**Fundamentos de Programación en Python:**

[**Módulo 1: Introducción a Python y a la programación**](https://397309018.netacad.com/courses/1091434/modules/items/71310711)

**En este módulo, aprenderás sobre:**

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

**¿Qué hace a un lenguaje?**

Podemos decir que cada idioma (máquina o natural) consta de los siguientes elementos:

**ALFABETO**

Un conjunto de símbolos utilizados para formar palabras de un determinado idioma

**LÉXICO**

(También conocido como diccionario) un conjunto de palabras que el idioma ofrece a sus usuarios

**SINTAXIS**

Un conjunto de reglas utilizadas para precisar si una determinada cadena de palabras forma una oración válida.

**SEMÁNTICA**

Un conjunto de reglas que determinan si una frase tiene sentido.

La IL (Lista de instrucciones) es, de hecho, **el alfabeto de un lenguaje máquina**. Es la lengua materna de la computadora.

Todos 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. Tales lenguajes son a menudo llamados **lenguajes de programación de alto nivel.**

Un programa escrito en un lenguaje de programación de alto nivel se llama **código fuente**. Del mismo modo, el archivo que contiene el código fuente se llama **archivo fuente**.

# Compilación vs. Interpretación

La programación de computadora es el acto de establecer una secuencia de instrucciones con la cual se causará el efecto deseado.

Tal composición tiene que ser correcta en muchos sentidos:

**Alfabéticamente:** Un programa debe escribirse en una secuencia de comandos reconocible.

**Léxicamente:** Cada lenguaje de programación tiene su diccionario y necesitas dominarlo.

**Sintácticamente:**Cada idioma tiene sus reglas y deben ser obedecidas.

**Semánticamente:**El programa tiene que tener sentido.

Supongamos que ha escrito correctamente un programa. ¿Cómo persuadimos a la computadora para que la ejecute? Tienes que convertir tu programa en lenguaje máquina.

Hay 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 que contiene el código de la máquina; ahora puedes distribuir el archivo en todo el mundo; el programa que realiza esta traducción se llama compilador o traductor.

**INTERPRETACIÓN:** 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.

# ¿Qué hace realmente el intérprete?

El intérprete lee el código fuente de arriba hacia abajo y de izquierda a derecha. En primer lugar, el intérprete verifica si todas las líneas subsiguientes son correctas.

Si el compilador encuentra un error, termina su trabajo inmediatamente. El único resultado en este caso es un **mensaje de error**. El intérprete le informará dónde se encuentra el error y qué lo causó. Sin embargo, estos mensajes pueden ser engañosos, ya que el intérprete no puede seguir tus intenciones exactas y puede detectar errores a cierta distancia de tus causas reales. Por ejemplo, si intentas usar una entidad de un nombre desconocido, causará un error, pero el error se descubrirá en el lugar donde se intenta usar la entidad, no donde se introdujo el nombre de la nueva entidad.

Si la línea se ve bien, el intérprete intenta ejecutarla (nota: cada línea generalmente se ejecuta por separado).

**Compilación vs. Interpretación - Ventajas y Desventajas**

|  | **COMPILACIÓN** | **INTERPRETACIÓN** |
| --- | --- | --- |
| **VENTAJAS** | * La ejecución del código traducido suele ser más rápida. * Solo el usuario 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 secreto. | * Puede ejecutar el código en cuanto lo complete; 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 compila el código por separado para cada arquitectura diferente. |
| **DESVENTAJAS** | * La compilación en sí misma puede llevar mucho tiempo; es posible que no puedas ejecutar tu código inmediatamente después de cualquier modificación. * Tienes que tener tantos compiladores como plataformas de hardware en los que deseas que se ejecute su código. | * No esperes que la interpretación incremente tu código a alta velocidad: tu código compartirá la potencia de la computadora con el intérprete, por lo que no puede ser realmente rápido. * Tanto tú como el usuario final deben tener el intérprete para ejecutar su 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 programación**, mientras que los programas fuente codificados que los usan se llaman **scripts**.

# ¿Qué es Python?

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

# ¿Quién creó Python?

# Una de las características sorprendentes de Python es el hecho de que en realidad es el trabajo de una persona.  Python fue creado por Guido van Rossum, nacido en 1956 en Haarlem, Países Bajos.

**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.

Es fácil de aprender, de enseñar, de utilizar, de comprender, y de obtener.

# Python alias CPython

Guido van Rossum utilizó el lenguaje de programación "C" para implementar la primera versión de su lenguaje y esta decisión aún está vigente. Todos los Pythons que vienen del PSF están escritos en el lenguaje "C".

# 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.

# Jython

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

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

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

# PyPy y RPython

Es un logotipo de **PyPy** - un Python dentro de un Python. En otras palabras, representa un entorno de Python escrito en un lenguaje similar a Python llamado **RPython** (Restricted Python). En realidad es un subconjunto de Python. El código fuente de PyPy no se ejecuta de manera interpretativa, sino que se traduce al lenguaje de programación C y luego se ejecuta por separado. PyPy es compatible con el lenguaje Python 3.

(para descargar python) Si eres un **usuario de Windows** , utiliza el archivo .exe descargado y sigue todos los pasos.

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

[**Módulo 2: Tipos de datos, variables, operaciones básicas de entrada y salida, operadores básicos**](https://397309018.netacad.com/courses/1091434/modules/items/71310712)

**En este módulo, aprenderás sobre:**

* Los tipos de datos y los métodos básicos de formateo, conversión de datos, entrada y salida de datos.
* Operadores.
* Variables.

**La función print()**

Mira la línea de código a continuación:

print("¡Hola, Mundo!")

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

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

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

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

¿De dónde provienen las funciones?

* Pueden venir **de Python mismo**. La función print es una de este tipo; dicha función es un valor agregado de Python junto con su entorno (está **integrada**);
* 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.
* 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.

Como se dijo anteriormente, una función puede tener:

* Un **efecto**.
* Un **resultado**.

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

Las funciones de Python 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.

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

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

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

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

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

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

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

print("¡Hola, Mundo!")

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

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

Casi cualquier cosa que ponga dentro de las comillas se tomará de manera literal, no como código, sino como **datos**.

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.

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

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

nombreFunción(argumento)

Veamos:

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

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

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

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

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

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

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

**La función print() - instrucciones**

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

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

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

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

**Las instrucciones en el código se ejecutan en el mismo orden** en que se colocaron en el archivo de origen; no se ejecuta la siguiente instrucción hasta que se complete la anterior.

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

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

Curiosamente, mientras **tú 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.

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

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**.

Si ponemos print() se representará como una línea en blanco, es decir un espacio.

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

**La función print() utilizando argumentos múltiples**

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

Los argumentos están **separados por comas**.

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

print("La Witsi Witsi Arañar" , "subió" , "a su telaraña."). **Console:** La Witsi Witsi Arañar subió a su telaraña.

**La función print() - los argumentos de palabras clave**

Python ofrece otro mecanismo para transmitir o pasar los argumentos, que puede ser útil cuando se desea convencer a la función print() de que cambie su comportamiento un poco. El mecanismo se llama **argumentos de palabras clave**.

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

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".

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

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

El argumento sep entrega el siguiente resultado:

Mi-nombre-es-Monty-Python.

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

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

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

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.

|  |  |
| --- | --- |
| print("Mi", "nombre", "es", sep="\_", end="\*")  print("Monty", "Python.", sep="\*", end="\*\n") | Mi\_nombre\_es\*Monty\*Python.\* |

Una cadena se puede multiplicar usando el siguiente truco: "string" \* 2 producirá "stringstring".

**Literales - los datos en si mismos**

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

123 es un literal, y c no lo es.

Se utilizan literales **para codificar datos y ponerlos dentro del código.**

A través de este ejemplo, encuentras dos tipos diferentes de literales: print("2") print(2).

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

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

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

**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.  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.

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.

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 numero 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 numero: 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 numero entero esta precedido por un código 0O o 0o (cero-o), el numero 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)

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.

**Flotantes**

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

Son números que tienen (o pueden tener) una parte fraccionaria después del punto decimal.

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

2.5

-0.4

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

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

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

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

0.4

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

En esencia, el valor 0.4 se puede escribir como:

.4

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

4.

Esto no cambiará su tipo ni su valor.

**Enteros vs. Flotantes**

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

Observa estos dos números:

4

4.0

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

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

El punto decimal es lo que determina si es flotante.

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

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

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

Para evitar escribir tantos ceros, los libros de texto emplean la forma abreviada, la cual probablemente hayas visto: 3 x 108.

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).

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:

0.0000000000000000000001

Cuando se corre en Python:

print(0.0000000000000000000001)

Este es el resultado:

1e-22 (**salida)**

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

**Cadenas**

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

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

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

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

Supongamos que se desea mostrar un muy sencillo mensaje:

Me gusta "Monty Python"

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

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

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

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

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

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

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

Este ejemplo funcionará también:

print('Me gusta "Monty Python"')

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

**Codificando cadenas**

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

Intenta imprimir una cadena que contenga el siguiente mensaje: I'm Monty Python.

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.

**Una cadena puede estar vacía** - puede no contener caracter alguno.

**Valores Booleanos**

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

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

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

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

Afortunadamente, las computadoras solo conocen dos tipos de respuestas:

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

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**. **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**.

**Python como una calculadora**

Python puede ser utilizado como una calculadora. Tomando esto, nos estamos adentrado en el terreno de los **operadores** y **expresiones**.

**Los Operadores Básicos**

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

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

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

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

+, -, \*, /, //, %, \*\*

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

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

**Operadores aritméticos: exponenciación**

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

Las matemáticas clásicas prefieren una notación con superíndices, como el siguiente: **23**. Los editores de texto puros no aceptan esa notación, por lo tanto Python utiliza \*\* en lugar de la notación matemática, por ejemplo, 2 \*\* 3.

**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**.

**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**.

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

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

Afortunadamente, Python puede ayudar con eso.

**Operadores aritméticos: división entera**

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

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

|  |  |
| --- | --- |
| print(6 // 3)  print(6 // 3.)  print(6. // 3)  print(6. // 3.) | 2  2.0  2.0  2.0 |

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

|  |  |
| --- | --- |
| print(6 // 4)  print(6. // 4) | 1  1.0 |

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

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

|  |  |
| --- | --- |
| print(-6 // 4)  print(6. // -4) | -2  -2.0 |

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**.

**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.

El resultado de la operación es el **residuo que queda de la división entera**. (sería el resto)

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.

**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.

**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 última línea de código del siguiente fragmento:

|  |  |
| --- | --- |
| print(-1.1) | -1.1 |

**Operadores y sus prioridades**

Muy seguido encontrarás más de un operador en una expresión. Probablemente recordaras de la escuela que las **multiplicaciones preceden a las sumas**. 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.

**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. Suele haber excepciones en torno a esto.

**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.

|  |  |  |
| --- | --- | --- |
| **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**.

**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.

**Puntos Clave**

1. Una **expresión** es una combinación de valores (o variables, operadores, llamadas a funciones, aprenderás de ello pronto) las cuales son evaluadas y dan como resultado un valor, por ejemplo, 1+2.

2. Los **operadores** son símbolos especiales o palabras clave que son capaces de operar en los valores y realizar operaciones matemáticas, por ejemplo, el \* multiplica dos valores: x\*y.

3. Los operadores aritméticos en Python: + (suma), - (resta), \* (multiplicación), / (división clásica: regresan un flotante si uno de los valores es de este tipo), % (módulo: divide el operando izquierdo entre el operando derecho y regresa el residuo de la operación, por ejemplo, 5%2=1), \*\* (exponenciación: el operando izquierdo se eleva a la potencia del operando derecho, por ejemplo, 2\*\*3=2\*2\*2=8), // (división entera: retorna el numero resultado de la división, pero redondeado al 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 hacia la derecha**, por ejemplo, 2\*\*2\*\*3=256.

**¿Qué son las Variables?**

Cómo 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).

Las variables no aparecen en un programa automáticamente. Como desarrollador, tú debes decidir cuantas variables deseas utilizar en tu programa. También las debes de nombrar.

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

* El nombre de la variable debe de estar compuesto por MAYUSCULAS, minúsculas, dígitos, y el carácter \_ (guion bajo).
* El nombre de la variable debe comenzar con una letra.
* El carácter guion bajo es considerado una letra.
* Las mayúsculas y minúsculas se tratan de forma distinta (un poco diferente que en el mundo real - *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.

**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 está reservada.

**Creando variables**

**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.** El primer carácter puede estar seguido de guiones bajos, letras, y dígitos. Las variables en Python son sensibles a mayúsculas y minúsculas.

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 que puede manejar variables también.

**Utilizando variables**

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

Sin embargo, no se permite utilizar una variable que no exista, (en otras palabras, una variable a la cual no se le a dado un valor). **RECUERDA** Se puede utilizar print() para combinar texto con variables utilizando el operador + para mostrar cadenas con variables.

**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

Entonces, ¿Cómo se lee esto en un programa?

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

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

var = 100

var = 200 + 300

print(var)

500 - ¿Por qué? 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.

**Operadores Abreviados**

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

Ejemplo: oveja = oveja + 1

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.

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

x \*= 2

oveja+= 1

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

**LABORATORIO**

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

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

var = "007"

print("Agente " + var)

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

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

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.

**La función input()**

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(). 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**.

* 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.

**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().

* 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.

**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.

**Conversión de datos o 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.

Ejemplo:

algo = float(input("Inserta un número: "))

resultado = algo \*\* 2.0

print(algo, "al cuadrado es", resultado)

**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**.

**Concatenación**

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

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

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

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

**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**.

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

Por ejemplo:

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

RECUERDA

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

**Conversión de tipos de datos: str()**

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

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

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

str(número)

**Puntos Clave**

1. La función print()**envía datos a la consola**, mientras que la función input()**obtiene datos de la consola**.

2. La función input() viene con un parámetro inicial: **un mensaje de tipo cadena para el usuario**. Permite escribir un mensaje antes de la entrada del usuario, por ejemplo:

nombre = input("Ingresa tu nombre: ")

print("Hola, " + nombre + ". ¡Un gusto conocerte!")

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

**En este módulo, aprenderás sobre:**

* Valores booleanos.
* Instrucciones if-elif-else.
* Bucles while y for.
* Control de flujo.
* Operaciones lógicas y bit a bit.
* Listas y arreglos.

## 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?*; a == b **compara** a y b.

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

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.

## 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.

**Operadores de Comparación: Mayor que**

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

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

ovejasNegras > ovejasBlancas # mayor que.

True lo confirma; False lo niega.

## Operadores de Comparación: Mayor o igual que

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

Hay dos signos subsecuentes, no uno.

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

## Operadores de Comparación: Menor o igual que

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

Mira este ejemplo simple:

velocidadMph < 85 # menor que.

velocidadMph ≤ 85 # menor o igual que.

La primera pregunta es estricta, la segunda no.

## Aprovechando las respuestas

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

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

respuesta = numerodeLeones >= numerodeLeonas

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

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

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

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

**Condiciones y ejecución condicional**

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

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

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

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

if cierto\_o\_no:

hacer\_esto\_si\_cierto

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

* La palabra clave if.
* Uno o más espacios en blanco.
* Una expresión (una pregunta o una respuesta) cuyo valor se 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 declaración?

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

Podemos escribir el siguiente fragmento de código:

if ClimaEsBueno:

irAcaminar()

almorzar()

**Ejecución condicional: La declaración if**

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

Echa un vistazo al siguiente código:

if contadorOvejas >= 120:

hacerCama()

tomarDucha()

dormirSoñar()

alimentarPerros()

Como puedes ver, tender la cama, tomar una ducha y dormir y soñar se ejecutan **condicionalmente**, cuando contadorOvejas alcanza el límite deseado.

Alimentar a los perros, sin embargo, **siempre se hace**(es decir, la función alimentarPerros no tiene sangría y no pertenece al bloque if, lo que significa que siempre se ejecuta).

Ahora vamos a discutir otra variante de la declaración condicional, que también permite realizar una acción adicional cuando no se cumple la condición.

**Ejecución condicional: la declaración if-else**

Comenzamos con una frase simple que decÃŒa: *Si el clima es bueno, saldremos a caminar*.

Nota: no hay una palabra sobre lo que suceder· 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 declaración condicional, la declaración *if-else*:

if condición\_true\_or\_false:

ejecuta\_si\_condición\_true

else:

ejecuta\_si\_condición\_false

Por lo tanto, hay una nueva palabra: else - esta es una **palabra 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 **Verdadero** (su valor no es igual a cero), la instrucción ejecuta\_si\_condición\_true se ejecuta, y la declaración condicional llega a su fin.
* Si la condición se evalúa como **Falso** (es igual a cero), la instrucción ejecuta\_si\_condición\_false se ejecuta, y la declaración condicional llega a su fin.

# La declaración if-else: más de ejecución condicional

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

if climaEsBueno:

irACaminar()

else:

irAlCine()

almorzar()

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 climaEsBueno:

irACaminar()

Diviertirse()

else:

irAlCine()

disfrutaLaPelicula()

almorzar()

## Declaraciones if-else anidadas

Ahora, analicemos dos casos especiales de la declaración 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 aquí:

if climaEsBueno:

if encontramosBuenRestaurante:

almorzar()

else:

comerSandwich()

else:

if hayBoletosDisponibles:

irAlCine()

else:

irDeCompras()

Aquí hay dos puntos importantes:

* Este uso de la declaraciónif 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*s y los*else*s.
* Considera como la sangría **mejora la legibilidad** y hace que el código sea más fácil de entender y rastrear.

