: el texto entre nosotros no es un código. No está diseñado para ser ejecutado, y se debe tomar tal como está.

Casi cualquier cosa que ponga dentro de las comillas se tomará de manera literal, no como código, sino como **datos**. Intenta jugar con esta cadena en particular - puedes modificarla. Ingresa contenido nuevo o borra parte del contenido existente.

Existe más de una forma de como especificar una cadena dentro del código de Python, pero por ahora, esta será suficiente.



Hasta ahora, has aprendido acerca de dos partes importantes del código- la función y la cadena. Hemos hablado de ellos en términos de sintaxis, pero ahora es el momento de discutirlos en términos de semántica.

## La función `print()`

El nombre de la función (***print*** en este caso) junto con los paréntesis y los argumentos, forman la **invocación de la función**.

Discutiremos esto en mayor profundidad mas adelante, pero por lo pronto, arrojaremos un poco más de luz al asunto.

```
print("¡Hola, Mundo!")
```

¿Qué sucede cuando Python encuentra una invocación como la que está a continuación?

```
nombreFunción(argumento)
```

Veamos:

- Primero, Python comprueba si el nombre especificado es **legal** (explora sus datos internos para encontrar una función existente del nombre; si esta búsqueda falla, Python cancela el código).
- En segundo lugar, Python comprueba si los requisitos de la función para el número de argumentos **le permiten invocar** la función de esta manera (por ejemplo, si una función específica exige exactamente dos argumentos, cualquier invocación que entregue solo un argumento se considerará errónea y abortará la ejecución del código).
- Tercero, Python **deja el código por un momento** y salta dentro de la función que se desea invocar; por lo tanto, también toma los argumentos y los pasa a la función.
- Cuarto, la función **ejecuta el código**, provoca el efecto deseado (si lo hubiera), evalúa el (los) resultado(s) deseado(s) y termina la tarea.
- Finalmente, Python **regresa al código** (al lugar inmediatamente después de la invocación) y reanuda su ejecución.

## Tiempo Estimado

5 minutos

## Nivel de dificultad

Muy fácil

## Objetivos

- Familiarizarse con la función `print()` y sus capacidades de formateo.
- Experimentar con el código de Python.

## Escenario

El comando `print()`, el cual es una de las directivas más sencillas de Python, simplemente imprime una línea de texto en la pantalla.

En tu primer laboratorio:

- Utiliza la función `print()` para imprimir la línea "¡Hola, Mundo!" en la pantalla.
- Una vez hecho esto, utiliza la función `print()` nuevamente, pero esta vez imprime tu nombre.
- Elimina las comillas dobles y ejecuta el código. Observa la reacción de Python. ¿Qué tipo de error se produce?
- Luego, elimina los paréntesis, vuelve a poner las comillas dobles y vuelve a ejecutar el código. ¿Qué tipo de error se produce esta vez?
- Experimenta tanto como puedas. Cambia las comillas dobles a comillas simples, utiliza múltiples funciones `print()` en la misma línea y luego en líneas diferentes. Observa que es lo que ocurre.

## La función `print()`

Tres preguntas importantes deben ser respondidas antes de continuar:

### 1. ¿Cuál es el efecto que causa la función `print()`?

El efecto es muy útil y espectacular. La función toma los argumentos (puede aceptar más de un argumento y también puede aceptar menos de un argumento) los convierte en un formato legible para el ser humano si es necesario (como puedes sospechar, las cadenas no requieren esta acción, ya que la cadena ya está legible) y **envía los datos resultantes al dispositivo de salida** (generalmente la consola); en otras palabras, cualquier cosa que se ponga en la función de `print()` aparecerá en la pantalla.

No es de extrañar entonces, que de ahora en adelante, utilizarás `print()` muy intensamente para ver los resultados de tus operaciones y evaluaciones.

## **2. ¿Qué argumentos espera print()?**

Cualquiera. Te mostraremos pronto que print() puede operar con prácticamente todos los tipos de datos ofrecidos por Python. Cadenas, números, caracteres, valores lógicos, objetos: cualquiera de estos se puede pasar con éxito a print().

## **3. ¿Qué valor evalúa la función print()?**

Ninguno. Su efecto es suficiente - print() no evalúa nada.

## La función `print()` - instrucciones

A estas alturas ya sabes que este programa contiene una invocación de función. A su vez, la invocación de función es uno de los posibles tipos de **instrucciones** de Python. Por lo tanto, este programa consiste de una sola instrucción.

Por supuesto, cualquier programa complejo generalmente contiene muchas más instrucciones que una. La pregunta es, ¿Cómo se acopla más de una instrucción en el código de Python?

La sintaxis de Python es bastante específica en esta área. A diferencia de la mayoría de los lenguajes de programación, Python requiere que **no haya más de una instrucción por una línea**.

Una línea puede estar vacía (por ejemplo, puede no contener ninguna instrucción) pero no debe contener dos, tres o más instrucciones. Esto está estrictamente prohibido.

Nota: Python hace una excepción a esta regla: permite que una instrucción se extienda por más de una línea (lo que puede ser útil cuando el código contiene construcciones complejas).

Vamos a expandir el código un poco, puedes verlo en el editor. Ejecútalo y nota lo que ves en la consola.

---

Tu consola Python ahora debería verse así:

La Witsi Witsi Araña subió a su telaraña. Vino la lluvia y se la llevó.

Esta es una buena oportunidad para hacer algunas observaciones:

- El programa **invoca la función `print()` dos veces**, como puedes ver hay dos líneas separadas en la consola: esto significa que `print()` comienza su salida desde una nueva línea cada vez que comienza su ejecución. Puedes cambiar este comportamiento, pero también puedes usarlo a tu favor.
- Cada invocación de `print()` contiene una cadena diferente, como su argumento y el contenido de la consola lo reflejan- esto significa que **las instrucciones en el código se ejecutan en el mismo orden** en que se colocaron en el archivo de origen; no se ejecuta la siguiente instrucción hasta que se complete la anterior (hay algunas excepciones a esta regla, pero puedes ignorarlas por ahora).

## La función `print()` - instrucciones

Hemos cambiado un poco el ejemplo: hemos agregado una invocación **vacía** de la función `print()`. La llamamos vacía porque no hemos agregado ningún argumento a la función.

Lo puedes ver en la ventana del editor. Ejecuta el código.

¿Qué ocurre?

---

Si todo sale bien, deberías ver algo como esto:

La Witsi Witsi Araña subió a su telaraña. Vino la lluvia y se la llevó.



Como puedes ver, la invocación de `print()` vacía no esta tan vacía como se esperaba - genera una línea vacía (esta interpretación también es correcta) su salida es solo una nueva línea.

Esta no es la única forma de producir una **nueva línea** en la consola de salida. Enseguida mostraremos otra manera.

## La función `print()` - los caracteres de escape y nueva línea

Hemos modificado el código de nuevo. Obsérvalo con cuidado.

Hay dos cambios muy sutiles: hemos insertado un par extraño de caracteres dentro del texto. Se ven así: `\n`.

Curiosamente, mientras **tu ves dos caracteres, Python ve solo uno.**

La barra invertida (`\`) tiene un significado muy especial cuando se usa dentro de las cadenas, es llamado **el carácter de escape**.

La palabra *escape* debe entenderse claramente- significa que la serie de caracteres en la cadena se escapa (detiene) por un momento (un momento muy corto) para introducir una inclusión especial.

En otras palabras, la barra invertida no significa nada, sino que es solo un tipo de anuncio, de que el siguiente carácter después de la barra invertida también tiene un significado diferente.

La letra `n` colocada después de la barra invertida proviene de la palabra *newline* (nueva línea).

Tanto la barra diagonal inversa como la *n* forman un símbolo especial denominado **carácter de nueva línea** (*newline character*), que incita a la consola a iniciar una **nueva línea de salida**.

---

Ejecuta el código. La consola ahora debería verse así:

La Witsi Witsi Araña subió a su telaraña. Vino la lluvia y se la llevó.

Como se puede observar, aparecen dos nuevas líneas en la canción infantil, en los lugares donde se ha utilizado `\n`.

## La función `print()` los caracteres de escape y nueva línea

El utilizar la diagonal invertida tiene dos características importantes:

1. Si deseas colocar solo una barra invertida dentro de una cadena, no olvides su naturaleza de escape: tienes que duplicarla, por ejemplo, la siguiente invocación causará un error:

```
print("\n")
```

Mientras que esta no lo hará:

```
print("\\n")
```

2. No todos los pares de escape (la barra invertida junto con otro carácter) significan algo.

Experimenta con el código en el editor, ejecútalo y observa lo que sucede.

## La función `print()` utilizando argumentos múltiples

Hasta ahora se ha probado el comportamiento de la función `print()` sin argumentos y con un argumento. También vale la pena intentar alimentar la función `print()` con más de un argumento.

Mira la ventana del editor. Esto es lo que vamos a probar ahora:

```
print("Witsi witsi araña" , "subió" , "su telaraña.")
```

Hay una invocación de la función `print()` pero contiene **tres argumentos**. Todos ellos son cadenas.

Los argumentos están **separados por comas**. Se han rodeado de espacios para hacerlos más visibles, pero no es realmente necesario y no se hará más.

En este caso, las comas que separan los argumentos desempeñan un papel completamente diferente a la coma dentro de la cadena. El primero es una parte de la sintaxis de Python, el segundo está destinado a mostrarse en la consola.

Si vuelves a mirar el código, verás que no hay espacios dentro de las cadenas.

Ejecuta el código y observa lo que pasa.

---

La consola ahora debería mostrar el siguiente texto:

La Witsi witsi araña subió su telaraña.

Los espacios, removidos de las cadenas, han vuelto a aparecer. ¿Puedes explicar porque?

Dos conclusiones surgen de este ejemplo:

- Una función `print()` invocada con más de un argumento genera la **salida en una sola línea**.
- La función `print()` **pone un espacio entre los argumentos emitidos** por iniciativa propia.

## La función `print()` - La manera posicional de pasar los argumentos

Ahora que sabes un poco acerca de la función `print()` y como personalizarla, te mostraremos como cambiarla.

Deberías de poder predecir la salida sin ejecutar el código en el editor.

La forma en que pasamos los argumentos a la función `print()` es la más común en Python, y se denomina **manera posicional** (este nombre proviene del hecho de que el significado del argumento está dictado por su posición, por ejemplo, el segundo argumento se emitirá después del primero, y no al revés).

Ejecuta el código y verifica si la salida coincide con tus predicciones.

## La función `print()` - los argumentos de palabras clave

Python ofrece otro mecanismo para transmitir o pasar los argumentos, que puede ser útil cuando se desea convenir a la función `print()` de que cambie su comportamiento un poco.

No se va a explicar en profundidad ahora. Se planea hacer esto cuando se trate el tema de funciones. Por ahora, simplemente queremos mostrarte como funciona. Siéntete libre de utilizarlo en tus propios programas.

El mecanismo se llama **argumentos de palabras clave**. El nombre se deriva del hecho de que el significado de estos argumentos no se toma de su ubicación (posición) sino de la palabra especial (palabra clave) utilizada para identificarlos.

La función `print()` tiene dos argumentos de palabras clave que se pueden utilizar para estos propósitos. El primero de ellos se llama `end`.

En la ventana del editor se puede ver un ejemplo muy simple de como utilizar un argumento de palabra clave.

