Fundamentos de Python 1:
Módulo 1

Introducción a Python y a la programación de computadoras.

En este módulo, aprenderás sobre:

- Los fundamentos de la programación de computadoras, es decir, cómo funciona la computadora, cómo se ejecuta el programa, cómo se define y construye el lenguaje de programación.
- La diferencia entre compilación e interpretación.
- Qué es Python, cómo se posiciona entre otros lenguajes de programación y qué distingue las diferentes versiones de 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 comprender 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 conocer 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 entre 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 lenguaje que la computadora pueda aceptar.

La palabra clave es el **lenguaje**.

Lenguajes naturales frente a 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 sola 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 este número, divídelo entre 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). 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 compone a un lenguaje?

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

- Un **alfabeto**: un conjunto de símbolos utilizados para formar palabras de un determinado lenguaje (por ejemplo, el alfabeto latino para el inglés, el alfabeto cirílico para el ruso, el kanji para el japonés, y así sucesivamente).
- Un **léxico**: (también conocido como diccionario) un conjunto de palabras que el lenguaje ofrece a sus usuarios (por ejemplo, la palabra "computadora" proviene del diccionario en inglés, mientras que "cmoptrue" no; la palabra "chat" está presente en los diccionarios de inglés y francés, pero sus significados son diferentes.
- Una **sintaxis**: 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).
- Una **semántica**: 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. Ambos (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 las computadoras que son mucho más complejos que las instrucciones 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**.

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

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

COMPILACIÓN - 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 máquina; ahora puedes distribuir el archivo en todo el mundo; el programa que realiza esta traducción se llama compilador o traductor.

INTERPRETACIÓN - 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 en particular está diseñado para caer en una de estas dos categorías.

Existen muy pocos idiomas que se pueden ser tanto compilados como interpretados. 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 hacía abajo y de izquierda a derecha. Hay algunas excepciones: se cubrirán más adelante en el curso.

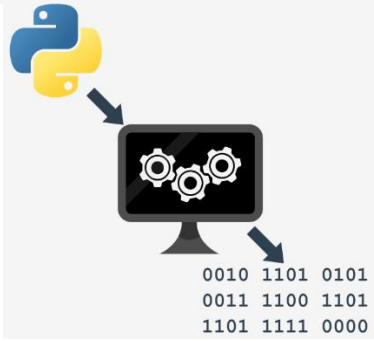
En primer lugar, el intérprete verifica si todas las líneas subsiguientes son correctas (utilizando los cuatro aspectos tratados anteriormente).

Si el compilador encuentra un error, termina su trabajo inmediatamente. El único resultado en este caso es un **mensaje de error**.

El intérprete te 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 sus 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 usarías 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", puede repetirse muchas veces, más veces que el número real de líneas en el archivo fuente, debido a que 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 lo hubiera, uno de estos modelos habría dejado de existir hace mucho tiempo. Ambos tienen sus ventajas y sus desventajas.

La compilación frente a la interpretación - ventajas y desventajas

	COMPILACIÓN	INTERPRETACIÓN
VENTAJAS	• La ejecución del código traducido suele ser más rápida. • Solo el programador debe tener el compilador; el usuario final puede usar el código sin él. • 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 un secreto.	• Puedes ejecutar el código en cuanto lo completes; no hay fases adicionales de traducción. • 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 se compila el código por separado para cada arquitectura diferente.
DESVENTAJAS	• La compilación en sí misma puede llevar mucho tiempo; es posible que	• No esperes que la interpretación incremente tu código a alta velocidad: tu código

	COMPILACIÓN	INTERPRETACIÓN
	no puedas ejecutar tu código inmediatamente después de cualquier modificación. • Tienes que tener tantos compiladores como plataformas de hardware en las que deseas que se ejecute tu código.	compartirá la potencia de la computadora con el intérprete, por lo que no puede ser realmente rápido. • Tanto tú como el usuario final deben tener el intérprete para ejecutar el código.

¿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 scripting**, 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.

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 en vivo para audiencias 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 al 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 existen muchos lenguajes de programación cuyos autores sean 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, testers, 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 el 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 PYPL PopularitY of Programming Language y el 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 al usarlo?

Existen 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 emplea 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 simples, lo que puede significar que la depuración del código de Python puede ser más difícil que con otros lenguajes. Afortunadamente, el cometer errores 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, por 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 más **aplicaciones de uso diario** en Python. Muchos **científicos** han abandonado las costosas herramientas patentadas y se han cambiado a Python. Muchos **testers** 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 existen 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.
- **Existe más de un Python**
- Existen 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.**
- 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. 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 el mismo, pero cuando se observan de cerca, se notan muchas diferencias.

- Si estás modificando una solución de 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.

- **NOTA**
- 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, 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 y Python 3.8.

Python alias CPython

Además de Python 2 y Python 3, existe más de una versión de cada uno.
En primer lugar, están los Pythons que se mantienen por 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 provienen del PSF están escritos en el lenguaje "C". Existen muchas razones para este enfoque. Una de ellas (probablemente la más importante) es que gracias a ello, 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 mucha expansión 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 de "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. ¿Puedes resolverlo?

Es el 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 sí los buscas, pero **este curso se centrará en CPython**.

Cómo obtener y utilizar Python

Existen 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 ensamblar 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 prompt del 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 utilizar 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 de Linux pueden descargar una copia en https://www.python.org/downloads/.

Comenzando tu trabajo con Python

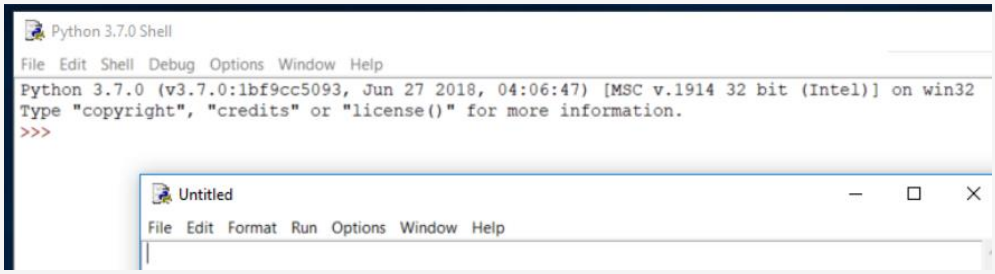
Ahora que tienes Python 3 instalado, es hora de verificar si funciona y de utilizarlo por primera vez.
Este será un procedimiento muy simple, pero debería ser suficiente para convencerte de que el entorno de Python es completo y funcional.
Existen muchas formas de utilizar Python, especialmente si vas a ser un desarrollador de Python.
Para comenzar tu trabajo, necesitas las siguientes herramientas:

- Un **editor** que te ayudará a escribir el código (debe 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 puedas ejecutar 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.

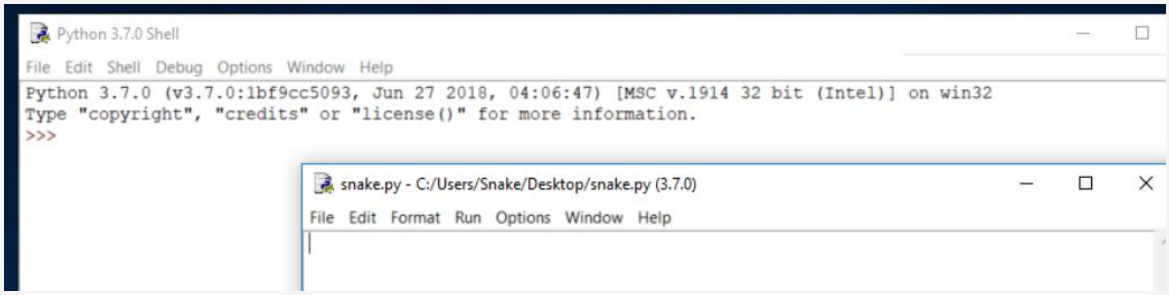
Además de sus muchos componentes útiles, la instalación estándar de Python 3 contiene una aplicación muy simple pero extremadamente útil llamada IDLE.
IDLE es un acrónimo de: Integrated Development and Learning Environment (Desarrollo Integrado y Entorno de Aprendizaje).
Navega por los menús de tu sistema 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 selecciona *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 del shell. Realizan diferentes funciones.
La ventana del editor actualmente no tiene título, pero es una buena práctica comenzar a trabajar nombrando el archivo fuente.
Haz clic en *File* (en la nueva ventana), luego haz clic sobre *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.



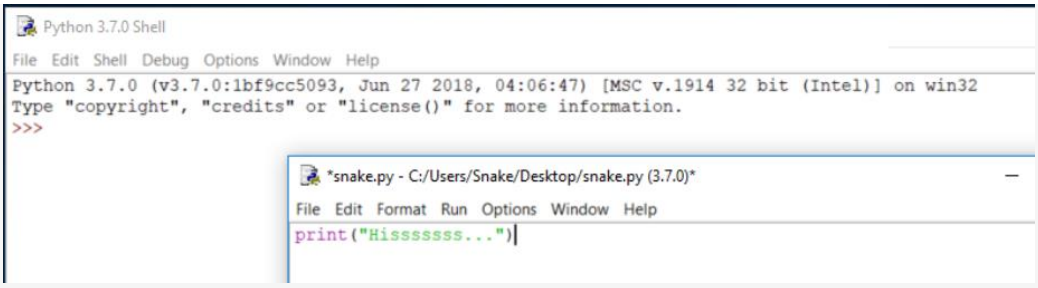
Nota: no establezcas ninguna extensión para el nombre de archivo que vas a utilizar. Python necesita que sus archivos tengan la extensión *.py*, por lo que debes confiar en los valores predeterminados de la ventana de diálogo. El uso de la extensión *.py* permite que el sistema operativo abra estos archivos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

Cómo escribir y ejecutar tu primer programa

Ahora solo coloca una línea en tu ventana de editor recién abierta y con nombre.
La línea se ve así:

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

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 las comillas (neutrales, rectas, etc.) que se usan comúnmente en los archivos fuente. No intentes utilizar citas tipográficas (curvadas, rizadas, etc.), utilizadas por los procesadores de texto avanzados, ya que Python no las acepta.



Guarda el archivo (*File* -> *Save*) y ejecuta el programa (*Run* -> *Run Module*).
Si todo sale 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 ejecutará de manera correcta y mostrará **Hissssss...** en la consola.
Intenta ejecutarlo una vez más. Y una vez más.
Ahora cierra ambas ventanas y vuelve al escritorio.

Cómo estropear y arreglar tu código

Ahora ejecuta IDLE nuevamente.

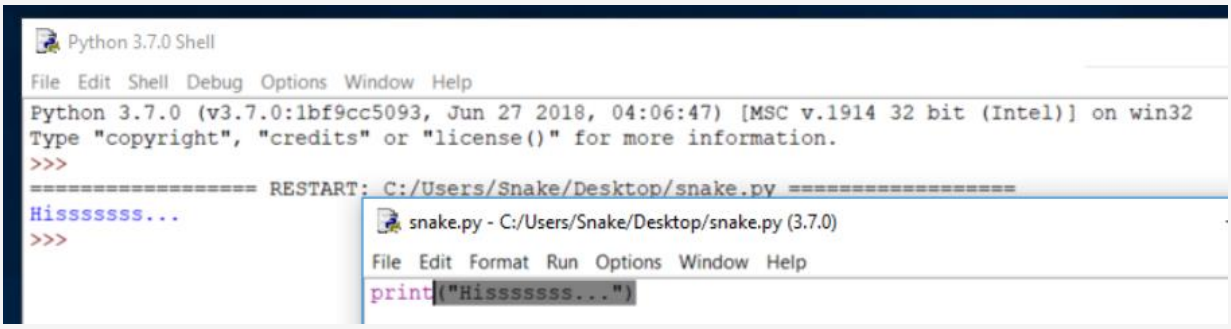
- Haz clic en *File*, *Open*, señala el archivo que guardaste anteriormente y deja que IDLE lo lea de nuevo.
- Intenta ejecutarlo de nuevo presionando *F5* cuando la ventana del editor esté activa.

Como puedes ver, IDLE puede guardar tu código y recuperarlo cuando lo necesites de nuevo.
IDLE contiene una característica adicional y muy útil.

- Primero, quita el paréntesis de cierre.
- Luego ingresa el paréntesis nuevamente.

Tu código debería parecerse al siguiente:

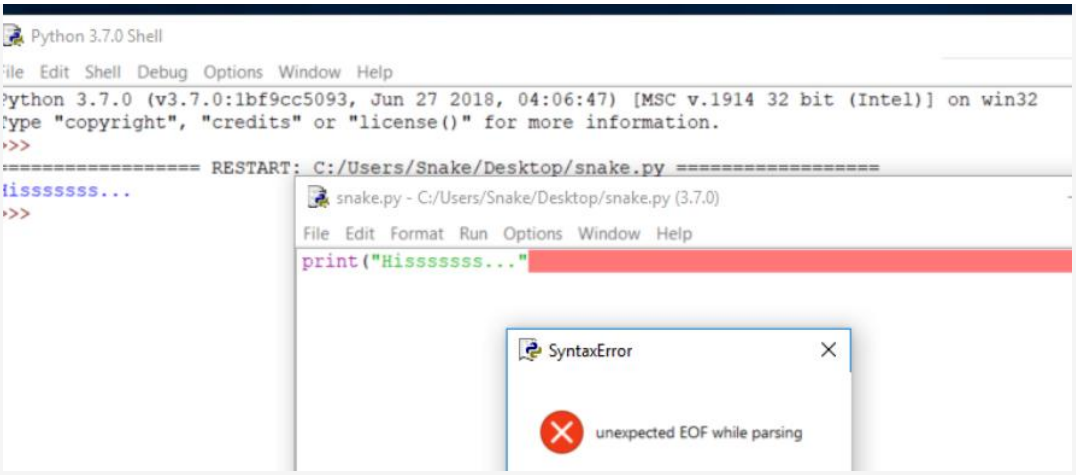
```
Hissssss...
```



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

Cómo estropear y arreglar tu código

Observa todas las ventanas con cuidado.
Aparece una nueva ventana, dice que el intérprete ha encontrado un EOF (*fin-de-archivo*) aunque (en su opinión) el código debería contener algo más de texto.
La ventana del editor muestra claramente donde ocurrió.

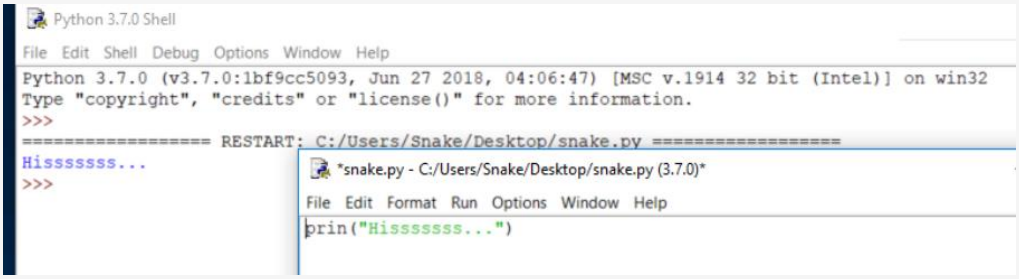


Ahora arregla el código. Debe verse así:

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

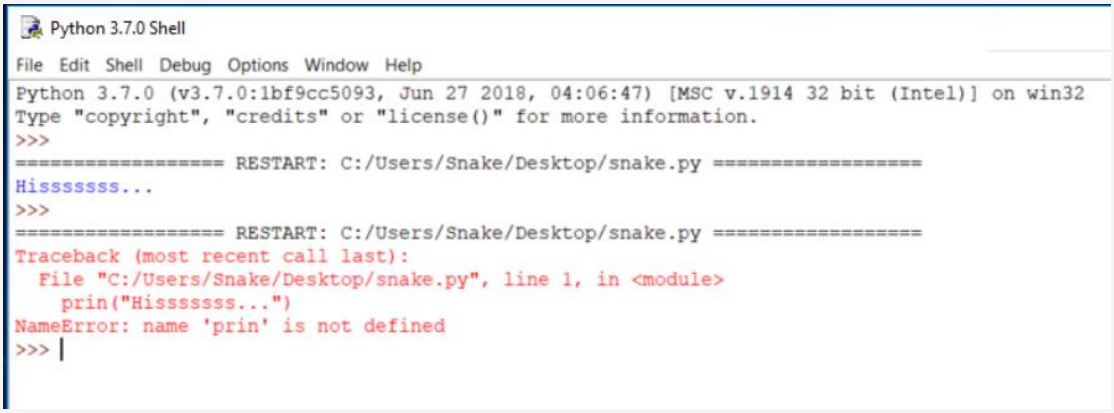
Ejecútalo para ver si sigue funcionando.

Vamos a arruinar el código una vez más. Elimina una letra de la palabra print. Ejecuta el código presionando *F5*. Como puedes ver, Python no puede reconocer la instrucción.



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.



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 si.

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 percata 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.

Fundamentos de Python 1:
Módulo 2
Tipos de datos, variables, operaciones básicas de entrada y salida, operadores básicos

En este módulo, aprenderás:

- Cómo escribir y ejecutar programas simples en Python.
- Qué son los literales, operadores y expresiones en Python.
- Qué son las variables y cuáles son las reglas que las gobiernan.
- Cómo realizar operaciones básicas de entrada y salida.

¡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, verás 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 el texto contenido entre comillas 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()

Observa 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 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** (por ejemplo, la raíz cuadrada de un valor o la longitud de un texto dado) y **devolverlo como el resultado de la función**; 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 existe un tercer componente de la función, muy importante, el o los **argumento**(s).
Las funciones matemáticas usualmente toman un argumento, por ejemplo, sen (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 desea 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 si, 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 puedes 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 más 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?

`nombre_función(argumento)`

Veamos:

como se procesa una función en Python:

- 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.

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 es legible).
- **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

Ya has visto un programa de computadora que contiene una invocación de función. La invocación de una función es uno de los muchos tipos posibles de **instrucciones** de Python.