## La declaración 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 declaración 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 elif desempeña su función.

Escribamos el mismo escenario usando Python:

if climaBueno:

iraCaminar()

elif hayBoletosDisponibles:

IralCine()

elif mesasLibres:

almorzar()

else:

jugarAjedrezEnCasa()

La forma de ensamblar las siguientes declaraciones *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.

**Analizando ejemplos de código**

Ahora te mostraremos algunos programas simples pero completos. Todos los programas resuelven el mismo problema: **encuentran el número mayor y lo imprimen**.

**Ejemplo 1:**

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

#lee dos números

numero1 = int (input("Ingresa el primer número:"))

numero2 = int (input("Ingresa el segundo número:"))

#elegir el número más grande

if numero1> numero2:

nmasGrande = numero1

else:

nmasGrande = numero2

#imprimir el resultado

print("El número más grande es:", nmasGrande)

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:

#lee dos números

numero1 = int (input("Ingresa el primer número:"))

numero2 = int (input("Ingresa el segundo número:"))

# elegir el número más grande

if numero1 > numero2: nmasGrande = numero1

else: nmasGrande = numero2

#imprimir el resultado

print("El número más grande es: ", nmasGrande)

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:

#lee tres números

numero1 = int (input("Ingresa el primer número:"))

numero2 = int (input("Ingresa el segundo número:"))

numero3 = int (input("Ingresa el tercer número:"))

#asumimos temporalmente que el primer número

#es el más grande

#lo verificaremos pronto

nmasGrande = numero1

#comprobamos si el segundo número es más grande que el mayor número actual

#y actualiza el número más grande si es necesario

if numero2 > nmasGrande:

    nmasGrande = numero2

#comprobamos si el tercer número es más grande que el mayor número actual

#y actualiza el número más grande si es necesario

if numero3 > nmasGrande:

    nmasGrande = numero3

#imprimir el resultado

print("El número más grande es:", nmasGrande)

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).

**Pseudocódigo e introducción a los ciclos o bucles**

Imagina un código que contiene 199 declaraciones 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:

línea 01 numeroMayor = -999999999

línea 02 numero = int(input())

línea 03 if numero == -1:

línea 04 print(numeroMayor)

línea 05 exit()

línea 06 if numero > numeroMayor:

línea 07 numeroMayor = numero

línea 08 vaya a la línea 02

¿Qué está pasando en él?

En primer lugar, podemos simplificar el programa si, al comienzo del código, asignamos la variable numeroMayor con 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:

# lee tres números

numero1 = int(input("Ingresa el primer número:"))

numero2 = int(input("Ingresa el segundo número:"))

numero3 = int(input("Ingresa el tercer número:"))

# verifica cuál de los números es el mayor

# y pásalo a la variable de mayor número

numeroMayor = max(numero1,numero2,numero3)

# imprimir el resultado

print("El número más grande es:", numeroMayor)

De la misma manera, puedes usar la función min() para devolver el número más bajo.

**Puntos clave**

**1**. Los operadores de **comparación** (o también denominados *relacionales*) se utilizan para comparar valores. La siguiente tabla ilustra cómo funcionan los operadores de comparación, asumiendo que x=0, y=1 y z=0:

|  |  |  |
| --- | --- | --- |
| **Operador** | **Descripción** | **Ejemplo** |
| == | Devuelve si los valores de los operandos son iguales, y False de lo contrario. | x == y # False  x == z # True |
| != | Devuelve True si los valores de los operandos no son iguales, y False de lo contrario. | x != y # True  x != z # False |
| > | DevuelveTrue si el valor del operando izquierdo es mayor que el valor del operando derecho, y False de lo contrario. | x > y # False  y > z # True |
| < | Devuelve True si el valor del operando izquierdo es menor que el valor del operando derecho, y False de lo contrario. | x < y # True y < z # False |
| ≥ | Devuelve True si el valor del operando izquierdo es mayor o igual al valor del operando derecho, y False de lo contrario. | x >= y # False  x >= z # True  y >= z # True |
| ≤ | Devuelve True si el valor del operando izquierdo es menor o igual al valor del operando derecho, y False de lo contrario. | x <= y # True x <= z # True  y <= z # False |

**2**. Cuando desea ejecutar algún código solo si se cumple una determinada condición, puede usar una **declaración condicional**:

* Una única declaración if, por ejemplo:

x = 10

if x == 10: # condición

    print("x es igual a 10") # ejecutado si la condición es verdadera

* Una serie de declaraciones if, por ejemplo:

x = 10

if x > 5: # condición uno

    print("x es mayor que 5") # ejecutado si la condición uno es verdadera

if x <10: # condición dos

    print("x es menor que 10") # ejecutado si la condición dos es verdadera

if x == 10: # condición tres

     print("x es igual a 10") # ejecutado si la condición tres es verdadera

Cada declaración if se prueba por separado.

* Una declaración de if-else, por ejemplo:

x = 10

if x < 10: # condición

    print ("x es menor que 10") # ejecutado si la condición es Verdadera

else:

    print ("x es mayor o igual a 10") # ejecutado si la condición es False

* Una serie de declaraciones if seguidas de un else, por ejemplo:

x = 10

if x > 5: # Verdadero

    print("x > 5")

if x > 8: # Verdadero

    print("x > 8")

if x > 10: # Falso

    print("x > 10")

else:

    print("Se ejecutará el else")

Cada if se prueba por separado. El cuerpo de else se ejecuta si el último if es False.

* La declaración  if-elif-else, por ejemplo:

x = 10

if x == 10: # Verdadero

    print("x == 10")

if x > 15: # Falso

    print("x > 15")

elif x > 10: # Falso

    print("x > 10")

elif x > 5: # Verdadero

    print("x > 5")

else:

    print("No se ejecutará el else")

Si la condición para if es False, el programa verifica las condiciones de los bloques elif posteriores: el primer elif que sea True es el que se ejecuta. Si todas las condiciones son False, se ejecutará el bloque else.

* Declaraciones condicionales anidadas, ejemplo:

x = 10

if x > 5: # Verdadero

    if x == 6: # Falso

        print("anidado: x == 6")

    elif x == 10: # Verdadero

        print("anidado: x == 10")

    else:

        print("anidado: else")

else:

    print("else")

**Ciclos o bucles en el código con while**

¿Estás de acuerdo con la declaración presentada a continuación?

“Mientras haya algo que hacer hazlo.”

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

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

while expresión\_condicional:

instrucción

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 declaraciones **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 expresión\_condicional:

instrucción\_uno

instruccion\_dos

instrucción\_tres

:

:

instrucción\_n

Ahora, es importante recordar que:

* Si deseas ejecutar **más de una declaración 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 ciclo**.
* 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 o ciclo infinito

Un ciclo infinito, también denominado **ciclo sin fin**, es una secuencia de instrucciones en un programa que se repite indefinidamente (ciclo sin fin).

Este es un ejemplo de un ciclo que no puede finalizar su ejecución:

while True:

print("Estoy atrapado dentro de un ciclo")

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

Si deseas obtener la mejor experiencia de aprendizaje al ver cómo se comporta un ciclo infinito, inicia IDLE, cree 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 ciclo". 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 ciclo. Hablaremos de ello más adelante en el curso.

Volvamos al bosquejo del algoritmo que te mostramos recientemente. Te mostraremos como usar este ciclo recién aprendido para encontrar el número más grande de un gran conjunto de datos ingresados.

Analiza el programa cuidadosamente. Localiza el cuerpo del ciclo y descubre **como se sale del cuerpo**:

# Almacenaremos el número más grande actual aquí

numero Mayor = -999999999

# Ingresa el primer valor

numero = int(input ("Introduzca un número o escriba -1 para detener:"))

# Si el número no es igual a -1, continuaremos

while numero != -1:

# ¿Es el número más grande que el número más grande?

if numero > numeroMayor:

# Sí si, actualiza el mayor númeroNúmero

numeroMayor = numero

# Ingresa el siguiente número

numero = int (input("Introduce un número o escribe -1 para detener:"))

# Imprimir el número más grande

print("El número más grande es:", numeroMayor)

# El ciclo(bucle) while: más ejemplos

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

# programa que lee una secuencia de números

# y cuenta cuántos números son pares y cuántos son impares

# programa termina cuando se ingresa cero

numerosImpares = 0

numerosPares = 0

# lee el primer número

numero = int (input ("Introduce un número o escriba 0 para detener:"))

# 0 termina la ejecución

while numero != 0:

# verificar si el número es impar

if numero % 2 == 1:

# aumentar el contador de números impares

numerosImpares += 1

else:

# aumentar el contador de números pares

numerosPares += 1

# lee el siguiente número

numero = int (input ("Introduce un número o escriba 0 para detener:"))

# imprimir resultados

print("Números impares: ", numerosImpares)

print("Números pares: ", numerosPares)

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 numero != 0: y while numero:

La condición que verifica si un número es impar también puede codificarse en estas formas equivalentes:

if numero % 2 == 1: e if numero % 2:

## Usando una variable contador para salir de un ciclo

Observa el fragmento de código a continuación:

contador = 5

while contador != 0:

print("Dentro del ciclo: ", contador)

contador -= 1

print("Fuera del ciclo", contador)

Este código está destinado a imprimir la cadena "Dentro del ciclo" y el valor almacenado en la variable contador durante un ciclo dado exactamente cinco veces. Una vez que la condición se haya cumplido (la variable contador ha alcanzado 0), se sale del ciclo y aparece el mensaje "Fuera del ciclo". así como el valor almacenado en contador se imprime.

Pero hay una cosa que se puede escribir de forma más compacta: la condición del ciclo while.

¿Puedes ver la diferencia?

contador=5

while contador:

print("Dentro del ciclo.", contador)

contador - = 1

print("Fuera del ciclo", contador)

¿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.

**Ciclos(bucles) en el código con**for

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

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

i = 0

while i < 100:

# hacer\_algo()

i += 1

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

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

En realidad, el ciclo 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):

#hacer algo()

pass

Hay algunos elementos nuevos. Déjanos contarte sobre ellos:

* La palabra reservada *for*abre el ciclo 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 ciclo; cuenta los giros del ciclo 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 ciclo 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 ciclo - no hace nada en absoluto; es una **instrucción vacía**: la colocamos aquí porque la sintaxis del ciclo 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 ciclo.

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 ciclo 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 ciclo 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 ciclo (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 ciclo 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:

pow = 1

for exp in range(16):

    print ("2 a la potencia de", exp, "es", pow)

    pow \* = 2

La variable exp se utiliza como una variable de control para el ciclo e indica el valor actual del *exponente*. La propia exponenciación se sustituye multiplicando por dos. Dado que 2 0es igual a 1, después 2 × 1 es igual a 21, 2 × 21 es igual a 22, 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.

El código debería decir:

2 a la potencia de 0 es 1

2 a la potencia de 1 es 1

2 a la potencia de 2 es 1

2 a la potencia de 3 es 1

2 a la potencia de 4 es 1

2 a la potencia de 5 es 1

2 a la potencia de 6 es 1

2 a la potencia de 7 es 1

2 a la potencia de 8 es 1

2 a la potencia de 9 es 1

2 a la potencia de 10 es 1

2 a la potencia de 11 es 1

2 a la potencia de 12 es 1

2 a la potencia de 13 es 1

2 a la potencia de 14 es 1

2 a la potencia de 15 es 1

**Las declaraciones break y continue**

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

* Parece que no es necesario continuar el ciclo en su totalidad; se debe abstener de seguir ejecutando el cuerpo del ciclo e ir más allá.
* Parece que necesitas comenzar el siguiente giro del ciclo 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 ciclo inmediatamente, e incondicionalmente termina la operación del ciclo; el programa comienza a ejecutar la instrucción más cercana después del cuerpo del ciclo.
* 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 de ruptura:")

for i in range(1,6):

if i == 3:

break

print("Dentro del ciclo.", i)

print("Fuera del ciclo.")

# continua - ejemplo

print("\nLa instrucción continue:")

for i in range(1,6):

if i == 3:

continue

print("Dentro del ciclo.", i)

print("Fuera del ciclo.")

En la consola se muestra:

La instrucción de ruptura:

Dentro del ciclo. 1

Dentro del ciclo. 2

Fuera del ciclo.

La instrucción continue:

Dentro del ciclo. 1

Dentro del ciclo. 2

Dentro del ciclo. 4

Dentro del ciclo. 5

Fuera del ciclo.

Es decir, que el break corta todo y sigue con la próxima instrucción, y el continue, omite esa instrucción de esa fila (if i==3, ejemplo) y continúa con lo siguiente del ciclo.

**Las declaraciones 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 usarías alguno de ellos.

La variante break va aquí:

numeroMayor = -99999999

contador = 0

while True:

    numero = int (input ("Ingresa un número o escribe -1 para finalizar el programa:"))

    if numero == -1:

        break

    contador = 1

    if numero > numeroMayor:

        numeroMayor = numero

if contador != 0:

    print("El número más grande es", numeroMayor)

else:

    print("No ha ingresado ningún número")

Ejecútalo, pruébalo y experimenta con él.

Y ahora la variante continue:

numeroMayor = -99999999

contador = 0

numero = int (input("Ingresa un número o escribe -1 para finalizar el programa:"))

while numero != -1:

    if numero == -1:

        continue

    contador = 1

    if numero > numeroMayor:

        numeroMayor = numero

    numero = int (input ("Ingresa un número o escribe -1 para finalizar el programa:"))

if contador:

    print("El número más grande es", numeroMayor)

else:

    print("No ha ingresado ningún número")

Otra vez: ejecútalo, pruébalo y experimenta con él.

**El while y la opción else**

Ambos ciclos, while y for, tienen una característica interesante (y rara vez se usa).

i = 1

while i < 5:

print (i)

i += 1

else:

print("else:", i)

Echa un vistazo al fragmento en el editor. Hay algo extraño al final: la palabra clave else.

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

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

**El ciclo for y la rama else**

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

for i in range(5):

print(i)

else:

print("else:", i)

La salida puede ser un poco sorprendente.

La variable i conserva su último valor.

0

1

2

3

4

else: 4

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 ciclo no se ejecutará aquí en absoluto. Nota: hemos asignado la variable i antes del ciclo.

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

Nota: **si la variable de control no existe antes de que comience el ciclo, no existirá cuando la ejecución llegue a la rama**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. Hay dos tipos de ciclos en Python: while y for:

* El ciclo while ejecuta una sentencia o un conjunto de declaraciones siempre que una condición booleana especificada sea verdadera, por ejemplo:

# Ejemplo 1

while True:

print("Atascado en un ciclo infinito")

# Ejemplo 2

contador = 5

while contador > 2:

print(contador)

contador -= 1

* El ciclo 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 ciclo for para iterar sobre una secuencia de números usando la función incorporada range. Mira los ejemplos a continuación:

# Ejemplo 1

palabra = "Python"

for letter in palabra:

print(letter, fin = "\*")

# 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 ciclo:

* Utiliza break para salir de un ciclo, por ejemplo:

texto = "OpenEDG Python Institute"

for letter in texto:

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 ciclos while y for también pueden tener una cláusula else en Python. La cláusula else se ejecuta después de que el ciclo 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:

# line of code

# line of code

# line of code  
Revisar

Solución de muestra:  
x = 1

while x < 11:

if x % 2 != 0:

print(x)

x += 1

**Ejercicio 3**

Crea un programa con un bucle for y una declaración 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 declaracióncontinue. 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="")

**Ejercicio 5**

¿Cuál es la salida del siguiente código?

n = 3

while n > 0:

print(n + 1)

n -= 1

else:

print(n)  
Revisar

4

3

2

0

**Ejercicio 6**

¿Cuál es la salida del siguiente código?

n = range(4)

for num in n:

print(num - 1)

else:

print(num)  
Revisar

-1

0

1

2

3

**Ejercicio 7**

¿Cuál es la salida del siguiente código?

for i in range(0, 6, 3):

print(i)  
Revisar

0

3

# 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 esta oración:

*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 estás en el centro comercial o 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 *y*. 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:

contador > 0 and valor == 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 y 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**:

print(var > 0)

print(not (var <= 0))  
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 cómo 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 vs. bits individuales

Los operadores lógicos toman sus argumentos como un todo, independientemente de cuántos 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

**Operadores bitwise**

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

Cubren todas las operaciones que mencionamos anteriormente en el contexto lógico, y un operador adicional. Este es el operador xor (como en **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) - exclusivo a nivel de bits o (xor).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operaciones bitwise (&, |, y ^)** | | | | |
| Arg A | Arg B | Arg B & Arg B | Arg A | Arg B | Arg A ^ Arg B |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

|  |  |
| --- | --- |
| **Operaciones bitwise (~)** | |
| Arg | ~Arg |
| 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 bitwise 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 vs 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:

og = 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 | 000000000000000000000000000 01111 |
| j | 000000000000000000000000000 10110 |
| bit = i & j | 000000000000000000000000000 00110 |

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 | 0000000000000000000000000000 1111 |
| bitneg = ~i | 1111111111111111111111111111 0000 |

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?**

Ahora te mostraremos para que puedes usar los operadores de bitwise. 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:

flagRegister = 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:

flagRegister = 000000000000000000000000000000x000

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 23=8. Se podría crear una máscara adecuada mediante la siguiente declaración:

theMask = 8

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

if flagRegister & theMask:

# mi bit está listo

else:

# mi bit se restablece

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 theMask. Restablecer el bit es simple, y se ve así (elige el que más te guste):

flagRegister = flagRegister & ~theMask

flagregister &= ~theMask

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:

flagRegister = flagRegister | theMask

flagRegister |= theMask

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:

flagRegister = flagRegister ^ theMask

flagRegister ^= theMask

**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.

valor << bits

valor >> 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.

var = 17

varRight = var >> 1

varLeft = var << 2

print(var, varLeft, varRight)

La invocación final de print() produce el siguiente resultado:

17 68 8

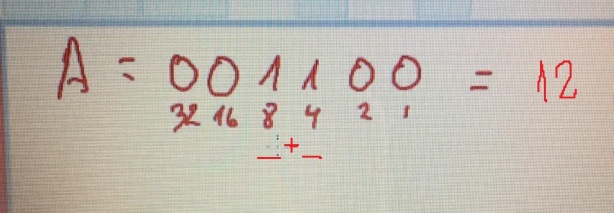
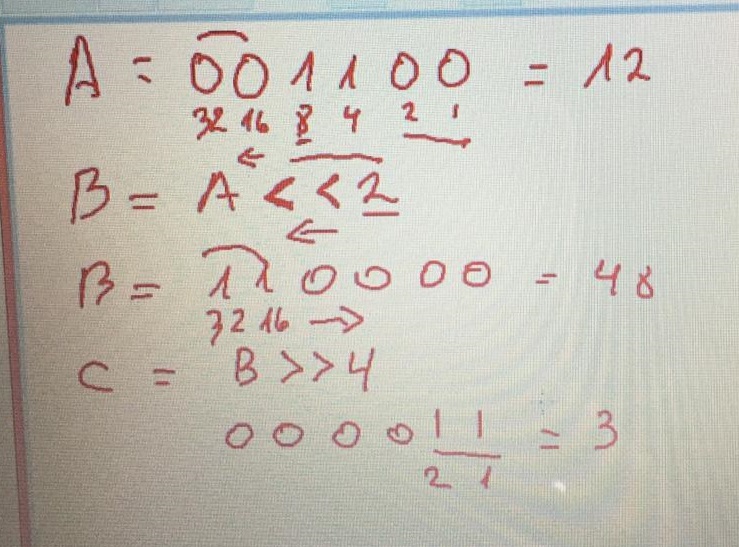
Nota:

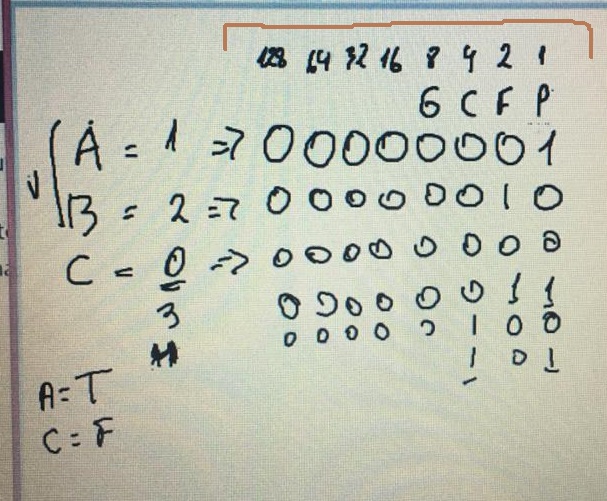
* 17 // 2 → 8 (**desplazarse hacia la derecha en un bit equivale a la división de enteros en dos**)
* 17 \* 4 → 68 (**desplazarse hacia la izquierda dos bits es lo mismo que multiplicar números enteros por cuatro**).

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

|  |  |  |
| --- | --- | --- |
| **Prioridad** | **Operador** |  |
| 1 | ! ~ (tipo) ++ -- + - | unario |
| 2 | \*\* |  |
| 3 | \* / % |  |
| 4 | + - | binario |
| 5 | << >> |  |
| 6 | <<=>> = |  |
| 7 | == != |  |
| 8 | & |  |
| 9 | | |  |
| 10 | && |  |
| 11 | || |  |
| 12 | = += -= \*= /= %= &= ^= |= >>= <<= |  |

Adjunto imagen con explicación del profe en clase en cuanto al valor de cada bit. (Un byte son 8 bits, cada uno de ellos tiene un valor asignado).



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

* x = 15, el cual es  0000 1111  en binario.
* y = 16, el cual es  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.

**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

False

**Ejercicio 2**

¿Cuál es la salida del siguiente fragmento de código?

x = 4

y = 1

a = x & y

b = x | y

c = ~ x

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 declaraciones 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).

numeros = [ 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 declaración 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**.

[0 es considerado Falso

El resto de los números es considerado Verdadero.]

**Listas de indexación**

¿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:

numeros = [10, 5, 7, 2, 1]

print("Contenido de la lista original:", numeros) # imprime el contenido de la lista original

numeros[0] = 111

print("Nuevo contenido de la lista:", numeros) # contenido de la lista actual.

El resultado en el programa será:

Contenido de la lista original: [10, 5, 7, 2, 1]

Nuevo contenido de la lista: [111, 5, 7, 2, 1]

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

numeros = [10, 5, 7, 2, 1]

print("Contenido de la lista original:", numeros) # imprimiendo contenido de la lista original.

numeros[0] = 111

print("\nPrevio contenido de la lista:", numeros) # imprimiendo contenido de la lista anterior.

numeros[1] = numeros[4] # copiando el valor del quinto elemento al segundo

print("Nuevo contenido de la lista:", numeros) # imprimiendo el contenido de la lista actual.

El resultado en el programa será:

Contenido de la lista: [10, 5, 7, 2, 1]

Previo contenido de la lista: [111, 5, 7, 2, 1]

Nuevo contenido de la lista: [111, 1, 7, 2, 1]

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(numeros[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(numeros) # 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]

## La función len()

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?

numeros = [10, 5, 7, 2, 1]

print("Contenido de la lista original:", numeros) # imprimiendo el contenido de la lista original

numeros [0] = 111

print("\nPrevio contenido de la lista:", numeros) # imprimiendo contenido de la lista anterior

numeros [1] = numeros [4] # copiando el valor del quinto elemento al segundo

print("Contenido de la lista anterior:", numeros) # imprimiendo contenido de la lista anterior

print("\nLongitud de la lista:", len(numeros)) # imprimiendo la longitud de la lista

En la consola el resultado será:

Contenido de la lista original: [10, 5, 7, 2, 1]

Previo contenido de la lista: [111, 5, 7, 2, 1]

Contenido de la lista anterior: [111, 1, 7, 2, 1]

Longitud de la lista: 5

**Eliminando elementos de una lista**

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 numeros[1]

print(len(numeros))

print(numeros)

numeros = [10, 5, 7, 2, 1]

print("Contenido de la lista original:", numeros) # imprimiendo contenido de la lista original

numeros[0] = 111

print("\nPrevio contenido de la lista:", numeros) # imprimiendo contenido de la lista anterior

numeros[1] = numeros[4] # copiando el valor del quinto elemento al segundo

print("Contenido de la lista anterior:", numeros) # imprimiendo contenido de la lista anterior

print ("\nLongitud de la lista:", len(numeros)) # imprimiendo la longitud de la lista anterior

del numeros[1] # eliminando el segundo elemento de la lista

print("Longitud de la nueva lista:", len(numeros)) # imprimiendo nueva longitud de la lista

print("\nNuevo contenido de la lista:", numeros) # imprimiendo el contenido de la lista actual

El resultado en la consola será:

Contenido de la lista original: [10, 5, 7, 2, 1]

Previo contenido de la lista: [111, 5, 7, 2, 1]

Contenido de la lista anterior: [111, 1, 7, 2, 1]

Longitud de la lista: 5

Longitud de la nueva lista: 4

Nuevo contenido de la lista: [111, 7, 2, 1]

----------------------------------------------------------------------------------------------------------------------------------------------

**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(numeros[4])

numeros[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**

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(numeros[-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(numeros[-2])

El fragmento de ejemplo mostrará 2.

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

numeros = [111, 7, 2, 1]

print(numeros[-1])

print(numeros[-2])

print(numeros[-4])

En la consola se verá:

1

2

111

**Funciones vs. 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:

resultado = funcion(argumento)

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

Una invocación de un método típico usualmente se ve así:

resultado = 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.

**Agregar elementos a una lista: append() e insert()**

Un nuevo elemento puede ser *añadido* al final de la lista existente:

lista.append(valor)

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.

lista.insert(ubicación,valor)

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:

numeros.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.

En código sería:

numeros = [111, 7, 2, 1]

print(len(numeros))

print(numeros)

numeros.append(4)

print(len(numeros))

print(numeros)

numeros.insert(0,222)

print(len(numeros))

print(numeros)

numeros.insert(1,333)

print(numeros)

En consola se muestra:

4

[111, 7, 2, 1]

5

[111, 7, 2, 1, 4]

6

[222, 111, 7, 2, 1, 4]

[222, 333, 111, 7, 2, 1, 4]

**Agregando elementos a una lista: continuación**

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.

miLista = [] # creando una lista vacía

for i in range (5):

miLista.append (i + 1)

print(miLista)

Salida en consola:

[1, 2, 3, 4, 5]

Será una secuencia de números enteros consecutivos del 1 hasta 5.

Hemos modificado un poco el fragmento:

miLista = [] # creando una lista vacía

for i in range(5):

miLista.insert(0, i + 1)

print(miLista)

¿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()).

miLista = [ ] # creando una lista vacía

for i in range (5):

miLista.insert(0, i + 1)

print(miLista)

Salida en consola:

[5, 4, 3, 2, 1]

**Haciendo uso de las listas**

El bucle for tiene una variante muy especial que puede **procesar las listas** de manera muy efectiva. Echemos un vistazo a eso.

Supongamos que desea **calcular la suma de todos los valores almacenados en la lista**miLista.

Necesitas una variable cuya suma se almacenará y se le asignará inicialmente un valor de 0 - su nombre es suma. 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.

En código:

miLista = [10, 1, 8, 3, 5]

suma = 0

for i in range(len(miLista)):

suma += miLista[i]

print(suma)

En consola:

27

## La segunda cara del ciclo 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:

miLista = [10, 1, 8, 3, 5]

suma = 0

for i in miLista:

suma += i

print(suma)

¿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 (miLista).
* 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.

**Las 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:

variable1 = 1

variable2 = 2

variable2 = variable1

variable1 = variable2

Si haces algo como esto, **perderás el valor previamente almacenado**envariable2. Cambiar el orden de las tareas no ayudará. Necesitas una tercera variable **que sirva como almacenamiento auxiliar**.

Así es como puedes hacerlo:

variable1 = 1

variable2 = 2

auxiliar = variable1

variable1 = variable2

variable2 = auxiliar

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

variable1 = 1

variable2 = 2

variable1, variable2 = variable2, variable1

Claro, efectivo y elegante, ¿no?

**Listas en acción**

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

miLista = [10, 1, 8, 3, 5]

miLista [0], miLista [4] = miLista [4], miLista [0]

miLista [1], miLista [3] = miLista [3], miLista [1]

print(miLista)

Ejecuta el fragmento. Su salida debería verse así:

[5, 3, 8, 1, 10]

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:

miLista = [10, 1, 8, 3, 5]

longitud = len(miLista)

for i in range (longitud // 2):

miLista[i], miLista[longitud-i-1] = miLista[longitud-i-1], miLista[i]

print(miLista)

Nota:

* Hemos asignado la variable longitud a la longitud de la lista actual (esto hace que nuestro código sea un poco más claro y más corto).
* Hemos lanzado el ciclo for para que se ejecute a través de su cuerpo longitud // 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 (longitud-i-1) (desde el final de la lista); en nuestro ejemplo, for i igual a 0 la (longitud-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:

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

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

miLista = [1, 1, None, True, 'Soy una cadena', 256, 0]

print(miLista [3]) # salida: soy una cadena

print(miLista [-1]) # salida: 0

miLista [1] = '?'

print (miLista) # salida: [1, '?', True, 'Soy una cadena', 256, 0]

miLista.insert (0, "first")

miLista.append ("last")

print (miLista ) # salida: ['first', 1, '?', True, 'Soy una cadena', 256, 0, 'last']

3. Las listas pueden estar **anidadas**, por ejemplo: miLista = [1, 'a', ["lista", 64, [0, 1], False]].

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

miLista = [1, 2, 3, 4]

del miLista[2]

print(miLista) # salida: [1, 2, 4]

del miLista # borra toda la lista

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

miLista = ["blanco", "purpura", "azul", "amarillo", "verde"]

for color in miLista :

print(color)

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

miLista = ["blanco", "purpura", "azul", "amarillo", "verde"]

print(len(miLista)) # la salidas es 5

del miLista[2]

print (len(miLista)) # la salidas es 4

7. Una invocación típica de **función**tiene el siguiente aspecto: resultado = funcion(argumento), mientras que una invocación típica de un **método** se ve así: resultado = 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]

lst2 = []

agregar = 0

for number in lst:

agregar += number

lst2.append (agregar)

print(lst2)

Revisar

[1, 3, 6, 10, 15]

**Ejercicio 3**

¿Qué sucede cuando ejecutas el siguiente fragmento de código?

lst = []

del lst

print(lst)

Revisar

NameError: el nombre 'lst' no está definido

**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.

miLista = [8, 10, 6, 2, 4] # lista para ordenar

for i in range(len(miLista) - 1): # necesitamos (5 - 1) comparaciones

if miLista[i] > miLista[i + 1]: # compara elementos adyacentes

miLista[i], miLista [i + 1] = miLista[i + 1], miLista[i] # si terminamos aquí significa que tenemos que intercambiar los elementos.

Deberías poder leer y comprender este programa sin ningún problema:

miLista = [8, 10, 6, 2, 4] # lista para ordenar

swapped = True # lo necesitamos verdadero (True) para ingresar al bucle while

while swapped:

swapped = False # no hay swaps hasta ahora

for i in range(len(miLista) - 1):

if miLista[i] > miLista[i + 1]:

swapped= True # ocurrió el intercambio!

miLista[i], miLista[i + 1] = miLista[i + 1], miLista[i]

print(miLista)

Ejecuta el programa y pruébalo.

**El ordenamiento burbuja - versión interactiva**

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 interactiva final**.

miLista = []

swapped = True

num = int (input("¿Cuántos elementos deseas ordenar?:"))

for i in range(num):

val = float(input("Introduce un elemento de la lista:"))

miLista.append(val)

while swapped:

swapped = False

for i in range(len(miLista) - 1):

if miLista[i] > miLista[i + 1]:

swapped = True

miLista[i], miLista[i + 1] = miLista[i + 1], miLista[i]

print("\nOrdenado:")

print(miLista)

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:

miLista = [8, 10, 6, 2, 4]

miLista.sort()

print(miLista)

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) # salida: [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**

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 lista1.
* La asigna a una nueva lista llamada lista2.
* Cambia el único elemento de lista1.
* Imprime la lista2.

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: lista2 = lista1copia el nombre de la matriz, no su contenido. En efecto, los dos nombres (lista1 y lista2) identifican la misma ubicación en la memoria de la computadora. Modificar uno de ellos afecta al otro, y viceversa.

**Rodajas Poderosas**

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

Una rodaja 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:

lista1 = [1]

lista2 = lista1[:]

lista1[0] = 2

print(lista2)

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 rodaja es la siguiente:

miLista[inicio:fin]

Como puedes ver, se asemeja a la indexación, pero los dos puntos en el interior hacen una gran diferencia.

Una rodaja 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:

miLista = [10, 8, 6, 4, 2]

nuevaLista = miLista [1:3] # Resta 3-1= 2, e imprime el inicio (1), y 2: [8,6].

print(nuevaLista)

La lista nuevaLista contendrá inicio-fin (3-1=2) elementos, los que tienen índices iguales a 1 y 2 (pero no 3)

La salida del fragmento es: [8, 6]

**Rodajas - índices negativos**

Observa el fragmento de código a continuación:

miLista[inicio:fin]

Para repetir:

* inicio es el índice del primer elemento **incluido en la rodaja**.
* fin es el índice del primer elemento **no incluido en la rodaja.**

Así es como **los índices negativos**funcionan con la rodaja:

miLista = [10, 8, 6, 4, 2]

nuevaLista = miLista [1:-1]

print(nuevaLista)

El resultado del fragmento es: [8, 6, 4].

Si elinicio especifica un elemento que se encuentra más allá del descrito por fin (desde el punto de vista inicial de la lista), la rodaja estará **vacía**:

miLista = [10, 8, 6, 4, 2]

nuevaLista = miLista [-1:1]

print(nuevaLista)

La salida del fragmento es: [].

**Rodajas: continuación**

Si omites inicio en tu rodaja, se supone que deseas obtener un segmento que comienza en el elemento con índice 0.

En otras palabras, la rodaja sería de esta forma:

miLista[:fin]

Es un equivalente más compacto:

miLista[0:fin]

Observa el fragmento de código a continuación:

miLista = [10, 8, 6, 4, 2]

nuevaLista = miLista [:3]

print(nuevaLista)

Es por esto que su salida es: [10, 8, 6].

Del mismo modo, si omites el fin en tu rodaja, se supone que deseas que el segmento termine en el elemento con el índice len(miLista).

En otras palabras, la rodaja sería de esta forma:

miLista[inicio:]

Es un equivalente más compacto:

miLista[inicio:len(miLista)]

Observa el siguiente fragmento de código:

miLista = [10, 8, 6, 4, 2]

nuevaLista = miLista[3:]

print(nuevaLista)

Por lo tanto, la salida es: [4, 2].