Para utilizarlo es necesario conocer algunas reglas:

- Un argumento de palabra clave consta de tres elementos: una **palabra clave** que identifica el argumento (end -termina aquí); un **signo de igual** (`=`); y un **valor** asignado a ese argumento.
- Cualquier argumento de palabra clave debe ponerse **después del último argumento posicional** (esto es muy importante).

En nuestro ejemplo, hemos utilizado el argumento de palabra clave `end` y lo hemos igualado a una cadena que contiene un espacio.

Ejecuta el código para ver como funciona.

---

La consola ahora debería mostrar el siguiente texto:

Mi nombre es Python. Monty Python.

Como puedes ver, el argumento de palabra clave `end` determina los caracteres que la función `print()` envía a la salida una vez que llega al final de sus argumentos posicionales.

El comportamiento predeterminado refleja la situación en la que el argumento de la palabra clave `end` se usa **implícitamente** de la siguiente manera: `end="\n"`.

## La función `print()` - los argumentos de palabras clave

Y ahora, es el momento de intentar algo más difícil.

Si observas detenidamente, verás que hemos utilizado el argumento `end`, pero su cadena asignada está vacía (no contiene ningún carácter).

¿Qué pasará ahora? Ejecuta el programa en el editor para averiguarlo.

Ya que al argumento `end` se le ha asignado a nada, la función `print()` tampoco genera nada, una vez que se hayan agotado los argumentos posicionales.

La consola ahora debería mostrar el siguiente texto:

Mi nombre es Monty Python.

Nota: **No se han enviado nuevas líneas a la salida.**

La cadena asignada al argumento de la palabra clave `end` puede ser de cualquier longitud. Experimenta con ello si gustas.

## La función `print()` - los argumentos de palabras clave

Se estableció anteriormente que la función `print()` separa los argumentos generados con espacios. Este comportamiento también puede ser cambiado.

El **argumento de palabra clave** que puede hacer esto se denomina `sep` (como *separador*).

Mira el código en el editor y ejecútalo.

El argumento sep entrega el siguiente resultado:

Mi-nombre-es-Monty-Python.

La función print() ahora utiliza un guión, en lugar de un espacio, para separar los argumentos generados.

Nota: el valor del argumento sep también puede ser una cadena vacía. Pruébalo tu mismo.

|  
|  
|

## La función `print()` - los argumentos de palabras clave

Ambos argumentos de palabras clave pueden **mezclarse en una invocación**, como aquí en la ventana del editor.

El ejemplo no tiene mucho sentido, pero representa visiblemente las interacciones entre `end` y `sep`.

¿Puedes predecir la salida?

Ejecuta el código y ve si coincide con tus predicciones.

Ahora que comprendes la función `print()`, estás listo para considerar aprender cómo almacenar y procesar datos en Python.

Sin `print()`, no se podría ver ningún resultado.

### LABORATORIO

## Tiempo Estimado

5 minutos

## Nivel de dificultad

Muy fácil

## Objetivos

- Familiarizarse con la función de `print()` y sus capacidades de formato.
- Experimentar con el código de Python.

## Escenario

Modifica la primera línea de código en el editor, utilizando las palabras clave `sep` y `end`, para que coincida con el resultado esperado. Recuerda, utilizar dos funciones `print()`.

No cambies nada en la segunda invocación de `print()`.

## Resultado Esperado

Fundamentos\*\*\*Programación\*\*\*en...Python

## Tiempo Estimado

5-10 minutos

## Nivel de dificultad

Fácil

## Objetivos

- Experimentar con el código Python existente.
- Descubrir y solucionar errores básicos de sintaxis.
- Familiarizarse con la función `print()` y sus capacidades de formato.

## Escenario

Recomendamos que **juegues con el código** que hemos escrito para ti y que realices algunas correcciones (quizás incluso destructivas). Siéntete libre de modificar cualquier parte del código, pero hay una condición: aprende de tus errores y saca tus propias conclusiones.

Intenta:

- Minimizar el número de invocaciones de la función `print()` insertando la secuencia `\n` en las cadenas.
- Hacer la flecha dos veces más grande (pero mantener las proporciones).
- Duplicar la flecha, colocando ambas flechas lado a lado; nota: una cadena se puede multiplicar usando el siguiente truco: `"string" * 2` producirá `"stringstring"` (te contaremos más sobre ello pronto).
- Elimina cualquiera de las comillas y observa detenidamente la respuesta de Python; presta atención a donde Python ve un error: ¿es el lugar en donde realmente existe el error?
- Haz lo mismo con algunos de los paréntesis.
- Cambia cualquiera de las palabras `print` en otra cosa (por ejemplo de minúscula a mayúscula, `Print`) - ¿Qué sucede ahora?
- Reemplaza algunas de las comillas por apóstrofes; observa lo que pasa detenidamente.

## Literales - los datos en si mismos

Ahora que tienes un poco de conocimiento acerca de algunas de las poderosas características que ofrece la función `print()`, es tiempo de aprender sobre cuestiones nuevas, y un nuevo término - el **literal**.

**Un literal se refiere a datos cuyos valores están determinados por el literal mismo.**

Debido a que es un concepto un poco difícil de entender, un buen ejemplo puede ser muy útil.

Observa los siguientes dígitos:

123

¿Puedes adivinar qué valor representa? claro que puedes - es *ciento veintitrés*.

Que tal este:

c

¿Representa algún valor? Tal vez. Puede ser el símbolo de la velocidad de la luz, por ejemplo. También puede representar la constante de integración. Incluso la longitud de una hipotenusa en el Teorema de Pitágoras. Existen muchas posibilidades.

No se puede elegir el valor correcto sin algo de conocimiento adicional.

Y esta es la pista: 123 es un literal, y c no lo es.

Se utilizan literales **para codificar datos y ponerlos dentro del código**. Ahora mostraremos algunas convenciones que se deben seguir al utilizar Python.

## Literales - los datos en si mismos

Comencemos con un sencillo experimento, observa el fragmento de código en el editor.

La primera línea luce familiar. La segunda parece ser errónea debido a la falta visible de comillas.

Intenta ejecutarlo.

Si todo salió bien, ahora deberías de ver dos líneas idénticas.

¿Qué paso? ¿Qué significa?

A través de este ejemplo, encuentras dos tipos diferentes de literales:

- Una **cadena**, la cual ya conoces.
- Y un número **entero**, algo completamente nuevo.

La función print() los muestra exactamente de la misma manera. Sin embargo, internamente, la memoria de la computadora los almacena de dos maneras completamente diferentes. La cadena existe como eso, solo una cadena, una serie de letras.

El número es convertido a una representación maquina (una serie de bits). La función print() es capaz de mostrar ambos en una forma legible para humanos.

Vamos a tomar algo de tiempo para discutir literales numéricas y su vida interna.

## Enteros

Quizá ya sepas un poco acerca de como las computadoras hacen cálculos con números. Tal vez has escuchado del **sistema binario**, y como es que ese es el sistema que las computadoras utilizan para almacenar números y como es que pueden realizar cualquier tipo de operaciones con ellos.

No exploraremos las complejidades de los sistemas numéricos posicionales, pero se puede afirmar que todos los números manejados por las computadoras modernas son de dos tipos:

- **Enteros**, es decir, aquellos que no tienen una parte fraccionaria.
- Y números **punto-flotantes** (o simplemente **flotantes**), los cuales contienen (o son capaces de contener) una parte fraccionaria.

Esta definición no es tan precisa, pero es suficiente por ahora. La distinción es muy importante, y la frontera entre estos dos tipos de números es muy estricta. Ambos tipos difieren significativamente en como son almacenados en una computadora y en el rango de valores que aceptan.

La característica del valor numérico que determina el tipo, rango y aplicación se denomina el **tipo**.

Si se codifica un literal y se coloca dentro del código de Python, la forma del literal determina la representación (tipo) que Python utilizará para **almacenarlo en la memoria**.

Por ahora, dejemos los números flotantes a un lado (regresaremos a ellos pronto) y analicemos como es que Python reconoce un número entero.

El proceso es casi como usar lápiz y papel, es simplemente una cadena de dígitos que conforman el número, pero hay una condición, no se deben insertar caracteres que no sean dígitos dentro del número.

Tomemos por ejemplo, el número once millones ciento once mil ciento once. Si tomaras ahorita un lápiz en tu mano, escribirías el siguiente número: 11,111,111, o así: 11.111.111, incluso de esta manera: 11 111 111.

Es claro que la separación hace que sea más fácil de leer, especialmente cuando el número tiene demasiados dígitos. Sin embargo, Python no acepta estas cosas. Esta **prohibido**. ¿Qué es lo que Python permite? El uso de **guion bajo** en los literales numéricos.\*

Por lo tanto, el número se puede escribir ya sea así: 11111111, o como sigue: 11\_111\_111.

\*Python 3.6 ha introducido el guion bajo en los literales numéricos, permitiendo colocar un guion bajo entre dígitos y después de especificadores de base para mejorar la legibilidad. Esta característica no está disponible en versiones anteriores de Python.

¿Cómo se codifican los números negativos en Python? Como normalmente se hace, agregando un signo de **menos**. Se puede escribir: -11111111, o -11\_111\_111.

Los números positivos no requieren un signo positivo antepuesto, pero es permitido, si se desea hacer. Las siguientes líneas describen el mismo número: +11111111 y 11111111.

## Enteros: números octales y hexadecimales

Existen dos convenciones adicionales en Python que no son conocidas en el mundo de las matemáticas. El primero nos permite utilizar un número en su representación **octal**.

Si un número entero está precedido por un código 0O o 0o (cero-o), el número será tratado como un valor octal.

Esto significa que el número debe contener dígitos en el rango del [0..7] únicamente.

0o123 es un número **octal** con un valor (decimal) igual a 83.

La función print() realiza la conversión automáticamente. Intenta esto:

```
print(0o123)
```

La segunda convención nos permite utilizar números en **hexadecimal**. Dichos números deben ser precedidos por el prefijo 0x o 0X (cero-x).

0x123 es un número **hexadecimal** con un valor (decimal) igual a 291. La función print() puede manejar estos valores también. Intenta esto:

```
print(0x123)
```



## Flotantes

Ahora es tiempo de hablar acerca de otro tipo, el cual esta designado para representar y almacenar los números que (como lo diría un matemático) tienen una **parte decimal no vacía**.

Son números que tienen (o pueden tener) una parte fraccionaria después del punto decimal, y aunque esta definición es muy pobre, es suficiente para lo que se desea discutir.

Cuando se usan términos como *dos y medio* o *menos cero punto cuatro*, pensamos en números que la computadora considera como números **punto-flotante**:

2.5 -0.4

Nota: *dos punto cinco* se ve normal cuando se escribe en un programa, sin embargo si tu idioma nativo prefiere el uso de una coma en lugar de un punto, se debe asegurar que **el número no contenga más comas**.

Python no lo aceptará, o (en casos poco probables) puede malinterpretar el número, debido a que la coma tiene su propio significado en Python.

Si se quiere utilizar solo el valor de dos punto cinco, se debe escribir como se mostró anteriormente. Nota que hay un punto entre el 2 y el 5 - no una coma.

Como puedes imaginar, el valor de **cero punto cuatro** puede ser escrito en Python como:

0.4

Pero no hay que olvidar esta sencilla regla, se puede omitir el cero cuando es el único dígito antes del punto decimal.

En esencia, el valor 0.4 se puede escribir como:

.4

Por ejemplo: el valor de 4.0 puede ser escrito como:

4.

Esto no cambiará su tipo ni su valor.