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 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 observa lo que aparece 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 fuente; 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.
                    <--------print()
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.

```
print("La Witsi Witsi Araña\nsubió a su telaraña.\n")
print()
print("Vino la lluvia\ny se la llevó.")
```

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("\")
```

Mientras que esta no lo hará:

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

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

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

```
print("La Witsi Witsi Araña" , "subió" , "a 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 observar 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 a 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() **coloca un espacio entre los argumentos emitidos** por iniciativa propia.

La función print() - la manera posicional de pasar 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() - la manera posicional de pasar 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 palabra clave

Python ofrece otro mecanismo para transmitir o pasar los argumentos, que puede ser útil cuando se desea convencer 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 palabra 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 palabra 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.

```
print("Mi nombre es", "Python.", end=" ")
```

print("Monty Python.")

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 palabra 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.

La función print() - los argumentos de palabra 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` (*separador*).

Observa el código en el editor y ejecútalo.

```
print("Mi", "nombre", "es", "Monty", "Python.", sep="-")
```

El argumento `sep` entrega el siguiente resultado:

`Mi-nombre-es-Monty-Python.`

La función `print()` ahora utiliza un guion, 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 palabra 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 aprender cómo almacenar y procesar datos en Python.

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

Puntos Clave

1. La función `print()` es una función **integrada** imprime/envía un mensaje específico a la pantalla/ventana de consola.
2. Las funciones integradas, al contrario de las funciones definidas por el usuario, están siempre disponibles y no tienen que ser importadas. Python 3.7.1 viene con 69 funciones incorporadas. Puedes encontrar su lista completa en orden alfabético en Python Standard Library.
3. Para llamar a una función (**invocación de función**), debe utilizarse el nombre de la función seguido de un paréntesis. Puedes pasar argumentos a una función colocándolos dentro de los paréntesis. Se Deben separar los argumentos con una coma, por ejemplo, `print("¡Hola,", "Mundo!")`. una función `print()` "vacía" imprime una línea vacía a la pantalla.
4. Las cadenas de Python están delimitadas por **comillas**, por ejemplo, `"Soy una cadena"`, o `'Yo soy una cadena, también'`.
5. Los programas de computadora son colecciones de **instrucciones**. Una instrucción es un comando para realizar una tarea específica cuando se ejecuta, por ejemplo, para imprimir un determinado mensaje en la pantalla.
6. En las cadenas de Python, la **barra diagonal inversa** (`\`) es un carácter especial que anuncia que el siguiente carácter tiene un significado diferente, por ejemplo, `\n` (el **carácter de nueva línea**) comienza una nueva línea de salida.
7. Los **argumentos posicionales** son aquellos cuyo significado viene dictado por su posición, por ejemplo, el segundo argumento se emite después del primero, el tercero se emite después del segundo, etc.

8. Los **argumentos de palabra clave** son aquellos cuyo significado no está dictado por su ubicación, sino por una palabra especial (palabra clave) que se utiliza para identificarlos.
9. Los parámetros `end` y `sep` se pueden usar para dar formato la salida de la función `print()`. El parámetro `sep` especifica el separador entre los argumentos emitidos (por ejemplo, `print("H", "E", "L", "L", "O", sep="-")`, mientras que el parámetro `end` especifica que imprimir al final de la declaración de impresión.

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.

```
print("2")
print(2)
```

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 máquina (una serie de bits). La función `print()` es capaz de mostrar ambos en una forma legible para humanos.

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 fraccionaría.

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. Está **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`.
NOTA *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 esta 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 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 frente a 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: 3×10^8.
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: **6.62607×10^{-34}**.
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:

```
0.000000000000000000001
```

Cuando se corre en Python:

```
print(0.000000000000000000001)
```

Este es el resultado:

```
1e-22
```

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

```
print('I\'m Monty Python.')
```

o

```
print("I'm Monty Python.")
```

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 carácter 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 **Álgebra Booleana** - una parte del álgebra 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 Sandbox. ¿Puedes explicar el resultado?

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`).

EXTRA

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 número en binario?
`1011`
2.3.1 Operadores: herramientas para la manipulación de datos:

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)
```

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 `+` (más) es un operador el cual es capaz de **sumar** dos números, dando el resultado de la suma.
Sin embargo, no todos los operadores de Python son tan simples como el signo de más, 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**.

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

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 frente a 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 frente a 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.

Observa el siguiente fragmento de código:

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

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

Si, sería `1.5` en ambos casos. Eso está 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`.

NOTA

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 más complicado:

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

¿Cuál es el resultado?

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 recordarás de la escuela que las **multiplicaciones preceden a las sumas**.

Seguramente recordarás 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 cálculo 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 del lado izquierdo**. 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` → `4`; `4 ** 3` → `64`
- `2 ** 3` → `8`; `2 ** 8` → `256`

Ejecuta el código, ¿Qué es lo que observas?

El resultado muestra claramente que **el operador de exponenciación utiliza enlazado del lado derecho**.

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 más alta (1) a la más 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? DEBE DAR 1

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()`?

Debe dar: `10.0`

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: regresa un flotante siempre), `%` (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 número resultado de la división, pero redondeado al número 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 del lado derecho**, por ejemplo, `2 ** 2 ** 3 = 256`.

Ejercicio 1

¿Cuál es la salida del siguiente fragmento de código?

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

Resultado: `16 8.0 8`

Ejercicio 2

¿Cuál es la salida del siguiente fragmento de código?

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

Resultado: `-0.5 0.5 0 -1`

Ejercicio 3

¿Cuál es la salida del siguiente fragmento de código?

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

resultado: `-2 2 512`

1.4.1 VARIABLES: CAJAS EN FORMA DE DATOS

¿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 MAYÚSCULAS, 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 - *Alicia* 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`, `dias_para_navidad`, `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)

NOTA

PEP 8 -- Style Guide for Python Code recomienda la siguiente convención de nomenclatura para variables y funciones en Python:

- Los nombres de las variables deben estar en minúsculas, con palabras separadas por guiones bajos para mejorar la legibilidad (por ejemplo: `var`, `mi_variable`).
- Los nombres de las funciones siguen la misma convención que los nombres de las variables (por ejemplo: `fun`, `mi_función`).
- También es posible usar letras mixtas (por ejemplo: `miVariable`), pero solo en contextos donde ese ya es el estilo predominante, para mantener la compatibilidad retroactiva con la convención adoptada.

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**).

RECUERDA

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 y puede manejar variables también. ¿Puedes predecir cuál será la salida (resultado) del código? `1`

Utilizando variables

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

```
var = 1
account_balance = 1000.0
client_name = 'John Doe'
print(var, account_balance, client_name)
print(var)
```

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

RECUERDA

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

```
var = "3.8.5"
print("Versión de Python: " + var)
```

¿Puedes predecir la salida del fragmento de código?
Revisar

```
Versión de Python: 3.8.5
```

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
```

salida

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 más 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 más 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)
```

resultado: `500` - ¿Porque? Bueno, primero, la variable `var` es creada y se le asigna el valor de 100. Después, a la misma variable se le asigna un nuevo valor: el resultado de sumarle 200 a 300, lo cual es 500.

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 triangulo 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:
$\sqrt{}(x) = x^{(½)}$
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.
Resultado `c = 5.0`

Operadores Abreviados

Es tiempo de explicar el siguiente conjunto de operadores que harán la vida del programador/desarrollador más 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:

```
sheep = sheep + 1
```

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

```
x *= 2
sheep += 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 imporante) y el operador es utilizado en el siguiente contexto:

```
variable = variable op expresión
```

It can be simplified and shown as follows:

```
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`

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. (2.1.4.1)
2. Cada variable debe de tener un nombre único - un **identificador**. Un nombre válido 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. (2.1.4.1)
3. Python es un lenguaje **de tipo dinámico**, lo que significa que no se necesita *declarar* variables en él. (2.1.4.3) 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`. (2.1.4.8)
5. Se les puede asignar valores nuevos a variables ya existentes utilizando el operador de asignación o un operador abreviado, por ejemplo (2.1.4.5):

```
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: (2.1.4.4)

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

Ejercicio 1

¿Cuál es el resultado del siguiente fragmento de código?

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

Resultado `3`

Ejercicio 2

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

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

Ejercicio 3

¿Cuál es el resultado del siguiente fragmento de código?

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

Resultado:

`11`

Ejercicio 4

¿Cuál es el resultado del siguiente fragmento de código?

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

resultado: 1.0
2 * b = 6
a = 6 → 6 / 6 = 1.0

2.5.1 COMENTARIOS.

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 `area_cuadrado` será muchísimo mejor que `tia_juana`.
El 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, sí 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.

TIP

Si deseas comentar o descomentar rápidamente varias líneas de código, selecciona las líneas que deseas modificar y utiliza el siguiente método abreviado de teclado: **CTRL** + **/** (Windows) or **CMD** + **/** (Mac OS).Es un truco muy útil, ¿no?
Intenta este código en Sandbox.

Ejemplo:

```
# este programa calcula los segundos en cierto número de horas determinadas
# este programa fue escrito hace dos días

a = 2 # número de horas
seconds = 3600 # número de segundos en una hora
```

```
print("Horas: ", a) #imprime el numero de horas
print("Segundos en Horas: ", a * seconds) # se imprime el numero de segundos en determinado numero de horas

#aquí también se debe de imprimir un "Adiós", pero el programador no tuvo tiempo de escribirlo
#este el es fin del programa que calcula el numero de segundos en 2 horas
```

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 última línea de código del siguiente fragmento), por ejemplo:

```
# Este programa imprime
# un saludo en pantalla
print("Hola!")  # Se invoca la función print()
# 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 *tú* estás 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")
```

Salida: `Cadena #2`

Ejercicio 2

¿Qué ocurrirá cuando se ejecute el siguiente código?

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

Salida: `SyntaxError: invalid syntax`

2.6.1 COMO HABLAR CON UNA COMPUTADORA

La función input()

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

¿Por qué? 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...")
anything = input()
```

```
print("Mmm...", anything, "...¿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.

La función input() con un argumento

La función `input()` puede hacer algo más: puede mostrar un mensaje al usuario sin la ayuda de la función `print()`.

Se ha modificado el ejemplo un poco, observa el código:

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

Nota:

- La función `input()` al ser invocada con un argumento, contiene una cadena con un mensaje.
- El mensaje será mostrado en consola antes de que el usuario tenga oportunidad de escribir algo.
- Después de esto `input()` hará su trabajo.

Esta variante de la invocación de la función `input()` simplifica el código y lo hace más claro.

El resultado de la función input()

Se ha dicho antes, pero hay que decirlo sin ambigüedades una vez más: el **resultado de la función** `input()` **es una cadena**.

Una cadena que contiene todos los caracteres que el usuario introduce desde el teclado. No es un entero ni un flotante.

Esto significa que **no se debe utilizar como un argumento para operaciones matemáticas**, por ejemplo, no se pueden utilizar estos datos para elevarlos al cuadrado, para dividirlos entre algo o por algo.

```
anything = input("Inserta un número: ")
something = anything ** 2.0
print(anything, "al cuadrado es", something)
```

La función input() - operaciones prohibidas

 # Probando mensajes de error.

```
anything = input("Inserta un número: ")
something = anything ** 2.0
print(anything, "al cuadrado es", something)
```

Observa el código en el editor. Ejecútalo, inserta cualquier número, y oprime *Enter*.

¿Qué es lo que ocurre?

Python debió haberte dado la siguiente salida:

```
Traceback (most recent call last):
File ".main.py", line 4, in <module>
something = anything ** 2.0
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'float'
```

La última línea lo explica todo, se intentó aplicar el operador `**` a `'str'` (una cadena) acompañado por un `'float'` (valor flotante).

Esto está prohibido.

Esto debe de ser obvio. ¿Puedes predecir el valor de `"ser o no ser"` elevado a la `2` potencia?

No podemos. Python tampoco puede.
¿Habremos llegado a un punto muerto? ¿Existirá alguna solución? Claro que la hay.

Conversión de datos (casting)

Python ofrece dos simples funciones para especificar un tipo de dato y resolver este problema, aquí están: `int()` y `float()`.
Sus nombres indican cual es su función:

- La función `int()` **toma un argumento** (por ejemplo, una cadena: `int(string)`) e intenta convertirlo a un valor entero; si llegase a fallar, el programa entero fallará también (existe una manera de solucionar esto, se explicará mas adelante).
- La función `float()` toma un argumento (por ejemplo, una cadena: `float(string)`) e intenta convertirlo a flotante (el resto es lo mismo).

Esto es muy simple y muy efectivo. Sin embargo, estas funciones se pueden invocar directamente pasando el resultado de la función `input()` directamente. No hay necesidad de emplear variables como almacenamiento intermedio.

```
leg_a = float(input("Inserta la longitud del primer cateto: "))

leg_b = float(input("Inserta la longitud del segundo cateto: "))

hypo = (leg_a**2 + leg_b**2) ** .5

print("La longitud de la hipotenusa es:", hypo)
```

Se ha implementado esta idea en el editor, observa el código.

```
anything = float(input("Inserta un número: "))
something = anything ** 2.0
print(anything, "al cuadrado es", something)
```

¿Puedes imaginar como la cadena introducida por el usuario fluye desde la función `input()` hacía la función `print()`?
Intenta ejecutar el código modificado. No olvides introducir un **número valido**.
Prueba con diferentes valores, pequeños, grandes, negativos y positivos. El cero también es un buen valor a introducir.

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.

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

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.

Toma en cuenta que en el programa que puede ver en el editor, la variable `hypo` se usa con un solo propósito: guardar el valor calculado entre la ejecución de la línea de código contigua.
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:

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

Operadores de cadenas - introducción

Es tiempo de regresar a estos dos operadores aritméticos: `+` y `*`.
Ambos tienen una función secundaría. 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. Por supuesto, puede ser utilizado más de una vez en una misma expresión, y en tal contexto se comporta con enlazado del lado izquierdo.

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:

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

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**.

```
string * number
number * string
```

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

Por ejemplo:

- `"James" * 3` produce `"JamesJamesJames"`
- `3 * "an"` produce `"ananan"`
- `5 * "2"` (o `"2" * 5`) produce `"22222"` (no `10`!)

RECUERDA

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(number)
```

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:

```
leg_a = float(input("Inserta la longitud del primer cateto: "))
leg_b = float(input("Inserta la longitud del segundo cateto: "))
print("La longitud de la hipotenusa es " + str((leg_a**2 + leg_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.

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:

```
name = input("Ingresa tu nombre: ")
print("Hola, " + name + ". ¡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*.

NOTA

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:

```
name = input("Ingresa tu nombre: ")
print("Hola, " + name + ". ¡Un gusto conocerte!")

print("\nPresiona la tecla Enter para finalizar el programa.")
input()
print("FIN.")
```

4. 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:

```
num_1 = input("Ingresa el primer número: ") # Ingresa 12
num_2 = input("Ingresa el segundo número: ") # Ingresa 21
print(num_1 + num_2) el programa retorna 1221
```

5. También se pueden multiplicar (`*` - replicación) cadenas, por ejemplo:

```
my_input = input("Ingresa algo: ") # Ejemplo: hola
print(my_input * 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")
```

resultado: `55`

Ejercicio 2

¿Cuál es la salida esperada del siguiente código

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

Resultado: `<class 'str'>`

FIN DEL MODULO 2.

Fundamentos de Python 1:
Módulo 3

En este módulo, aprenderás sobre:

- Datos de tipo booleano.

- Operadores relacionales.
- Cómo tomar decisiones en Python (if, if-else, if-elif, else).
- Cómo repetir la ejecución de código usando los bucles (while, for).
- Cómo realizar operaciones lógicas y de bit a bit en Python.
- Listas en Python (construcción, indexación y segmentación; manipulación de contenido).
- Cómo ordenar una lista usando algoritmos de clasificación de burbujas.
- Listas multidimensionales y sus aplicaciones.

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:

- Si, 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 assigna a la varable a el valor de b.
- == es una pregunta *¿Son estos valores iguales?* así que a == b **compara** a y b.

Es un **operador binario con enlazado del lado izquierdo**. 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

True - por supuesto, 2 es igual a 2. Python responderá True (recuerda este par de literales predefinidos, True y False - también son palabras clave reservadas de Python).

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

2 == 2 Revisar

Esta pregunta no es tan fácil como la primera. Por suerte, Python es capaz de convertir el valor entero en su equivalente real, y en consecuencia, la respuesta es True.

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

1 == 2 Revisar

Esto debería ser fácil. La respuesta será (o mejor dicho, siempre es) False.

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
```

Toma 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:

```
black_sheep == 2 * white_sheep
```

Debido a la baja prioridad del operador `==`, la pregunta será tratada como la siguiente:

```
black_sheep == (2 * white_sheep)
```

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: 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).

Observa este ejemplo simple:

```
current_velocity_mph < 85  # Menor que
current_velocity_mph ≤ 85  # Menor o igual que
```

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

Haciendo uso de las respuestas

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

```
answer = number_of_lions >= number_of_lionesses
```

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	`==`, `!=`	

LABORATORIO

Tiempo Estimado
5 minutos
Nivel de Dificultad
Muy Fácil
Objetivos

- Familiarizarse con la función `input()`.
- Familiarizarse con los operadores de comparación en Python.

Escenario
Usando uno de los operadores de comparación en Python, escribe un programa simple de dos líneas que tome el parámetro `n` como entrada, que es un entero, e imprime `False` si `n` es menor que `100`, y `True` si `n` es mayor o igual que `100`.
No debes crear ningún bloque `if` (hablaremos de ellos muy pronto). Prueba tu código usando los datos que te proporcionamos.
Datos de Prueba
Ejemplo de entrada: `55`
Resultado esperado: `False`
Ejemplo de entrada: `99`
Resultado esperado: `False`
Ejemplo de entrada: `100`
Resultado esperado: `True`
Ejemplo de entrada: `101`
Resultado esperado: `True`
Ejemplo de entrada: `-5`
Resultado esperado: `False`
Ejemplo de entrada: `+123`
Resultado esperado: `True`

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 sentencia condicional).
Existen varias variantes de la misma. Comenzaremos con la más simple, aumentando la dificultad lentamente.
La primera forma de una sentencia condicional, que puede ver a continuación, está escrita de manera muy informal pero figurada:

```
if true_or_not:
    do_this_if_true
```

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

- La palabra clave reservada `if`.
- Uno o más espacios en blanco.
- Una expresión (una pregunta o una respuesta) cuyo valor se interpretar únicamente en términos de `True` (cuando su valor no sea cero) y `False` (cuando sea igual a cero).
- Unos **dos puntos** seguido 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 mas 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 sentencia?