**Rodajas: continuación**

Como hemos dicho antes, el omitir inicio y fin hace una copia **de toda la lista**:

miLista = [10, 8, 6, 4, 2]

nuevLista = miLista [:]

print(nuevLista)

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 rodajas**:

miLista = [10, 8, 6, 4, 2]

del miLista[1:3]

print(miLista)

Nota: En este caso, la rodaja **¡no produce ninguna lista nueva!**

La salida del fragmento es:[10, 4, 2].

También es posible eliminar **todos los elementos** a la vez:

miLista = [10, 8, 6, 4, 2]

del miLista[:]

print(miLista)

La lista se queda vacía y la salida es: [].

Al eliminar la rodaja del código, su significado cambia dramáticamente.

Echa un vistazo:

miLista = [10, 8, 6, 4, 2]

del miLista

print(miLista)

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**

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 miLista

elem not in miLista

El primero de ellos (in) verifica si un elemento dado (su 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 (su 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.

miLista = [0, 3, 12, 8, 2]

print(5 in miLista)

print(5 not in miLista)

print(12 in miLista)

**Consola:**

False

True

True

**Listas - algunos programas simples**

Ahora queremos mostrarte algunos programas simples que utilizan listas.

El primero de ellos intenta encontrar el mayor valor 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.

miLista = [17, 3, 11, 5, 1, 9, 7, 15, 13]

mayor = miLista[0]

for i in range(1, len(miLista)):

if miLista [i]> mayor:

mayor = miLista[i]

print(mayor)

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

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

miLista = [17, 3, 11, 5, 1, 9, 7, 15, 13]

mayor = miLista [0]

for i in miLista:

if i > mayor:

mayor = i

print(mayor)

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 rodaja:

miLista = [17, 3, 11, 5, 1, 9, 7, 15, 13]

mayor = miLista [0]

for i in miLista [1:]:

if i > mayor:

mayor = i

print(mayor)

**Listas - algunos programas simples**

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

miLista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Encontrar = 5

Encontrado = False

for i in range(len(miLista)):

Encontrado = miLista[i] == Encontrar

if Encontrado:

break

if Encontrado:

print("Elemento encontrado en el índice", i)

else:

print("ausente")

Nota:

* El valor buscado se almacena en la variable Encontrar.
* El estado actual de la búsqueda se almacena en la variable Encontrado (True/False).
* Cuando Encontrado 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?

El programa te dará la respuesta:

sorteados = [5, 11, 9, 42, 3, 49]

seleccionados = [3, 7, 11, 42, 34, 49]

aciertos = 0

for numeros in seleccionados:

if numeros in sorteados:

aciertos += 1

print(aciertos)

Nota:

* La lista sorteados almacena todos los números ganadores.
* La lista de seleccionados almacena con números con que se juega.
* La variable aciertos cuenta tus aciertos.

La salida del programa es: 4.

# Puntos clave

1. Si tienes una lista l1, la siguiente asignación: l2 = l1 no hace una copia de la lista l1, pero hace que las variables l1 y l2 **apunten a la misma lista en la memoria**. Por ejemplo:

vehiculosUno = ['carro', 'bicicleta', 'moto']

print(vehiculosUno) # salida: ['carro', 'bicicleta', 'moto']

vehiculosDos = vehiculosUno

del vehiculosUno[0] # borra 'carro'

print(vehiculosDos) # salida: ['bicicleta', 'moto']

2. Si deseas copiar una lista o parte de la lista, puede hacerlo haciendo uso de**rodajas(slicing)**:

colores = ['rojo', 'verde', 'naranja']

copiaTodosColores = colores[:] # copia la lista completa

copiaParteColores = colores[0:2] # copia parte de la lista

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

listaMuestra = ["A", "B", "C", "D", "E"]

nuevaLista = listaMuestra[2:-1]

print(nuevaLista) # salida: ['C', 'D']

4. Los parámetros inicio y finson**opcionales**al partir en rodajas una lista: lista[inicio:fin], por ejemplo:

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

rodajaUno = miLista [2:]

rodajaDos = miLista [:2]

rodajaTres = miLista [-2:]

print(rodajaUno) # salidas: [3, 4, 5]

print(rodajaDos) # salidas: [1, 2]

print(rodajaTres) # salidas: [4, 5]

5. Puedes **eliminar rodajas**utilizando la instrucción del:

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

del miLista [0:2]

print(miLista) # salida: [3, 4, 5]

del miLista[:]

print(miLista) # elimina 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:

miLista = ["A", "B", 1, 2]

print("A" in miLista) # salida: True

print("C" not in miLista) # salida: False

print(2 not in miLista) # salidas: False

**Ejercicio 1**

¿Cuál es la salida del siguiente fragmento de código?

l1 = ["A", "B", "C"]

l2 = l1

l3 = l2

del l1[0]

del l2[0]

print(l3)  
Revisar

['C']

**Ejercicio 2**

¿Cuál es la salida del siguiente fragmento de código?

l1 = ["A", "B", "C"]

l2 = l1

l3 = l2

del l1[0]

del l2

print(l3)

Revisar

['B', 'C']

**Ejercicio 3**

¿Cuál es la salida del siguiente fragmento de código?

l1 = ["A", "B", "C"]

l2 = l1

l3 = l2

del l1[0]

del l2[:]

print(l3)  
Revisar

[]

**Ejercicio 4**

¿Cuál es la salida del siguiente fragmento de código?

l1 = ["A", "B", "C"]

l2 = l1[:]

l3 = l2[:]

del l1[0]

del l2[0]

print(l3)  
Revisar

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

**Ejercicio 5**

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

miLista = [1, 2, "in", True, "ABC"]

print(1 ??? miLista) # salida True

print("A" ??? miLista) # salida True

print(3 ??? miLista) # salida True

print(False ??? miLista) # salida False

Revisar

miLista = [1, 2, "in", True, "ABC"]

print(1 in miLista) # salidas True

print("A" not in miLista) # salida True

print(3 not in miLista) # salida True

print(False in miLista) # 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 solo 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:

fila = []

for i in range(8):

row.append(PEON\_BLANCO)

Crea una lista que contiene ocho elementos que representan la segunda fila del tablero de ajedrez: la que está llena de peones (supon que PEON\_BLANCO 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:

fila = [PEON\_BLANCO for i in range(8)]

La parte del código colocada dentro de los paréntesis especifica:

* Los datos que se utilizarán para completar la lista (PEON\_BLANCO)
* 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:

cuadrados = [x \*\* 2 for x in range(10)]

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

Ejemplo # 2:

dos = [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:

probabilidades = [x for x in cuadrados if x % 2 != 0]

El fragmento hace una lista con solo los elementos impares de la lista cuadrados.

**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:

tablero = []

for i in range(8):

fila = [EMPTY for i in range(8)]

tablero.append(fila)

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 tablero.
* La parte exterior se repite ocho veces.
* En total, la lista tablero 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 tablero 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:

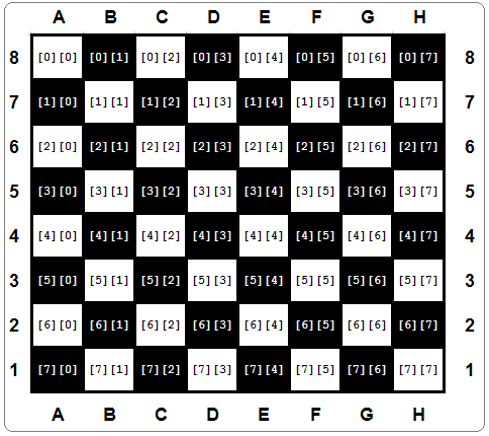
tablero = [[EMPTY for i in range(8)] for j in range(8)]

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:

tablero[0][0] = TORRE

tablero[0][7] = TORRE

tablero[7][0] = TORRE

tablero[7][7] = TORRE

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

tablero[4][2] = CABALLO

Y ahora un peón a E5:

tablero[3][4] = PEON

Y ahora - experimenta con el código en el editor.

EMPTY = "-"

TORRE = "TORRE"

tablero = []

for i in range(8):

fila = [EMPTY for i in range(8)]

tablero.append (fila)

tablero[0][0] = TORRE

tablero[0][7] = TORRE

tablero[7][0] = TORRE

tablero[7][7] = TORRE

print(tablero)

Consola:

[['TORRE', '-', '-', '-', '-', '-', '-', 'TORRE'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-', '-', '-', '-'], ['TORRE', '-', '-', '-', '-', '-', '-', 'TORRE']]

**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 ℃.

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, dpara 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 mágicamente aquí

#

suma = 0.0

for day in temps:

suma += day[11]

promedio= suma / 31

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

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, ve el código:

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

#

# la matriz se actualiza mágicamente aquí

#

mas\_alta = -100.0

for day in temps:

for temp in day:

if temp > mas\_alta:

mas\_alta = temp

print("La temperatura más alta fue:", mas\_alta)  
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 mágicamente aquí

#

hotDays = 0

for day in temps:

if day[11] > 20.0:

hotDays += 1

print(hotDays, " fueron los días calurosos

# Arreglos tridimensionales

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:

habitaciones = [[[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:

habitaciones[1][9][13] = True

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

habitaciones[0][4][1] = False

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

vacante = 0

for numeroHabitacion in range(20):

if not habitaciones[2][14][numeroHabitacion]:

vacante += 1

La variable vacante 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 lista de comprensión es la siguiente:

[expresión for elemento in lista if condicional]

El cual es un equivalente del siguiente código:

for elemento in lista:

if condicional:

expresión

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

cubos = [num \*\* 3 for num in range (5)]

print(cubos) # salidas: [0, 1, 8, 27, 64]

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

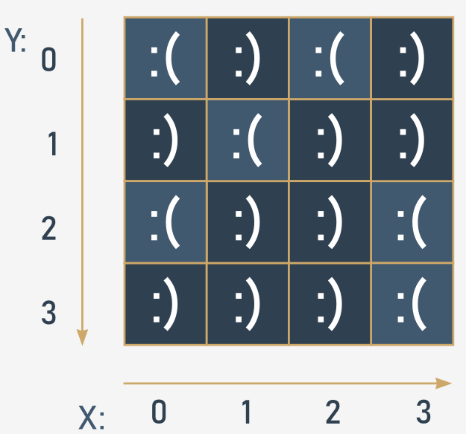
  
  
# Una tabla de cuatro columnas y cuatro filas: un arreglo bidimensional (4x4)

table = [[":(", ":)", ":(", ":)"],

[":)", ":(", ":)", ":)"],

[":(", ":)", ":)", ":("],

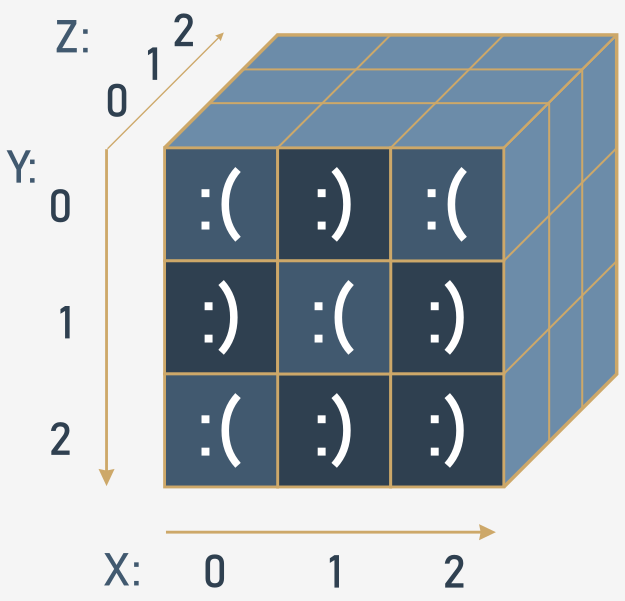
[":)", ":)", ":)", ":("]]

print(tabla)

print(tabla [0][0]) # salida: ':('

print(tabla [0][3]) # salida: ':)'

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)

cubo = [[[':(', 'x', 'x'],

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

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

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

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

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

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

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

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

print(cubo)

print(cubo [0][0][0]) # salida: ':('

print(cubo [2][2][0]) # salida: ':)'

[**Módulo 4 Funciones, tuplas, diccionarios y procesamiento de datos**](https://397309018.netacad.com/courses/1091434/modules/items/71310714)

**En este módulo, aprenderás:**

* Cómo definir y utilizar funciones.
* Cómo pasar argumentos y las distintas formas de hacerlo.
* El alcance de los nombres.
* Tuplas y diccionarios.
* Procesamiento de datos.

**¿Por qué necesitamos funciones?**

Hasta ahorita has implementado varias veces el uso de **funciones**, pero solo se han visto algunas de sus ventajas. Solo se han invocado funciones para utilizarlas como herramientas, con el fin de hacer la vida mas fácil, y para simplificar tareas tediosas y repetitivas.

Cuando se desea mostrar o imprimir algo en consola se utiliza print(). Cuando se desea leer el valor de una variable se emplea input(), combinados posiblemente con int() o float().

También se ha hecho uso de algunos **métodos**, las cuales también son funciones, pero declarados de una manera muy especifica.

Ahora aprenderás a escribir tus propias funciones, y como utilizarlas. Escribiremos varias de ellas juntos, desde muy sencillas hasta algo complejas. Se requerirá de tu concentración y atención.

Muy a menudo ocurre que un cierto fragmento de código **se repite muchas veces en un programa**. Se repite de manera literal o, con algunas modificaciones menores, empleando algunas otras variables dentro del programa. También ocurre que un programador ha comenzado a copiar y pegar ciertas partes del código en más de una ocasión en el mismo programa.

Puede ser muy frustrante percatarse de repente que existe un error en el código copiado. El programador tendrá que escarbar bastante para encontrar todos los lugares en el código donde hay que corregir el error. Además, existe un gran riesgo de que las correcciones produzcan errores adicionales.

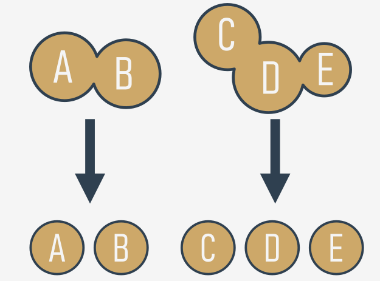
Definamos la primer condición por la cual es una buena idea comenzar a escribir funciones propias: **si un fragmento de código comienza a aparecer en más de una ocasión, considera la posibilidad de aislarlo en la forma de una función** invocando la función desde el lugar en el que originalmente se encontraba.

Puede suceder que el algoritmo que se desea implementar sea tan complejo que el código comience a crecer de manera incontrolada y, de repente, ya no se puede navegar por él tan fácilmente.

Se puede intentar solucionar este problema comentando el código, pero pronto te darás cuenta que esto empeorará la situación - **demasiados comentarios hacen que el código sea más difícil de leer y entender**. Algunos dicen que **una función bien escrita debe ser comprensible con tan solo una mirada**.

Un buen desarrollador **divide el código** (o mejor dicho: el problema) en piezas aisladas, y **codifica cada una de ellas en la forma de una función**.

Esto simplifica considerablemente el trabajo del programa, debido a que cada pieza se codifica por separado y consecuentemente se prueba por separado. A este proceso se le llama comúnmente **descomposición**.



Existe una segunda condición: **si un fragmento de código se hace tan extenso que leerlo o entenderlo se hace complicado, considera dividirlo pequeños problemas por separado e implementa cada uno de ellos como una función independiente**.

Esta descomposición continua hasta que se obtiene un conjunto de funciones cortas, fáciles de comprender y probar.

**Descomposición**

Es muy común que un programa sea tan largo y complejo que no puede ser asignado a un solo desarrollador, y en su lugar un **equipo de desarrolladores** trabajarán en el. El problema, debe ser dividido entre varios desarrolladores de una manera en que se pueda asegurar su eficiencia y cooperación.

Es inconcebible que más de un programador deba escribir el mismo código al mismo tiempo, por lo tanto, el trabajo debe de ser dividido entre todos los miembros del equipo.

Este tipo de descomposición tiene diferentes propósitos, no solo se trata de **compartir el trabajo**, sino también de **compartir la responsabilidad** entre varios desarrolladores.

Cada uno debe escribir un conjunto bien definido y claro de funciones, las cuales al ser **combinadas dentro de un módulo** (esto se clarificara un poco mas adelante) nos dará como resultado el producto final.

Esto nos lleva directamente a la tercera condición: si se va a dividir el trabajo entre varios programadores, **se debe descomponer el problema para permitir que el producto sea implementado como un conjunto de funciones escritas por separado empacadas juntas en diferentes módulos**.

**¿De dónde provienen las funciones?**

En general, las funciones provienen de al menos tres lugares:

* De Python mismo: varias funciones (como print()) son una **parte integral de Python**, y siempre están disponibles sin algún esfuerzo adicional del programador; se les llama a estas funciones**funciones integradas**.
* De los **módulos preinstalados** de Python: muchas de las funciones, las cuales comúnmente son menos utilizadas que las integradas, están disponibles en módulos instalados juntamente con Python; para poder utilizar estas funciones el programador debe realizar algunos pasos adicionales (se explicará acerca de esto en un momento).
* **Directamente del código**: tu puedes escribir tus propias funciones, colocarlas dentro del código, y usarlas libremente.
* Existe una posibilidad más, pero se relaciona con clases, se omitirá por ahora.