## Enteros vs. Flotantes

El punto decimal es esencialmente importante para reconocer números punto-flotantes en Python.

Observa estos dos números:

4 4.0

Se puede pensar que son idénticos, pero Python los ve de una manera completamente distinta.

4 es un número **entero**, mientras que 4.0 es un número **punto-flotante**.

El punto decimal es lo que determina si es flotante.

Por otro lado, no solo el punto hace que un número sea flotante. Se puede utilizar la letra e.

Cuando se desea utilizar números que son muy pequeños o muy grandes, se puede implementar la **notación científica**.

Por ejemplo, la velocidad de la luz, expresada en *metros por segundo*. Escrita directamente se vería de la siguiente manera: 300000000.

Para evitar escribir tantos ceros, los libros de texto emplean la forma abreviada, la cual probablemente hayas visto:  $3 \times 10^8$ .

Se lee de la siguiente manera: tres por diez elevado a la octava potencia.

En Python, el mismo efecto puede ser logrado de una manera similar, observa lo siguiente:

3E8

La letra E (también se puede utilizar la letra minúscula e - proviene de la palabra **exponente**) la cual significa *por diez a la n potencia*.

Nota:

- El **exponente** (el valor después de la  $E$ ) debe ser un valor entero.
- La **base** (el valor antes de la  $E$ ) puede o no ser un valor entero.

# Codificando Flotantes

Veamos ahora como almacenar números que son muy pequeños (en el sentido de que están muy cerca del cero).

Una constante de física denominada "*La Constante de Planck*" (denotada como  $h$ ), de acuerdo con los libros de texto, tiene un valor de: **6.62607 x 10<sup>-34</sup>**.

Si se quisiera utilizar en un programa, se debería escribir de la siguiente manera:

6.62607E-34

Nota: el hecho de que se haya escogido una de las posibles formas de codificación de un valor flotante no significa que Python lo presentará de la misma manera.

Python podría en ocasiones elegir una **notación diferente**.

Por ejemplo, supongamos que se ha elegido utilizar la siguiente notación:

[illegible]

Cuando se corre en Python:

[illegible]

Este es el resultado:

1e-22

**salida**

Python siempre elige **la presentación más corta del número**, y esto se debe de tomar en consideración al crear literales.

# Cadenas

Las cadenas se emplean cuando se requiere procesar texto (como nombres de cualquier tipo, direcciones, novelas, etc.), no números.

Ya conoces un poco acerca de ellos, por ejemplo, que **las cadenas requieren comillas** así como los flotantes necesitan punto decimal.

Este es un ejemplo de una cadena: "Yo soy una cadena."

Sin embargo, hay una cuestión. ¿Cómo se puede codificar una comilla dentro de una cadena que ya está delimitada por comillas?

Supongamos que se desea mostrar un muy sencillo mensaje:

Me gusta "Monty Python"

¿Cómo se puede hacer esto sin generar un error? Existen dos posibles soluciones.

La primera se basa en el concepto ya conocido del **carácter de escape**, el cual recordarás se utiliza empleando la **diagonal invertida**. La diagonal invertida puede también escapar de la comilla. Una comilla precedida por una diagonal invertida cambia su significado, no es un limitador, simplemente es una comilla. Lo siguiente funcionará como se desea:

```
print("Me gusta \"Monty Python\"")
```

Nota: ¿Existen dos comillas con escape en la cadena, puedes observar ambas?

La segunda solución puede ser un poco sorprendente. Python puede utilizar **una apóstrofe en lugar de una comilla**. Cualquiera de estos dos caracteres puede delimitar una cadena, pero para ello se debe ser **consistente**.

Si se delimita una cadena con una comilla, se debe cerrar con una comilla.

Si se inicia una cadena con un apóstrofe, se debe terminar con un apóstrofe.

Este ejemplo funcionará también:

```
print('Me gusta "Monty Python"')
```

Nota: en este ejemplo no se requiere nada de escapes.

## Codificando cadenas

Ahora, la siguiente pregunta es: ¿Cómo se puede insertar un apóstrofe en una cadena la cual está limitada por dos apóstrofes?

A estas alturas ya se debería tener una posible respuesta o dos.

Intenta imprimir una cadena que contenga el siguiente mensaje:

I'm Monty Python.

¿Sabes cómo hacerlo? Haz clic en *Revisar* para saber si estas en lo cierto:

[Revisar](#)

Como se puede observar, la diagonal invertida es una herramienta muy poderosa, puede escapar no solo comillas, sino también apóstrofes.

Ya se ha mostrado, pero se desea hacer énfasis en este fenómeno una vez mas - **una cadena puede estar vacía** - puede no contener caracter alguno.

Una cadena vacía sigue siendo una cadena:

```
" ""
```

## Valores Booleanos

Para concluir con los literales de Python, existen dos más.

No son tan obvios como los anteriores y se emplean para representar un valor muy abstracto - **la veracidad**.

Cada vez que se le pregunta a Python si un número es más grande que otro, el resultado es la creación de un tipo de dato muy específico - un valor **booleano**.

El nombre proviene de George Boole (1815-1864), el autor de *Las Leyes del Pensamiento*, las cuales definen el **Algebra Booleana** - una parte del algebra que hace uso de dos valores: Verdadero y Falso, denotados como 1 y 0.

Un programador escribe un programa, y el programa hace preguntas. Python ejecuta el programa, y provee las respuestas. El programa debe ser capaz de reaccionar acorde a las respuestas recibidas.

Afortunadamente, las computadoras solo conocen dos tipos de respuestas:

- Si, esto es verdad.
- No, esto es falso.

Nunca habrá una respuesta como: *No lo sé* o *probablemente si, pero no estoy seguro*.

Python, es entonces, un reptil **binario**.

Estos dos valores booleanos tienen denotaciones estrictas en Python:

True False

No se pueden cambiar, se deben tomar estos símbolos como son, incluso respetando las **mayúsculas y minúsculas**.

Reto: ¿Cuál será el resultado del siguiente fragmento de código?

```
print(True > False) print(True < False)
```

Ejecuta el código en la terminal. ¿Puedes explicar el resultado?



# Tiempo Estimado

5 minutos

# Nivel de dificultad

Fácil

## Objetivos

- Familiarizarse con la función `print()` y sus capacidades de formato.
- Practicar el codificar cadenas.
- Experimentar con el código de Python.

## Escenario

Escribe una sola línea de código, utilizando la función `print()`, así como los caracteres de nueva línea y escape, para obtener la salida esperada de tres líneas.

## Salida Esperada

```
"Estoy" ""aprendiendo"" ""Python""
```

## Puntos Clave

1. **Literales** son notaciones para representar valores fijos en el código. Python tiene varios tipos de literales, es decir, un literal puede ser un número por ejemplo, 123), o una cadena (por ejemplo, "Yo soy un literal.").
2. El **Sistema Binario** es un sistema numérico que emplea 2 como su base. Por lo tanto, un número binario está compuesto por 0s y 1s únicamente, por ejemplo, 1010 es 10 en decimal.  
Los sistemas de numeración Octales y Hexadecimales son similares pues emplean 8 y 16 como sus bases respectivamente. El sistema hexadecimal utiliza los números decimales más seis letras adicionales.
3. **Los Enteros** (o simplemente **int**) son uno de los tipos numéricos que soporta Python. Son números que no tienen una parte fraccionaria, por ejemplo, 256, o -1 (enteros negativos).
4. Los números **Punto-Flotante** (o simplemente **flotantes**) son otro tipo numérico que soporta Python. Son números que contienen (o son capaces de contener) una parte fraccionaria, por ejemplo, 1.27.
5. Para codificar un apóstrofe o una comilla dentro de una cadena se puede utilizar el carácter de escape, por ejemplo, 'I'm happy.', o abrir y cerrar la cadena utilizando un conjunto de símbolos distintos al símbolo que se desea codificar, por ejemplo, "I'm happy." para codificar un apóstrofe, y 'Él dijo "Python", no "typhoon"' para codificar comillas.
6. **Los Valores Booleanos** son dos objetos constantes Verdadero y Falso empleados para representar valores de verdad (en contextos numéricos 1 es True, mientras que 0 es False).

Existe un literal especial más utilizado en Python: el literal None. Este literal es llamado un objeto de NonType (ningún tipo), y puede ser utilizado para representar **la ausencia de un valor**. Pronto se hablará más acerca de ello.

### Ejercicio 1

¿Qué tipos de literales son los siguientes dos ejemplos?

"Hola", "007"

Revisar

### Ejercicio 2

¿Qué tipo de literales son los siguientes cuatro ejemplos?

"1.5", 2.0, 528, False

Revisar

### Ejercicio 3

¿Cuál es el valor en decimal del siguiente numero en binario?

## Python como una calculadora

Ahora, se va a mostrar un nuevo lado de la función print(). Ya se sabe que la función es capaz de mostrar los valores de los literales que le son pasados por los argumentos.

De hecho, puede hacer algo más. Observa el siguiente fragmento de código:

```
print(2+2)
```

Reescribe el código en el editor y ejecútalo. ¿Puedes adivinar la salida?

Deberías de ver el número cuatro. Tómate la libertad de experimentar con otros operadores.

Sin tomar esto con mucha seriedad, has descubierto que Python puede ser utilizado como una calculadora. No una muy útil, y definitivamente no una de bolsillo, pero una calculadora sin duda alguna.

Tomando esto más seriamente, nos estamos adentrado en el terreno de los **operadores** y **expresiones**.

## Los Operadores Básicos

Un **operador** es un símbolo del lenguaje de programación, el cual es capaz de realizar operaciones con los valores.

Por ejemplo, como en la aritmética, el signo de + (mas) es un operador el cual es capaz de **sumar** dos numeros, dando el resultado de la suma.

Sin embargo, no todos los operadores de Python son tan simples como el signo de mas, veamos algunos de los operadores disponibles en Python, las reglas que se deben seguir para emplearlos, y como interpretar las reglas que realizan.

Se comenzará con los operadores que están asociados con las operaciones aritméticas más conocidas:

`+, -, *, /, //, %, **`

El orden en el que aparecen no es por casualidad. Hablaremos más de ello cuando se hayan visto todos.

**Recuerda:** Cuando los datos y operadores se unen, forman juntos **expresiones**. La expresión más sencilla es el literal.

## Operadores aritméticos: exponenciación

Un signo de `**` (doble asterisco) es un operador de **exponenciación** (potencia). El argumento a la izquierda es la **base**, el de la derecha, el **exponente**.

Las matemáticas clásicas prefieren una notación con superíndices, como el siguiente:  $2^3$ . Los editores de texto puros no aceptan esa notación, por lo tanto Python utiliza `**` en lugar de la notación matemática, por ejemplo, `2 ** 3`.

Observa los ejemplos en la ventana del editor.

Nota: En los ejemplos, los dobles asteriscos están rodeados de espacios, no es obligatorio hacerlo pero hace que el código sea mas **legible**.

Los ejemplos muestran una característica importante de los **operadores numéricos** de Python.

Ejecuta el código y observa cuidadosamente los resultados que arroja. ¿Puedes observar algo?

**Recuerda:** Es posible formular las siguientes reglas con base en los resultados:

- Cuando **ambos** `**` argumentos son enteros, el resultado es entero también.
- Cuando **al menos un** `**` argumento es flotante, el resultado también es flotante.

Esta es una distinción importante que se debe recordar.

## Operadores aritméticos: multiplicación

Un símbolo de `*` (asterisco) es un operador de **multiplicación**.