- Si la expresión `true_or_not` **representa la verdad** (es decir, su valor no es igual a cero), **las sentencias con sangría se ejecutarán**.
- Si la expresión `true_or_not` **no representa la verdad** (es decir, su valor es igual a cero), **las sentencias con sangría se omitirán** , y la siguiente instrucción ejecutada será la siguiente al nivel de la sangría original.

En la vida real, a menudo expresamos un deseo:
si 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 `go_for_a_walk()` y `have_lunch()`, podemos escribir el siguiente fragmento de código:

```
if the_weather_is_good:
    go_for_a_walk()
have_lunch()
```

Ejecución condicional: la sentencia if

Si un determinado desarrollador de Python sin dormir se queda dormido cuando cuenta 120 ovejas, y el procedimiento de inducción del sueño se puede implementar como una función especial llamada `sleep_and_dream()`, el código toma la siguiente forma:

```
if sheep_counter >= 120: # #evalúa una expresión condicional
    sleep_and_dream() #se ejecuta si la expresión condicional es True

```

Puedes leerlo como sigue: si `sheep_counter` es mayor o igual que `120`, entonces duerme y sueña (es decir, ejecuta la función `sleep_and_dream`).

Hemos dicho que las **sentencias condicionales deben tener sangría**. Esto crea una estructura muy legible, demostrando claramente todas las rutas de ejecución posibles en el código.
Analiza el siguiente código:

```
if sheep_counter >= 120:
    make_a_bed()
    take_a_shower()
    sleep_and_dream()
feed_the_sheepdogs()
```

Como puedes ver, tender la cama, tomar una ducha y dormir y soñar se ejecutan **condicionalmente**, cuando `sheep_counter` alcanza el límite deseado.
Alimentar a los perros, sin embargo, **siempre se hace** (es decir, la función `feed_the_sheepdogs()` no tiene sangría y no pertenece al bloque `if`, lo que significa que siempre se ejecuta).
Ahora vamos a discutir otra variante de la sentencia condicional, que también permite realizar una acción adicional cuando no se cumple la condición.

Ejecución condicional: la sentencia if-else

Comenzamos con una frase simple que decía: *Si el clima es bueno, saldremos a caminar*.
Nota: no hay una palabra sobre lo que sucederá si el clima es malo. Solo sabemos que no saldremos al aire libre, pero no sabemos que podríamos hacer. Es posible que también queramos planificar algo en caso de mal tiempo.
Podemos decir, por ejemplo: *Si el clima es bueno, saldremos a caminar, de lo contrario, iremos al cine*.
Ahora sabemos lo que haremos **si se cumplen las condiciones**, y sabemos lo que haremos **si no todo sale como queremos**. En otras palabras, tenemos un "Plan B".
Python nos permite expresar dichos planes alternativos. Esto se hace con una segunda forma, ligeramente mas compleja, de la sentencia condicional, la sentencia *if-else*:

```
if true_or_false_condition:
    perform_if_condition_true
else:
    perform_if_condition_false
```

Por lo tanto, hay una nueva palabra: `else` - esta es una **palabra clave reservada**.
La parte del código que comienza con `else` dice que hacer si no se cumple la condición especificada por el `if` (observa los **dos puntos** después de la palabra).
La ejecución de *if-else* es la siguiente:

- Si la condición se evalúa como **True** (su valor no es igual a cero), la instrucción `perform_if_condition_true` se ejecuta, y la sentencia condicional llega a su fin.
- Si la condición se evalúa como **False** (es igual a cero), la instrucción `perform_if_condition_false` se ejecuta, y la sentencia condicional llega a su fin.

La sentencia if-else: más sobre ejecución condicional

Al utilizar esta forma de sentencia condicional, podemos describir nuestros planes de la siguiente manera:

```
if the_weather_is_good:
    go_for_a_walk()
else:
    go_to_a_theater()
have_lunch()
```

Si el clima es bueno, saldremos a caminar. De lo contrario, iremos al cine. No importa si el clima es bueno o malo, almorzaremos después (después de la caminata o después de ir al cine).
Todo lo que hemos dicho sobre la sangría funciona de la misma manera dentro de **la rama *else*** :

```
if the_weather_is_good:
    go_for_a_walk()
    have_fun()
else:
    go_to_a_theater()
    enjoy_the_movie()
have_lunch()
```

Sentencias if-else anidadas

Ahora, analicemos dos casos especiales de la sentencia condicional.
Primero, considera el caso donde la instrucción **colocada después del** `if` **es otro** `if`.
Lee lo que hemos planeado para este Domingo. Si hay buen clima, saldremos a caminar. Si encontramos un buen restaurante, almorzaremos allí. De lo contrario, vamos a comer un sandwich. Si hay mal clima, iremos al cine. Si no hay boletos, iremos de compras al centro comercial más cercano.
Escribamos lo mismo en Python. Considera cuidadosamente el código siguiente:

```
if the_weather_is_good:
    if nice_restaurant_is_found:
        have_lunch()
    else:
        eat_a_sandwich()
else:
    if tickets_are_available:
        go_to_the_theater()
    else:
        go_shopping()
```

Aquí hay dos puntos importantes:

- Este uso de la sentencia `if` se conoce como **anidamiento**; recuerda que cada `else` se refiere al `if` que se encuentra **en el mismo nivel de sangría**; se necesita saber esto para determinar cómo se relacionan los *if* y los *else* .
- Considera como la sangría **mejora la legibilidad** y hace que el código sea más fácil de entender y rastrear.

La sentencia elif

El segundo caso especial presenta otra nueva palabra clave de Python: **elif**. Como probablemente sospechas, es una forma más corta de **else-if**.
`elif` se usa para **verificar más de una condición**, y para **detener** cuando se encuentra la primera sentencia verdadera.
Nuestro siguiente ejemplo se parece a la anidación, pero las similitudes son muy leves. Nuevamente, cambiaremos nuestros planes y los expresaremos de la siguiente manera: si hay buen clima, saldremos a caminar, de lo contrario, si obtenemos entradas, iremos al cine, de lo contrario, si hay mesas libres en el restaurante, vamos a almorzar; si todo falla, regresaremos a casa y jugaremos ajedrez.
¿Has notado cuantas veces hemos usado la palabra *de lo contrario*? Esta es la etapa en la que la palabra clave reservada `elif` desempeña su función.
Escribamos el mismo escenario empleando Python:

```
if the_weather_is_good:
    go_for_a_walk()
elif tickets_are_available:
    go_to_the_theater()
```

```
elif table_is_available:
    go_for_lunch()
else:
    play_chess_at_home()
```

La forma de ensamblar las siguientes sentencias *if-elif-else* a veces se denomina **cascada**.
Observa de nuevo como la sangría mejora la legibilidad del código.
Se debe prestar atención adicional a este caso:

- **No debes usar** `else` **sin un** `if` precedente.
- `else` siempre es la **última rama de la cascada**, independientemente de si has usado `elif` o no.
- `else` es una parte **opcional** de la cascada, y puede omitirse.
- Si hay una rama `else` en la cascada, solo se ejecuta una de todas las ramas.
- Si no hay una rama `else`, es posible que no se ejecute ninguna de las opciones disponibles.

Esto puede sonar un poco desconcertante, pero ojalá que algunos ejemplos simples ayuden a comprenderlo mejor.

Analizando ejemplos de código

Ahora te mostraremos algunos programas simples pero completos. No los explicaremos a detalle, porque consideramos que los comentarios (y los nombres de las variables) dentro del código son guías suficientes.
Todos los programas resuelven el mismo problema: **encuentran el número mayor de una serie de números y lo imprimen**.

Ejemplo 1:

Comenzaremos con el caso más simple: **¿Cómo identificar el mayor de los dos números?**

```
#Se leen dos números
number1 = int(input("Ingresa el primer número: "))
number2 = int(input("Ingresa el segundo número: "))

# Elige el número más grande
if number1 > number2:
    larger_number = number1
else:
    larger_number = number2
# Imprime el resultado
print("El número más grande es:", larger_number)
```

El fragmento de código anterior debe estar claro: lee dos valores enteros, los compara y encuentra cuál es el más grande.

Ejemplo 2:

Ahora vamos a mostrarte un hecho intrigante. Python tiene una característica interesante, mira el código a continuación:

```
#Se leen dos números
number1 = int(input("Ingresa el primer número: "))
number2 = int(input("Ingresa el segundo número: "))
# Elige el número más grande
if number1 > number2: larger_number = number1
else: larger_number = number2
# Imprime el resultado
print("El número más grande es:", larger_number)
```

Nota: si alguna de las ramas de *if-elif-else* contiene una sola instrucción, puedes codificarla de forma más completa (no es necesario que aparezca una línea con sangría después de la palabra clave), pero solo continúa la línea después de los dos puntos).
Sin embargo, este estilo puede ser engañoso, y no lo vamos a usar en nuestros programas futuros, pero definitivamente vale la pena saber si quieres leer y entender los programas de otra persona.
No hay otras diferencias en el código.

Ejemplo 3:

Es hora de complicar el código: encontremos el mayor de los tres números. ¿Se ampliará el código? Un poco.
Suponemos que el primer valor es el más grande. Luego verificamos esta hipótesis con los dos valores restantes.
Observa el siguiente código:

```
# Se leen tres números
number1 = int(input("Ingresa el primer número: "))
number2 = int(input("Ingresa el segundo número: "))
number3 = int(input("Ingresa el tercer número: "))
# Asumimos temporalmente que el primer número
# es el más grande.
# Lo verificaremos pronto.
largest_number = number1

# Comprobamos si el segundo número es más grande que el mayor número actual
# y actualiza el número más grande si es necesario.
if number2 > largest_number:
    largest_number = number2

# Comprobamos si el tercer número es más grande que el mayor número actual
# y actualiza el número más grande si es necesario.
if number3 > largest_number:
    largest_number = number3
# Imprime el resultado.
print("El número más grande es:", largest_number)
```

Este método es significativamente más simple que tratar de encontrar el número más grande comparando todos los pares de números posibles (es decir, el primero con el segundo, el segundo con el tercero y el tercero con el primero). Intenta reconstruir el código por ti mismo.

Pseudocódigo e introducción a los bucles (ciclos)

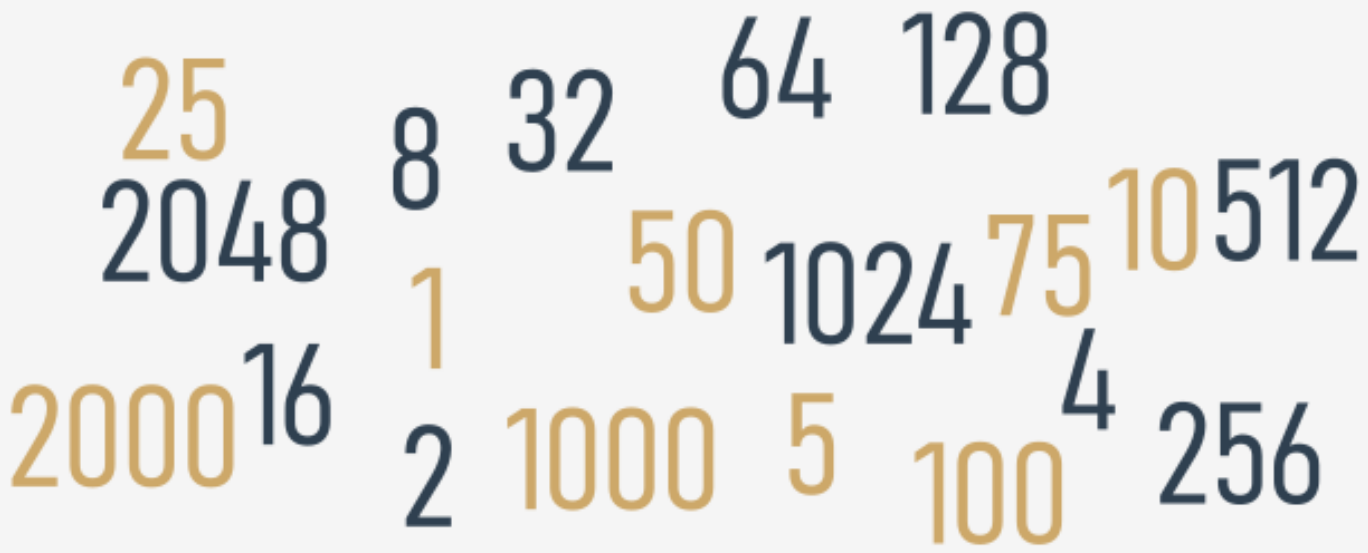
Ahora deberías poder escribir un programa que encuentre el mayor de cuatro, cinco, seis o incluso diez números.

Ya conoces el esquema, por lo que ampliar el tamaño del problema no será particularmente complejo.

¿Pero qué sucede si te pedimos que escribas un programa que encuentre el mayor de doscientos números? ¿Te imaginas el código?

Necesitarás doscientas variables. Si doscientas variables no son lo suficientemente complicadas, intenta imaginar la búsqueda del número más grande de un millón.

Imagina un código que contiene 199 sentencias condicionales y doscientas invocaciones de la función `input()`. Por suerte, no necesitas lidiar con eso. Hay un enfoque más simple.



Por ahora ignoraremos los requisitos de la sintaxis de Python e intentaremos analizar el problema sin pensar en la programación real. En otras palabras, intentaremos escribir el **algoritmo**, y cuando estemos contentos con él, lo implementaremos.

En este caso, utilizaremos un tipo de notación que no es un lenguaje de programación real (no se puede compilar ni ejecutar), pero está formalizado, es conciso y se puede leer. Se llama **pseudocódigo**.
Veamos nuestro pseudocódigo a continuación:

```
largest_number = -999999999
number = int(input())
if number == -1:
    print(largest_number)
    exit()
if number > largest_number:
    largest_number = number
# Ir a la línea 02
```

¿Qué está pasando en él?
En primer lugar, podemos simplificar el programa si, al principio del código, le asignamos a la variable `largest_number` un valor que será más pequeño que cualquiera de los números ingresados. Usaremos `-999999999` para ese propósito.
En segundo lugar, asumimos que nuestro algoritmo no sabrá por adelantado cuántos números se entregarán al programa. Esperamos que el usuario ingrese todos los números que desee; el algoritmo funcionará bien con cien y con mil números. ¿Cómo hacemos eso?

Hacemos un trato con el usuario: cuando se ingresa el valor `-1`, será una señal de que no hay más datos y que el programa debe finalizar su trabajo.
De lo contrario, si el valor ingresado no es igual a `-1`, el programa leerá otro número, y así sucesivamente.
El truco se basa en la suposición de que cualquier parte del código se puede realizar más de una vez, precisamente, tantas veces como sea necesario.
La ejecución de una determinada parte del código más de una vez se denomina **bucle**. El significado de este término es probablemente obvio para ti.
Las líneas `02` a `08` forman un bucle. Los **pasaremos tantas veces como sea necesario** para revisar todos los valores ingresados.
¿Puedes usar una estructura similar en un programa escrito en Python? Si, si puedes.

Información Adicional

Python a menudo viene con muchas funciones integradas que harán el trabajo por ti. Por ejemplo, para encontrar el número más grande de todos, puede usar una función incorporada de Python llamada `max()`. Puedes usarlo con múltiples argumentos. Analiza el código de abajo:

```
# Se leen tres números.

number1 = int(input("Ingresa el primer número: "))

number2 = int(input("Ingresa el segundo número: "))

number3 = int(input("Ingresa el tercer número: ")

# Verifica cuál de los números es el mayor

# y pásalo a la variable largest_number

largest_number = max(number1, number2, number3)

# Imprime el resultado.
```

```
print("El número más grande es:", largest_number)
```

De la misma manera, puedes usar la función `min()` para devolver el número más pequeño. Puedes reconstruir el código anterior y experimentar con él en Sandbox.

Vamos a hablar sobre estas (y muchas otras) funciones pronto. Por el momento, nuestro enfoque se centrará en la ejecución condicional y los bucles para permitirte ganar más confianza en la programación y enseñarte las habilidades que te permitirán comprender y aplicar los dos conceptos en tu codigo. Entonces, por ahora, no estamos tomando atajos.

Bucles (ciclos) en el código con while

¿Estás de acuerdo con la sentencia presentada a continuación?

```
mientras haya algo que hacer
  hazlo
```

Toma en cuenta que este registro también declara que, si no hay nada que hacer, nada ocurrirá.

En general, en Python, un bucle se puede representar de la siguiente manera:

```
while conditional_expression:
  instruction
```

Si observas algunas similitudes con la instrucción *if*, está bien. De hecho, la diferencia sintáctica es solo una: usa la palabra `while` en lugar de la palabra `if`.

La diferencia semántica es más importante: cuando se cumple la condición, *if* realiza sus sentencias **sólo una vez**; *while* **repite la ejecución siempre que la condición se evalúe como** `True`.

Nota: todas las reglas relacionadas con **sangría** también se aplican aquí. Te mostraremos esto pronto.

Observa el algoritmo a continuación:

```
while conditional_expression:
  instruction_one
  instruction_two
  instruction_three
  :
  :
  instruction_n
```

Ahora, es importante recordar que:

- Si deseas ejecutar **más de una sentencia dentro de un** `while`, debes (como con `if`) **poner sangría** a todas las instrucciones de la misma manera.
- Una instrucción o conjunto de instrucciones ejecutadas dentro del `while` se llama el **cuerpo del bucle**.
- Si la condición es `False` (igual a cero) tan pronto como se compruebe por primera vez, el cuerpo no se ejecuta ni una sola vez (ten en cuenta la analogía de no tener que hacer nada si no hay nada que hacer).
- El cuerpo debe poder cambiar el valor de la condición, porque si la condición es `True` al principio, el cuerpo podría funcionar continuamente hasta el infinito. Observa que hacer una cosa generalmente disminuye la cantidad de cosas por hacer.