**Tu primer función**

Observa el fragmento de código en el editor.

print("Ingresa un valor: ")

a = int(input())

print("Ingresa un valor: ")

b = int(input())

print("Ingresa un valor: ")

c = int(input())

Es bastante sencillo, es un ejemplo de como **transformar una parte de código que se esta repitiendo en una función**.

El mensaje enviado a la consola por la función print() es siempre el mismo. El código es funcional y no contiene errores, sin embargo imagina tendrías que hacer si tu jefe pidiera cambiar el mensaje para que fuese mas cortés, por ejemplo, que comience con la frase "Por favor,".

Tendrías que tomar algo de tiempo para cambiar el mensaje en todos los lugares donde aparece (podrías hacer uso de copiar y pegar, pero eso no lo haría mas sencillo). Es muy probable que cometas errores durante el proceso de corrección, eso traería frustración a ti y a tu jefe.

¿Es posible separar ese código *repetido*, darle un nombre y hacerlo reutilizable? Significaría que **el cambio hecho en un solo lugar será propagado a todos los lugares donde se utilice**.

Para que esto funcione, dicho código debe ser invocado cada vez que se requiera.

Es posible, esto es exactamente para lo que existen las funciones.

¿Cómo es que se crea dicha función?

Se necesita **definirla**. Aquí, la palabra *define* es significativa.

Así es como se ve la definición más simple de una función:

def nombreFuncion():

cuerpoFuncion

* Siempre comienza con la **palabra reservada**def (que significa *definir*)
* Después de def va el **nombre de la función** (las reglas para darle nombre a las funciones son las mismas que para las variables).
* Después del nombre de la función, hay un espacio para un par de **paréntesis** (ahorita no contienen algo, pero eso cambiará pronto).
* La línea debe de terminar con **dos puntos**.
* La línea inmediatamente después de def marca el comienzo del **cuerpo de la función** - donde varias o (al menos una). **instrucción anidada**, será ejecutada cada vez que la función sea invocada; nota: la **función termina donde el anidamiento termina**, se debe ser cauteloso.

A continuación, se **definirá** la función. Se llamará mensaje - aquí esta:

def mensaje():

print("Ingresa un valor: ")

La función es muy sencilla, pero completamente **utilizable**. Se ha nombrado mensaje, pero eso es opcional, tu puedes cambiarlo. Hagamos uso de ella.

El código ahora contiene la definición de la función:

def mensaje():

print("Ingresa un valor: ")

print("Se comienza aquí.")

print("Se termina aquí.")

Nota: no se esta utilizando la función, no se esta **invocando** en el código.

Al correr el programa, se mostrará lo siguiente:

Se comienza aquí.

Se termina aquí.

Esto significa que Python lee la definición de la función y la recuerda, pero no la ejecuta sin permiso.

Se ha modificado el código, se ha insertado la **invocación de la función** entre los dos mensajes:

def mensaje():

print("Ingresa un valor: ")

print("Se comienza aquí.")

mensaje()

print("Se termina aquí.")

La salida ahora se ve diferente:

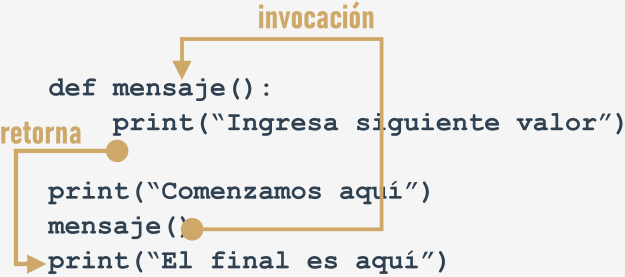
Se comienza aquí.

Ingresa un valor:

Se termina aquí.

**El funcionamiento de las funciones**

Observa la imagen:



La imagen intenta mostrar el proceso completo:

* Cuando se **invoca** una función, Python recuerda el lugar donde esto ocurre y *salta* hacia dentro de la función invocada.
* El cuerpo de la función es entonces **ejecutado**.
* Al llegar al final de la función, Python **regresa** al lugar inmediato después de donde ocurrió la invocación.

Existen dos consideraciones muy importantes, la primera de ella es:

**No se debe invocar una función antes de que se haya definido.**

Recuerda: Python lee el código de arriba hacia abajo. No va a adelantarse en el código para determinar si la función invocada esta definida mas adelante, el lugar correcto para definirla es antes de ser invocada.

Se ha insertado un error en el código anterior - ¿Puedes notar la diferencia?

print("Se comienza aquí.")

mensaje()

print("Se termina aquí.")

def mensaje():

print("Ingresa un valor: ")

Se ha movido la función al final del código. ¿Podrá Python encontrarla cuando la ejecución llegue a la invocación?

No, no podrá. El mensaje de error dirá:

NameError: name 'mensaje' is not defined

No intentes forzar a Python a encontrar funciones que no están definidas en el lugar correcto.

La segunda consideración es mas sencilla:

**Una función y una variable no pueden compartir el mismo nombre**.

El siguiente fragmento de código es erróneo:

def mensaje():

print("Ingresa un valor: ")

mensaje = 1

El asignar un valor al nombre "mensaje" causa que Python olvide su rol anterior. La función con el nombre de mensaje ya no estará disponible.

Afortunadamente, es posible **combinar o mezclar el código con las funciones**- no es forzoso colocar todas las funciones al inicio del archivo fuente.

Observa el siguiente código:

print("Se comienza aquí.")

def mensaje():

print("Ingresa un valor: ")

mensaje()

print("Se termina aquí.")

Puede verse extraño, pero es completamente correcto, y funciona como se necesita.

Regresemos al ejemplo inicial para implementar la función de manera correcta:

def mensaje():

print("Ingresa un valor: ")

mensaje()

a = int(input())

mensaje()

b = int(input())

mensaje()

c = int(input())

El modificar el mensaje de entrada es ahora sencillo: se puede hacer con solo **modificar el código una única vez** - dentro del cuerpo de la función.

**Puntos Clave**

1. Una **función** es un bloque de código que realiza una tarea especifica cuando la función es llamada (invocada). Las funciones son útiles para hacer que el código sea reutilizable, que este mejor organizado y más legible. Las funciones contienen parámetros y pueden regresar valores.

2. Existen al menos cuatro tipos de funciones básicas en Python:

* **Funciones integradas** las cuales son partes importantes de Python (como lo es la función print()). Puedes ver una lista completa de las funciones integradas de Python en la siguiente liga: <https://docs.python.org/3/library/functions.html>.
* También están las que se encuentran en **módulos pre-instalados** (se hablará acerca de ellas en el *Módulo 5* de este curso).
* **Funciones definidas por el usuario** las cuales son escritas por los programadores para los programadores, puedes escribir tus propias funciones y utilizarlas libremente en tu código.
* Las funciones lambda (aprenderás acerca de ellas en el *Módulo 6* del curso).

3. Las funciones propias se pueden definir utilizando la palabra reservada def y con la siguiente sintaxis:

def tuFuncion (parámetros opcionales):

# el cuerpo de la función

Se puede definir una función sin que haga uso de argumentos, por ejemplo:

def mensaje(): # definiendo una función

print("Hola") # cuerpo de la función

mensaje() # invocación de la función

También es posible definir funciones con argumentos, como la siguiente que contiene un solo parámetro:

def hola(nombre): # definiendo una función

print("Hola,", nombre) # cuerpo de la función

nombre = input("Ingresa tu nombre: ")

hola(nombre) # invocación de la función

Se hablará mas acerca de funciones con parámetros en la siguiente sección.

**Ejercicio 1**

La función input() es un ejemplo de:  
  
a) una función definida por el usuario  
b) una función integrada

Revisar

**b** - es una función integrada

**Ejercicio 2**

¿Qué es lo que ocurre cuando se invoca una función antes de ser definida? Ejemplo:

hola()

def hola():

print("hola!")  
Revisar

Se genera una excepción (la excepción NameError)

**Ejercicio 3**

¿Qué es lo que ocurrirá cuando se ejecute el siguiente código?

def hola():

print("hola")

hola(5)  
Revisar

Se genera una excepción (la excepción TypeError) - la función hola() no toma argumentos.

**Funciones con parámetros**

El potencial completo de una función se revela cuando puede ser equipada con una interface que es capaz de aceptar datos provenientes de la invocación. Dichos datos pueden modificar el comportamiento de la función, haciéndola mas flexible y adaptable a condiciones cambiantes.

Un parámetro es una variable, pero existen dos factores que hacen a un parámetro diferente:

* **Los parámetros solo existen dentro de las funciones en donde han sido definidos**, y el único lugar donde un parámetro puede ser definido es entre los paréntesis después del nombre de la función, donde se encuentra la palabra reservada def.
* **La asignación de un valor a un parámetro de una función se hace en el momento en que la función se manda llamar o se invoca**, especificando el argumento correspondiente.

def funcion(parametro):

###

Recuerda que:

* **Los parámetros solo existen dentro de las funciones** (este es su entorno natural).
* **Los argumentos existen fuera de las funciones**, y son los que pasan los valores a los parámetros correspondientes.

Existe una clara división entre estos dos mundos.

Enriquezcamos la función anterior agregándole un parámetro, se utilizará para mostrar al usuario el valor de un número que la función pide.

Se tendrá que modificar la definición def de la función, así es como se ve ahora:

def mensaje(numero):

###

Esta definición especifica que nuestra función opera con un solo parámetro con el nombre de numero. Se puede utilizar como una variable normal, pero **solo dentro de la función** - no es visible en otro lugar.

Ahora hay que mejorar el cuerpo de la función:

def mensaje(numero):

print("Ingresa el número:", numero)

Se ha hecho buen uso del parámetro. Nota: No se le ha asignado al parámetro algún valor. ¿Es correcto?

Si, lo es.

Un valor para el parámetro llegará del entorno de la función.

Recuerda: **especificar uno o más parámetros en la definición de la función**es un requerimiento, y se debe de cumplir durante la invocación de la misma. Se debe **proveer el mismo número de argumentos como haya parámetros definidos**.

El no hacerlo provocará un error.

**Funciones con parámetros: continuación**

Intenta ejecutar el código en el editor.

Esto es lo que aparecerá en consola:

TypeError: mensaje() missing 1 required positional argument: 'numero'

Esto significa que se esta invocando la función pero esta faltando el argumento.

Aquí esta ya de manera correcta:

def mensaje(numero):

print("Ingresa un número:", numero)

mensaje(1)

De esta manera ya esta correcto. El código producirá la siguiente salida:

Ingresa un número: 1

¿Puedes ver como funciona? El valor del argumento utilizado durante la invocación (1) ha sido pasado a la función, dándole un valor inicial al parámetro con el nombre de numero.

Existe una circunstancia importante que se debe mencionar.

Es posible tener una **variable con el mismo nombre del parámetro de la función**.

El siguiente código muestra un ejemplo de esto:

def mensaje(numero):

print("Ingresa un número:", numero)

numero = 1234

mensaje(1)

print(numero)

Una situación como la anterior, activa un mecanismo denominado **sombreado**:

* El parámetro x sombrea cualquier variable con el mismo nombre, pero...
* ... solo dentro de la función que define el parámetro.

El parámetro llamado numero es una entidad completamente diferente de la variable llamada numero.

Esto significa que el código anterior producirá la siguiente salida:

Ingresa un número: 1

1234

# Funciones con parámetros: continuación

Una función puede tener **tantos parámetros como se desee**, pero entre más parámetros, es más difícil memorizar su rol y propósito.

Modifiquemos la función- ahora tiene **dos parámetros**:

def mensaje(que, numero):

print("Ingresa", que, "número", numero)

Esto significa que para invocar la función, se necesitan **dos argumentos**.

El primer valor va a contener el nombre del valor deseado.

Aquí esta:

def mensaje(que, numero):

print("Ingresa", que, "número", numero)

mensaje("teléfono", 11)

mensaje("precio", 5)

mensaje("número", "número")

Estas es la salida del código anterior:

Ingresa teléfono número 11

Ingresa precio número 5

Ingresa número número número

# Paso de parámetros posicionales

La técnica que asigna cada argumento al parámetro correspondiente, es llamada **paso de parámetros posicionales**, los argumentos pasados de esta manera son llamados **argumentos posicionales**.

Ya se ha utilizado, pero Python ofrece mucho más. Se abordará este tema a continuación.

def miFuncion(a, b, c):

print(a, b, c)

miFuncion(1, 2, 3)

Nota: el paso de parámetros posicionales es usado de manera intuitiva por las personas en muchas situaciones. Por ejemplo, es generalmente aceptado que cuando nos presentamos mencionamos primero nuestro nombre(s) y después nuestro apellido, por ejemplo, "Me llamo Juan Pérez."

Sin embargo, En Hungría se hace al revés.Implementemos esa costumbre en Python. La siguiente función es utilizada para presentar a alguien:

def presentar(primerNombre, segundoNombre):

print("Hola, mi nombre es", primerNombre, segundoNombre)

presentar("Luke", "Skywalker")

presentar("Jesse", "Quick")

presentar("Clark", "Kent")

¿Puedes predecir la salida? Ejecuta el código y verifícalo por ti mismo.

Ahora imaginemos que la función esta siendo utilizada en Hungría. En este caso, el código sería de la siguiente manera:

def presentar(primerNombre, segundoNombre):

print("Hola, mi nombre es", primerNombre, segundoNombre)

presentar("Skywalker" ,"Luke" )

presentar("Quick", "Jesse")

presentar("Kent", "Clark")

La salida será diferente. ¿La puedes predecir?

# Paso de argumentos con palabras clave

Python ofrece otra manera de pasar argumentos, donde **el significado del argumento esta definido por su nombre**, no su posición, a esto se le denomina **paso de argumentos con palabras clave**.

Observa el siguiente código:

def presentar (primerNombre, segundoNombre):

print("Hola, mi nombre es", primerNombre, segundoNombre)

presentar(primerNombre = "James", segundoNombre = "Bond")

presentar(segundoNombre = "Skywalker", primerNombre = "Luke")

El concepto es claro: los valores pasados a los parámetros son precedidos por el nombre del parámetro al que se le va a pasar el valor, seguido por el signo de =.

La posición no es relevante aquí, cada argumento conoce su destino con base en el nombre utilizado.

Debes de poder predecir la salida. Ejecuta el código y verifica tu respuesta.

Por supuesto que **no se debe de utilizar el nombre de un parámetro que no existe**.

El siguiente código provocará un error de ejecución:

def presentar (primerNombre, segundoNombre):

print("Hola, mi nombre es ", primerNombre, segundoNombre)

introduction(apellido="Skywalker", primerNombre="Luke")

Esto es lo que Python arrojará:

TypeError: presentar() got an unexpected

**El combinar argumentos posicionales y de palabras clave**

Es posible combinar ambos tipos si se desea, solo hay una regla inquebrantable: se deben colocar primero los **argumentos posicionales y después los de palabras clave**.

Piénsalo por un momento y entenderás el porque.

Para mostrarte como funciona, se utilizara la siguiente función de tres parámetros:

def suma(a, b, c):

print(a, "+", b, "+", c, "=", a + b + c)

Su propósito es el de evaluar y presentar la suma de todos sus argumentos.

La función, al ser invocada de la siguiente manera:

suma(1, 2, 3)

Dará como salida:

1 + 2 + 3 = 6

Hasta ahorita es un ejemplo puro de **un argumento posicional**.También, se puede reemplazar la invocación actual por una con palabras clave, como la siguiente:

suma(c = 1, a = 2, b = 3)

El programa dará como salida lo siguiente:

2 + 3 + 1 = 6

Ten presente el orden de los valores.

Ahora intentemos mezclar ambas.

Observa la siguiente invocación de la función:

suma(3, c = 1, b = 2)

Vamos a analizarla:

* El argumento (3) para el parametro a es pasado utilizando la manera posicional.
* Los argumentos para c y b son especificados con palabras clave.

Esto es lo que se verá en la consola:

3 + 2 + 1 = 6

Se cuidadoso, ten cuidado de no cometer errores. Si se intenta pasar mas de un valor a un argumento, ocurrirá un error y se mostrará lo siguiente:

Observa la siguiente invocación, se le esta asignando dos veces un valor al parámetro a:

suma(3, a = 1, b = 2)

La respuesta de Python es:

TypeError: suma() got multiple values for argument 'a'

Observa el siguiente código. Es un código completamente correcto y funcional, pero no tiene mucho sentido:

suma(4, 3, c = 2)

Todo es correcto, pero el dejar solo un argumento con palabras clave es algo extraño - ¿Qué es lo que opinas?

# Funciones con parámetros: mas detalles

En ocasiones ocurre que algunos valores de ciertos argumentos son mas utilizados que otros. Dichos argumentos tienen **valores predefinidos** los cuales pueden ser considerados cuando los argumentos correspondientes han sido omitidos.

Uno de los apellidos más comunes en Latinoamérica es *González*. Tomémoslo para el ejemplo.

El valor por default para el parámetro se asigna de la siguiente manera:

def presentar(primerNombre, segundoNombre="González"):

print("Hola, mi nombre es", primerNombre, segundoNombre)

Solo se tiene que colocar el nombre del parámetro seguido del signo de = y el valor por default.

Invoquemos la función de manera normal:

presentar("Jorge", "Pérez")

¿Puedes predecir la salida del programa? Ejecútalo y revisa si era lo esperado.