Ejecuta el código y revisa si la regla de *entero vs flotante* aún funciona.

```
print(2 * 3) print(2 * 3.) print(2. * 3) print(2. * 3.)
```

## Operadores aritméticos: división

Un símbolo de `/` (diagonal) es un operador de **división**.

El valor después de la diagonal es el **dividendo**, el valor antes de la diagonal es el **divisor**.



Ejecuta el código y analiza los resultados.

```
print(6 / 3) print(6 / 3.) print(6. / 3) print(6. / 3.)
```

Deberías de poder observar que hay una excepción a la regla.

**El resultado producido por el operador de división siempre es flotante**, sin importar si a primera vista el resultado es flotante:  $1 / 2$ , o si parece ser completamente entero:  $2 / 1$ .

¿Esto ocasiona un problema? Sí, en ocasiones se podrá necesitar que el resultado de una división sea entero, no flotante.

Afortunadamente, Python puede ayudar con eso.

## Operadores aritméticos: división entera

Un símbolo de `//` (doble diagonal) es un operador de **división entera**. Difiere del operador estándar `/` en dos detalles:

- El resultado carece de la parte fraccionaria, está ausente (para los enteros), o siempre es igual a cero (para los flotantes); esto significa que **los resultados siempre son redondeados**.
- Se ajusta a la regla *entero vs flotante*.

Ejecuta el ejemplo debajo y observa los resultados:

```
print(6 // 3) print(6 // 3.) print(6. // 3) print(6. // 3.)
```

Como se puede observar, *una división de entero entre entero* da un **resultado entero**. Todos los demás casos producen flotantes.

Hagamos algunas pruebas mas avanzadas.

Observa el siguiente fragmento de código:

```
print(6 // 4) print(6. // 4)
```

Imagina que se utilizó `/` en lugar de `//` - ¿Podrías predecir los resultados?

Sí, sería 1.5 en ambos casos. Eso esta claro.

Pero, ¿Qué resultado se debería esperar con una división `//`?

Ejecuta el código y observa por ti mismo.

Lo que se obtiene son dos unos, uno entero y uno flotante.

El resultado de la división entera siempre se redondea al valor entero inferior mas cercano del resultado de la división no redondeada.

Esto es muy importante: **el redondeo siempre va hacia abajo**.

Observa el código e intenta predecir el resultado nuevamente:

```
print(-6 // 4) print(6. // -4)
```

Nota: Algunos de los valores son negativos. Esto obviamente afectara el resultado. ¿Pero cómo?

El resultado es un par de dos negativos. El resultado real (no redondeado) es -1.5 en ambo casos. Sin embargo, los resultados se redondean. El **redondeo se hace hacia el valor inferior entero**, dicho valor es -2, por lo tanto los resultados son: -2 y -2.0.

La division entera también se le suele llamar en inglés **floor division**. Más adelante te cruzarás con este término.

## Operadores: residuo (módulo)

El siguiente operador es uno muy peculiar, porque no tiene un equivalente dentro de los operadores aritméticos tradicionales.

Su representación gráfica en Python es el símbolo de % (porcentaje), lo cual puede ser un poco confuso.

Piensa en el como una diagonal (operador de división) acompañado por dos pequeños círculos.

El resultado de la operación es el **residuo que queda de la división entera**.

En otras palabras, es el valor que sobra después de dividir un valor entre otro para producir un resultado entero.

Nota: el operador en ocasiones también es denominado **módulo** en otros lenguajes de programación.

Observa el fragmento de código â intenta predecir el resultado y después ejecútalo:

```
print(14 % 4)
```

Como puedes observar, el resultado es dos. Esta es la razón:

- $14 // 4$  da como resultado un 3 → esta es la parte entera, es decir el **cociente**.
- $3 * 4$  da como resultado 12 → como resultado de **la multiplicación entre el cociente y el divisor**.
- $14 - 12$  da como resultado 2 → este es el **residuo**.
- 

El siguiente ejemplo es un poco mas complicado:

```
print(12 % 4.5)
```

¿Cuál es el resultado?

Revisar

## Operadores: como no dividir

Como probablemente sabes, la **división entre cero no funciona**.

No intentes:

- Dividir entre cero.
- Realizar una división entera entre cero.
- Encontrar el residuo de una división entre cero.

## Operadores: suma

El símbolo del operador de **suma** es el + (signo de más), el cual esta completamente alineado a los estándares matemáticos.

De nuevo, observa el siguiente fragmento de código:

```
print(-4 + 4) print(-4. + 8)
```

El resultado no debe de sorprenderte. Ejecuta el código y revisa los resultados.

## El operador de resta, operadores unarios y binarios

El símbolo del operador de **resta** es obviamente - (el signo de menos), sin embargo debes notar que este operador tiene otra función - **puede cambiar el signo de un número**.

Esta es una gran oportunidad para mencionar una distinción muy importante entre operadores **unarios** y **binarios**. En aplicaciones de resta, el **operador de resta espera dos argumentos**: el izquierdo (un **minuendo** en términos aritméticos) y el derecho (un **sustraendo**).

Por esta razón, el operador de resta es considerado uno de los operadores binarios, así como los demás operadores de suma, multiplicación y división.

Pero el operador negativo puede ser utilizado de una forma diferente, observa la ultima línea de código del siguiente fragmento:

```
print(-4 - 4) print(4. - 8) print(-1.1)
```

Por cierto: también hay un operador + unario. Se puede utilizar de la siguiente manera:

```
print(+2)
```

El operador conserva el signo de su único argumento, el de la derecha.

Aunque dicha construcción es sintácticamente correcta, utilizarla no tiene mucho sentido, y sería difícil encontrar una buena razón para hacerlo.

Observa el fragmento de código que está arriba - ¿Puedes adivinar el resultado o salida?

## Operadores y sus prioridades

Hasta ahora, se ha tratado cada operador como si no tuviera relación con los otros. Obviamente, dicha situación tan simple e ideal es muy rara en la programación real.

También, muy seguido encontrarás más de un operador en una expresión, y entonces esta presunción ya no es tan obvia.

Considera la siguiente expresión:

```
2 + 3 * 5
```

Probablemente recordaras de la escuela que las **multiplicaciones preceden a las sumas**.

Seguramente recordaras que primero se debe multiplicar 3 por 5, mantener el 15 en tu memoria y después sumar el 2, dando como resultado el 17.

El fenómeno que causa que algunos operadores actúen antes que otros es conocido como **la jerarquía de prioridades**.

Python define la jerarquía de todos los operadores, y asume que los operadores de mayor jerarquía deben realizar sus operaciones antes que los de menor jerarquía.

Entonces, si se sabe que la `*` tiene una mayor prioridad que la `+`, el resultado final debe de ser obvio.

## Operadores y sus enlaces

El **enlace** de un operador determina el orden en que se computan las operaciones de los operadores con la misma prioridad, los cuales se encuentran dentro de una misma expresión.

La mayoría de los operadores de Python tienen un enlazado hacia la izquierda, lo que significa que el calculo de la expresión es realizado de izquierda a derecha.

Este simple ejemplo te mostrará como funciona. Observa:

```
print(9 % 6 % 2)
```

Existen dos posibles maneras de evaluar la expresión:

- De izquierda a derecha: primero `9 % 6` da como resultado 3, y entonces `3 % 2` da como resultado 1.
- De derecha a izquierda: primero `6 % 2` da como resultado 0, y entonces `9 % 0` causa **un error fatal**.

Ejecuta el ejemplo y observa lo que se obtiene.

El resultado debe ser 1. El operador tiene un **enlazado hacia la izquierda**. Pero hay una excepción interesante.

## Operadores y sus enlaces: exponenciación

Repite el experimento, pero ahora con exponentes.

Utiliza este fragmento de código:

```
print(2 ** 2 ** 3)
```

Los dos posibles resultados son:

- $2 ** 2 \rightarrow 4$ ;  $4 ** 3 \rightarrow 64$
- $2 ** 3 \rightarrow 8$ ;  $2 ** 8 \rightarrow 256$

Ejecuta el código, ¿Qué es lo que observas?

El resultado muestra claramente que **el operador de exponenciación utiliza enlazado hacia la derecha**.

## Lista de prioridades

Como eres nuevo a los operadores de Python, no se presenta por ahora una lista completa de las prioridades de los operadores.

En lugar de ello, se mostrarán solo algunos, y se irán expandiendo conforme se vayan introduciendo operadores nuevos.

Observa la siguiente tabla:

Prioridad	Operador	
1	+, -	unario
2	**	
3	*, /, %	
4	+, -	binario

Nota: se han enumerado los operadores en orden **de la mas alta (1) a la mas baja (4) prioridad**.

Intenta solucionar la siguiente expresión:

```
print(2 * 3 % 5)
```

Ambos operadores (\* y %) tienen la misma prioridad, el resultado solo se puede obtener conociendo el sentido del enlazado. ¿Cuál será el resultado?

Revisar

## Operadores y paréntesis

Por supuesto, se permite hacer uso de **paréntesis**, lo cual cambiará el orden natural del cálculo de la operación.

De acuerdo con las reglas aritméticas, **las sub-expresiones dentro de los paréntesis siempre se calculan primero**.

Se pueden emplear tantos paréntesis como se necesiten, y seguido son utilizados para **mejorar la legibilidad** de una expresión, aun si no cambian el orden de las operaciones.

Un ejemplo de una expresión con múltiples paréntesis es la siguiente:

```
print((5 * ((25 % 13) + 100) / (2 * 13)) // 2)
```

Intenta calcular el valor que se calculará en la consola. ¿Cuál es el resultado de la función print()?

## Puntos Clave

1. Una **expresión** es una combinación de valores (o variables, operadores, llamadas a funciones, aprenderás de ello pronto) las cuales son evaluadas y dan como resultado un valor, por ejemplo, 1+2.
2. Los **operadores** son símbolos especiales o palabras clave que son capaces de operar en los valores y realizar operaciones matemáticas, por ejemplo, el \* multiplica dos valores: x\*y.
3. Los operadores aritméticos en Python: + (suma), - (resta), \* (multiplicación), / (división clásica: regresan un flotante si uno de los valores es de este tipo), % (módulo: divide el operando izquierdo entre el operando derecho y regresa el residuo de la operación, por ejemplo, 5%2=1), \*\* (exponenciación: el operando izquierdo se eleva a la potencia del operando derecho, por ejemplo, 2\*\*3=2\*2\*2=8), // (división entera: retorna el numero resultado de la división, pero redondeado al numero entero inferior más cercano, por ejemplo, 3//2.0=1.0).
4. Un operador **unario** es un operador con solo un operando, por ejemplo, -1, o +3.
5. Un operador **binario** es un operador con dos operados, por ejemplo, 4+5, o 12%5.
6. Algunos operadores actúan antes que otros, a esto se le llama - **jerarquía de prioridades**:
  - Unario + y - tienen la prioridad más alta.
  - Después: \*\*, después: \*, /, y %, y después la prioridad más baja: binaria + y -.
7. Las sub-expresiones dentro de **paréntesis** siempre se calculan primero, por ejemplo, 15-1\*(5\*(1+2))=0.