Un bucle infinito

Un bucle infinito, también denominado **bucle sin fin**, es una secuencia de instrucciones en un programa que se repite indefinidamente (bucle sin fin).
Este es un ejemplo de un bucle que no puede finalizar su ejecución:

```
while True:
    print("Estoy atrapado dentro de un bucle.")
```

Este bucle imprimirá infinitamente `"Estoy atrapado dentro de un bucle".` En la pantalla.

NOTA

Si deseas obtener la mejor experiencia de aprendizaje al ver cómo se comporta un bucle infinito, inicia IDLE, crea un nuevo archivo, copia y pega el código anterior, guarda tu archivo y ejecuta el programa. Lo que verás es la secuencia interminable de cadenas impresas de `"Estoy atrapado dentro de un bucle".` en la ventana de la consola de Python. Para finalizar tu programa, simplemente presiona *Ctrl-C* (o *Ctrl-Break* en algunas computadoras). Esto provocará la excepción `KeyboardInterrupt` y permitirá que tu programa salga del bucle. Hablaremos de ello más adelante en el curso.

Volvamos al bosquejo del algoritmo que te mostramos recientemente. Te mostraremos como usar este bucle recién aprendido para encontrar el número más grande de un gran conjunto de datos ingresados.
Analiza el programa cuidadosamente. Localiza donde comienza el bucle (línea 8) y descubre **como se sale del cuerpo del bucle**:

```
# Almacena el actual número más grande aquí.
largest_number = -999999999
# Ingresa el primer valor.
number = int(input("Introduce un número o escribe -1 para detener: "))
# Si el número no es igual a -1, continuaremos
while number != -1:
    # ¿Es el número más grande que el valor de largest_number?
    if number > largest_number:
        # Sí si, se actualiza largest_number.
        largest_number = number
    # Ingresa el siguiente número.
    number = int(input("Introduce un número o escribe -1 para detener: "))

# Imprime el número más grande
print("El número más grande es:", largest_number)
```

Comprueba como este código implementa el algoritmo que te mostramos anteriormente.

El bucle while: más ejemplos

Veamos otro ejemplo utilizando el bucle `while`. Sigue los comentarios para descubrir la idea y la solución.

```
# Un programa que lee una secuencia de números
# y cuenta cuántos números son pares y cuántos son impares.
# El programa termina cuando se ingresa un cero.
odd_numbers = 0
even_numbers = 0

# Lee el primer número.
number = int(input("Introduce un número o escribe 0 para detener: "))

# 0 termina la ejecución.
while number != 0:
    # Verificar si el número es impar.
    if number % 2 == 1:
        # Incrementar el contador de números impares odd_numbers.
```

```
        odd_numbers += 1
    else:
        # Incrementar el contador de números pares even_numbers.
        even_numbers += 1
    # Leer el siguiente número.
    number = int(input("Introduce un número o escribe 0 para detener: "))

# Imprimir resultados.
print("Cuenta de números impares:", odd_numbers)
print("Cuenta de números pares:", even_numbers)
```

Ciertas expresiones se pueden simplificar sin cambiar el comportamiento del programa.
Intenta recordar cómo Python interpreta la verdad de una condición y ten en cuenta que estas dos formas son equivalentes:
`while number != 0:` y `while number:`
La condición que verifica si un número es impar también puede codificarse en estas formas equivalentes:
`if number % 2 == 1:` y `if number % 2:`

Empleando una variable counter para salir del bucle

Observa el fragmento de código a continuación:

```
counter = 5
while counter != 0:
    print("Dentro del bucle.", counter)
    counter -= 1
print("Fuera del bucle.", counter)
```

Este código está destinado a imprimir la cadena `"Dentro del bucle."` y el valor almacenado en la variable `counter` durante un bucle dado exactamente cinco veces. Una vez que la condición se haya cumplido (la variable `counter` ha alcanzado `0`), se sale del bucle y aparece el mensaje `"Fuera del bucle".` así como tambien el valor almacenado en `counter` se imprime.
Pero hay una cosa que se puede escribir de forma más compacta: la condición del bucle `while`.
¿Puedes ver la diferencia?

```
counter = 5
while counter:
    print("Dentro del bucle.", counter)
    counter -= 1
print("Fuera del bucle.", counter)
```

¿Es más compacto que antes? Un poco. ¿Es más legible? Eso es discutible.
RECUERDA

No te sientas obligado a codificar tus programas de una manera que siempre sea la más corta y la más compacta. La legibilidad puede ser un factor más importante. Manten tu código listo para un nuevo programador.

Bucles en tu código con for

Otro tipo de bucle disponible en Python proviene de la observación de que a veces es más importante **contar los "giros o vueltas" del bucle** que verificar las condiciones.

Imagina que el cuerpo de un bucle debe ejecutarse exactamente cien veces. Si deseas utilizar el bucle `while` para hacerlo, puede tener este aspecto:

```
i = 0
```

```
while i < 100:
    # do_something()
    i += 1
```

Sería bueno si alguien pudiera hacer esta cuenta aburrida por ti. ¿Es eso posible?

Por supuesto que lo es, hay un bucle especial para este tipo de tareas, y se llama `for`.

En realidad, el bucle `for` está diseñado para realizar tareas más complicadas, **puede "explorar" grandes colecciones de datos elemento por elemento**. Te mostraremos como hacerlo pronto, pero ahora presentaremos una variante más sencilla de su aplicación.

Echa un vistazo al fragmento:

```
for i in range(100):
    # do_something()
    pass
```

Existen algunos elementos nuevos. Déjanos contarte sobre ellos:

- La palabra reservada *for* abre el bucle `for`; nota - No hay condición después de eso; no tienes que pensar en las condiciones, ya que se verifican internamente, sin ninguna intervención.
- Cualquier variable después de la palabra reservada *for* es la **variable de control** del bucle; cuenta los giros del bucle y lo hace automáticamente.
- La palabra reservada *in* introduce un elemento de sintaxis que describe el rango de valores posibles que se asignan a la variable de control.
- La función `range()` (esta es una función muy especial) es responsable de generar todos los valores deseados de la variable de control; en nuestro ejemplo, la función creará (incluso podemos decir que **alimentará** el bucle con) valores subsiguientes del siguiente conjunto: 0, 1, 2 .. 97, 98, 99; nota: en este caso, la función `range()` comienza su trabajo desde 0 y lo finaliza un paso (un número entero) antes del valor de su argumento.
- Nota la palabra clave *pass* dentro del cuerpo del bucle - no hace nada en absoluto; es una **instrucción vacía** : la colocamos aquí porque la sintaxis del bucle `for` exige al menos una instrucción dentro del cuerpo (por cierto, `if`, `elif`, `else` y `while` expresan lo mismo).

Nuestros próximos ejemplos serán un poco más modestos en el número de repeticiones de bucle.

Echa un vistazo al fragmento de abajo. ¿Puedes predecir su salida?

```
for i in range(10):
    print("El valor de i es actualmente", i)
```

Ejecuta el código para verificar si tenías razón.
Nota:

- El bucle se ha ejecutado diez veces (es el argumento de la función `range()`).
- El valor de la última variable de control es `9` (no `10`, ya que **comienza desde** `0` , no desde `1`).

La invocación de la función `range()` puede estar equipada con dos argumentos, no solo uno:

```
for i in range(2, 8):
    print("El valor de i es actualmente", i)
```

En este caso, el primer argumento determina el valor inicial (primero) de la variable de control.
El último argumento muestra el primer valor que no se asignará a la variable de control.
Nota: la función `range()` **solo acepta enteros como argumentos** y genera secuencias de enteros.
¿Puedes adivinar la salida del programa? Ejecútalo para comprobar si ahora también estabas en lo cierto.
El primer valor mostrado es `2` (tomado del primer argumento de `range()`).
El último es `7` (aunque el segundo argumento de `range()` es `8`).

Más sobre el bucle for y la función range() con tres argumentos

La función `range()` también puede aceptar **tres argumentos**: Echa un vistazo al código del editor.

El tercer argumento es un **incremento**: es un valor agregado para controlar la variable en cada giro del bucle (como puedes sospechar, el valor predeterminado del incremento **es 1**).
¿Puedes decirnos cuántas líneas aparecerán en la consola y qué valores contendrán?

```
for i in range(2, 8, 3):
    print("El valor de i es actualmente", i)
```

Ejecuta el programa para averiguar si tenías razón.

Deberías poder ver las siguientes líneas en la ventana de la consola:

```
El valor de i es actualmente 2
El valor de i es actualmente 5
```

¿Sabes por qué? El primer argumento pasado a la función `range()` nos dice cual es el número **de inicio** de la secuencia (por lo tanto, `2` en la salida). El segundo argumento le dice a la función dónde **detener** la secuencia (la función genera números hasta el número indicado por el segundo argumento, pero no lo incluye). Finalmente, el tercer argumento indica el **paso**, que en realidad significa la diferencia entre cada número en la secuencia de números generados por la función.
`2` (número inicial) → `5` (`2` incremento por 3 es igual a `5` - el número está dentro del rango de 2 a 8)
→ `8` (`5` incremento por 3 es igual a `8` - el número no está dentro del rango de 2 a 8, porque el parámetro de parada no está incluido en la secuencia de números generados por la función).

Nota: si el conjunto generado por la función `range()` está vacío, el bucle no ejecutará su cuerpo en absoluto.
Al igual que aquí, no habrá salida:

```
for i in range(1, 1):
    print("El valor de i es actualmente", i)
```

Nota: el conjunto generado por `range()` debe ordenarse en **un orden ascendente**. No hay forma de forzar el `range()` para crear un conjunto en una forma diferente. Esto significa que el segundo argumento de `range()` debe ser mayor que el primero.
Por lo tanto, tampoco habrá salida aquí:

```
for i in range(2, 1):
    print("El valor de i es actualmente", i)
```

Echemos un vistazo a un programa corto cuya tarea es escribir algunas de las primeras potencias de dos:

```
power = 1
for expo in range(16):
    print("2 a la potencia de", expo, "es", power)
    power *= 2
```

La variable `expo` se utiliza como una variable de control para el bucle e indica el valor actual del *exponente*. La propia exponenciación se sustituye multiplicando por dos. Dado que 2^0 es igual a 1, después 2×1 es igual a 2^1, 2×2^1 es igual a 2^2, y así sucesivamente. ¿Cuál es el máximo exponente para el cual nuestro programa aún imprime el resultado?
Ejecuta el código y verifica si la salida coincide con tus expectativas.

Las sentencias break y continue

Hasta ahora, hemos tratado el cuerpo del bucle como una secuencia indivisible e inseparable de instrucciones que se realizan completamente en cada giro del bucle. Sin embargo, como desarrollador, podrías enfrentar las siguientes opciones:

- Parece que no es necesario continuar el bucle en su totalidad; se debe abstener de seguir ejecutando el cuerpo del bucle e ir más allá.
- Parece que necesitas comenzar el siguiente giro del bucle sin completar la ejecución del turno actual.

Python proporciona dos instrucciones especiales para la implementación de estas dos tareas. Digamos por razones de precisión que su existencia en el lenguaje no es necesaria: un programador experimentado puede codificar

cualquier algoritmo sin estas instrucciones. Tales adiciones, que no mejoran el poder expresivo del lenguaje, sino que solo simplifican el trabajo del desarrollador, a veces se denominan **dulces sintácticos** o azúcar sintáctica.

Estas dos instrucciones son:

- `break`: sale del bucle inmediatamente, e incondicionalmente termina la operación del bucle; el programa comienza a ejecutar la instrucción más cercana después del cuerpo del bucle.
- `continue`: se comporta como si el programa hubiera llegado repentinamente al final del cuerpo; el siguiente turno se inicia y la expresión de condición se prueba de inmediato.

Ambas palabras son **palabras clave reservadas**.

Ahora te mostraremos dos ejemplos simples para ilustrar como funcionan las dos instrucciones. Mira el código en el editor. Ejecuta el programa y analiza la salida. Modifica el código y experimenta.

```
# break - ejemplo

print("La instrucción break:")

for i in range(1, 6):

    if i == 3:

        break

    print("Dentro del bucle.", i)

print("Fuera del bucle.")

# continue - ejemplo

print("\nLa instrucción continue:")

for i in range(1, 6):

    if i == 3:

        continue

    print("Dentro del bucle.", i)

print("Fuera del bucle.")
```

Las sentencias break y continue: más ejemplos

Regresemos a nuestro programa que reconoce el más grande entre los números ingresados. Lo convertiremos dos veces, usando las instrucciones de `break` y `continue`.

Analiza el código y determina como las usarías.

La variante empleando `break` es la siguiente:

```
largest_number = -99999999

counter = 0
```

```
while True:
    number = int(input("Ingresa un número o escribe -1 para finalizar el programa: "))
    if number == -1:
        break
    counter += 1
    if number > largest_number:
        largest_number = number
if counter != 0:
    print("El número más grande es", largest_number)
else:
    print("No has ingresado ningún número.")
```

Ejecútalo, pruébalo y experimenta con él.

Y ahora la variante con `continue`:

```
largest_number = -99999999
counter = 0
number = int(input("Ingresa un número o escribe -1 para finalizar el programa: "))
while number != -1:
    if number == -1:
        continue
    counter += 1
    if number > largest_number:
        largest_number = number
    number = int(input("Ingresa un número o escribe -1 para finalizar el programa: "))
if counter:
    print("El número más grande es", largest_number)
else:
    print("No has ingresado ningún número.")
```

Observa con atención, el usuario ingresa el primer número **antes** de que el programa ingrese al bucle `while`. El número siguiente se ingresa cuando el programa **ya está en el bucle**.

De nuevo: ejecútalo, pruébalo y experimenta con él.

El bucle while y la rama else

Ambos bucles `while` y `for`, tienen una característica interesante (y rara vez se usa).
Te mostraremos como funciona: intenta juzgar por ti mismo si es utilizable.
En otras palabras, trata de convencerte si la función es valiosa y útil, o solo es azúcar sintáctica.

Echa un vistazo al fragmento en el editor.

```
i = 1
while i < 5:
    print(i)
    i += 1
else:
    print("else:", i)
```

Hay algo extraño al final: la palabra reservada `else`.

Como pudiste haber sospechado, los bucles **también pueden tener la rama** `else`**, como los** `if`.

La rama `else` del bucle **siempre se ejecuta una vez, independientemente de si el bucle ha entrado o no en su cuerpo**.

¿Puedes adivinar la salida? Ejecuta el programa para comprobar si tenías razón.

Modifica el fragmento un poco para que el bucle no tenga oportunidad de ejecutar su cuerpo ni una sola vez:

```
i = 5
while i < 5:
    print(i)
    i += 1
else:
    print("else:", i)
```

El estado del `while` es `False` al principio, ¿puedes verlo?

Ejecuta y prueba el programa, y verifica si se ha ejecutado o no la rama `else`.

El bucle for y la rama else

Los bucles `for` se comportan de manera un poco diferente: echa un vistazo al fragmento en el editor y ejecútalo.

La salida puede ser un poco sorprendente.

La variable `i` conserva su último valor.

```
for i in range(5):
    print(i)
else:
    print("else:", i)
```

Modifica el código un poco para realizar un experimento más.

```
i = 111
for i in range(2, 1):
    print(i)
else:
    print("else:", i)
```

¿Puedes adivinar la salida?

El cuerpo del bucle no se ejecutará aquí en absoluto. Nota: hemos asignado la variable `i` antes del bucle.

Ejecuta el programa y verifica su salida.

Cuando el cuerpo del bucle no se ejecuta, la variable de control conserva el valor que tenía antes del bucle.

Nota: **si la variable de control no existe antes de que comience el bucle, no existirá cuando la ejecución llegue a la rama** `else`.

¿Cómo te sientes acerca de esta variante de `else`?

Ahora vamos a informarte sobre otros tipos de variables. Nuestras variables actuales solo pueden **almacenar un valor a la vez**, pero hay variables que pueden hacer mucho más; pueden **almacenar tantos valores como desees**.

Puntos Clave

1. Existen dos tipos de bucles en Python: `while` y `for`:
   - El bucle `while` ejecuta una sentencia o un conjunto de sentencias siempre que una condición booleana especificada sea verdadera, por ejemplo:

```
# Ejemplo 1
while True:
    print("Atascado en un bucle infinito.")

# Ejemplo 2
counter = 5
while counter > 2:
    print(counter)
    counter -= 1
```

- El bucle `for` ejecuta un conjunto de sentencias muchas veces; se usa para iterar sobre una secuencia (por ejemplo, una lista, un diccionario, una tupla o un conjunto; pronto aprenderás sobre ellos) u otros objetos que son iterables (por ejemplo, cadenas). Puedes usar el bucle `for` para iterar sobre una secuencia de números usando la función incorporada `range`. Mira los ejemplos a continuación:

```
# Ejemplo 1
word = "Python"
for letter in word:
    print(letter, end="*")

# Ejemplo 2
for i in range(1, 10):
    if i % 2 == 0:
        print(i)
```

2. Puedes usar las sentencias `break` y `continue` para cambiar el flujo de un bucle:

- Utiliza `break` para salir de un bucle, por ejemplo:

```
text = "OpenEDG Python Institute"
for letter in text:
    if letter == "P":
        break
    print(letter, end="")
```

- Utiliza `continue` para omitir la iteración actual, y continuar con la siguiente iteración, por ejemplo:

```
text = "pyxpyxpyx"
for letter in text:
    if letter == "x":
        continue
    print(letter, end="")
```

3. Los bucles `while` y `for` también pueden tener una cláusula `else` en Python. La cláusula `else` se ejecuta después de que el bucle finalice su ejecución siempre y cuando no haya terminado con `break`, por ejemplo:

```
n = 0

while n != 3:
    print(n)
    n += 1
```

```
else:
    print(n, "else")

print()

for i in range(0, 3):
    print(i)
else:
    print(i, "else")
```

4. La función `range()` genera una secuencia de números. Acepta enteros y devuelve objetos de rango. La sintaxis de `range()` tiene el siguiente aspecto: `range(start, stop, step)`, donde:

- `start` es un parámetro opcional que especifica el número de inicio de la secuencia (0 por defecto)
- `stop` es un parámetro opcional que especifica el final de la secuencia generada (no está incluido).
- y `step` es un parámetro opcional que especifica la diferencia entre los números en la secuencia es (1 por defecto.)