¿Y? No parece haber cambiado algo, pero cuando se invoca la función de una manera inusual, como esta:

presentar("Enrique")

o así:

presentar (primerNombre="Guillermo")

no habrá errores, ambas invocaciones funcionarán, la consola mostrará los siguientes resultados:

Hola, mi nombre es Enrique González

Hola, mi nombre es Guillermo González

Pruébalo.

Puedes hacerlo con mas parámetros, si te resulta útil. Ambos parámetros tendrán sus valores por default, observa el siguiente código:

def presentar(primerNombre="Juan", segundoNombre="González"):

print("Hola, mi nombre es ", primerNombre, segundoNombre)

Esto hace que la siguiente invocación sea completamente valida:

presentar ()

Y esta es la salida esperada:

Hola, mi nombre es Juan González

Si solo se especifica un argumento de palabra clave, el restante tomará el valor por default:

presentar(segundoNombre="Rodríguez")

La salida es:

Hola, mi nombre es Juan Rodríguez   
Felicidades, has aprendido las maneras básicas de comunicación con funciones.

# Puntos Clave

# 1. Se puede pasar información a las funciones utilizando parámetros. Las funciones pueden tener tantos parámetros como sean necesarios.

Un ejemplo de una función con un parámetro:

def hola(nombre):

print("Hola,", nombre)

hola("Greg")

Un ejemplo de una función de dos parámetros:

def holaTodos(nombre1, nombre2):

print("Hola,", nombre2)

print("Hola,", nombre1)

holaTodos("Sebastián", "Felipe")

Un ejemplo de una función de tres parámetros:

def direccion(calle, ciudad, codigoPostal):

print("Tu dirección es:", calle, ciudad, codigoPostal)

c = input("Calle: ")

cp = input("Código Postal: ")

cd = input("Ciudad: ")

address(c, cd, cp)

2. Puedes pasar argumentos a una función utilizando las siguientes técnicas:

* **Paso de argumentos posicionales** en la cual el orden de los parámetros es relevante (Ejemplo 1).
* **Paso de argumentos con palabras clave**en la cual el orden de los argumentos es irrelevante (Ejemplo 2).
* Una mezcla de argumentos posicionales y con palabras clave (Ejemplo 3).

**Ejemplo 1**

def resta(a, b):

print(a - b)

resta(5, 2) # salida: 3

resta(2, 5) # salida: -3

**Ejemplo 2**

def resta(a, b):

print(a - b)

resta(a=5, b=2) # salida: 3

resta(b=2, a=5) # salida: 3

**Ex. 3**

def resta(a, b):

print(a - b)

resta(5, b=2) # salida: 3

resta(5, 2) # salida: 3

Es importante recordar que **primero se especifican los argumentos posicionales y después los de palabras clave**. Es por esa razón que si se intenta ejecutar el siguiente código:

def resta(a, b):

print(a - b)

resta(5, b=2) # salida: 3

resta(a=5, 2) # Syntax Error

Python no lo ejecutará y marcará un error de sintaxis SyntaxError.

3. Se puede utilizar la técnica de argumentos con palabras clave para asignar valores **predefinidos** a los argumentos:

def nombre(nombre, apellido="Pérez"):

print(nombre, apellido)

nombre("Andy") # salida: Andy Pérez

nombre("Bety", "Rodríguez") # salida: Bety Johnson (el argumento de palabra clave es reemplazado por " Rodríguez ")  
**Ejercicio 1**

¿Cuál es la salida del siguiente código?

def intro(a="James Bond", b="Bond"):

print("Mi nombre es", b + ".", a + ".")

intro()

Revisar

Mi nombre es Bond. James Bond.

**Ejercicio 2**

¿Cuál es la salida del siguiente código?

def intro(a="James Bond", b="Bond"):

print("Mi nombre es", b + ".", a + ".")

intro(b="Sergio López")

Revisar

Mi nombre es Sergio López. James Bond.

**Ejercicio 3**

¿Cuál es la salida del siguiente fragmento de código?

def intro(a, b="Bond"):

print("Mi nombre es", b + ".", a + ".")

intro("Susan")

Revisar

Mi nombre es Bond. Susan.

**Ejercicio 4**

¿Cuál es la salida del siguiente código?

def suma(a, b=2, c):

print(a + b + c)

suma(a=1, c=3)

Revisar

SyntaxError - a non-default argument (c) follows a default argument (b=2)

# Efectos y resultados: la instrucción return

Todas las funciones presentadas anteriormente tienen algún tipo de efecto: producen un texto y lo envían a la consola.

Por supuesto, las funciones, al igual que las funciones matemáticas, pueden tener resultados.

Para lograr que las **funciones devuelvan un valor** (pero no solo para ese propósito) se utiliza la instrucción return (regresar o retornar).

Esta palabra nos da una idea completa de sus capacidades. Nota: es una **palabra reservada** de Python.

La instrucción return tiene **dos variantes diferentes**: considerémoslas por separado.

## return sin una expresión

La primera consiste en la palabra reservada en sí, sin nada que la siga.

Cuando se emplea dentro de una función, provoca la **terminación inmediata de la ejecución de la función, y un retorno instantáneo (de ahí el nombre) al punto de invocación**.

Nota: si una función no está destinada a producir un resultado, **emplear la instrucción** return **no es obligatorio**, se ejecutará implícitamente al final de la función.

De cualquier manera, se puede emplear para **terminar las actividades de una función**, antes de que el control llegue a la última línea de la función.

Consideremos la siguiente función:

def felizAñoNuevo(deseos = True):

print("Tres ...")

print("Dos ...")

print("Uno ...")

if not deseos:

return

print("¡Feliz año nuevo!")

Cuando se invoca sin ningún argumento:

felizAñoNuevo()

La función produce un poco de ruido; la salida se verá así:

Tres ...

Dos...

Uno...

¡Feliz año nuevo!

Al proporcionar False como argumento:

felizAñoNuevo(False)

Se modificará el comportamiento de la función; la instrucción return provocará su terminación justo antes de los deseos. Esta es la salida actualizada:

Tres ...

Dos...

Uno ...

## return con una expresión

La segunda variante de return está **extendida con una expresión**:

funcion():

return expresión

Hay dos consecuencias de usarla:

* Provoca la **terminación inmediata de la ejecución de la función** (nada nuevo en comparación con la primer variante).
* Además, la función **evaluará el valor de la expresión y lo devolverá (de ahí el nombre una vez mas) como el resultado de la función**.

Este ejemplo es sencillo:

def funcion\_aburrida():

return 123

x = funcion\_aburrida()

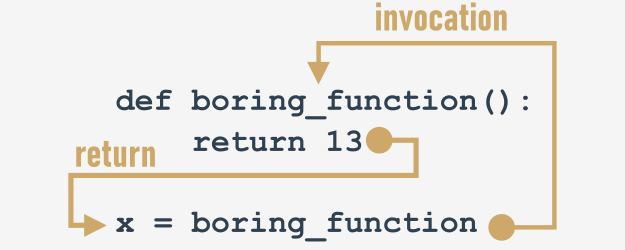
print ("La funcion\_aburrida ha devuelto su resultado. Es: ", x)

El fragmento de código escribe el siguiente texto en la consola:

La funcion\_aburrida ha devuelto su resultado. Es: 123

Vamos a investigarlo.

Analiza la siguiente figura:



La instrucción return, enriquecida con la expresión (la expresión es muy simple aquí), "transporta" el valor de la expresión al lugar donde se ha invocado la función.

El resultado se puede usar libremente aquí, por ejemplo, para ser asignado a una variable.

También puede ignorarse por completo y perderse sin dejar rastro.

Ten en cuenta que no estamos siendo muy educados aquí: la función devuelve un valor y lo ignoramos (no lo usamos de ninguna manera):

def funcion\_aburrida():

print("'Modo aburrimiento' ON.")

return 123

print("¡Esta lección es interesante!)

funcion\_aburrida()

print("Esta lección es aburrida ...")

El programa produce el siguiente resultado:

¡Esta lección es interesante!

'Modo aburrimiento' ON.

Esta lección es aburrida ...

¿Esta mal? De ninguna manera.

La única desventaja es que el resultado se ha perdido irremediablemente.

No olvides:

* Siempre se te **permite ignorar el resultado de la función** y estar satisfecho con el efecto de la función (si la función tiene alguno).
* Si una función intenta devolver un resultado útil, debe contener la segunda variante de la instrucción return.

Espera un segundo, ¿Significa esto que también hay resultados inútiles? Sí, en cierto sentido.

**Unas pocas palabras acerca de None**

Permítenos presentarte un valor muy curioso (para ser honestos, un valor que es ninguno) llamado None.

Sus datos no representan valor razonable alguno; en realidad, no es un valor en lo absoluto; por lo tanto, **no debe participar en ninguna expresión**.

Por ejemplo, un fragmento de código como el siguiente:

print(None + 2)

Causará un error de tiempo de ejecución, descrito por el siguiente mensaje de diagnóstico:

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'

Nota: None es una **palabra reservada**.

Solo hay dos tipos de circunstancias en las que None se puede usar de manera segura:

* Cuando **se le asigna a una variable** (o se devuelve como **el resultado de una función**).
* Cuando **se compara con una variable** para diagnosticar su estado interno.

Al igual que aquí:

valor = None

if valor == None:

print("Lo siento, no tienes ningún valor")

No olvides esto: si una función no devuelve un cierto valor utilizando una cláusula de expresión return, se asume que **devuelve implícitamente**None.

Vamos a probarlo.

**Algunas palabras acerca de None: continuación**

Echa un vistazo al código en el editor.

Es obvio que la función strangeFunction devuelve True cuando su argumento es par.

¿Qué es lo que regresa de otra manera?

Podemos usar el siguiente código para verificarlo:

print(strangeFunction(2))

print(strangeFunction(1))

Esto es lo que vemos en la consola:

True

None

No te sorprendas la próxima vez que veas None como el resultado de la función, puede ser el síntoma de un error sutil dentro de la función.

# Efectos y resultados: listas y funciones

Hay dos preguntas adicionales que deben responderse aquí.

El primero es: **¿Se puede enviar una lista a una función como un argumento?**

¡Por supuesto que se puede! Cualquier entidad reconocible por Python puede desempeñar el papel de un argumento de función, aunque debes asegurarte de que la función sea capaz de hacer uso de él.

Entonces, si pasas una lista a una función, la función tiene que manejarla como una lista.

Una función como la siguiente:

def sumaDeLista(lst):

sum = 0

for elem in lst:

sum += elem

return sum

y se invoca así:

print(sumaDeLista([5, 4, 3]))

Regresará 12 como resultado, pero habrá problemas si la invocas de esta manera riesgosa:

print(sumaDeLista(5))

La respuesta de Python será la siguiente:

TypeError: 'int' object is not iterable

Esto se debe al hecho de que el **bucle**for**no puede iterar un solo valor entero.**

# Efectos y resultados: listas y funciones - continuación

La segunda pregunta es: **¿Puede una lista ser el resultado de una función?**

¡Si, por supuesto! Cualquier entidad reconocible por Python puede ser un resultado de función.

Observa el código en el editor. La salida del programa será así:

[4, 3, 2, 1, 0]

Ahora puedes escribir funciones con y sin resultados.

Vamos a profundizar un poco más en los problemas relacionados con las variables en las funciones. Esto es esencial para crear funciones efectivas y seguras.

# Puntos clave

1. Puedes emplear la palabra clave return para decirle a una función que devuelva algún valor. La instrucción return termina la función, por ejemplo:

def multiply(a, b):

return a \* b

print(multiply(3, 4)) # salida: 12

def multiply(a, b):

return

print(multiply(3, 4)) # salida: None

2. El resultado de una función se puede asignar fácilmente a una variable, por ejemplo:

def deseos():

return "¡Felíz Cumpleaños!"

d = deseos()

print(d) # salida: ¡Felíz Cumpleaños!

Observa la diferencia en la salida en los siguientes dos ejemplos:

# Ejemplo 1

def deseos():

print("Mis deseos")

return "¡Felíz Cumpleaños!"

deseos() # salida: Mis deseos

# Ejemplo 2

def deseos():

print("Mis Deseos")

return "¡Feliz Cumpleaños!"

print(deseos()) # salidas: Mis Deseos

# ¡Feliz Cumpleaños!

3. Puedes usar una lista como argumento de una función, por ejemplo:

def HolaaTodos(myList):

for nombre in myList:

print("Hola,", nombre)

HolaaTodos(["Adam", "John", "Lucy"])

4. Una lista también puede ser un resultado de función, por ejemplo:

def createList(n):

myList = []

for i in range(n):

myList.append(i)

return myList

print(createList(5))

**Ejercicio 1**

¿Cuál es la salida del siguiente fragmento de código?

def hola():

return

print("¡Hola!")

hola()

Revisar

La función devolverá un valor None implícito.

**Ejercicio 2**

¿Cuál es la salida del siguiente fragmento de código?

def isInt(data):

if type(data) == int:

return True

elif type(data) == float:

return False

print(isInt(5))

print(isInt(5.0))

print(isInt("5"))

Revisar

True

False

None

**Ejercicio 3**

¿Cuál es la salida del siguiente fragmento de código?

def evenNumLst(ran):

lst = []

for num in range(ran):

if num % 2 == 0:

lst.append(num)

return lst

print(evenNumLst(11))

Revisar

[0, 2, 4, 6, 8, 10]

**Ejercicio 4**

¿Cuál es la salida del siguiente fragmento de código?

def listUpdater(lst):

updList = []

for elem in lst:

elem \*\*= 2

updList.append(elem)

return updList

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

print(listUpdater(l))

Revisar

[1, 4, 9, 16, 25]

# Las funciones y sus alcances (scopes)

Comencemos con una definición:

El **alcance de un nombre** (por ejemplo, el nombre de una variable) es la parte del código donde el nombre es reconocido correctamente.

Por ejemplo, el alcance del parámetro de una función es la función en si misma. El parámetro es inaccesible fuera de la función.

Vamos a revisarlo. Observa el código en el editor. ¿Que ocurrirá cuando se ejecute?

El programa no correrá. El mensaje de error dirá:

NameError: name 'x' is not defined

Esto era de esperarse.

Vamos a conducir algunos experimentos para mostrar cómo es que Python define los alcances y como los puedes utilizar para tu beneficio.

**Las funciones y sus alcances (scopes): continuación**

Comencemos revisando si una variable creada fuera de una función es visible dentro de una función. En otras palabras, ¿El nombre de la variable se propaga dentro del cuerpo de la función?

Observa el código en el editor. Ahí esta nuestro conejillo de indias.

def miFuncion():

print("¿Conozco a la variable?", var)

var = 1

miFuncion()

print(var)

El resultado de la prueba es positivo, el código da como salida:

¿Conozco a la variable? 1

1

La respuesta es: **una variable que existe fuera de una función tiene alcance dentro del cuerpo de la función**.

Esta regla tiene una excepción muy importante. Intentemos encontrarla.

Hagamos un pequeño cambio al código:

def miFuncion():

var = 2

print("¿Conozco a la variable?", var)

var = 1

miFuncion()

print(var)

El resultado ha cambiado también el código arroja una salida con una ligera diferencia:

¿Conozco a la variable? 2

1

¿Qué es lo que ocurrió?

* La variable var creada dentro de la función no es la misma que la que se definió fuera de ella, parece ser que hay dos variables diferentes con el mismo nombre.
* La variable de la función es una sombra de la variable fuera de la función.

La regla anterior se puede definir de una manera mas precisa y adecuada:

**Una variable que existe fuera de una función tiene un alcance dentro del cuerpo de la función, excluyendo a aquellas que tienen el mismo nombre.**

También significa que **el alcance de una variable existente fuera de una función solo se puede implementar dentro de una función cuando su valor es leído**. El asignar un valor hace que la función cree su propia variable.

Asegúrate bien de entender esto correctamente y de realizar tus propios experimentos.

# Las funciones y sus alcances (scopes): la palabra reservada global

Al llegar a este punto, debemos hacernos la siguiente pregunta: ¿Una función es capaz de modificar una variable que fue definida fuera de ella? Esto sería muy incómodo.

Afortunadamente, la respuesta es *no*.

Existe un método especial en Python el cual puede **extender el alcance de una variable incluyendo el cuerpo de las funciones** para poder no solo leer los valores de las variables sino también modificarlos.

Este efecto es causado por la palabra reservada llamada global:

global name

global name1, name2, ...

El utilizar la palabra reservada dentro de una función con el nombre o nombres de las variables separados por comas, obliga a Python a abstenerse de crear una nueva variable dentro de la función; se empleará la que se puede acceder desde el exterior.

En otras palabras, este nombre se convierte en global (tiene un **alcance global**, y no importa si se esta leyendo o asignando un valor).

Observa el código en el editor.

def miFuncion():

global var

var = 2

print("¿Conozco a aquella variable?", var)

var = 1

miFuncion()

print(var)

Se ha agregado la palabra global a la función.

El código ahora da como salida:

¿Conozco a aquella variable? 2

2

Esto debe de ser suficiente evidencia para mostrar lo que la palabra reservada global puede hacer.

**Como interactúa la función con sus argumentos**

Ahora descubramos como la función interactúa con sus argumentos.

El código en editor nos enseña algo. Como puedes observar, la función cambia el valor de su parámetro. ¿Este cambio afecta el argumento?

def miFuncion(n):

print("Yo obtuve", n)

n += 1

print("Yo ahora tengo", n)

var = 1

miFuncion(var)

print(var)

Ejecuta el programa y verifícalo.