8. Los operadores de **exponenciación** utilizan **enlazado hacia la derecha**, por ejemplo,  $2^{2^{2^3}}=256$ .

### Ejercicio 1

¿Cuál es la salida del siguiente fragmento de código?

```
print((2**4), (2*4.), (2*4))  
Revisar
```

### Ejercicio 2

¿Cuál es la salida del siguiente fragmento de código?

```
print((-2/4), (2/4), (2//4), (-2//4))  
Revisar
```

### Ejercicio 3

¿Cuál es la salida del siguiente fragmento de código?

```
print((2%-4), (2%4), (2**3**2))
```

## ¿Qué son las Variables?

Es justo que Python nos permita codificar literales, las cuales contengan valores numéricos y cadenas.

Ya hemos visto que se pueden hacer operaciones aritméticas con estos números: sumar, restar, etc. Esto se hará una infinidad de veces en un programa.

Pero es normal preguntar como es que se pueden **almacenar los resultados** de estas operaciones, para poder emplearlos en otras operaciones, y así sucesivamente.

¿Cómo almacenar los resultados intermedios, y después utilizarlos de nuevo para producir resultados subsecuentes?

Python ayudará con ello. Python ofrece "cajas" (contenedores) especiales para este propósito, estas cajas son llamadas **variables** - el nombre mismo sugiere que el contenido de estos contenedores puede variar en casi cualquier forma.

¿Cuáles son los componentes o elementos de una variable en Python?

- Un nombre.
- Un valor (el contenido del contenedor).

Comencemos con lo relacionado al nombre de la variable.

Las variables no aparecen en un programa automáticamente. Como desarrollador, tu debes decidir cuantas variables deseas utilizar en tu programa.

También las debes de nombrar.

Si se desea **nombrar una variable**, se deben seguir las siguientes reglas:

- El nombre de la variable debe de estar compuesto por MAYUSCULAS, minúsculas, dígitos, y el carácter \_ (guion bajo).
- El nombre de la variable debe comenzar con una letra.
- El carácter guion bajo es considerado una letra.
- Las mayúsculas y minúsculas se tratan de forma distinta (un poco diferente que en el mundo real - *Ali-cia* y *ALICIA* son el mismo nombre, pero en Python son dos nombres de variable distintos, subsecuentemente, son dos variables diferentes).
- El nombre de las variables no pueden ser igual a alguna de las palabras reservadas de Python (se explicará más de esto pronto).



## Nombres correctos e incorrectos de variables

Nota que la misma restricción aplica a los nombres de funciones.

Python no impone restricciones en la longitud de los nombres de las variables, pero eso no significa que un nombre de variable largo sea mejor que uno corto.

Aquí se muestran algunos nombres de variable que son correctos, pero que no siempre son convenientes:

MiVariable, i, t34, Tasa\_Cambio, contador, DiasParaNavidad, ElNombreEsTanLargoQueSeCometeranErroresConEl, \_.

Además, Python permite utilizar no solo las letras latinas, sino caracteres específicos de otros idiomas que utilizan otros alfabetos.

Estos nombres de variables también son correctos:

Adiós\_Señora, sûr\_la\_mer, Einbahnstraße, переменная.

Ahora veamos algunos **nombres incorrectos**:



10t (no comienza con una letra), Tasa Cambio (contiene un espacio).

## Palabras Clave

Observa las palabras que juegan un papel muy importante en cada programa de Python.

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

Son llamadas **palabras clave** o (mejor dicho) **palabras reservadas**. Son reservadas porque **no se deben utilizar como nombres**: ni para variables, ni para funciones, ni para cualquier otra cosa que se desee crear.

El significado de la palabra reservada está **predefinido**, y no debe cambiar.

Afortunadamente, debido al hecho de que Python es sensible a mayúsculas y minúsculas, cualquiera de estas palabras se pueden modificar cambiando una o varias letras de mayúsculas a minúsculas o viceversa, creando una nueva palabra, la cual no esta reservada.

Por ejemplo - **no se puede nombrar** a la variable así:

```
import
```

No se puede tener una variable con ese nombre, esta prohibido, pero se puede hacer lo siguiente:

```
Import
```

Estas palabras podrían parecer un misterio ahorita, pero pronto se aprenderá acerca de su significado.

## Creando variables

¿Qué se puede poner dentro de una variable?

Cualquier cosa.

Se puede utilizar una variable para almacenar cualquier tipo de los valores que ya se han mencionado, y muchos mas de los cuales aun no se han explicado.

El valor de la variable en lo que se ha puesto dentro de ella. Puede variar tanto como se necesite o requiera. El valor puede ser entero, después flotante, y eventualmente ser una cadena.

Hablemos de dos cosas importantes - **como son creadas las variables**, y **como poner valores dentro de ellas** (o mejor dicho, como dar o **pasarles valores**).

**Una variable se crea cuando se le asigna un valor.** A diferencia de otros lenguajes de programación, no es necesario declararla.

Si se le asigna cualquier valor a una variable no existente, la variable será **automáticamente creada**. No se necesita hacer algo más.

La creación (o su sintaxis) es muy simple: **solo utiliza el nombre de la variable deseada, después el signo de igual (=) y el valor que se desea colocar dentro de la variable.**



Observa el siguiente fragmento de código:

```
var = 1 print(var)
```

Consiste de dos simples instrucciones:

- La primera crea una variable llamada var, y le asigna un literal con un valor entero de 1.
- La segunda imprime el valor de la variable recientemente creada en la consola.

Nota: print() tiene una función más â puede manejar variables también. ¿Puedes predecir cual será la salida (resultado) del código?

## Utilizando variables

Se tiene permitido utilizar cuantas declaraciones de variables sean necesarias para lograr el objetivo del programa, por ejemplo:

```
var = 1 balance_cuenta = 1000.0 nombreCliente = 'John Doe' print(var, balance_cuenta, nombreCliente) print(var)
```

Sin embargo, no se permite utilizar una variable que no exista, (en otras palabras, una variable a la cual no se le a dado un valor).

Este ejemplo **ocasionara un error**:

```
var = 1 print(Var)
```

Se ha tratado de utilizar la variable llamada Var, la cual no tiene ningún valor (nota: var y Var son entidades diferentes, y no tienen nada en común dentro de Python).

Se puede utilizar `print()` para combinar texto con variables utilizando el operador `+` para mostrar cadenas con variables, por ejemplo:

```
var = "3.7.1" print("Versión de Python: " + var)
```

¿Puedes predecir la salida del fragmento de código?

## Asignar un valor nuevo a una variable ya existente

¿Cómo se le asigna un valor nuevo a una variable que ya ha sido creada? De la misma manera. Solo se necesita el signo de igual.

El signo de igual es de hecho un **operador de asignación**. Aunque esto suene un poco extraño, el operador tiene una sintaxis simple y una interpretación clara y precisa.

Asigna el valor del argumento de la derecha al de la izquierda, aún cuando el argumento de la derecha sea una expresión arbitraria compleja que involucre literales, operadores y variables definidas anteriormente.

Observa el siguiente código:

```
var = 1 print(var) var = var + 1 print(var)
```

El código envía dos líneas a la consola:

```
1 2
```

La primer línea del código **crea una nueva variable** llamada `var` y le asigna el valor de 1.

La declaración se lee de la siguiente manera: asigna el valor de 1 a una variable llamada `var`.

De manera mas corta: asigna 1 a `var`.

Algunos prefieren leer el código así: `var` se convierte en 1.

La tercera línea **le asigna a la misma variable un nuevo valor** tomado de la variable misma, sumándole 1. Al ver algo así, un matemático probablemente protestaría, ningún valor puede ser igualado a si mismo mas uno. Esto es una contradicción. Pero Python trata el signo `=` no como *igual a*, sino como *asigna un valor*.

Entonces, ¿Cómo se lee esto en un programa?

Toma el valor actual de la variable `var`, sumale 1 y guárdalo en la variable `var`.

En efecto, el valor de la variable `var` ha sido **incrementado** por uno, lo cual no está relacionado con comparar la variable con otro valor.

¿Puedes predecir cuál será el resultado del siguiente fragmento de código?

```
var = 100 var = 200 + 300 print(var)
```

## Resolviendo problemas matemáticos simples

Ahora deberías de ser capaz de construir un corto programa el cual resuelva problemas matemáticos sencillos como el Teorema de Pitágoras:

*El cuadrado de la hipotenusa es igual a la suma de los cuadrados de los dos catetos.*

El siguiente código evalúa la longitud de la hipotenusa (es decir, el lado más largo de un triángulo rectángulo, el opuesto al ángulo recto) utilizando el Teorema de Pitágoras:

```
a = 3.0 b = 4.0 c = (a ** 2 + b ** 2) ** 0.5 print("c =", c)
```

Nota: se necesita hacer uso del operador `**` para evaluar la raíz cuadrada:

$$\sqrt{x} = x^{(1/2)}$$

y

$$c = \sqrt{a^2 + b^2}$$

¿Puedes predecir la salida del código?

Revisa abajo y ejecuta el código en el editor para confirmar tus predicciones.

## Tiempo Estimado

10 minutos

## Nivel de dificultad

Fácil

## Objetivos

- Familiarizarse con el concepto de almacenar y trabajar con diferentes tipos de datos en Python.
- Experimentar con el código en Python.

## Escenario

A continuación una historia:

Érase una vez en la Tierra de las Manzanas, Juan tenía tres manzanas, María tenía cinco manzanas, y Adán tenía seis manzanas. Todos eran muy felices y vivieron por muchísimo tiempo. Fin de la Historia.

Tu tarea es:

- Crear las variables: `juan`, `maria`, y `adan`.
- Asignar valores a las variables. El valor debe de ser igual al numero de manzanas que cada quien tenía.
- Una vez almacenados los números en las variables, imprimir las variables en una línea, y separar cada una de ellas con una coma.
- Después se debe crear una nueva variable llamada `totalManzanas` y se debe igualar a la suma de las tres variables anteriores.
- Imprime el valor almacenado en `totalManzanas` en la consola.
- **Experimenta con tu código:** crea nuevas variables, asigna diferentes valores a ellas, y realiza varias operaciones aritméticas con ellas (por ejemplo, `+`, `-`, `*`, `/`, `//`, etc.). Intenta poner una cadena con un entero juntos en la misma línea, por ejemplo, "Numero Total de Manzanas:" y `totalManzanas`.

## Operadores Abreviados

Es tiempo de explicar el siguiente conjunto de operadores que harán la vida del programador/desarrollador mas fácil.

Muy seguido, se desea utilizar la misma variable al lado derecho y al lado izquierdo del operador `=`.

Por ejemplo, si se necesita calcular una serie de valores sucesivos de la potencia de 2, se puede usar el siguiente código:

```
x = x * 2
```

También, puedes utilizar una expresión como la siguiente si no puedes dormir y estas tratando de resolverlo con alguno de los métodos tradicionales:

```
oveja = oveja + 1
```

Python ofrece una manera mas corta de escribir operaciones como estas, lo cual se puede codificar de la siguiente manera:

```
x *= 2 oveja += 1
```

A continuación se intenta presentar una descripción general para este tipo de operaciones.

Si op es un operador de dos argumentos (esta es una condición muy importante) y el operador es utilizado en el siguiente contexto:

```
variable = variable op expresión
```

Puede ser simplificado de la siguiente manera:

```
variable op= expresión
```

Observa los siguientes ejemplos. Asegúrate de entenderlos todos.

```
i = i + 2 * j    i += 2 * j
```

```
var = var / 2    var /= 2
```

```
rem = rem % 10    rem %= 10
```

```
j = j - (i + var + rem)    j -= (i + var + rem)
```

```
x = x ** 2    x **= 2
```

# Tiempo estimado

10 minutos

# Nivel de dificultad

Fácil

## Objetivos

- Familiarizarse con el concepto de variables y trabajar con ellas.
- Realizar operaciones básicas y conversiones.
- Experimentar con el código de Python.