Código de ejemplo:

```
for i in range(3):
    print(i, end=" ")  # Salidas: 0 1 2

for i in range(6, 1, -2):
    print(i, end=" ")  # Salidas: 6, 4, 2
```

Puntos Clave: continuación

Ejercicio 1

Crea un bucle `for` que cuente de 0 a 10, e imprima números impares en la pantalla. Usa el esqueleto de abajo:

```
for i in range(1, 11):
    # Línea de código.
        # Línea de código.
```

Revisar

Solución de muestra:

```
for i in range(0, 11):
    if i % 2 != 0:
        print(i)
```

Ejercicio 2

Crea un bucle `while` que cuente de 0 a 10, e imprima números impares en la pantalla. Usa el esqueleto de abajo:

```
x = 1
while x < 11:
    # Línea de código.
        # Línea de código.
    # Línea de código.
```

Ejercicio 3

Crea un programa con un bucle `for` y una sentencia `break`. El programa debe iterar sobre los caracteres en una dirección de correo electrónico, salir del bucle cuando llegue al símbolo `@` e imprimir la parte antes de `@` en una línea. Usa el esqueleto de abajo:

```
for ch in "john.smith@pythoninstitute.org":
    if ch == "@":
        # Línea de código.
    # Línea de código.
```

Revisar
Solución de muestra:

```
for ch in "john.smith@pythoninstitute.org":
    if ch == "@":
        break
    print(ch, end="")
```

Ejercicio 4
Crea un programa con un bucle `for` y una sentencia `continue`. El programa debe iterar sobre una cadena de dígitos, reemplazar cada `0` con `x`, e imprimir la cadena modificada en la pantalla. Usa el esqueleto de abajo:

```
for digit in "0165031806510":
    if digit == "0":
        # Línea de código.
        # Línea de código.
    # Línea de código.
```

Revisar
Solución de muestra:

```
for digit in "0165031806510":
    if digit == "0":
        print("x", end="")
        continue
    print(digit, end="")
```

Lógica de computadora
¿Te has dado cuenta de que las condiciones que hemos usado hasta ahora han sido muy simples, por no decir, bastante primitivas? Las condiciones que utilizamos en la vida real son mucho más complejas. Veamos este enunciado:
Si tenemos tiempo libre, y el clima es bueno, saldremos a caminar.

Hemos utilizado la conjunción `and` (y), lo que significa que salir a caminar depende del cumplimiento simultáneo de estas dos condiciones. En el lenguaje de la lógica, tal conexión de condiciones se denomina **conjunción**. Y ahora otro ejemplo:
Si tu estás en el centro comercial o yo estoy en el centro comercial, uno de nosotros le comprará un regalo a mamá.

La aparición de la palabra `or` (o) significa que la compra depende de al menos una de estas condiciones. En lógica, este compuesto se llama una **disyunción**.
Está claro que Python debe tener operadores para construir conjunciones y disyunciones. Sin ellos, el poder expresivo del lenguaje se debilitaría sustancialmente. Se llaman **operadores lógicos**.
and
Un operador de conjunción lógica en Python es la palabra *and*. Es un operador binario **con una prioridad inferior a la expresada por los operadores de comparación**. Nos permite codificar condiciones complejas sin el uso de paréntesis como este:

```
counter > 0 and value == 100
```

El resultado proporcionado por el operador `and` se puede determinar sobre la base de la **tabla de verdad**.

Si consideramos la conjunción de `A and B`, el conjunto de valores posibles de argumentos y los valores correspondientes de conjunción se ve de la siguiente manera:

Argumento A	**Argumento** B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

or

Un operador de disyunción es la palabra `or`. Es un operador binario **con una prioridad más baja que** `and` (al igual que `+` en comparación con `*`). Su tabla de verdad es la siguiente:

Argumento A	**Argumento** B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

not

Además, hay otro operador que se puede aplicar para condiciones de construcción. Es un **operador unario que realiza una negación lógica**. Su funcionamiento es simple: convierte la verdad en falso y lo falso en verdad
Este operador se escribe como la palabra `not`, y su **prioridad es muy alta: igual que el unario** `+` **y** `-`. Su tabla de verdad es simple:

Argumento	not **Argumento**
False	True
True	False

Expresiones lógicas

Creemos una variable llamada `var` y asignémosle `1`. Las siguientes condiciones son **equivalentes a pares**:

```
# Ejemplo 1:
print(var > 0)
print(not (var <= 0))


# Ejemplo 2:
print(var != 0)
print(not (var == 0))
```

Puedes estar familiarizado con las leyes de De Morgan. Dicen que:
La negación de una conjunción es la separación de las negaciones.
La negación de una disyunción es la conjunción de las negaciones.

Escribamos lo mismo usando Python:

```
not (p and q) == (not p) or (not q)
not (p or q) == (not p) and (not q)
```

Observa como se han utilizado los paréntesis para codificar las expresiones: las colocamos allí para mejorar la legibilidad.
Deberíamos agregar que ninguno de estos operadores de dos argumentos se puede usar en la forma abreviada conocida como `op=`. Vale la pena recordar esta excepción.

Valores lógicos frente a bits individuales

Los operadores lógicos toman sus argumentos como un todo, independientemente de cuantos bits contengan. Los operadores solo conocen el valor: cero (cuando todos los bits se restablecen) significa `False`; no cero (cuando se establece al menos un bit) significa `True`.

El resultado de sus operaciones es uno de estos valores: `False` o `True`. Esto significa que este fragmento de código asignará el valor `True` a la variable `j` si `i` no es cero; de lo contrario, será `False`.

```
i = 1

j = not not i
```

Operadadores bit a bit

Sin embargo, hay cuatro operadores que le permiten **manipular bits de datos individuales**. Se denominan **operadores bit a bit**.

Cubren todas las operaciones que mencionamos anteriormente en el contexto lógico, y un operador adicional. Este es el operador `xor` (significa **o exclusivo**), y se denota como `^` (signo de intercalación).

Aquí están todos ellos:

- `&` (ampersand) - conjunción a nivel de bits.
- `|` (barra vertical) - disyunción a nivel de bits.
- `~` (tilde) - negación a nivel de bits.
- `^` (signo de intercalación) - o exclusivo a nivel de bits (xor).

Operaciones bit a bit (`&`, `|`, y `^`)

Argumento A	**Argumento** B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Operaciones bit a bit (~)

Argumento	~ **Argumento**
0	1

```
1                    0
```

Hagámoslo más fácil:

- `&` requieres exactamente dos `1`s para proporcionar `1` como resultado.
- `|` requiere al menos un `1` para proporcionar `1` como resultado.
- `^` requiere exactamente un `1` para proporcionar `1` como resultado.

Agreguemos un comentario importante: los argumentos de estos operadores **deben ser enteros**. No debemos usar flotantes aquí.

La diferencia en el funcionamiento de los operadores lógicos y de bits es importante: **los operadores lógicos no penetran en el nivel de bits de su argumento**. Solo les interesa el valor entero final.

Los operadores bit a bit son más estrictos: tratan con **cada bit por separado**. Si asumimos que la variable entera ocupa 64 bits (lo que es común en los sistemas informáticos modernos), puede imaginar la operación a nivel de bits como una evaluación de 64 veces del operador lógico para cada par de bits de los argumentos. Su analogía es obviamente imperfecta, ya que en el mundo real todas estas 64 operaciones se realizan al mismo tiempo (simultáneamente).

Operaciones lógicas frente a operaciones de bit: continuación
Ahora te mostraremos un ejemplo de la diferencia en la operación entre las operaciones lógicas y de bit.
Supongamos que se han realizado las siguientes tareas:

```
i = 15
j = 22
```

Si asumimos que los enteros se almacenan con 32 bits, la imagen a nivel de bits de las dos variables será la siguiente:

```
i: 00000000000000000000000000001111
j: 00000000000000000000000000010110
```

Se ejecuta la asignación:

```
log = i and j
```

Estamos tratando con una conjunción lógica aquí. Vamos a trazar el curso de los cálculos. Ambas variables `i` y `j` no son ceros, por lo que se considerará que representan a `True`. Al consultar la tabla de verdad para el operador `and`, podemos ver que el resultado será `True`. No se realizan otras operaciones.

```
log: True
```

Ahora la operación a nivel de bits - aquí está:

```
bit = i & j
```

El operador `&` operará con cada par de bits correspondientes por separado, produciendo los valores de los bits relevantes del resultado. Por lo tanto, el resultado será el siguiente:

```
i                    00000000000000000000000000001111
```

j	00000000000000000000000000010110
bit = i & j	00000000000000000000000000000110

Estos bits corresponden al valor entero de seis.

Veamos ahora los operadores de negación. Primero el lógico:

```
logneg = not i
```

La variable logneg se establecerá en False: no es necesario hacer nada más.
La negación a nivel de bits es así:

```
bitneg = ~i
```

Puede ser un poco sorprendente: el valor de la variable bitneg es -16. Esto puede parecer extraño, pero no lo es en absoluto. Si deseas obtener más información, debes consultar el sistema de números binarios y las reglas que rigen los números de complemento de dos.

i	00000000000000000000000000001111
bitneg = ~i	11111111111111111111111111110000

Cada uno de estos operadores de dos argumentos se puede utilizar en **forma abreviada**. Estos son los ejemplos de sus notaciones equivalentes:

x = x & y	x &= y
x = x | y	x |= y
x = x ^ y	x ^= y

¿Cómo tratamos los bits individuales?

```
var = 17
var_right = var >> 1
var_left = var << 2
print(var, var_left, var_right)
```

Ahora te mostraremos para que puedes usar los operadores de bit a bit. Imagina que eres un desarrollador obligado a escribir una pieza importante de un sistema operativo. Se te ha dicho que puedes usar una variable asignada de la siguiente forma:

```
flag_register = 0x1234
```

La variable almacena la información sobre varios aspectos de la operación del sistema. **Cada bit de la variable almacena un valor de si/no**. También se te ha dicho que solo uno de estos bits es tuyo, el tercero (recuerda que los bits se numeran desde cero y el número de bits cero es el más bajo, mientras que el más alto es el número 31). Los bits restantes no pueden cambiar, porque están destinados a almacenar otros datos. Aquí está tu bit marcado con la letra x:

```
flag_register = 0000000000000000000000000000x000
```

Es posible que tengas que hacer frente a las siguientes tareas:
1. **Comprobar el estado de tu bit**: deseas averiguar el valor de su bit; comparar la variable completa con cero no hará nada, porque los bits restantes pueden tener valores completamente impredecibles, pero puedes usar la siguiente propiedad de conjunción:

```
x & 1 = x
x & 0 = 0
```

Si aplicas la operación `&` a la variable `flagRegister` junto con la siguiente imagen de bits:
`00000000000000000000000000001000`
(observa el `1` en la posición de tu bit) como resultado, obtendrás una de las siguientes cadenas de bits:

- `00000000000000000000000000001000` si tu bit se estableció en `1`
- `00000000000000000000000000000000` si tu bit se reseteo a `0`

Dicha secuencia de ceros y unos, cuya tarea es tomar el valor o cambiar los bits seleccionados, se denomina **máscara de bits**.
Construyamos una máscara de bits para detectar el estado de tus bits. Debería apuntar a **el tercer bit**. Ese bit tiene el peso de 2^3=8. Se podría crear una máscara adecuada mediante la siguiente sentencia:

```
the_mask = 8
```

También puedes hacer una secuencia de instrucciones dependiendo del estado de tu bit, aquí está:

```
if flag_register & the_mask:
   # Mi bit se estableció en 1.
else:
   # Mi bit se restableció a 0.
```

2. **Reinicia tu bit**: asigna un cero al bit, mientras que todos los otros bits deben permanecer sin cambios; usemos la misma propiedad de la conjunción que antes, pero usemos una máscara ligeramente diferente, exactamente como se muestra a continuación:
`11111111111111111111111111110111`
Tenga en cuenta que la máscara se creó como resultado de la negación de todos los bits de la variable `the_mask`.
Restablecer el bit es simple, y se ve así (elige el que más te guste):

```
flag_register = flag_register & ~the_mask
flag_register &= ~the_mask
```

3. **Establece tu bit** : asigna un `1` a tu bit, mientras que todos los bits restantes deben permanecer sin cambios; usa la siguiente propiedad de disyunción:

```
x | 1 = 1
x | 0 = x
```

Ya estás listo para configurar su bit con una de las siguientes instrucciones:

```
flag_register = flag_register | the_mask
flag_register |= the_mask
```

4. **Niega tu bit**: reemplaza un `1` con un `0` y un `0` con un `1`. Puedes utilizar una propiedad interesante del operador `~x`:

```
x ^ 1 = ~x
x ^ 0 = x
```

Niega tu bit con las siguientes instrucciones:

```
flag_register = flag_register ^ the_mask
flag_register ^= the_mask
```

Desplazamiento izquierdo binario y desplazamiento derecho binario
Python ofrece otra operación relacionada con los bits individuales: **shifting**. Esto se aplica solo a los valores de **número entero**, y no debe usar flotantes como argumentos para ello
Ya aplicas esta operación muy a menudo y muy inconscientemente. ¿Cómo multiplicas cualquier número por diez? Echa un vistazo:
12345 × 10 = 123450

Como puede ver, **multiplicar por diez es de hecho un desplazamiento** de todos los dígitos a la izquierda y llenar el vacío resultante con cero.
¿División entre diez? Echa un vistazo:
12340 ÷ 10 = 1234
Dividir entre diez no es más que desplazar los dígitos a la derecha.

La computadora realiza el mismo tipo de operación, pero con una diferencia: como dos es la base para los números binarios (no 10), **desplazar un valor un bit a la izquierda corresponde a multiplicarlo por dos**; respectivamente, **desplazar un bit a la derecha es como dividir entre dos** (observe que se pierde el bit más a la derecha).
Los **operadores de cambio** en Python son un par de **dígrafos**: `< <` y `> >`, sugiriendo claramente en qué dirección actuará el cambio.

```
value << bits
value >> bits
```

El argumento izquierdo de estos operadores es un valor entero cuyos bits se desplazan. El argumento correcto determina el tamaño del turno.
Esto demuestra que esta operación ciertamente no es conmutativa
La prioridad de estos operadores es muy alta. Los verás en la tabla de prioridades actualizada, que te mostraremos al final de esta sección.

Echa un vistazo a los cambios en la ventana del editor.
La invocación final de `print()` produce el siguiente resultado:

```
17 68 8
```

salida

Nota:

- `17 >> 1` → `17 // 2` (**17** dividido entre **2 a la potencia de 1**) → `8` (desplazarse hacia la derecha en un bit equivale a la división entera entre dos)
- `17 << 2` → `17 * 4` (**17** multiplicado por **2 a la potencia de 2**) → `68` (desplazarse hacia la izquierda dos bits es lo mismo que multiplicar números enteros por cuatro)

Y aquí está la **tabla de prioridades actualizada** , que contiene todos los operadores presentados hasta ahora:

Prioridad	Operador	
1	`~`, `+`, `-`	unario
2	`**`	
3	`*`, `/`, `//`, `%`	
4	`+`, `-`	binario
5	`<<`, `>>`	
6	`<`, `<=`, `>`, `>=`	
7	`==`, `!=`	
8	`&`	
9	`|`	
10	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `>>=`, `<<=`	

Puntos Clave

1. Python es compatible con los siguientes operadores lógicos:
   - `and` → si ambos operandos son verdaderos, la condición es verdadera, por ejemplo, `(True and True)` es `True`.
   - `or` → si alguno de los operandos es verdadero, la condición es verdadera, por ejemplo, `(True or False)` es `True`.
   - `not` → devuelve False si el resultado es verdadero y devuelve True si es falso, por ejemplo, `not True` es `False`.

2. 2. Puedes utilizar operadores bit a bit para manipular bits de datos individuales. Los siguientes datos de muestra:

- `x = 15`, which is `0000 1111` en binario.
- `y = 16`, which is `0001 0000` en binario.

Se utilizarán para ilustrar el significado de operadores bit a bit en Python. Analiza los ejemplos a continuación::

- `&` hace un *bit a bit and (y)*, por ejemplo, `x & y = 0`, el cual es `0000 0000` en binario.
- `|` hace un *bit a bit or (o)*, por ejemplo, `x | y = 31`, el cual es `0001 1111` en binario.
- `~` hace un *bit a bit not (no)*, por ejemplo, `~ x = 240`, el cual es `1111 0000` en binario.
- `^` hace un *bit a bit xor*, por ejemplo, `x ^ y = 31`, el cual es `0001 1111` en binario.
- `>>` hace un *desplazamiento bit a bit a la derecha*, por ejemplo, `y >> 1 = 8`, el cual es `0000 1000` en binario.
- `<<` hace un *desplazamiento bit a bit a la izquierda*, por ejemplo, `y << 3 =` , el cual es `1000 0000` en binario.

* `-16` (decimal del complemento a 2 con signo) -- lee más acerca de la operación Complemento a dos.

Ejercicio 1

¿Cuál es la salida del siguiente fragmento de código?

```
x = 1
y = 0

z = ((x == y) and (x == y)) or not(x == y)
print(not(z))
```

Revisar

Ejercicio 2

¿Cuál es la salida del siguiente fragmento de código?

```
x = 4
y = 1

a = x & y
b = x | y
c = ~x  # !difícil!
d = x ^ 5
e = x >> 2
f = x << 2

print(a, b, c, d, e, f)
```

Revisar: `0 5 -5 1 1 16`

¿Por qué necesitamos listas?