La salida del código es:

Yo obtuve 1

Yo ahora tengo 2

1

La conclusión es obvia - **al cambiar el valor del parámetro este no se propaga fuera de la función**(más específicamente, no cuando la variable es un valor escalar, como en el ejemplo).

Esto también significa que una función recibe el **valor del argumento**, no el argumento en sí. Esto es cierto para los valores escalares. Es decir, se puede modificar el argumento pero su valor seguirá siendo el mismo.

Vale la pena revisar cómo funciona esto con las listas (¿Recuerdas las peculiaridades de asignar rodajas de listas en lugar de asignar la lista entera?)

El siguiente ejemplo arrojará luz sobre el asunto:

def miFuncion(miLista1):

print(miLista1)

miLista1 = [0, 1]

miLista2 = [2, 3]

miFuncion(miLista2)

print(miLista2)

La salida del código es:

[2, 3]

[2, 3]

Parece ser que se sigue aplicando la misma regla.

La diferencia se puede observar en el siguiente ejemplo:

def miFuncion(miLista1):

print(miLista1)

del miLista1[0]

miLista2 = [2, 3]

miFuncion(miLista2)

print(miLista2)

No se modifica el valor del parámetro miLista1 (ya se sabe que no afectará el argumento), en lugar de ello se modificará la lista identificada por el.

El resultado puede ser sorprendente. Ejecuta el código y verifícalo:

[2, 3]

[3]

¿Lo puedes explicar?

Intentémoslo:

* Si el argumento es una lista, el cambiar el valor del parámetro correspondiente no afecta la lista (Recuerda: las variables que contienen listas son almacenadas de manera diferente que las escalares).
* Pero si se modifica la lista identificada por el parámetro (Nota: ¡La lista no el parámetro!), la lista reflejará el cambio.

Es tiempo de escribir algunos ejemplos de funciones. Lo harás en la siguiente sección.

Si una variable está definida dentro de una función, sólo la leerá esa función, exceptuando que se use global para que la variable sea leída también fuera de la función.

Si al final del código yo pongo print(una variable), me va a imprimir la variable que esté definida fuera de la función, si pongo por ejemplo fun() me va a imprimir la variable que está definida dentro de la función.

Cuando se usa la palabra reservada “global”, el programa va a leer la variable global cuando al llamarla la función está expresada después de la definición de otra variable. EJ: a =3, fun(), print(a), en este caso va a mostrar la variable a que está dentro de la función.

Cuando se define otra variable con el mismo nombre después de la función, se va a tomar el valor de la variable que está definida después de la función.

# Puntos Clave

1. Una variable que existe fuera de una función tiene alcance dentro del cuerpo de la función. (Ejemplo 1) al menos que la función defina una variable con el mismo nombre. (Ejemplo 2, y Ejemplo 3), por ejemplo:

Ejemplo 1:

var = 2

def multByVar(x):

return x \* var

print(multByVar(7)) # salida: 14

Ejemplo 2:

def mult(x):

var = 5

return x \* var

print(mult(7)) # salida: 35

Ejemplo 3:

def multip(x):

var = 7

return x \* var

var = 3

print(multip(7)) # salida: 49

2. Una variable que existe dentro de una función tiene un alcance solo dentro del cuerpo de la función (Ejemplo 4), por ejemplo:

Ejemplo 4:

def sum(x):

var = 7

return x + var

print(sum(4)) # salida: 11

print(var) # NameError

3. Se puede emplear la palabra reservada global seguida por el nombre de una variable para que el alcance de la variable sea global, por ejemplo:

var = 2

print(var) # salida: 2

def retVar():

global var

var = 5

return var

print(retVar()) # salida: 5

print(var) # salida: 5

**Ejercicio 1**

¿Qué ocurrirá cuando se intente ejecutar el siguiente código?

def message():

alt = 1

print("Hola, mundo!")

print(alt)

Revisar

Se arrojará una excepción NameError(NameError: name 'alt' is not defined)

**Ejercicio 2**

¿Cuál es la salida del siguiente fragmento de código?

a = 1

def fun():

a = 2

print(a)

fun()

print(a)

Revisar

2

1

**Ejercicio 3**

¿Cuál es la salida del siguiente fragmento de código?

a = 1

def fun():

global a

a = 2

print(a)

fun()

a = 3

print(a)

Revisar

2

3

**Ejercicio 4**

¿Cuál es la salida del siguiente fragmento de código?

a = 1

def fun():

global a

a = 2

print(a)

a = 3

fun()

print(a)

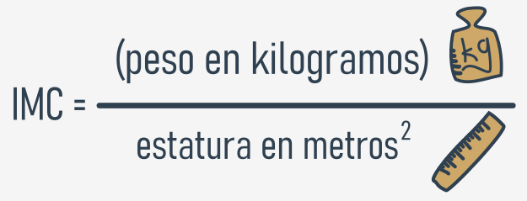
Revisar

2

2

**Funciones Simples: Calcular el IMC**

Definamos una función que calcula el Índice de Masa Corporal (IMC).



Como puedes observar, la formula ocupa dos valores:

* peso (originalmente en kilogramos)
* altura (originalmente en metros)

La nueva función tendrá **dos parámetros**. Su nombre será imc, pero si prefieres utilizar otro nombre, adelante.

Codifiquemos la función.

La función esta completa aquí abajo (y en la ventana de editor):

def imc(peso, altura):

return peso / altura \*\* 2

print(imc(52.5, 1.65))

El resultado del ejemplo anterior es el siguiente:

19.283746556473833

La función hace lo que deseamos, pero es un poco sencilla - asume que los valores de ambos parámetros son significativos. Se debe comprobar que son confiables.

Vamos a comprobar ambos y regresar None si cualquiera de los dos es incorrecto.

# Algunas funciones simples: calcular el IMC y convertir unidades del sistema inglés al sistema métrico

Observa el código en el editor. Hay dos cosas a las cuales hay que prestar atención.

Primero, se asegura que los datos que sean ingresados sean correctos, de lo contrario la salida será:

None

Segundo, observa como el símbolo de **diagonal invertida** (\) es empleado. Si se termina una línea de código con el, Python entenderá que la línea continua en la siguiente.

Esto puede ser útil cuando se tienen largas líneas de código y se desea que sean más legibles.Sin embargo, hay algo que omitimos: las medias en sistema inglés. La función no es útil para personas que utilicen libras, pies y pulgadas.

¿Qué podemos hacer por ellos?

Escribimos dos funciones sencillas para **convertir unidades del sistema inglés al sistema métrico**. Comencemos con las pulgadas.

Es bien conocido que 1 lb = 0.45359237 kg. Esto lo emplearemos en nuestra nueva función.

Esta función se llamará lbakg:

def lbakg(lb):

return lb \* 0.45359237

print(lbakg(1))

El resultado de la prueba es el siguiente:

0.45359237

Haremos lo mismo ahora con los pies y pulgadas: 1 pie = 0.3048 m, y 1 pulgada = 2.54 cm = 0.0254 m.

La función se llamará piepulgam:

def piepulgam(pie, pulgada):

return pie \* 0.3048 + pulgada \* 0.0254

print(piepulgam(1, 1))

El resultado de una prueba rápida es:

0.3302

Resulta como esperado.

Vamos a convertir seis pies a metros:

print(piespulgam(6, 0))

Esta es la salida:

1.8288000000000002

Es muy posible que en ocasiones se desee utilizar solo pies sin pulgadas. ¿Python nos ayudará? Por supuesto que si.

Se ha modificado el código un poco:

def piepulgam(pie, pulgada = 0.0):

return pie \* 0.3048 + pulgada \* 0.0254

print(piepulgam(6))

Ahora el parámetro pulgada tiene como valor predeterminado el 0.0.

El código produce la siguiente salida, esto es lo que se esperaba:

1.8288000000000002

Finalmente, el código es capaz de responder a la pregunta: ¿Cual es el IMC de una persona que tiene 5'7" de altura y un peso de 176 lbs?

Este es el código que debemos de escribir:

def piespulgam(pies, pulgadas = 0.0):

return pies \* 0.3048 + pulgadas \* 0.0254

def lbsakg(lb):

return lb \* 0.45359237

def imc(peso, altura):

if altura < 1.0 or altura > 2.5 or \

peso < 20 or peso > 200:

return None

return peso / altura \*\* 2

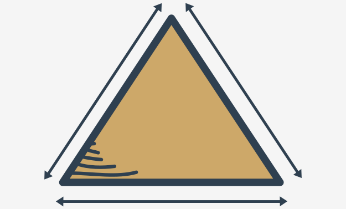
print(imc(peso = lbsakg(176), altura = piespulgam(5, 7)))

La respuesta es:

27.565214082533313

# Algunas funciones simples: continuación

Ahora trabajaremos con triángulos. Comenzaremos con una función que verifique si tres lados de ciertas longitudes pueden formar un triángulo.



En la escuela aprendimos que *la suma arbitraria de dos lados tiene que ser mayor que la longitud del tercer lado*.

No será algo difícil. La función tendrá **tres parámetros** - uno para cada lado.

Regresará True si todos los lados pueden formar un triángulo, y False de lo contrario. En este caso, esUnTringulo es un buen nombre para dicha función.

Observa el código en el editor. Ahí se encuentra la función. Ejecuta el programa.

def esUnTriangulo(a, b, c):

if a + b <= c:

return False

if b + c <= a:

return False

if c + a <= b:

return False

return True

print(esUnTriangulo (1, 1, 1))

print(esUnTriangulo (1, 1, 3))

Parece que funciona perfectamente: estos son los resultados:

True

False

¿Se podrá hacer más compacta?. Parece tener demasiadas palabras.

Esta es la versión más compacta:

def esUnTriangulo(a, b, c):

if a + b <= c or b + c <= a or \

c + a <= b:

return False

return True

print(esUnTriangulo(1, 1, 1))

print(esUnTriangulo(1, 1, 3))

¿Se puede compactar aún más?

Por supuesto, observa:

def esUnTriangulo (a, b, c):

return a + b > c and b + c > a and c + a > b

print(esUnTriangulo (1, 1, 1))

print(esUnTriangulo (1, 1, 3))

Se ha negado la condición (se invirtieron los operadores relacionales y se reemplazaron los ors con ands, obteniendo una **expresión universal para probar triángulos**).

Coloquemos la función en un programa más grande. Se le pedirá al usuario los tres valores y se hará uso de la función.

# Algunas funciones simples: triángulos y el teorema de Pitágoras

Observa el código en el editor. Le pide al usuario tres valores. Después hace uso de la función esUnTriangulo. El código esta listo para correrse.

def esUnTriangulo(a, b, c):

return a + b > c and b + c > a and c + a > b

a = float(input("Ingresa la longitud del primer lado: "))

b = float(input("Ingresa la longitud del segundo lado: "))

c = float(input("Ingresa la longitud del tercer lado: "))

if esUnTriangulo(a, b, c):

print("Felicidades, puede ser un triángulo.")

else:

print("Lo siento, no puede ser un triángulo.")

En el segundo paso, intentaremos verificar si un triángulo es un **triángulo rectángulo**.

Para ello haremos uso del **Teorema de Pitágoras**:

**c2 = a2 + b2**

¿Cómo saber cual de los tres lados es la hipotenusa?

**La hipotenusa es el lado mas largo**.

Aquí esta el código:

def esUnTriangulo(a, b, c):

return a + b > c and b + c > a and c + a > b

def esUnTrianguloRectangulo(a, b, c):

if not esUnTriangulo (a, b, c):

return False

if c > a and c > b:

return c \*\* 2 == a \*\* 2 + b \*\* 2

if a > b and a > c:

return a \*\* 2 == b \*\* 2 + c \*\* 2

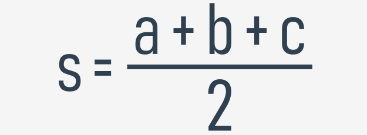
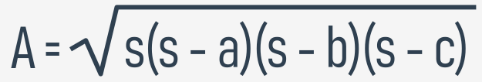
print(esUnTrianguloRectangulo(5, 3, 4))

print(esUnTrianguloRectangulo(1, 3, 4))

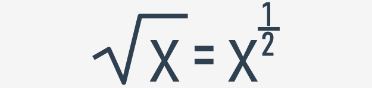
Observa cómo se establece la relación entre la hipotenusa y los dos catetos. Se eligió el lado más largo y se aplicó el **Teorema de Pitágoras** para verificar que todo estuviese en orden. Esto requiere tres revisiones en total.

# Algunas funciones simples: evaluando el campo de un triángulo

También es posible evaluar el campo de un triángulo. **La Fórmula de Heron** será útil aquí:

Vamos a emplear el operador de exponenciación para calcular la raíz cuadrada - puede ser extraño, pero funciona.

  
Este es el código resultante:

def esUnTriangulo(a, b, c):

return a + b > c and b + c > a and c + a > b

def heron(a, b, c):

p = (a + b + c) / 2

return (p \* (p - a) \* (p - b) \* (p - c)) \*\* 0.5

def campoTriangulo(a, b, c):

if not esUnTriangulo(a, b, c):

return None

return heron(a, b, c)

print(campoTriangulo(1., 1., 2. \*\* .5))

Lo probaremos con un triángulo rectángulo la mitad de un cuadrado y con un lado igual a 1. Esto significa que su campo debe ser igual a 0.5.

Es extraño pero el código produce la siguiente salida:

0.49999999999999983

Es muy cercano a 0.5, pero no es exactamente 0.5,¿Que significa?, ¿Es un error?

No, no lo es. Son solo los **cálculos de valores punto flotantes**. Pronto se discutirá el tema.

# Algunas funciones simples: factoriales

La siguiente función a definir calcula **factoriales**. ¿Recuerdas cómo se calcula un factorial?

0! = 1 (¡Si!, es verdad.)

1! = 1

2! = 1 \* 2

3! = 1 \* 2 \* 3

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

:

:

n! = 1 \* 2 \*\* 3 \* 4 \* ... \* n-1 \* n

Se expresa con un **signo de exclamación**, y es igual al **producto** de todos los números naturales previos al argumento o número dado.

Escribamos el código. Creemos una función con el nombre factorialFun. Aquí está el código:

def factorialFun(n):

if n < 0:

return None

if n < 2:

return 1

producto = 1

for i in range(2, n + 1):

producto \*= i

return producto

for n in range(1, 6): # probando

print(n, factorialFun(n))

Observa como se sigue el procedimiento matemático, y como se emplea el bucle for para **encontrar el producto**.

Estos son los resultados obtenidos de un código de prueba:

1 1

2 2

3 6

4 24

5 120

**Algunas funciones simples: Serie Fibonacci**

¿Estás familiarizado con la serie **Fibonacci**?

Son una **secuencia de números enteros** los cuales siguen una regla sencilla:

* El primer elemento de la secuencia es igual a uno (**Fib1 = 1**).
* El segundo elemento también es igual a uno (**Fib2 = 1**).
* Cada numero después de ellos son la suman de los dos números anteriores (**Fibi = Fibi-1 + Fibi-2**).

Aquí están algunos de los primeros números en la serie Fibonacci:

fib1 = 1

fib2 = 1

fib3 = 1 + 1 = 2

fib4 = 1 + 2 = 3

fib5 = 2 + 3 = 5

fib6 = 3 + 5 = 8

fib7 = 5 + 8 = 13

¿Que opinas acerca de **implementarlo como una función**?

Creemos nuestra propia función fib y probémosla, aquí esta:

def fib(n):

if n < 1:

return None

if n < 3:

return 1

elem1 = elem2 = 1

sum = 0

for i in range(3, n + 1):

sum = elem1 + elem2

elem1, elem2 = elem2, sum

return sum

for n in range(1, 10): # probando

print(n, "->", fib(n))

Analiza el codigo del bucle for cuidadosamente, descifra como se **mueven las variables**elem1**y**elem2**a través de los números subsecuentes de la serie Fibonacci.**

Al probar el código, se generan los siguientes resultados:

1 -> 1

2 -> 1

3 -> 2

4 -> 3

5 -> 5

6 -> 8

7 -> 13

8 -> 21

9 -> 34

# Algunas funciones simples: recursividad

Existe algo más que se desea mostrar: es la **recursividad**.

Este término puede describir muchos conceptos distintos, pero uno de ellos, hace referencia a la programación computacional.

Aquí, la recursividad es una **técnica donde una función se invoca a si misma**.

Tanto el factorial como la serie Fibonacci, son las mejores opciones para ilustrar este fenómeno.

**La serie de Fibonacci es un claro ejemplo de recursividad**.

**Fibi = Fibi-1 + Fibi-2**

El número ith se refiere al número i-1, y así sucesivamente hasta llegar a los primeros dos.

¿Puede ser empleado en el código? Por supuesto que puede. Puede hacer el código más corto y claro.

La segunda versión de la función fib() hace uso directo de la recursividad:

def fib(n):

if n < 1:

return None

if n < 3:

return 1

return fib(n - 1) + fib(n - 2)  
El código es mucho más claro ahora.

¿Pero es realmente seguro?, ¿Implica algún riesgo?

Si, existe algo de riesgo. **Si no se considera una condición que detenga las invocaciones recursivas, el programa puede entrar en un bucle infinito**. Se debe ser cuidadoso.

El factorial también tiene un lado **recursivo**. Observa:

n! = 1 × 2 × 3 × ... × n-1 × n

Es obvio que:

1 × 2 × 3 × ... × n-1 = (n-1)!