## Escenario

Millas y kilómetros son unidades de longitud o distancia.

Teniendo en mente que 1 equivale aproximadamente a 1.61 kilómetros, complementa el programa en el editor para que convierta de:

- Millas a kilómetros.
- Kilómetros a millas.

No se debe cambiar el código existente. Escribe tu código en los lugares indicados con `###`. Prueba tu programa con los datos que han sido provistos en el código fuente.

---

Pon mucha atención a lo que está ocurriendo dentro de la función `print()`. Analiza como es que se proveen múltiples argumentos para la función, y como es que se muestra el resultado.

Nota que algunos de los argumentos dentro de la función `print()` son cadenas (por ejemplo "millas son", y otros son variables (por ejemplo `millas`).

Hay una cosa interesante mas que está ocurriendo. ¿Puedes ver otra función dentro de la función `print()` ? Es la función `round()`. Su trabajo es redondear la salida del resultado al numero de decimales especificados en el paréntesis, y regresar un valor flotante (dentro de la función `round()` se puede encontrar el nombre de la variable, el nombre, una coma, y el numero de decimales que se desean mostrar). Se hablará mas de esta función muy pronto, no te preocupes si no todo queda muy claro. Solo se quiere impulsar tu curiosidad.

---

Después de completar el laboratorio , abre Sandbox (el arenero), y experimenta más. Intenta escribir diferentes convertidores, por ejemplo, un convertidor de USD a EUR, un convertidor de temperatura, etc. â ¡deja que tu imaginación vuele! Intenta mostrar los resultados combinando cadenas y variables. Intenta utilizar y experimentar con la función `round()` para redondear tus resultados a uno, dos o tres decimales. Revisa que es lo que sucede si no se provee un dígito al redondear. Recuerda probar tus programas.

Experimenta, saca tus propias conclusiones, y aprende. Se curioso.

# Resultado Esperado

7.38 millas son 11.88 kilómetros 12.25 kilómetros son 7.61 millas



## Tiempo Estimado

10-15 minutos

## Nivel de Dificultad

Fácil

## Objetivos

- Familiarizarse con los conceptos de números, operadores y operaciones aritméticas en Python.
- Realizar cálculos básicos.

## Escenario

Observa el código en el editor: lee un valor flotante, lo coloca en una variable llamada `x`, e imprime el valor de la variable llamada `y`. Tu tarea es completar el código para evaluar la siguiente expresión:

$$3x^3 - 2x^2 + 3x - 1$$

El resultado debe ser asignado a `y`.

Recuerda que la notación algebraica clásica muy seguido omite el operador de multiplicación, aquí se debe de incluir de manera explícita. Nota como se cambia el tipo de dato para asegurarnos de que `x` es del tipo flotante.

Mantén tu código limpio y legible, y pruébalo utilizando los datos que han sido proporcionados. No te desanimes por no lograrlo en el primer intento. Se persistente y curioso.

## Prueba de Datos

Datos de Muestra

`x = 0` `x = 1` `x = -1`

Salida Esperada

`y = -1.0` `y = 3.0` `y = -9.0`

## Puntos Clave

1. Una **variable** es una ubicación nombrada reservada para almacenar valores en la memoria. Una variable es creada o inicializada automáticamente cuando se le asigna un valor por primera vez.
2. Cada variable debe de tener un nombre único - un **identificador**. Un nombre valido debe ser aquel que no contiene espacios, debe comenzar con un guion bajo (`_`), o una letra, y no puede ser una palabra reservada de Python. El primer carácter puede estar seguido de guiones bajos, letras, y dígitos. Las variables en Python son sensibles a mayúsculas y minúsculas.

3. Python es un lenguaje **de tipo dinámico**, lo que significa que no se necesita *declarar* variables en él. Para asignar valores a las variables, se utiliza simplemente el operador de asignación, es decir el signo de igual (=) por ejemplo, `var = 1`.

4. También es posible utilizar **operadores de asignación compuesta** (operadores abreviados) para modificar los valores asignados a las variables, por ejemplo, `var += 1`, or `var /= 5 * 2`.

5. Se les puede asignar valores nuevos a variables ya existentes utilizando el operador de asignación o un operador abreviado:

```
var = 2 print(var) var = 3 print(var) var += 1 print(var)
```

6. Se puede combinar texto con variables empleado el operador +, y utilizar la función `print()` para mostrar o imprimir los resultados, por ejemplo:

```
var = "007" print("Agente " + var)
```

### Ejercicio 1

¿Cuál es el resultado del siguiente fragmento de código?

```
var = 2 var = 3 print(var)
```

Revisar

### Ejercicio 2

¿Cuáles de los siguientes nombres de variables son ilegales en Python?

```
my_var m 101 averylongvariablename m101 m 101 Del del
```

Revisar

### Ejercicio 3

¿Cuál es el resultado del siguiente fragmento de código?

```
a = '1' b = "1" print(a + b)
```

Revisar

### Ejercicio 4

¿Cuál es el resultado del siguiente fragmento de código?

```
a = 6 b = 3 a /= 2 * b print(a)
```

## Poner comentarios en el código: ¿por qué, cuándo y dónde?

Quizá en algún momento será necesario poner algunas palabras en el código dirigidas no a Python, sino a las personas quienes estén leyendo el código con el fin de explicarles como es que funciona, o tal vez especificar el significado de las variables, también para documentar quien es el autor del programa y en que fecha fue escrito.

Un texto insertado en el programa el cual es, **omitido en la ejecución**, es denominado un **comentario**.

¿Cómo se colocan este tipo de comentarios en el código fuente? Tiene que ser hecho de cierta manera para que Python no intente interpretarlo como parte del código.

Cuando Python se encuentra con un comentario en el programa, el comentario es completamente transparente, desde el punto de vista de Python, el comentario es solo un espacio vacío, sin importar que tan largo sea.

En Python, un comentario es un texto que comienza con el símbolo # y se extiende hasta el final de la línea.

Si se desea colocar un comentario que abarca varias líneas, se debe colocar este símbolo en cada línea.

Justo como el siguiente código:

```
# Esta programa calcula la hipotenusa (c) # a y b son las longitudes de los catetos a = 3.0 b = 4.0 c = (a ** 2 + b  
** 2) ** 0.5 # se utiliza ** en lugar de la raíz cuadrada print("c =", c)
```

Los desarrolladores buenos y responsables **describen cada pieza importante de código**, por ejemplo, el explicar el rol de una variable; aunque la mejor manera de comentar una variable es dándole un nombre que no sea ambiguo.

Por ejemplo, si una variable determinada esta diseñada para almacenar el área de un cuadrado, el nombre areaCuadrado será muchísimo mejor que tiaJuana.

El primer nombre dado a la variable se puede definir como **auto-comentable**.

Los comentarios pueden ser útiles en otro aspecto, se pueden utilizar para **marcar un fragmento de código que actualmente no se necesita**, cual sea la razón. Observa el siguiente ejemplo, si se **descomenta** la línea resaltada, esto afectara la salida o resultado del código:

```
# Este es un programa de prueba x = 1 y = 2 # y = y + x print(x + y)
```

Esto es frecuentemente realizado cuando se esta probando un programa, con el fin de aislar un fragmento de código donde posiblemente se encuentra un error.

# Tiempo Estimado

5 minutos

# Nivel de Dificultad

Muy Fácil

## Objetivos

- Familiarizarse con el concepto de comentarios en Python.
- Utilizar y no utilizar los comentarios.
- Reemplazar los comentarios con código.
- Experimentar con el código de Python.

## Escenario

El código en el editor contiene comentarios. Intenta mejorarlo: agrega o quita comentarios donde consideres que sea apropiado (en ocasiones el remover un comentario lo hace mas legible), además, cambia el nombre de las variables donde consideres que esto mejorará la comprensión del código.

Los comentarios son muy importantes. No solo hacen que el programa sea **más fácil de entender**, pero también sirven para **deshabilitar aquellas partes de código que no son necesarias** (por ejemplo, cuando se necesita probar cierta parte del código, e ignorar el resto). Los buenos programadores **describen** cada parte importante del código, y dan **nombres significativos** a variables, debido a que en ocasiones es mucho más sencillo dejar el comentario dentro del código mismo.

Es bueno utilizar nombres de variables **legibles**, y en ocasiones es mejor **dividir el código** en partes con nombres (por ejemplo en funciones). En algunas situaciones, es una buena idea escribir los pasos de como se realizaron los cálculos de una forma sencilla y clara.

Una cosa mas: puede ocurrir que un comentario contenga una pieza de información incorrecta o errónea, nunca se debe de hacer eso a propósito.

## Puntos Clave

1. Los comentarios pueden ser utilizados para colocar información adicional en el código. Son omitidos al momento de la ejecución. Dicha información es para los lectores que están manipulando el código. En Python, un comentario es un fragmento de texto que comienza con un #. El comentario se extiende hasta el final de la línea.

2. Si deseas colocar un comentario que abarque varias líneas, es necesario colocar un `#` al inicio de cada línea.

Además, se puede utilizar un comentario para marcar un fragmento de código que no es necesaria en el momento y no se desea ejecutar. (observa la ultima línea de código del siguiente fragmento), por ejemplo:

```
# Este programa imprime # un saludo en pantalla print("Hola!") # Se invoca la función print() function #  
print("Soy Python.")
```

3. Cuando sea posible, se deben **auto comentar los nombres** de las variables, por ejemplo, si se están utilizando dos variables para almacenar la altura y longitud de algo, los nombres altura y longitud son una mejor elección que `mivar1` y `mivar2`.

4. Es importante utilizar los comentarios para que los programas sean más fáciles de entender, además de emplear variables legibles y significativas en el código. Sin embargo, es igualmente importante **no utilizar** nombres de variables que sean confusos, o dejar comentarios que contengan información incorrecta.

5. Los comentarios pueden ser muy útiles cuando *tu* estas leyendo tu propio código después de un tiempo (es común que los desarrolladores olviden lo que su propio código hace), y cuando *otros* están leyendo tu código (les puede ayudar a comprender que es lo que hacen tus programas y como es que lo hacen).

### Ejercicio 1

¿Cuál es la salida del siguiente fragmento de código?

```
# print("Cadena #1") print("Cadena #2")  
Revisar
```

### Ejercicio 2

¿Qué ocurrirá cuando se ejecute el siguiente código?

```
# Esto es un comentario en varias líneas # print("Hola!")
```

## La función `input()`

Ahora se introducirá una nueva función, la cual pareciese ser un reflejo de la función `print()`.

¿Por que? Bueno, `print()` envía datos a la consola.

Esta nueva función obtiene datos de ella.

`print()` no tiene un resultado utilizable. La importancia de esta nueva función es que **regresa un valor muy utilizable**.

La función se llama `input()`. El nombre de la función lo dice todo.

La función `input()` es capaz de leer datos que fueron introducidos por el usuario y pasar esos datos al programa en ejecución.

El programa entonces puede manipular los datos, haciendo que el código sea verdaderamente interactivo.

Todos los programas **leen y procesan datos**. Un programa que no obtiene datos de entrada del usuario es un **programa sordo**.