Puede suceder que tengas que leer, almacenar, procesar y, finalmente, imprimir docenas, quizás cientos, tal vez incluso miles de números. ¿Entonces qué? ¿Necesitas crear una variable separada para cada valor? ¿Tendrás que pasar largas horas escribiendo sentencias como la que se muestra a continuación?

```
var1 = int(input())
var2 = int(input())
var3 = int(input())
var4 = int(input())
var5 = int(input())
var6 = int(input())
```

:
:
Si no crees que esta sea una tarea complicada, toma un papel y escribe un programa que:

- Lea cinco números.
- Los imprima en orden desde el más pequeño hasta el más grande (Este tipo de procesamiento se denomina **ordenamiento**).

Debes percatarte que ni siquiera tienes suficiente papel para completar la tarea.
Hasta ahora, has aprendido como declarar variables que pueden almacenar exactamente un valor dado a la vez. Tales variables a veces se denominan **escalares** por analogía con las matemáticas. Todas las variables que has usado hasta ahora son realmente escalares.
Piensa en lo conveniente que sería declarar una variable que podría **almacenar más de un valor**. Por ejemplo, cien, o mil o incluso diez mil. Todavía sería una y la misma variable, pero muy amplia y espaciosa. ¿Suena atractivo? Quizás, pero ¿cómo manejarías un contenedor así lleno de valores diferentes? ¿Cómo elegirías solo el que necesitas?
¿Y si solo pudieras numerarlos? Y luego di: *dame el valor número 2; asigna el valor número 15; aumenta el número del valor 10000*.
Te mostraremos como declarar tales **variables de múltiples valores**. Haremos esto con el ejemplo que acabamos de sugerir. Escribiremos un programa **que ordene una secuencia de números**. No seremos particularmente ambiciosos: asumiremos que hay exactamente cinco números.
Vamos a crear una variable llamada `numeros`; se le asigna no solo un número, sino que se llena con una lista que consta de cinco valores (nota: la lista **comienza con un corchete abierto y termina con un corchete cerrado**; el espacio entre los corchetes es llenado con cinco números separados por comas).

```
numbers = [10, 5, 7, 2, 1]
```

Digamos lo mismo utilizando una terminología adecuada: `numeros` **es una lista que consta de cinco valores, todos ellos números**. También podemos decir que esta sentencia crea una lista de longitud igual a cinco (ya que contiene cinco elementos).
Los elementos dentro de una lista **pueden tener diferentes tipos**. Algunos de ellos pueden ser enteros, otros son flotantes y otros pueden ser listas.
Python ha adoptado una convención que indica que los elementos de una lista están **siempre numerados desde cero**. Esto significa que el elemento almacenado al principio de la lista tendrá el número cero. Como hay cinco elementos en nuestra lista, al último de ellos se le asigna el número cuatro. No olvides esto.
Pronto te acostumbrarás y se convertirá en algo natural.
Antes de continuar con nuestra discusión, debemos indicar lo siguiente: nuestra lista **es una colección de elementos, pero cada elemento es un escalar**.

Indexando Listas

```
numbers = [10, 5, 7, 2, 1]
print("Contenido de la lista:", numbers)  # Imprimiendo contenido de la lista original.
```

¿Cómo cambias el valor de un elemento elegido en la lista?
Vamos a **asignar un nuevo valor de** `111` **al primer elemento** en la lista. Lo hacemos de esta manera:

```
numbers = [10, 5, 7, 2, 1]
print("Contenido de la lista original:", numbers)  # Imprimiendo contenido de la lista original.
numbers[0] = 111
print("Nuevo contenido de la lista: ", numbers)  # Contenido de la lista actual.
```

Y ahora queremos **copiar el valor del quinto elemento al segundo elemento**. ¿Puedes adivinar cómo hacerlo?

```
numbers = [10, 5, 7, 2, 1]
print("Contenido de la lista original:", numbers)  # Imprimiendo contenido de la lista original.
numbers[0] = 111
print("\nPrevio contenido de la lista:", numbers)  # Imprimiendo contenido de la lista anterior.
numbers[1] = numbers[4]  # Copiando el valor del quinto elemento al segundo elemento.
print("Nuevo contenido de la lista:", numbers)  # Imprimiendo el contenido de la lista actual.
```

El valor dentro de los corchetes que selecciona un elemento de la lista se llama un **índice**, mientras que la operación de seleccionar un elemento de la lista se conoce como **indexación**.
Vamos a utilizar la función `print()` para imprimir el contenido de la lista cada vez que realicemos los cambios. Esto nos ayudará a seguir cada paso con más cuidado y ver que sucede después de una modificación de la lista en particular.

Nota: todos los índices utilizados hasta ahora son literales. Sus valores se fijan en el tiempo de ejecución, pero **cualquier expresión también puede ser un índice**. Esto abre muchas posibilidades.

Accediendo al contenido de la lista

Se puede acceder a cada uno de los elementos de la lista por separado. Por ejemplo, se puede imprimir:

```
print(numbers[0]) # Accediendo al primer elemento de la lista.
```

Suponiendo que todas las operaciones anteriores se hayan completado con éxito, el fragmento enviará `111` a la consola.

Como puedes ver en el editor, la lista también puede imprimirse como un todo, como aquí:

```
print(numbers)  # Imprimiendo la lista completa.
```

Como probablemente hayas notado antes, Python decora la salida de una manera que sugiere que todos los valores presentados forman una lista. La salida del fragmento de ejemplo anterior se ve así:

```
[111, 1, 7, 2, 1]
```

output

La función len()

```
numbers = [10, 5, 7, 2, 1]
print("Contenido de la lista original:", numbers)  # Imprimiendo el contenido de la lista original.
numbers[0] = 111
print("\nContenido de la lista con cambio:", numbers)  # Imprimiendo contenido de la lista con 111.
numbers[1] = numbers[4]  # Copiando el valor del quinto elemento al segundo elemento.
print("Contenido de la lista con intercambio:", numbers)  # Imprimiendo contenido de la lista con intercambio.
print("\nLongitud de la lista:", len(numbers))  # Imprimiendo la longitud de la lista.
```

La longitud **de una lista** puede variar durante la ejecución. Se pueden agregar nuevos elementos a la lista, mientras que otros pueden eliminarse de ella. Esto significa que la lista es una entidad muy dinámica.

Si deseas verificar la longitud actual de la lista, puedes usar una función llamada `len()` (su nombre proviene de *length* - *longitud*).

La función toma **el nombre de la lista como un argumento y devuelve el número de elementos almacenados actualmente** dentro de la lista (en otras palabras, la longitud de la lista).

Observa la última línea de código en el editor, ejecuta el programa y verifica que valor imprimirá en la consola. ¿Puedes adivinar?

Eliminando elementos de una lista

```
numbers = [10, 5, 7, 2, 1]
print("Contenido de la lista original:", numbers)  # Imprimiendo el contenido de la lista original.
numbers[0] = 111
print("\nContenido de la lista con cambio:", numbers)  # Imprimiendo contenido de la lista con 111.
numbers[1] = numbers[4]  # Copiando el valor del quinto elemento al segundo elemento.
print("Contenido de la lista con intercambio:", numbers)  # Imprimiendo contenido de la lista con intercambio.
print("\nLongitud de la lista:", len(numbers))  # Imprimiendo la longitud de la lista.
###
del numbers[1]  # Eliminando el segundo elemento de la lista.
print("Longitud de la nueva lista:", len(numbers))  # Imprimiendo nueva longitud de la lista.
print("\nNuevo contenido de la lista:", numbers)  # Imprimiendo el contenido de la lista actual.
###
```

Cualquier elemento de la lista puede ser **eliminado** en cualquier momento, esto se hace con una instrucción llamada `del` (eliminar). Nota: es una instrucción, no una función.

Tienes que apuntar al elemento que quieres eliminar, desaparecerá de la lista y la longitud de la lista se reducirá en uno.

Mira el fragmento de abajo. ¿Puedes adivinar qué salida producirá? Ejecuta el programa en el editor y comprueba.

```
del numbers[1]
```

`print(len(numbers))`
`print(numbers)`

No puedes acceder a un elemento que no existe , no puedes obtener su valor ni asignarle un valor. Ambas instrucciones causarán ahora errores de tiempo de ejecución:

```
print(numbers[4])
numbers[4] = 1
```

Agrega el fragmento de código anterior después de la última línea de código en el editor, ejecute el programa y verifique que sucede.

Nota: hemos eliminado uno de los elementos de la lista; ahora solo hay cuatro elementos en la lista. Esto significa que el elemento número cuatro no existe.

Los índices negativos son válidos

```
numbers = [111, 7, 2, 1]
print(numbers[-1])
print(numbers[-2])
```

Puede parecer extraño, pero los índices negativos son válidos y pueden ser muy útiles.

Un elemento con un índice igual a `-1` es **el último en la lista**.

`print(numbers[-1])`

El código del ejemplo mostrará `1`. Ejecuta el programa y comprueba.

Del mismo modo, el elemento con un índice igual a `-2` es **el anterior al último en la lista**.

`print(numbers[-2])`

El fragmento de ejemplo dará como salida un `2`.

El último elemento accesible en nuestra lista es `numeros[-4]` (el primero). ¡No intentes ir más lejos!

Funciones frente a métodos

Un método **es un tipo específico de función**: se comporta como una función y se parece a una función, pero difiere en la forma en que actúa y en su estilo de invocación.

Una función **no pertenece a ningún dato**: obtiene datos, puede crear nuevos datos y (generalmente) produce un resultado.

Un método hace todas estas cosas, pero también puede **cambiar el estado de una entidad seleccionada**.

Un método es propiedad de los datos para los que trabaja, mientras que una función es propiedad de todo el código.

Esto también significa que invocar un método requiere alguna especificación de los datos a partir de los cuales se invoca el método.

Puede parecer desconcertante aquí, pero lo trataremos en profundidad cuando profundicemos en la programación orientada a objetos.

En general, una invocación de función típica puede tener este aspecto:

`result = function(arg)`

La función toma un argumento, hace algo y devuelve un resultado.

Una invocación de un método típico usualmente se ve así:

`result = data.method(arg)`

Nota: el nombre del método está precedido por el nombre de los datos que posee el método. A continuación, se agrega un **punto**, seguido del **nombre del método** y un par de **paréntesis que encierran los argumentos**.

El método se comportará como una función, pero puede hacer algo más: puede **cambiar el estado interno de los datos** a partir de los cuales se ha invocado.

Puedes preguntar: ¿por qué estamos hablando de métodos, y no de listas?

Este es un tema esencial en este momento, ya que le mostraremos como agregar nuevos elementos a una lista existente. Esto se puede hacer con métodos propios de las listas, no por funciones.

Agregando elementos a una lista: append() e insert()

```
numbers = [111, 7, 2, 1]
print(len(numbers))
print(numbers)
###
numbers.append(4)
print(len(numbers))
print(numbers)
```

```
###
numbers.insert(0, 222)
print(len(numbers))
print(numbers)
#
```

Un nuevo elemento puede ser *añadido* al final de la lista existente:
`list.append(value)`

Dicha operación se realiza mediante un método llamado `append()`. Toma el valor de su argumento y lo coloca **al final de la lista** que posee el método.
La longitud de la lista aumenta en uno.
El método `insert()` es un poco más inteligente: puede agregar un nuevo elemento **en cualquier lugar de la lista**, no solo al final.
`list.insert(location, value)`

Toma dos argumentos:

- El primero muestra la ubicación requerida del elemento a insertar. Nota: todos los elementos existentes que ocupan ubicaciones a la derecha del nuevo elemento (incluido el que está en la posición indicada) se desplazan a la derecha, para hacer espacio para el nuevo elemento.
- El segundo es el elemento a insertar.

Observa el código en el editor. Ve como usamos los métodos `append()` e `insert()`. Presta atención a lo que sucede después de usar `insert()`: el primer elemento anterior ahora es el segundo, el segundo el tercero, y así sucesivamente.
Agrega el siguiente fragmento después de la última línea de código en el editor:
`numbers.insert(1, 333)`
Imprime el contenido de la lista final en la pantalla y ve que sucede. El fragmento de código sobre el fragmento de código inserta `333` en la lista, por lo que es el segundo elemento. El segundo elemento anterior se convierte en el tercero, el tercero en el cuarto, y así sucesivamente.

Agregando elementos a una lista: continuación

```
my_list = []  # Creando una lista vacía.
for i in range(5):
    my_list.append(i + 1)
print(my_list)
```

Puedes **iniciar la vida de una lista creándola vacía** (esto se hace con un par de corchetes vacíos) y luego agregar nuevos elementos según sea necesario.
Echa un vistazo al fragmento en el editor. Intenta adivinar su salida después de la ejecución del bucle `for`. Ejecuta el programa para comprobar si tenías razón.
Será una secuencia de números enteros consecutivos del `1` (luego agrega uno a todos los valores agregados) hasta `5`.
Hemos modificado un poco el fragmento:

```
my_list = []  # Creando una lista vacía.
for i in range(5):
    my_list.insert(0, i + 1)
print(my_list)
```

¿Qué pasará ahora? Ejecuta el programa y comprueba si esta vez también tenías razón.
Deberías obtener la misma secuencia, pero en **orden inverso** (este es el mérito de usar el método `insert()`).

Haciendo uso de las listas

```
my_list = [10, 1, 8, 3, 5]
total = 0
for i in range(len(my_list)):
    total += my_list[i]
print(total)
```

El bucle `for` tiene una variante muy especial que puede **procesar las listas** de manera muy efectiva. Echemos un vistazo a eso.
Supongamos que deseas **calcular la suma de todos los valores almacenados en la lista** `my_list`.

Necesitas una variable cuya suma se almacenará y se le asignará inicialmente un valor de `0` - su nombre será `total`. (Nota: no la vamos a nombrar `sum` ya que Python utiliza el mismo nombre para una de sus funciones integradas: `sum()`. **Utilizar ese nombre sería considerado una mala práctica**. Luego agrega todos los elementos de la lista usando el bucle `for`. Echa un vistazo al fragmento en el editor.
Comentemos este ejemplo:

- A la lista se le asigna una secuencia de cinco valores enteros.
- La variable `i` toma los valores `0`, `1`,`2`,`3`, y `4`, y luego indexa la lista, seleccionando los elementos siguientes: el primero, segundo, tercero, cuarto y quinto.
- Cada uno de estos elementos se agrega junto con el operador `+=` a la variable `suma`, dando el resultado final al final del bucle.
- Observa la forma en que se ha empleado la función `len()`, hace que el código sea independiente de cualquier posible cambio en el contenido de la lista.

El segundo aspecto del bucle for

Pero el bucle `for` puede hacer mucho más. Puede ocultar todas las acciones conectadas a la indexación de la lista y entregar todos los elementos de la lista de manera práctica.
Este fragmento modificado muestra como funciona:

```
my_list = [10, 1, 8, 3, 5]
total = 0
for i in my_list:
    total += i
print(total)
```

¿Qué sucede aquí?

- La instrucción `for` especifica la variable utilizada para navegar por la lista (`i`) seguida de la palabra clave `in` y el nombre de la lista siendo procesado (`my_list`).
- A la variable `i` se le asignan los valores de todos los elementos de la lista subsiguiente, y el proceso ocurre tantas veces como hay elementos en la lista.
- Esto significa que usa la variable `i` como una copia de los valores de los elementos, y no necesita emplear índices.
- La función `len()` tampoco es necesaria aquí.

Listas en acción

Dejemos de lado las listas por un breve momento y veamos un tema intrigante.
Imagina que necesitas reorganizar los elementos de una lista, es decir, revertir el orden de los elementos: el primero y el quinto, así como el segundo y cuarto elementos serán intercambiados. El tercero permanecerá intacto.
Pregunta: ¿Cómo se pueden intercambiar los valores de dos variables?
Echa un vistazo al fragmento:

```
variable_1 = 1
variable_2 = 2
variable_2 = variable_1
variable_1 = variable_2
```

Si haces algo como esto, **perderás el valor previamente almacenado** en `variable_2`. Cambiar el orden de las tareas no ayudará. Necesitas una tercera variable **que sirva como almacenamiento auxiliar**.
Así es como puedes hacerlo:

```
variable_1 = 1
variable_2 = 2
auxiliary = variable_1
variable_1 = variable_2
variable_2 = auxiliary
```

Python ofrece una forma más conveniente de hacer el intercambio, echa un vistazo:

```
variable_1 = 1
variable_2 = 2
variable_1, variable_2 = variable_2, variable_1
```

Claro, efectivo y elegante, ¿no?

Listas en acción

Ahora puedes **intercambiar** fácilmente los elementos de la lista para **revertir su orden**:

```
my_list = [10, 1, 8, 3, 5]

my_list[0], my_list[4] = my_list[4], my_list[0]
my_list[1], my_list[3] = my_list[3], my_list[1]

print(my_list)
```

Ejecuta el fragmento. Su salida debería verse así:

```
[5, 3, 8, 1, 10]
```

salida

Se ve bien con cinco elementos.

¿Seguirá siendo aceptable con una lista que contenga 100 elementos? No, no lo hará.
¿Puedes usar el bucle `for` para hacer lo mismo automáticamente, independientemente de la longitud de la lista? Si, si puedes.

Así es como lo hemos hecho:

```
my_list = [10, 1, 8, 3, 5]
length = len(my_list)

for i in range(length // 2):
    my_list[i], my_list[length - i - 1] = my_list[length - i - 1], my_list[i]

print(my_list)
```

Nota:

- Hemos asignado la variable `length` a la longitud de la lista actual (esto hace que nuestro código sea un poco más claro y más corto).
- Hemos preparado el bucle `for` para que se ejecute su cuerpo `length // 2` veces (esto funciona bien para listas con longitudes pares e impares, porque cuando la lista contiene un número impar de elementos, el del medio permanece intacto).
- Hemos intercambiado el elemento i (desde el principio de la lista) por el que tiene un índice igual a `(length-i-1)` (desde el final de la lista); en nuestro ejemplo, for `i` igual a `0` a la `(length-i-1)` da `4`; for `i` igual a `3`, da `3`: esto es exactamente lo que necesitábamos.

Las listas son extremadamente útiles y las encontrarás muy a menudo.

Puntos Clave

1. La lista **es un tipo de dato** en Python que se utiliza para **almacenar múltiples objetos**. Es una **colección ordenada y mutable** de elementos separados por comas entre corchetes, por ejemplo:

```
my_list = [1, None, True, "Soy una cadena", 256, 0]
```

2. Las listas se pueden **indexar y actualizar** , por ejemplo:

```
my_list = [1, None, True, 'Soy una cadena', 256, 0]
print(my_list[3])  # salida: Soy una cadena
print(my_list[-1])  # salida: 0
my_list[1] = '?'
print(my_list)  # salida: [1, '?', True, 'Soy una cadena', 256, 0]
```

```
my_list.insert(0, "primero")
my_list.append("último")
print(my_list)  # outputs: ['primero', 1, '?', True, 'Soy una cadena', 256, 0, 'último']
```

3. Las listas pueden estar **anidadas**, por ejemplo:

```
my_list = [1, 'a', ["lista", 64, [0, 1], False]]
```

Aprenderás más sobre el anidamiento en el módulo 3.7; por el momento, solo queremos que sepas que algo como esto también es posible.

4. Los elementos de la lista y las listas se pueden **eliminar**, por ejemplo:

```
my_list = [1, 2, 3, 4]
del my_list[2]
print(my_list)  # salida: [1, 2, 4]

del my_list  # borra la lista entera
```

Nuevamente, aprenderás más sobre esto en el módulo 3.6, no te preocupes. Por el momento, intenta experimentar con el código anterior y verifica cómo cambiarlo afecta la salida.

5.Las listas pueden ser **iteradas** mediante el uso del bucle `for`, por ejemplo:

```
my_list = ["blanco", "purpura", "azul", "amarillo", "verde"]

for color in my_list:
    print(color)
```

6. La función `len()` se puede usar para **verificar la longitud de la lista**, por ejemplo:

```
my_list = ["blanco", "purpura", "azul", "amarillo", "verde"]
print(len(my_list))  # salida 5
del my_list[2]
print(len(my_list))  # salida 4
```

7. Una invocación típica de **función** tiene el siguiente aspecto: `result = function(arg)`, mientras que una invocación típica de un **método** se ve así: `result = data.method(arg)`.

Ejercicio 1

¿Cuál es la salida del siguiente fragmento de código?

```
lst = [1, 2, 3, 4, 5]
lst.insert(1, 6)
del lst[0]
lst.append(1)
print(lst)
```

Revisar

```
[6, 2, 3, 4, 5, 1]
```

Ejercicio 2

¿Cuál es la salida del siguiente fragmento de código?

```
lst = [1, 2, 3, 4, 5]
lst_2 = []
add = 0
for number in lst:
    add += number
    lst_2.append(add)
print(lst_2)
```

Revisar

```
[1, 3, 6, 10, 15]
```

Ejercicio 3

¿Cuál es la salida del siguiente fragmento de código?

```
lst = []
del lst
print(lst)
```

Revisar `NameError: name 'lst' is not defined`

Ejercicio 4

¿Cuál es la salida del siguiente fragmento de código?

```
lst = [1, [2, 3], 4]
print(lst[1])
print(len(lst))
```

Revisar

```
[2, 3]
3
```

Ordenamiento Burbuja

Ahora que puedes hacer malabarismos con los elementos de las listas, es hora de aprender como **ordenarlos**. Se han inventado muchos algoritmos de clasificación, que difieren mucho en velocidad, así como en complejidad. Vamos a mostrar un algoritmo muy simple, fácil de entender, pero desafortunadamente, tampoco es muy eficiente. Se usa muy raramente, y ciertamente no para listas extensas.

Digamos que una lista se puede ordenar de dos maneras:

- Ascendente (o más precisamente, no descendente): si en cada par de elementos adyacentes, el primer elemento no es mayor que el segundo.
- Descendente (o más precisamente, no ascendente): si en cada par de elementos adyacentes, el primer elemento no es menor que el segundo.

En las siguientes secciones, ordenaremos la lista en orden ascendente, de modo que los números se ordenen de menor a mayor.

Aquí está la lista:

8	10	6	2	4

Intentaremos utilizar el siguiente enfoque: tomaremos el primer y el segundo elemento y los compararemos; si determinamos que están en el orden incorrecto (es decir, el primero es mayor que el segundo), los intercambiaremos; Si su orden es válido, no haremos nada. Un vistazo a nuestra lista confirma lo último: los elementos 01 y 02 están en el orden correcto, así como `8<10`.

Ahora observa el segundo y el tercer elemento. Están en las posiciones equivocadas. Tenemos que intercambiarlos:

8	**6**	**10**	2	4

Vamos más allá y observemos los elementos tercero y cuarto. Una vez más, esto no es lo que se supone que es. Tenemos que intercambiarlos:

8	6	**2**	**10**	4

Ahora comprobemos los elementos cuarto y quinto. Si, ellos también están en las posiciones equivocadas. Ocurre otro intercambio:

8	6	2	**4**	**10**

El primer paso a través de la lista ya está terminado. Todavía estamos lejos de terminar nuestro trabajo, pero algo curioso ha sucedido mientras tanto. El elemento más grande, `10`, ya ha llegado al final de la lista. Ten en cuenta que este es el **lugar deseado** para el. Todos los elementos restantes forman un lío pintoresco, pero este ya está en su lugar

Ahora, por un momento, intenta imaginar la lista de una manera ligeramente diferente, es decir, de esta manera:

10
4
2
6
8

Observa - El `10` está en la parte superior. Podríamos decir que flotó desde el fondo hasta la superficie, al igual que las burbujas **en una copa de champán**. El método de clasificación deriva su nombre de la misma observación: se denomina **ordenamiento de burbuja**.

Ahora comenzamos con el segundo paso a través de la lista. Miramos el primer y el segundo elemento, es necesario un intercambio:

6	**8**	2	4	10

Tiempo para el segundo y tercer elemento: también tenemos que intercambiarlos:

6	**2**	**8**	4	10

Ahora el tercer y cuarto elementos, y la segunda pasada, se completa, ya que 8 ya está en su lugar:

6	2	**4**	**8**	10

Comenzamos el siguiente pase inmediatamente. Observe atentamente el primer y el segundo elemento: se necesita otro cambio:

2	**6**	4	8	10

Ahora 6 necesita ir a su lugar. Cambiamos el segundo y el tercer elemento:

2	**4**	**6**	8	10

La lista ya está ordenada. No tenemos nada más que hacer. Esto es exactamente lo que queremos.
Como puedes ver, la esencia de este algoritmo es simple: **comparamos los elementos adyacentes y, al intercambiar algunos de ellos, logramos nuestro objetivo**.
Codifiquemos en Python todas las acciones realizadas durante un solo paso a través de la lista, y luego consideraremos cuántos pases necesitamos para realizarlo. No hemos explicado esto hasta ahora, pero lo haremos pronto.

Ordenando una lista

¿Cuántos pases necesitamos para ordenar la lista completa?
Resolvamos este problema de la siguiente manera: **introducimos otra variable**, su tarea es observar si se ha realizado algún intercambio durante el pase o no. Si no hay intercambio, entonces la lista ya está ordenada, y no hay que hacer nada más. Creamos una variable llamada swapped, y le asignamos un valor de False para indicar que no hay intercambios. De lo contrario, se le asignará True.

```
my_list = [8, 10, 6, 2, 4]  # lista a ordenar

for i in range(len(my_list) - 1):  # necesitamos (5 - 1) comparaciones
    if my_list[i] > my_list[i + 1]:  # compara elementos adyacentes
        my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]  # Si terminamos aquí, tenemos que intercambiar elementos.
```

Deberías poder leer y comprender este programa sin ningún problema:

```
my_list = [8, 10, 6, 2, 4]  # lista a ordenar
swapped = True  # Lo necesitamos verdadero (True) para ingresar al bucle while.

while swapped:
    swapped = False  # no hay intercambios hasta ahora
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True  # ¡ocurrió el intercambio!
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]

print(my_list)
```

Ejecuta el programa y pruébalo.

El ordenamiento burbuja - versión interactiva

```
my_list = []
swapped = True
num = int(input("¿Cuántos elementos deseas ordenar?: "))
for i in range(num):
    val = float(input("Ingresa un elemento de la lista: "))
    my_list.append(val)
```

```
while swapped:
    swapped = False
    for i in range(len(my_list) - 1):
        if my_list[i] > my_list[i + 1]:
            swapped = True
            my_list[i], my_list[i + 1] = my_list[i + 1], my_list[i]
print("\nOrdenada:")
print(my_list)
```

En el editor, puedes ver un programa completo, enriquecido por una conversación con el usuario, y que permite ingresar e imprimir elementos de la lista: **El ordenamiento burbuja: versión final interactiva**.

Python, sin embargo, tiene sus propios mecanismos de clasificación. Nadie necesita escribir sus propias clases, ya que hay un número suficiente de **herramientas listas para usar**.

Te explicamos este sistema de clasificación porque es importante aprender como procesar los contenidos de una lista y mostrarte como puede funcionar la clasificación real.

Si quieres que Python ordene tu lista, puedes hacerlo de la siguiente manera:

```
my_list = [8, 10, 6, 2, 4]
my_list.sort()
print(my_list)
```

Es tan simple como eso.

La salida del fragmento es la siguiente:

```
[2, 4, 6, 8, 10]
```

Como puedes ver, todas las listas tienen un método denominado `sort()`, que las ordena lo más rápido posible. Ya has aprendido acerca de algunos de los métodos de lista anteriormente, y pronto aprenderás más sobre otros.

Puntos Clave

1. Puedes usar el método `sort()` para ordenar los elementos de una lista, por ejemplo:

```
lst = [5, 3, 1, 2, 4]
print(lst)

lst.sort()
print(lst)  # outputs: [1, 2, 3, 4, 5]
```

2.También hay un método de lista llamado `reverse()`, que puedes usar para invertir la lista, por ejemplo:

```
lst = [5, 3, 1, 2, 4]
print(lst)

lst.reverse()
print(lst)  # salida: [4, 2, 1, 3, 5]
```

Ejercicio 1

¿Cuál es la salida del siguiente fragmento de código?

```
lst = ["D", "F", "A", "Z"]
lst.sort()

print(lst)
```

Revisar `['A', 'D', 'F', 'Z']`

Ejercicio 2

¿Cuál es la salida del siguiente fragmento de código?

```
a = 3
b = 1
c = 2

lst = [a, c, b]
lst.sort()
```

```
print(lst)
```

Revisar `[1, 2, 3]`

Ejercicio 3

¿Cuál es la salida del siguiente fragmento de código?

```
a = "A"
b = "B"
c = "C"
d = " "

lst = [a, b, c, d]
lst.reverse()

print(lst)
```

Revisar `[' ', 'C', 'B', 'A']`

La vida al interior de las listas

list_1 = [1]
list_2 = list_1
list_1[0] = 2
print(list_2)

Ahora queremos mostrarte una característica importante y muy sorprendente de las listas, que las distingue de las variables ordinarias.

Queremos que lo memorices, ya que puede afectar tus programas futuros y causar graves problemas si se olvida o se pasa por alto.

Echa un vistazo al fragmento en el editor.

El programa:

- Crea una lista de un elemento llamada `list_1`.
- La asigna a una nueva lista llamada `list_2`.
- Cambia el único elemento de `list_1`.
- Imprime la `list_2`.

La parte sorprendente es el hecho de que el programa mostrará como resultado: `[2]`, no `[1]`, que parece ser la solución obvia.

Las listas (y muchas otras entidades complejas de Python) se almacenan de diferentes maneras que las variables ordinarias (escalares).

Se podría decir que:

- El nombre de una variable ordinaria es el **nombre de su contenido**.
- El nombre de una lista es el nombre de una ubicación de memoria **donde se almacena la lista**.

Lee estas dos líneas una vez más, la diferencia es esencial para comprender de que vamos a hablar a continuación.

La asignación: `list_2 = list_1` copia el nombre del arreglo no su contenido. En efecto, los dos nombres (`list_1` y `list_2`) identifican la misma ubicación en la memoria de la computadora. Modificar uno de ellos afecta al otro, y viceversa.

¿Cómo te las arreglas con eso?

Rebanadas Poderosas

Afortunadamente, la solución está al alcance de tu mano: su nombre es **rebanada**.

Una rebanada es un elemento de la sintaxis de Python que permite **hacer una copia nueva de una lista, o partes de una lista**.

En realidad, copia el contenido de la lista, no el nombre de la lista.

Esto es exactamente lo que necesitas. Echa un vistazo al fragmento de código a continuación:

```
list_1 = [1]
list_2 = list_1[:]
list_1[0] = 2
print(list_2)
```

Su salida es `[1]`.

Esta parte no visible del código descrito como `[:]` puede producir una lista completamente nueva.

Una de las formas más generales de la rebanada es la siguiente:
`my_list[start:end]`

Como puedes ver, se asemeja a la indexación, pero los dos puntos en el interior hacen una gran diferencia.
Una rebanada de este tipo **crea una nueva lista (de destino), tomando elementos de la lista de origen: los elementos de los índices desde el principio hasta el** `fin - 1`.
Nota: no hasta el `fin`, sino hasta `fin-1`. Un elemento con un índice igual a `fin` es el primer elemento el cual **no participa en la segmentación**.
Es posible utilizar valores negativos tanto para el inicio como para el fin(al igual que en la indexación).
Echa un vistazo al fragmento:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:3]
print(new_list)
```

La lista `new_list` contendrá `fin - inicio` (3 - 1 = 2) elementos â los que tienen índices iguales a `1` y `2` (pero no `3`).
La salida del fragmento es: `[8, 6]`

```
# Copiando la lista entera.
list_1 = [1]
list_2 = list_1[:]
list_1[0] = 2
print(list_2)
# Copiando parte de la lista.
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:3]
print(new_list)
```

Rebanadas - índices negativos

Observa el fragmento de código a continuación:
`my_list[start:end]`
Para confirmar:

- `start` es el índice del primer elemento **incluido en la rebanada**.
- `end` es el índice del primer elemento **no incluido en la rebanada.**

Así es como **los índices negativos** funcionan en las rebanadas:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[1:-1]
print(new_list)
```

El resultado del fragmento es: `[8, 6, 4]}`
Si `start` especifica un elemento que se encuentra más allá del descrito por `end` (desde el punto de vista inicial de la lista), la rebanada estará **vacía**:

```
my_list = [10, 8, 6, 4, 2]
new_list = my_list[-1:1]
print(new_list)
```

La salida del fragmento es:

```
[]
```

Rebanadas: continuación

Si omites el `start` en tu rebanada, se supone que deseas obtener un segmento que comienza en el elemento con índice `0`.
En otras palabras, la rebanada sería de esta forma:
`my_list[:end]`
es un equivalente más compacto de:
`my_list[0:end]`
Observa el fragmento de código a continuación:

```
my_list = [10, 8, 6, 4, 2]
```

`new_list = my_list[:3]`
`print(new_list)`
Es por esto que su salida es: `[10, 8, 6]`.
Del mismo modo, si omites el `end` en tu rebanada, se supone que deseas que el segmento termine en el elemento con el índice `len(my_list)`.
En otras palabras, la rebanada sería de esta forma:
`my_list[start:]`
es un equivalente más compacto de:
`my_list[start:len(my_list)]`
Observa el siguiente fragmento de código:
`my_list = [10, 8, 6, 4, 2]`
`new_list = my_list[3:]`
`print(new_list)`
Por lo tanto, la salida es: `[4, 2]`

Rebanadas: continuación

Como hemos dicho anteriormente, el omitir el `inicio` y `fin` crea una copia **de toda la lista**:
`my_list = [10, 8, 6, 4, 2]`
`new_list = my_list[:]`
`print(new_list)`
El resultado del fragmento es: `[10, 8, 6, 4, 2]`.
La instrucción `del` descrita anteriormente puede **eliminar más de un elemento de la lista a la vez, también puede eliminar rebanadas**:
`my_list = [10, 8, 6, 4, 2]`
`del my_list[1:3]`
`print(my_list)`
Nota: En este caso, la rebanada **¡no produce ninguna lista nueva!**
La salida del fragmento es: `[10, 4, 2]`.
También es posible eliminar **todos los elementos** a la vez:
`my_list = [10, 8, 6, 4, 2]`
`del my_list[:]`
`print(my_list)`
La lista se queda vacía y la salida es: `[]`.
Al eliminar la rebanada del código, su significado cambia dramáticamente.
Echa un vistazo:
`my_list = [10, 8, 6, 4, 2]`
`del my_list`
`print(my_list)`
La instrucción `del` **eliminará la lista, no su contenido**.
La función `print()` de la última línea del código provocará un error de ejecución.

Los operadores in y not in

my_list = [0, 3, 12, 8, 2]
print(5 in my_list)
print(5 not in my_list)
print(12 in my_list)
Python ofrece dos operadores muy poderosos, capaces de **revisar la lista para verificar si un valor específico está almacenado dentro de la lista o no**.
Estos operadores son:
`elem in my_list`
`elem not in my_list`
El primero de ellos (`in`) verifica si un elemento dado (el argumento izquierdo) está actualmente almacenado en algún lugar dentro de la lista (el argumento derecho) - el operador devuelve `True` en este caso.
El segundo (`not in`) comprueba si un elemento dado (el argumento izquierdo) está ausente en una lista - el operador devuelve `True` en este caso

Observa el código en el editor. El fragmento muestra ambos operadores en acción. ¿Puedes adivinar su salida? Ejecuta el programa para comprobar si tenías razón.