Observa el ejemplo:

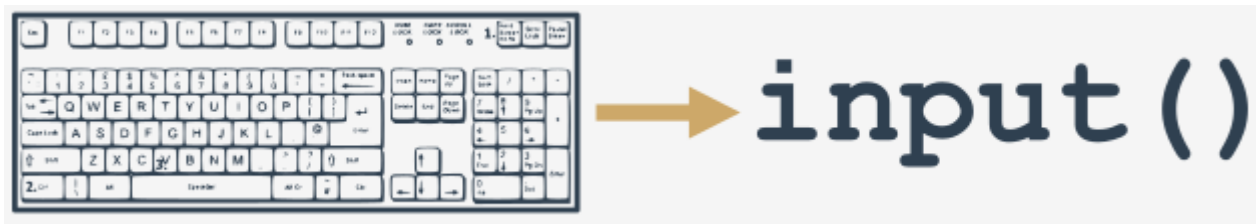
```
print("Dime algo...") algo = input() print("Mmm...", algo, "...¿en serio?")
```

Se muestra un ejemplo muy sencillo de como utilizar la función `input()`.

Nota:

- El programa **solicita al usuario que inserte algún dato** desde la consola (seguramente utilizando el teclado, aunque también es posible introducir datos utilizando la voz o alguna imagen).
- La función `input()` es invocada sin argumentos (es la manera mas sencilla de utilizar la función); la función **pondrá la consola en modo de entrada**; aparecerá un cursor que parpadea, y podrás introducir datos con el teclado, al terminar presiona la tecla *Enter*; todos los datos introducidos serán **enviados al programa** a través del resultado de la función.
- Nota: el resultado debe ser asignado a una variable; esto es crucial, si no se hace los datos introducidos se perderán.
- Después se utiliza la función `print()` para mostrar los datos que se obtuvieron, con algunas observaciones adicionales.

Intenta ejecutar el código y permite que la función te muestre lo que puede hacer.



## Más acerca de la función `input()` y tipos de conversión

El tener un equipo compuesto por `input()`-`int()`-`float()` abre muchas nuevas posibilidades.

Eventualmente serás capaz de escribir programas completos, los cuales acepten datos en forma de números, los cuales serán procesados y se mostrarán los resultados.

Por supuesto, estos programas serán muy primitivos y no muy utilizables, debido a que no pueden tomar decisiones, y consecuentemente no son capaces de reaccionar acorde a cada situación.

Sin embargo, esto no es un problema; se explicará como solucionarlo pronto.

---

El siguiente ejemplo hace referencia al programa anterior que calcula la longitud de la hipotenusa. Vamos a reescribirlo, para que pueda leer las longitudes de los catetos desde la consola.

Revisa la ventana del editor, así es como se ve ahora.

Este programa le preguntó al usuario los dos catetos, calcula la hipotenusa e imprime el resultado.

Ejecútalo de nuevo e intenta introducir valores negativos.

El programa desafortunadamente, no reacciona correctamente a este error.

Vamos a ignorar esto por ahora. Regresaremos a ello pronto.

---

Debido a que la función `print()` acepta una expresión como argumento, se puede **quitar la variable** del código.

Como se muestra en el siguiente código:

```
cateto_a = float(input("Inserta la longitud del primer cateto: "))
cateto_b = float(input("Inserta la longitud del segundo cateto: "))
print("La longitud de la hipotenusa es: ", (cateto_a**2 + cateto_b**2) **.5)
```

## Operadores de cadenas - introducción

Es tiempo de regresar a estos dos operadores aritméticos: `+` y `*`.

Ambos tienen una función secundaria. Son capaces de hacer algo más que **sumar y multiplicar**.

Los hemos visto en acción cuando sus argumentos son (flotantes o enteros).

Ahora veremos que son capaces también de manejar o manipular cadenas, aunque, en una manera muy específica.

## Concatenación

El signo de `+` (más), al ser aplicado a dos cadenas, se convierte en **un operador de concatenación**:

`string + string`

Simplemente **concatena** (junta) dos cadenas en una. Además, puede ser utilizado más de una vez en una misma expresión.

En contraste con el operador aritmético, el operador de concatenación no es **conmutativo**, por ejemplo, `"ab" + "ba"` no es lo mismo que `"ba" + "ab"`.

No olvides, si se desea que el signo `+` sea un **concatenador**, no un sumador, solo se debe asegurar que **ambos argumentos sean cadenas**.

No se pueden mezclar los tipos de datos aquí.

---

Este es un programa sencillo que muestra como funciona el signo `+` como concatenador:

```
nom = input("¿Me puedes dar tu nombre por favor? ")
ape = input("¿Me puedes dar tu apellido por favor? ")
print("Gracias.")
print("\nTu nombre es " + nom + " " + ape + ".")
```

Nota: El utilizar `+` para concatenar cadenas te permite construir la salida de una manera más precisa, en comparación de utilizar únicamente la función `print()`, aún cuando se enriquezca con los argumentos `end=` y `sep=`.

Ejecuta el código y comprueba si la salida es igual a tus predicciones.

## Replicación

El signo de `*` (asterisco), cuando es aplicado a una cadena y a un número (o a un número y cadena) se convierte en **un operador de replicación**.

`cadena * número`   `número * cadena`

Replica la cadena el numero de veces indicado por el número.

Por ejemplo:

- "James" \* 3 nos da "JamesJamesJames".
- 3 \* "an" nos da "ananan".
- 5 \* "2" (o "2" \* 5) da como resultado "22222" (no 10).

Un número menor o igual que cero produce una **cadena vacía**.

Este sencillo programa "dibuja" un rectángulo, haciendo uso del operador (+), pero en un nuevo rol:

```
print("+" + 10 * "-" + "+") print(("|" + " " * 10 + "\\n") * 5, end="") print("+" + 10 * "-" + "+")
```

Nota como se ha utilizado el paréntesis en la segunda línea de código.

¡Intenta practicar para crear otras figuras o tus propias obras de arte!

- 2.1.6.7 Cómo hablar con una computadora: operadores de cadenas



## Replicación

El signo de \* (asterisco), cuando es aplicado a una cadena y a un número (o a un número y cadena) se convierte en un **operador de replicación**.

`cadena * número número * cadena`

Replica la cadena el numero de veces indicado por el número.

Por ejemplo:

- `"James" * 3` nos da `"JamesJamesJames"`.
- `3 * "an"` nos da `"ananan"`.
- `5 * "2"` (o `"2" * 5`) da como resultado `"22222"` (no 10).

Un número menor o igual que cero produce una **cadena vacía**.

Este sencillo programa "dibuja" un rectángulo, haciendo uso del operador (+), pero en un nuevo rol:

```
print("+" + 10 * "-" + "+") print(("|" + " " * 10 + "\n") * 5, end="") print("+" + 10 * "-" + "+")
```

Nota como se ha utilizado el paréntesis en la segunda línea de código.

¡Intenta practicar para crear otras figuras o tus propias obras de arte!

## Conversión de tipos de datos: `str()`

A estas alturas ya sabes como emplear las funciones `int()` y `float()` para convertir una cadena a un número.

Este tipo de conversión no es en un solo sentido. También se puede **convertir un numero a una cadena**, lo cual es más fácil y rápido, esta operación es posible hacerla siempre.

Una función capaz de hacer esto se llama `str()`:

```
str(número)
```

Sinceramente, puede hacer mucho más que transformar números en cadenas, eso lo veremos después.

## El "triángulo rectángulo" de nuevo

Este es el programa del "triángulo rectángulo" visto anteriormente:

```
cateto_a = float(input("Ingresa la longitud del primer cateto: ")) cateto_b = float(input("Ingresa la longitud del segundo cateto: ")) print("La longitud de la hipotenusa es: " + str((cateto_a**2 + cateto_b**2) **.5))
```

Se ha modificado un poco para mostrar cómo es que la función `str()` trabaja. Gracias a esto, podemos **pasar el resultado entero a la función `print()` como una sola cadena**, sin utilizar las comas.

Has hecho algunos pasos importantes en tu camino hacia la programación de Python.

Ya conoces los tipos de datos básicos y un conjunto de operadores fundamentales. Sabes cómo organizar la salida y cómo obtener datos del usuario. Estos son fundamentos muy sólidos para el Módulo 3. Pero antes de pasar al siguiente módulo, hagamos unos cuantos laboratorios y resumamos todo lo que has aprendido en esta sección.

### 2.1.6.10 LABORATORIO: Operadores y expresiones

## Tiempo Estimado

5-10 minutos

## Nivel de Dificultad

Fácil

## Objetivos

- Familiarizarse con la entrada y salida de datos en Python.
- Evaluar expresiones simples.

## Escenario

La tarea es completar el código para evaluar y mostrar el resultado de cuatro operaciones aritméticas básicas. El resultado debe ser mostrado en consola.

Quizá no podrás proteger el código de un usuario que intente dividir entre cero. Por ahora, no hay que preocuparse por ello.

Prueba tu código - ¿Produce los resultados esperados?

- 2.1.6.10 LABORATORIO: Operadores y expresiones

## Tiempo estimado

20 minutos

## Nivel de dificultad

Intermedio

## Objetivos

Familiarizarse con los conceptos de números, operadores y expresiones aritméticas en Python.

Comprender la precedencia y asociatividad de los operadores de Python, así como el correcto uso de los paréntesis.

## Escenario

La tarea es completar el código para poder evaluar la siguiente expresión:

$$\frac{1}{x + \frac{1}{x + \frac{1}{x + \frac{1}{x}}}}$$

El resultado debe de ser asignado a y. Se cauteloso, observa los operadores y priorízalos. Utiliza cuantos paréntesis sean necesarios.

Puedes utilizar variables adicionales para acortar la expresión (sin embargo, no es muy necesario). Prueba tu código cuidadosamente.

## Datos de Prueba

Entrada de muestra: 1

Salida esperada:

y = 0.6000000000000001

Entrada de muestra: 10

Salida esperada:

y = 0.09901951266867294

Entrada de muestra: 100

Salida esperada:

y = 0.009999000199950014

Entrada de muestra: -5

Salida esperada:

y = -0.19258202567760344

- 2.1.6.11 LABORATORIO: Operadores y expresiones

## Tiempo estimado

15-20 minutos

## Nivel de dificultad

Fácil

## Objetivos

- Mejorar la habilidad de implementar números, operadores y operaciones aritméticas en Python.
- Utilizar la función `print()` y sus capacidades de formateo.
- Aprender a expresar fenómenos del día a día en términos de un lenguaje de programación.

## Escenario

La tarea es preparar un código simple para evaluar o encontrar el **tiempo final** de un periodo de tiempo dado, expresándolo en horas y minutos. Las horas van de 0 a 23 y los minutos de 0 a 59. El resultado debe ser mostrado en la consola.

Por ejemplo, si el evento comienza a las **12:17** y dura **59 minutos**, terminará a las **13:16**.

No te preocupes si tu código no es perfecto, está bien si acepta una hora inválida, lo más importante es que el código produzca una salida correcta acorde a la entrada dada.

Prueba el código cuidadosamente. Pista: utilizar el operador `%` puede ser clave para el éxito.

## Datos de Prueba

Entrada de muestra: 12 17 59

Salida esperada: 13:16

Entrada de muestra: 23 58 642

Salida esperada: 10:40

Entrada de muestra: 0 1 2939

Salida esperada: 1:0

## Puntos Clave

1. La función `print()` **envía datos a la consola**, mientras que la función `input()` **obtiene datos de la consola**.
2. La función `input()` viene con un parámetro inicial: **un mensaje de tipo cadena para el usuario**. Permite escribir un mensaje antes de la entrada del usuario, por ejemplo:  

```
nombre = input("Ingresa tu nombre: ") print("Hola, " + nombre + ". ¡Un gusto conocerte!")
```
3. Cuando la función `input()` es llamada o invocada, el flujo del programa se detiene, el símbolo del cursor se mantiene parpadeando (le está indicando al usuario que tome acción ya que la consola está en modo de entrada) hasta que el usuario haya ingresado un dato y/o haya presionado la tecla *Enter*.