Listas - algunos programas simples

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = my_list[0]
for i in range(1, len(my_list)):
    if my_list[i] > largest:
        largest = my_list[i]
print(largest)
```

Ahora queremos mostrarte algunos programas simples que utilizan listas.

El primero de ellos intenta encontrar el valor mayor en la lista. Mira el código en el editor.

El concepto es bastante simple: asumimos temporalmente que el primer elemento es el más grande y comparamos la hipótesis con todos los elementos restantes de la lista.

El código da como resultado el `17` (como se espera).

El código puede ser reescrito para hacer uso de la forma recién introducida del bucle `for`:

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = my_list[0]
for i in my_list:
    if i > largest:
        largest = i
print(largest)
```

El programa anterior realiza una comparación innecesaria, cuando el primer elemento se compara consigo mismo, pero esto no es un problema en absoluto.

El código da como resultado el `17` también (nada inusual).

Si necesitas ahorrar energía de la computadora, puedes usar una rebanada:

```
my_list = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = my_list[0]
for i in my_list[1:]:
    if i > largest:
        largest = i
print(largest)
```

La pregunta es: ¿Cuál de estas dos acciones consume más recursos informáticos: solo una comparación o rebanar casi todos los elementos de una lista?

Listas - algunos programas simples

Ahora encontremos la ubicación de un elemento dado dentro de una lista:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
to_find = 5
found = False

for i in range(len(my_list)):
    found = my_list[i] == to_find
    if found:
        break
if found:
    print("Elemento encontrado en el índice", i)
else:
    print("ausente")
```

Nota:

- El valor buscado se almacena en la variable `to_find`.
- El estado actual de la búsqueda se almacena en la variable `found` (`True`/`False`).
- Cuando `found` se convierte en `True`, se sale del bucle `for`.

Supongamos que has elegido los siguientes números en la lotería: `3`, `7`, `11`, `42`, `34`, `49`.

Los números que han salido sorteados son: `5`, `11`, `9`, `42`, `3`, `49`.

La pregunta es: ¿A cuántos números le has atinado?
Este programa te dará la respuesta:

```
drawn = [5, 11, 9, 42, 3, 49]
bets = [3, 7, 11, 42, 34, 49]
hits = 0

for number in bets:
   if number in drawn:
      hits += 1

print(hits)
```

Nota:

- La lista `drawn` almacena todos los números sorteados.
- La lista `bets` almacena los números con que se juega.
- La variable `hits` cuenta tus aciertos.

La salida del programa es: `4`.

Puntos Clave

1. Si tienes una lista `list_1`, y la siguiente asignación: `list_2 = list_1` esto no hace una copia de la lista `list_1`, pero hace que las variables `list_1` y `list_2` **apunten a la misma lista en la memoria**. Por ejemplo:

```
vehicles_one = ['carro', 'bicicleta', 'motor']
print(vehicles_one) # salida: [carro', 'bicicleta', 'motor']
vehicles_two = vehicles_one
del vehicles_one[0] # elimina 'carro'
print(vehicles_two) # salida: ['bicicleta', 'motor']
```

2. Si deseas copiar una lista o parte de la lista, puedes hacerlo haciendo uso de **rebanadas**:

```
colors = ['rojo', 'verde', 'naranja']
copy_whole_colors = colors[:]  # copia la lista entera
copy_part_colors = colors[0:2]  # copia parte de la lista
```

3. También puede utilizar **índices negativos** para hacer uso de rebanadas. Por ejemplo:

```
sample_list = ["A", "B", "C", "D", "E"]
new_list = sample_list[2:-1]
print(new_list)  # outputs: ['C', 'D']
```

4. Los parámetros `start` y `end` son **opcionales** al partir en rebanadas una lista: `list[start:end]`, por ejemplo:

```
my_list = [1, 2, 3, 4, 5]
slice_one = my_list[2: ]
slice_two = my_list[ :2]
slice_three = my_list[-2: ]

print(slice_one)  # salida: [3, 4, 5]
print(slice_two)  # salida: [1, 2]
print(slice_three)  # salida: [4, 5]
```

5. Puedes **eliminar rebanadas** utilizando la instrucción `del`:

```
my_list = [1, 2, 3, 4, 5]
del my_list[0:2]
print(my_list)  # salida: [3, 4, 5]

del my_list[:]
print(my_list)  # delimina el contenido de la lista, genera: []
```

6. Puedes probar si algunos elementos **existen en una lista o no** utilizando las palabras clave `in` y `not in`, por ejemplo:

```
my_list = ["A", "B", 1, 2]
```

```
print("A" in my_list)  # salida: True
print("C" not in my_list)  # salida: True
print(2 not in my_list)  # salida: False
```

Ejercicio 1

¿Cuál es la salida del siguiente fragmento de código?

```
list_1 = ["A", "B", "C"]
list_2 = list_1
list_3 = list_2

del list_1[0]
del list_2[0]
print(list_3)
```

Revisar: ['C']

Ejercicio 2

¿Cuál es la salida del siguiente fragmento de código?

```
list_1 = ["A", "B", "C"]
list_2 = list_1
list_3 = list_2
del list_1[0]
del list_2
print(list_3)
```

Revisar ['B', 'C']

Ejercicio 3

¿Cuál es la salida del siguiente fragmento de código?

```
list_1 = ["A", "B", "C"]
list_2 = list_1
list_3 = list_2
del list_1[0]
del list_2[:]
print(list_3)
```

Revisar []

Ejercicio 4

¿Cuál es la salida del siguiente fragmento de código?

```
list_1 = ["A", "B", "C"]
list_2 = list_1[:]
list_3 = list_2[:]
del list_1[0]
del list_2[0]
print(list_3)
```

Revisar ['A', 'B', 'C']

Ejercicio 5

Inserta `in` o `not in` en lugar de `???` para que el código genere el resultado esperado.

```
my_list = [1, 2, "in", True, "ABC"]
print(1 ??? my_list)  # salida True
print("A" ??? my_list)  # salida True
print(3 ??? my_list)  # salida True
print(False ??? my_list)  # salida False
```

Revisar:

```
my_list = [1, 2, "in", True, "ABC"]
print(1 in my_list)  # salida True
print("A" not in my_list)  # salida True
```

```
print(3 not in my_list)  # salida True
print(False in my_list)  # salida False
```

Listas dentro de listas

Las listas pueden constar de escalares (es decir, números) y elementos de una estructura mucho más compleja (ya has visto ejemplos como cadenas, booleanos o incluso otras listas en las lecciones del Resumen de la Sección anterior). Veamos más de cerca el caso en el que los elementos de una lista **son listas**.

A menudo encontramos estos **arreglos** en nuestras vidas. Probablemente el mejor ejemplo de esto sea un **tablero de ajedrez**.

Un tablero de ajedrez está compuesto de filas y columnas. Hay ocho filas y ocho columnas. Cada columna está marcada con las letras de la A a la H. Cada línea está marcada con un número del uno al ocho.

La ubicación de cada campo se identifica por pares de letras y dígitos. Por lo tanto, sabemos que la esquina inferior derecha del tablero (la que tiene la torre blanca) es A1, mientras que la esquina opuesta es H8.

Supongamos que podemos usar los números seleccionados para representar cualquier pieza de ajedrez. También podemos asumir que **cada fila en el tablero de ajedrez es una lista**.

Observa el siguiente código:

```
row = []

for i in range(8):
    row.append(WHITE_PAWN)
```

Crea una lista que contiene ocho elementos que representan la segunda fila del tablero de ajedrez: la que está llena de peones (supon que `WHITE_PAWN` es un **símbolo predefinido** que representa un peón blanco).

El mismo efecto se puede lograr mediante una **comprensión de lista**, la sintaxis especial utilizada por Python para completar o llenar listas masivas.

Una comprensión de lista es en realidad una lista, pero **se creó sobre la marcha durante la ejecución del programa, y no se describe de forma estática**.

Echa un vistazo al fragmento:

```
row = [WHITE_PAWN for i in range(8)]
```

La parte del código colocada dentro de los paréntesis especifica:

- Los datos que se utilizarán para completar la lista (`WHITE_PAWN`)
- La cláusula que especifica cuántas veces se producen los datos dentro de la lista (`for i in range(8)`)

Permítenos mostrarte otros **ejemplos de comprensión de lista**:

Ejemplo #1:

```
squares = [x ** 2 for x in range(10)]
```

El fragmento de código genera una lista de diez elementos y la rellena con cuadrados de diez números enteros que comienzan desde cero (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

Ejemplo #2:

```
twos = [2 ** i for i in range(8)]
```

El fragmento crea un arreglo de ocho elementos que contiene las primeras ocho potencias del numero dos (1, 2, 4, 8, 16, 32, 64, 128)

Ejemplo #3:

```
odds = [x for x in squares if x % 2 != 0 ]
```

El fragmento crea una lista con solo los elementos impares de la lista `squares`.

Listas dentro de listas: arreglos bidimensionales

Supongamos también que un **símbolo predefinido** denominado `EMPTY` designa un campo vacío en el tablero de ajedrez.

Entonces, si queremos crear una lista de listas que representan todo el tablero de ajedrez, se puede hacer de la siguiente manera:

```
board = []

for i in range(8):
    row = [EMPTY for i in range(8)]
    board.append(row)
```

Nota:

- La parte interior del bucle crea una fila que consta de ocho elementos (cada uno de ellos es igual a `EMPTY`) y lo agrega a la lista del `board`.
- La parte exterior se repite ocho veces.
- En total, la lista `board` consta de 64 elementos (todos iguales a `EMPTY`).

Este modelo imita perfectamente el tablero de ajedrez real, que en realidad es una lista de elementos de ocho elementos, todos ellos en filas individuales. Resumamos nuestras observaciones:

- Los elementos de las filas son campos, ocho de ellos por fila.
- Los elementos del tablero de ajedrez son filas, ocho de ellos por tablero de ajedrez.

La variable `board` ahora es un **arreglo bidimensional**. También se le llama, por analogía a los términos algebraicos, una **matriz**.

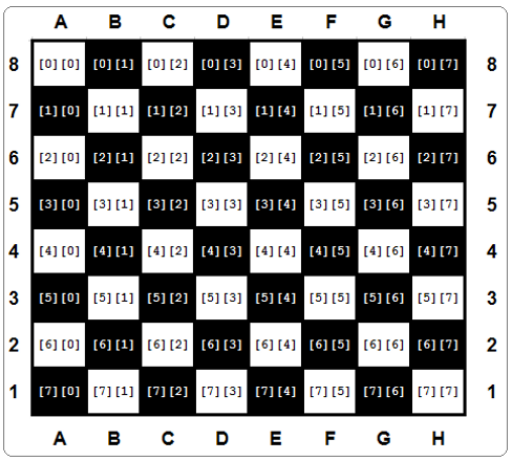
Como las listas de comprensión puede ser **anidadas**, podemos acortar la creación del tablero de la siguiente manera:

```
board = [[EMPTY for i in range(8)] for j in range(8)]
```

La parte interna crea una fila, y la parte externa crea una lista de filas.

Listas dentro de listas: arreglos bidimensionales - continuación

El acceso al campo seleccionado del tablero requiere dos índices: el primero selecciona la fila; el segundo: el número del campo dentro de la fila, el cual es un número de columna.



Echa un vistazo al tablero de ajedrez. Cada campo contiene un par de índices que se deben dar para acceder al contenido del campo:

Echando un vistazo a la figura que se muestra arriba, coloquemos algunas piezas de ajedrez en el tablero. Primero, agreguemos todas las torres:

```
board[0][0] = ROOK
board[0][7] = ROOK
board[7][0] = ROOK
board[7][7] = ROOK
```

Si deseas agregar un caballo a C4, hazlo de la siguiente manera:

```
board[4][2] = KNIGHT
```

Y ahora un peón a E5:

```
board[3][4] = PAWN
```

Y ahora - experimenta con el código en el editor:

```
EMPTY = "-"
PAWN = "PEON"
ROOK = "TORRE"
KNIGHT = "CABALLO"
board = []
for i in range(8):
    row = [EMPTY for i in range(8)]
    board.append(row)
board[0][0] = ROOK
board[0][7] = ROOK
board[7][0] = ROOK
board[7][7] = ROOK
print(board)
```

Naturaleza multidimensional de las listas: aplicaciones avanzadas

Profundicemos en la naturaleza multidimensional de las listas. Para encontrar cualquier elemento de una lista bidimensional, debes usar dos *coordenadas*:

- Una vertical (número de fila).
- Una horizontal (número de columna).

Imagina que desarrollas una pieza de software para una estación meteorológica automática. El dispositivo registra la temperatura del aire cada hora y lo hace durante todo el mes. Esto te da un total de 24 × 31 = 744 valores.

Intentemos diseñar una lista capaz de almacenar todos estos resultados.

Primero, debes decidir qué tipo de datos sería adecuado para esta aplicación. En este caso, sería mejor un `float`, ya que este termómetro puede medir la temperatura con una precisión de 0.1 °C.

Luego tomarás la decisión arbitraria de que las filas registrarán las lecturas cada hora exactamente (por lo que la fila tendrá 24 elementos) y cada una de las filas se asignará a un día del mes (supongamos que cada mes tiene 31 días, por lo que necesita 31 filas). Aquí está el par apropiado de comprensiones(`h` es para las horas, `d`para el día):

```
temps = [[0.0 for h in range(24)] for d in range(31)]
```

Toda la matriz está llena de ceros ahora. Puede suponer que se actualiza automáticamente utilizando agentes de hardware especiales. Lo que tienes que hacer es esperar a que la matriz se llene con las mediciones.

Ahora es el momento de determinar la temperatura promedio mensual del mediodía. Suma las 31 lecturas registradas al mediodía y divida la suma por 31. Puedes suponer que la temperatura de medianoche se almacena primero. Aquí está el código:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# La matriz se actualiza aquí.
#

total = 0.0

for day in temps:
    total += day[11]

average = total / 31

print("Temperatura promedio al mediodía:", average)
```

Nota: La variable `day` utilizada por el bucle `for` no es un escalar: cada paso a través de la matriz `temps` lo asigna a la siguiente fila de la matriz; Por lo tanto, es una lista. Se debe indexar con `11` para acceder al valor de temperatura medida al mediodía.

Ahora encuentra la temperatura más alta durante todo el mes, analiza el código:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# La matriz se actualiza aquí.
#

highest = -100.0

for day in temps:
    for temp in day:
        if temp > highest:
            highest = temp

print("La temperatura más alta fue:", highest)
```

Nota:

- La variable `day` itera en todas las filas de la matriz `temps`.
- La variable `temp` itera a través de todas las mediciones tomadas en un día.

Ahora cuenta los días en que la temperatura al mediodía fue de al menos 20 ℃:

```
temps = [[0.0 for h in range(24)] for d in range(31)]
#
# La matriz se actualiza aquí.
#

hot_days = 0

for day in temps:
```

```
    if day[11] > 20.0:
        hot_days += 1

print(hot_days, "fueron los días calurosos.")
```

Arreglos tridimensionales

En el editor: rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]

Python no limita la profundidad de la inclusión lista en lista. Aquí puedes ver un ejemplo de un arreglo tridimensional:

Imagina un hotel. Es un hotel enorme que consta de tres edificios, de 15 pisos cada uno. Hay 20 habitaciones en cada piso. Para esto, necesitas un arreglo que pueda recopilar y procesar información sobre las habitaciones ocupadas/libres.

Primer paso: El tipo de elementos del arreglo. En este caso, sería un valor Booleano (`True`/`False`).

Paso dos: análisis de la situación. Resume la información disponible: tres edificios, 15 pisos, 20 habitaciones.

Ahora puedes crear el arreglo:

```
rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]
```

El primer índice (`0` a `2`) selecciona uno de los edificios; el segundo (`0` a `14`) selecciona el piso, el tercero (`0` a `19`) selecciona el número de habitación. Todas las habitaciones están inicialmente desocupadas.

Ahora ya puedes reservar una habitación para dos recién casados: en el segundo edificio, en el décimo piso, habitación 14:

```
rooms[1][9][13] = True
```

y desocupar el segundo cuarto en el quinto piso ubicado en el primer edificio:

```
rooms[0][4][1] = False
```

Verifica si hay disponibilidad en el piso 15 del tercer edificio:

```
vacancy = 0
for room_number in range(20):
    if not rooms[2][14][room_number]:
        vacancy += 1
```

La variable `vacancy` contiene `0` si todas las habitaciones están ocupadas, o en dado caso el número de habitaciones disponibles.

¡Felicitaciones! Has llegado al final del módulo. ¡Sigue con el buen trabajo!

Puntos Clave

1. **La comprensión de listas** te permite crear nuevas listas a partir de las existentes de una manera concisa y elegante. La sintaxis de una comprensión de lista es la siguiente:

```
[expression for element in list if conditional]
```

El cual es un equivalente del siguiente código:

```
for element in list:
    if conditional:
        expression
```

Este es un ejemplo de una comprensión de lista: el código siguiente crea una lista de cinco elementos con los primeros cinco números naturales elevados a la potencia de 3:

```
cubed = [num ** 3 for num in range(5)]
print(cubed)  # outputs: [0, 1, 8, 27, 64]
```

2. Puedes usar **listas anidadas** en Python para crear **matrices** (es decir, listas bidimensionales). Por ejemplo:


```
#  Una tabla de cuatro columnas y cuatro filas: un arreglo bidimensional (4x4)
```

```
table = [[":(", ":)", ":(", ":)"],
        [":)", ":(", ":)", ":)"],
        [":(", ":)", ":)", ":("],
        [":)", ":)", ":)", ":("]]
print(table)
print(table[0][0])  # outputs: ':('
print(table[0][3])  # outputs: ':)'
```

3. Puedes anidar tantas listas en las listas como desee y, por lo tanto, crear listas n-dimensionales, por ejemplo, arreglos de tres, cuatro o incluso sesenta y cuatro dimensiones. Por ejemplo:

```
# Cubo - un arreglo tridimensional (3x3x3)

cube = [[[':(', 'x', 'x'],
        [':)', 'x', 'x'],
        [':(', 'x', 'x']],

       [[':)', 'x', 'x'],
        [':(', 'x', 'x'],
        [':)', 'x', 'x']],

       [[':(', 'x', 'x'],
        [':)', 'x', 'x'],
        [':)', 'x', 'x']]]

print(cube)
print(cube[0][0][0])  # outputs: ':('
print(cube[2][2][0])  # outputs: ':)'
```