Puedes probar la funcionalidad completa de la función `input()` localmente en tu máquina. Por razones de optimización, se ha limitado el máximo número de ejecuciones en Edube a solo algunos segundos únicamente. Ve a Sandbox, copia y pega el código que está arriba, ejecuta el programa y espera unos segundos. Tu programa debe detenerse después de unos segundos. Ahora abre IDLE, y ejecuta el mismo programa ahí -¿Puedes notar alguna diferencia?

Consejo: La característica mencionada anteriormente de la función `input()` puede ser utilizada para pedirle al usuario que termine o finalice el programa. Observa el siguiente código:

```
nombre = input("Ingresa tu nombre: ") print("Hola, " + nombre + ". ¡Un gusto conocerte!") print("\nPresiona la tecla Enter para finalizar el programa.") input() print("FIN.")
```

3. El resultado de la función `input()` es una cadena. Se pueden unir cadenas unas con otras a través del operador de concatenación (+). Observa el siguiente código:

```
num1 = input("Ingresa el primer número: ") # Ingresa 12 num2 = input("Ingresa el segundo número: ") # Ingresa 21 print(num1 + num2) # el programa regresa 1221
```

4. También se pueden multiplicar (\* - replicación) cadenas, por ejemplo:

```
miEntrada = ("Ingresa Algo: ") # Ejemplo: hola print(miEntrada * 3) # Salida esperada: holaholahola
```

### Ejercicio 1

¿Cuál es la salida del siguiente código?

```
x = int(input("Ingresa un número: ")) # el usuario ingresa un 2 print(x * "5")  
Revisar
```

### Ejercicio 2

¿Cuál es la salida esperada del siguiente código?

```
x = input("Ingresa un número: ") # el usuario ingresa un 2 print(type(x))
```

## Datos de Prueba

Entrada de muestra: 1

Salida esperada:

$y = 0.6000000000000001$

Entrada de muestra: 10

Salida esperada:

$y = 0.09901951266867294$

Entrada de muestra: 100

Salida esperada:

$y = 0.009999000199950014$

Entrada de muestra: -5

Salida esperada:

$y = -0.19258202567760344$

## Módulo 3

Valores booleanos, ejecución condicional, bucles, listas y procesamiento de listas, operaciones lógicas y bit a bit.

# Preguntas y respuestas

Un programador escribe un programa y **el programa hace preguntas**.

Una computadora ejecuta el programa y **proporciona las respuestas**. El programa debe ser capaz de **reaccionar de acuerdo con las respuestas recibidas**.

Afortunadamente, las computadoras solo conocen dos tipos de respuestas:

- Sí, es cierto.
- No, esto es falso.

Nunca obtendrás una respuesta como *Déjame pensar ..., no lo sé, o probablemente sí, pero no lo sé con seguridad*.

**Para hacer preguntas, Python utiliza un conjunto de operadores muy especiales.** Revisemos uno tras otro, ilustrando sus efectos en algunos ejemplos simples.

## Comparación: operador de igualdad

Pregunta: ¿**Son dos valores iguales**?

Para hacer esta pregunta, se utiliza el `==` Operador (igual igual).

No olvides esta importante distinción:

- `=` es un **operador de asignación**, por ejemplo, `a = b` asigna a la variable `a` el valor de `b`.
- `==` es una pregunta *¿Son estos valores iguales?*; `a == b` **compara** `a` y `b`.

Es un **operador binario con enlazado a la izquierda**. Necesita dos argumentos y **verifica si son iguales**.

## Ejercicios

Ahora vamos a hacer algunas preguntas. Intenta adivinar las respuestas.

---

**Pregunta #1:** ¿Cuál es el resultado de la siguiente comparación?

`2 == 2`    Revisar

---

**Pregunta # 2:** ¿Cuál es el resultado de la siguiente comparación?

`2 == 2.`    Revisar

---

**Pregunta # 3:** ¿Cuál es el resultado de la siguiente comparación?

`1 == 2`

## Igualdad: El operador *igual a* (==)

El operador == (igual a) compara los valores de dos operandos. Si son iguales, el resultado de la comparación es True. Si no son iguales, el resultado de la comparación es False.

Observa la comparación de igualdad a continuación: ¿Cuál es el resultado de esta operación?

```
var == 0
```

Ten en cuenta que no podemos encontrar la respuesta si no sabemos qué valor está almacenado actualmente en la variable (var).

Si la variable se ha cambiado muchas veces durante la ejecución del programa, o si se ingresa su valor inicial desde la consola, Python solo puede responder a esta pregunta en el tiempo de ejecución del programa.

Ahora imagina a un programador que sufre de insomnio, y tiene que contar las ovejas negras y blancas por separado siempre y cuando haya exactamente el doble de ovejas negras que de las blancas.

La pregunta será la siguiente:

```
ovejasNegras == 2 * ovejasBlancas
```

Debido a la baja prioridad de el operador == ,la pregunta será tratada como la siguiente:

```
ovejasNegras == (2 * ovejaBlancas)
```

---

Entonces, vamos a practicar la comprensión del operador == - ¿Puedes adivinar la salida del código a continuación?

```
var = 0 # asignando 0 a var print(var == 0) var = 1 # asignando 1 a var print(var == 0)
```

Ejecuta el código y comprueba si tenías razón.

## Desigualdad: el operador *no es igual a* (!=)

El operador != (no es igual a) también compara los valores de dos operandos. Aquí está la diferencia: si son iguales, el resultado de la comparación es False. Si no son iguales, el resultado de la comparación es True.

Ahora echa un vistazo a la comparación de desigualdad a continuación: ¿Puedes adivinar el resultado de esta operación?

```
var = 0 # asignando 0 a var print(var != 0) var = 1 # asignando 1 a var print(var != 0)
```

Ejecuta el código y comprueba si tenías razón.



## Operadores de Comparación: Mayor que

También se puede hacer una pregunta de comparación usando el operador `>` (mayor que).

Si deseas saber si hay más ovejas negras que blancas, puedes escribirlo de la siguiente manera:

`ovejasNegras > ovejasBlancas` # mayor que.

True lo confirma; False lo niega.

## Operadores de Comparación: Mayor o igual que

El operador *mayor que* tiene otra variante especial, una variante **no estricta**, pero se denota de manera diferente que la notación aritmética clásica: `>=` (mayor o igual que).

Hay dos signos subsecuentes, no uno.

Ambos operadores (estrictos y no estrictos), así como los otros dos que se analizan en la siguiente sección, son **operadores binarios con enlace en el lado izquierdo**, y su **prioridad es mayor que la mostrada por `==` y `!=`**.

Si queremos saber si tenemos que usar un gorro o no, nos hacemos la siguiente pregunta:

`centigradosAfuera ≥ 0.0` # mayor o igual a.

## Operadores de Comparación: Menor o igual que

Como probablemente ya hayas adivinado, los operadores utilizados en este caso son: El operador `<` (menor que) y su hermano no estricto: `<=` (menor o igual que).

Mira este ejemplo simple:

`velocidadMph < 85` # menor que. `velocidadMph ≤ 85` # menor o igual que.

Vamos a comprobar si existe un riesgo de ser multados (la primera pregunta es estricta, la segunda no).

## Aprovechando las respuestas

¿Qué puedes hacer con la respuesta (es decir, el resultado de una operación de comparación) que se obtiene de la computadora?

Hay al menos dos posibilidades: primero, puedes memorizarlo (**almacenarlo en una variable**) y utilizarlo más tarde. ¿Cómo haces eso? Bueno, utilizarías una variable arbitraria como esta:

```
respuesta = numerodeLeones >= numerodeLeonas
```

El contenido de la variable te dirá la respuesta a la pregunta.

La segunda posibilidad es más conveniente y mucho más común: puedes utilizar la respuesta que obtengas para **tomar una decisión sobre el futuro del programa**.

Necesitas una instrucción especial para este propósito, y la discutiremos muy pronto.

Ahora necesitamos actualizar nuestra **tabla de prioridades**, y poner todos los nuevos operadores en ella. Ahora se ve como a continuación:

Prioridad	Operador	
1	+, -	unario
2	**	
3	*, /, %	
4	+, -	binario
5	<, <=, >, >=	
6	==, !=	

# Condiciones y ejecución condicional

Ya sabes como hacer preguntas a Python, pero aún no sabes como hacer un uso razonable de las respuestas. Se debe tener un mecanismo que le permita hacer algo **si se cumple una condición, y no hacerlo si no se cumple**.

Es como en la vida real: haces ciertas cosas o no cuando se cumple una condición específica, por ejemplo, sales a caminar si el clima es bueno, o te quedas en casa si está húmedo y frío.

Para tomar tales decisiones, Python ofrece una instrucción especial. Debido a su naturaleza y su aplicación, se denomina **instrucción condicional** (o declaración condicional).

Existen varias variantes de la misma. Comenzaremos con la más simple, aumentando la dificultad lentamente.

La primera forma de una declaración condicional, que puede ver a continuación, está escrita de manera muy informal pero figurada:

```
if cierto_o_no: hacer_esto_si_cierto
```

Esta declaración condicional consta de los siguientes elementos, estrictamente necesarios en este orden:

- La palabra clave **if**.
- Uno o más espacios en blanco.
- Una expresión (una pregunta o una respuesta) cuyo valor se interpretará únicamente en términos de **True** (cuando su valor no sea cero) y **False** (cuando sea igual a cero).
- Unos **dos puntos** seguidos de una nueva línea.
- Una instrucción **con sangría** o un conjunto de instrucciones (se requiere absolutamente al menos una instrucción); la **sangría** se puede lograr de dos maneras: insertando un número particular de espacios (la recomendación es usar **cuatro espacios de sangría**), o usando el *tabulador*; nota: si hay más de una instrucción en la parte con sangría, la sangría debe ser la misma en todas las líneas; aunque puede parecer lo mismo si se mezclan tabuladores con espacios, es importante que todas las sangrías **sean exactamente iguales** Python 3 **no permite mezclar espacios y tabuladores** para la sangría.

- 

¿Cómo funciona esta declaración?

- Si la expresión `cierto_o_no` **representa la verdad** (es decir, su valor no es igual a cero), **la(s) declaración(es) con sangría se ejecutará**.
- Si la expresión `cierto_o_no` **no representa la verdad** (es decir, su valor es igual a cero), **las declaraciones con sangría se omitirá**, y la siguiente instrucción ejecutada será la siguiente al nivel de la sangría original.

En la vida real, a menudo expresamos un deseo:

*if el clima es bueno, saldremos a caminar*

*después, almorzaremos*

Como puedes ver, almorzar no es **una actividad condicional** y no depende del clima.

Sabiendo que condiciones influyen en nuestro comportamiento y asumiendo que tenemos las funciones sin parámetros `irACaminar()` y `almorzar()`, podemos escribir el siguiente fragmento de código:

```
if ClimaEsBueno:  
    irAcaminar()  
almorzar()
```

## **Módulo 4**

Funciones, tuplas, diccionarios y procesamiento de datos.