Entonces, finalmente, el resultado es:

n! = (n-1)! × n

Esto se empleará en nuestra nueva solución.

Aquí esta:

def factorialFun(n):

if n < 0:

return None

if n < 2:

return 1

return n \* factorialFun(n - 1)

¿Funciona? Claro que sí. Pruébalo por ti mismo.

Nuestro viaje *funcional* está por terminar. La siguiente sección abordara dos tipos de datos en Python: tuplas y diccionarios.

# Puntos Clave

1. Una función puede invocar otras funciones o incluso a sí misma. Cuando una función se invoca a si misma, se le conoce como **recursividad**, y la función que se invoca a si misma y contiene una condición de terminación (la cual le dice a la función que ya no siga invocándose a si misma) es llamada una función **recursiva**.

2. Se pueden emplear funciones recursivas en Python para crear funciones **limpias, elegantes, y dividir el código en trozos más pequeños**. Sin embargo, se debe tener mucho cuidado ya que es **muy fácil cometer un error y crear una función la cual nunca termine.** También se debe considerar que **las funciones recursivas consumen mucha memoria**, y por lo tanto pueden ser en ocasiones ineficientes.

Al emplear la recursividad, se deben de tomar en cuenta tanto sus ventajas como desventajas.

La función factorial es un ejemplo clásico de como se puede implementar el concepto de recursividad:

# Implementación recursiva de la función factorial

def factorial(n):

if n == 1: # la condición de terminación

return 1

else:

return n \* factorial(n - 1)

print(factorial(4)) # 4 \* 3 \* 2 \* 1 = 24  
  
**Ejercicio 1**

¿Qué ocurrirá al intentar ejecutar el siguiente fragmento de código y porque?

def factorial(n):

return n \* factorial(n - 1)

print(factorial(4))

Revisar

La función no tiene una condición de terminación, por lo tanto Python arrojara una excepción (RecursionError: maximum recursion depth exceeded)

**Ejercicio 2**

¿Cuál es la salida del siguiente fragmento de código?

def fun(a):

if a > 30:

return 3

else:

return a + fun(a + 3)

print(fun(25))

Revisar

56

# Tipos de secuencias y mutabilidad

Antes de comenzar a hablar acerca de **tuplas** y **diccionarios**, se deben introducir dos conceptos importantes: **tipos de secuencia** y **mutabilidad**.

Un **tipo de secuencia es un tipo de dato en Python el cual es capaz de almacenar mas de un valor (o ninguno si la secuencia esta vacía), los cuales pueden ser secuencialmente (de ahí el nombre) examinados**, elemento por elemento.

Debido a que el bucle for es una herramienta especialmente diseñada para iterar a través de las secuencias, podemos definirlas de la siguiente manera: **una secuencia es un tipo de dato que puede ser escaneado por el bucle**for.

Hasta ahora, has trabajado con una secuencia en Python, la lista. La lista es un clásico ejemplo de una secuencia de Python. Aunque existen otras secuencias dignas de mencionar, las cuales se presentaran a continuación.

La segunda noción - **la mutabilidad** - es una propiedad de cualquier tipo de dato en Python que describe su disponibilidad para poder cambiar libremente durante la ejecución de un programa. Existen dos tipos de datos en Python: **mutables** e **inmutables**.

**Los datos mutables pueden ser actualizados libremente en cualquier momento**, a esta operación se le denomina "in situ".

*In situ* es una expresión en Latín que se traduce literalmente como *en posición, en el lugar o momento*. Por ejemplo, la siguiente instrucción modifica los datos "in situ":

list.append(1)

**Los datos inmutables no pueden ser modificados de esta manera**.

Imagina que una lista solo puede ser asignada y leída. No podrías adjuntar ni remover un elemento de la lista. Si se agrega un elemento al final de la lista provocaría que la lista se cree desde cero.

Se tendría que crear una lista completamente nueva, la cual contenga los elementos ya existentes mas el nuevo elemento.

El tipo de datos que se desea tratar ahora se llama **tupla**.**Una tupla es una secuencia inmutable**. Se puede comportar como una lista pero no puede ser modificada en el momento.

## ¿Qué es una tupla?

Lo primero que distingue una lista de una tupla es la sintaxis empleada para crearlas. Las **tuplas utilizan paréntesis**, mientras que las listas usan corchetes, aunque también es **posible crear una tupla tan solo separando los valores por comas**.

Observa el ejemplo:

tupla1 = (1, 2, 4, 8)

tupla2 = 1., .5, .25, .125

Se definieron dos tuplas, ambas contienen **cuatro elementos**.

A continuación se imprimen en consola:

print(tupla1)

print(tupla2)

Esto es lo que se muestra en consola:

(1, 2, 4, 8)

(1.0, 0.5, 0.25, 0.125)

Nota: **cada elemento de una tupla puede ser de distinto tipo** (punto flotante, entero, cadena, etc.).

## ¿Cómo crear una tupla?

¿Es posible crear una tupla vacía? Si, solo se necesitan unos paréntesis:

tuplaVacia = ()

Si se desea crear una tupla de **un solo elemento**, se debe de considerar el hecho de que, debido a la sintaxis (una tupla debe de poder distinguirse de un valor entero ordinario), se debe de colocar una coma al final:

tuplaUnElemento1 = (1, )

tuplaUnElemento2 = 1.,

El quitar las comas no arruinará el programa en el sentido sintáctico, pero serán variables no tuplas.

# ¿Cómo utilizar una tupla?

Si deseas leer los elementos de una tupla, lo puedes hacer de la misma manera que se hace con las listas.

Observa el código en el editor.

miTupla = (1, 10, 100, 1000)

print(miTupla[0])

print(miTupla[-1])

print(miTupla[1:])

print(miTupla[:-2])

for elem in miTupla:

print(elem)

El programa debe de generar la siguiente salida, ejecútalo y comprueba:

1

1000

(10, 100, 1000)

(1, 10)

1

10

100

1000

Las similitudes pueden ser engañosas - **no intentes modificar en contenido de la tupla**¡No es una lista!

Todas estas instrucciones (con excepción de primera) causarán un error de ejecución:

miTupla = (1, 10, 100, 1000)

miTupla.append(10000)

del miTupla[0]

miTupla[1] = -10

Este es el mensaje que Python arrojará en la ventana de consola:

AttributeError: 'tuple' object has no attribute 'append'

**¿Cómo utilizar una tupla? continuación**

¿Qué más pueden hacer las tuplas?

* La función len() acepta tuplas, y regresa el numero de elementos contenidos dentro.
* El operador + puede unir tuplas (ya se ha mostrado esto antes).
* El operador \* puede multiplicar las tuplas, así como las listas.
* Los operadores in y not in funcionan de la misma manera que en las listas.

El fragmento de código en el editor presenta todo esto.

miTupla = (1, 10, 100)

t1 = miTupla + (1000, 10000)

t2 = miTupla \* 3

print(len(t2))

print(t1)

print(t2)

print(10 in miTupla)

print(-10 not in miTupla)

La salida es la siguiente:

9

(1, 10, 100, 1000, 10000)

(1, 10, 100, 1, 10, 100, 1, 10, 100)

True

True

Una de las propiedades de las tuplas más útiles es que pueden **aparecer en el lado izquierdo del operador de asignación**. Este fenómeno ya se vio con anterioridad, cuando fue necesario encontrar una manera de intercambiar los valores entre dos variables.

Observa el siguiente fragmento de código:

var = 123

t1 = (1, )

t2 = (2, )

t3 = (3, var)

t1, t2, t3 = t2, t3, t1

print(t1, t2, t3)

Muestra tres tuplas interactuando en efecto, los valores almacenados en ellas "circulan" entre ellas. t1 se convierte en t2, t2 se convierte en t3, y t3 se convierte en t1.

Nota: el ejemplo presenta un importante hecho más: los **elementos de una tupla pueden ser variables**, no solo literales. Además, pueden ser expresiones si se encuentran en el lado derecho del operador de asignación.

# ¿Qué es un diccionario?

El **diccionario** es otro tipo de estructura de datos de Python. No **es una secuencia** (pero puede adaptarse fácilmente a un procesamiento secuencial) y además es **mutable**.

Para explicar lo que es un diccionario en Python, es importante comprender de manera literal lo que es un diccionario.

Un diccionario en Python funciona de la misma manera que **un diccionario bilingüe**. Por ejemplo, se tiene la palabra en español "gato" y se necesita su equivalente en francés. Lo que se haría es buscar en el diccionario para encontrar la palabra "gato". Eventualmente la encontrarás, y sabrás que la palabra equivalente en francés es "chat".

En el mundo de Python, la palabra que se está buscando se denomina clave(key). La palabra que se obtiene del diccionario es denominada valor.

Esto significa que un diccionario es un conjunto de pares de **claves y valores**. Nota:

* Cada clave debe de ser **única**. No es posible tener una clave duplicada.
* Una clave puede ser **un tipo de dato de cualquier tipo**: puede ser un número (entero o flotante), o incluso una cadena.
* Un diccionario no es una lista. Una lista contiene un conjunto de valores numerados, mientras que un **diccionario almacena pares de valores**.
* La función len() aplica también para los diccionarios, regresa la cantidad de pares (clave-valor) en el diccionario.
* Un diccionario es **una herramienta de un solo sentido**. Si fuese un diccionario español-francés, podríamos buscar en español para encontrar su contraparte en francés mas no viceversa.

A continuación veamos algunos ejemplos:

## ¿Cómo crear un diccionario?

Si deseas asignar algunos pares iniciales a un diccionario, utiliza la siguiente sintaxis:

dict = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

numerosTelefono = {'jefe' : 5551234567, 'Suzy' : 22657854310}

diccionarioVacio = {}

print(dict)

print(numerosTelefono)

print(diccionarioVacio)

En este primer ejemplo, el diccionario emplea claves y valores las cuales ambas son cadenas. En el segundo, las claves con cadenas pero los valores son enteros. El orden inverso (claves → números, valores → cadenas) también es posible, así como la combinación número a número.

La lista de todos los pares es **encerrada con llaves**, mientras que los pares son **separados por comas**, y las **claves y valores por dos puntos**.

El primer diccionario es muy simple, es un diccionario Español-Francés. El segundo es un directorio telefónico muy pequeño.

Los diccionarios vacíos son construidos por **un par vacío de llaves** - nada inusual.

El diccionario entero se puede imprimir con una invocación a la función print(). El fragmento de código **puede** producir la siguiente salida:

{'perro': 'chien', 'caballo': 'cheval', 'gato': 'chat'}

{'Suzy': 5557654321, 'boss': 5551234567}

{}

¿Has notado que el orden de los pares impresos es diferente a la asignación inicial?, ¿Qué significa esto?

Primeramente, recordemos que **los diccionarios no son listas** - no guardan el orden de sus datos, el orden no tiene significado (a diferencia de los diccionarios reales). El orden en que un diccionario **almacena sus datos esta fuera de nuestro control**. Esto es normal. (\*)

**NOTA**

(\*) En Python 3.6x los diccionarios se han convertido en colecciones **ordenadas** de manera predeterminada. Tu resultado puede variar dependiendo en la versión de Python que se este utilizando.

**¿Cómo utilizar un diccionario?**

Si deseas obtener cualquiera de los valores, debes de proporcionar una clave válida:

print(dict['gato'])

print(numerosTelefono['Suzy'])

El obtener el valor de un diccionario es semejante a la indexación, gracias a los corchetes alrededor del valor de la clave.

Nota:

* Si una clave es una cadena, se tiene que especificar como una cadena.
* **Las claves son sensibles a las mayúsculas y minúsculas**: 'Suzy' sería diferente a 'suzy'.

El fragmento de código da las siguientes salidas:

chat

5557654321

Ahora algo muy importante: **No se puede utilizar una clave que no exista**. Hacer algo como lo siguiente:

print(numerosTelefono['presidente'])

Provocará un error de ejecución. Inténtalo.Afortunadamente, existe una manera simple de evitar dicha situación. El operador in, junto con su acompañante, not in, pueden salvarnos de esta situación.

El siguiente código busca de manera segura palabras en francés:

dict = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

words = ['gato', 'leon', 'caballo']

for word in words:

if word in dict:

print(word, "->", dict[word])

else:

print(word, "no está en el diccionario")

La salida del código es la siguiente:

gato -> chat

leon no está en el diccionario

caballo -> cheval

# ¿Cómo utilizar un diccionario? El método keys()

¿Pueden los diccionarios ser **examinados** utilizando el bucle for, como las listas o tuplas?

No y si.

No, porque un diccionario **no es un tipo de dato secuencial** - el bucle for no es útil aquí.

Si, porque hay herramientas simples y muy efectivas que pueden **adaptar cualquier diccionario a los requerimientos del bucle**for (en otras palabras, se construye un enlace intermedio entre el diccionario y una entidad secuencial temporal).

El primero de ellos es un método denominado keys(), el cual es parte de todo diccionario. El método **retorna o regresa una lista de todas las claves dentro del diccionario**. Al tener una lista de claves se puede acceder a todo el diccionario de una manera fácil y útil.

A continuación se muestra un ejemplo:

dict = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

for key in dict.keys():

print(key, "->", dict[key]

El código produce la siguiente salida:

caballo -> cheval

perro -> chien

gato -> chat

## La función sorted()

¿Deseas que la salida este **ordenada**? Solo hay que agregar al bucle for lo siguiente:

for key in sorted(dict.keys()):

La función sorted() hará su mejor esfuerzo y la salida será la siguiente:

caballo -> cheval

gato -> chat

perro -> chien

# ¿Cómo utilizar un diccionario? Los métodos item() y values()

Otra manera de hacerlo es utilizar el método items(). Este método **regresa una lista de tuplas** (este es el primer ejemplo en el que las tuplas son mas que un ejemplo de si mismas) **donde cada tupla es un par de cada clave con su valor**.

Así es como funciona:

dict = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

for spanish, french in dict.items():

print(spanish, "->", french)

Nota la manera en que la tupla ha sido utilizada como una variable del bucle for.

El ejemplo imprime lo siguiente:

cat -> chat

dog -> chien

horse -> cheval

También existe un método denominado values(), funciona de manera muy similar al de keys(), pero **regresa una lista de valores**.

Este es un ejemplo sencillo:

dict = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

for french in dict.values():

print(french)

Como el diccionario no es capaz de automáticamente encontrar la clave de un valor dado, el rol de este método es algo limitado.

Esta es la salida esperada:

cheval

chien

chat

# ¿Cómo utilizar un diccionario? Modificar, agregar y eliminar valores

El asignar un nuevo valor a una clave existente es sencillo, debido a que los diccionarios son completamente **mutables**, no existen obstáculos para modificarlos.

Se va a reemplazar el valor "chat" por "minou", lo cual no es muy adecuado, pero funcionará con nuestro ejemplo.

Observa:

dict = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

dict['gato'] = 'minou'

print(dict)

La salida es:

{'perro': 'chien', 'caballo': 'cheval', 'gato': 'minou'}

## Agregando nuevas claves

El agregar una nueva clave con su valor a un diccionario es tan simple como cambiar un valor. Solo se tiene que asignar un valor a una nueva **clave que no haya existido antes**.

Nota: este es un comportamiento muy diferente comparado a las listas, las cuales no permiten asignar valores a índices no existentes.

A continuación se agrega un par nuevo al diccionario, un poco extraño pero valido:

dict = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

dict['cisne'] = 'cygne'

print(dict)

El ejemplo muestra como salida:

{'cisne': 'cygne', 'caballo': 'cheval', 'perro': 'chien', 'gato': 'chat'}

**EXTRA**

También es posible insertar un elemento al diccionario utilizando el método update(), por ejemplo:

dict = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

dict.update({"pato" : "canard"})

print(dict)

## Eliminando claves

¿Puedes deducir como eliminar una clave de un diccionario?

Nota: al eliminar la clave también se **removerá el valor asociado**. **Los valores no pueden existir sin sus claves**.

Esto se logra con la instrucción del.

A continuación un ejemplo:

dict = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

del dict['perro']

print(dict)

Nota: **el eliminar una clave no existente, provocará un error**.

El ejemplo da como salida:

{'gato': 'chat', 'caballo': 'cheval'}

**EXTRA**

Para eliminar el ultimo elemento de la lista, se puede emplear el método popitem():

dict = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

dict.popitem()

print(dict) # salida: {'gato' : 'chat', 'perro' : 'chien'}

En versiones anteriores de Python, por ejemplo, antes de la 3.6.7, el método popitem() elimina un elemento al azar del diccionario.

**Las tuplas y los diccionarios pueden trabajar juntos**

Se ha preparado un ejemplo sencillo, mostrando como las tuplas y los diccionarios pueden trabajar juntos.

Imaginemos el siguiente problema:

* Necesitas un programa para calcular los promedios de tus alumnos.
* El programa pide el nombre del alumno seguido de su calificación.
* Los nombres son ingresados en cualquier orden.
* El ingresar la palabra exit da por terminado el ingreso de nombres.
* Una lista con todos los nombres y el promedio de cada alumno debe ser mostrada al final.

Observa el código en el editor, se muestra la solución.

grupo = {}

while True:

nombre = input("Ingresa el nombre del estudiante (o exit para detenerse): ")

if nombre == 'exit':

break

calif = int(input("Ingresa la calificación del alumno (0-10): "))

if nombre in grupo:

grupo[nombre] += (calif,)

else:

grupo[nombre] = (calif,)

for nombre in sorted(grupo.keys()):

sum = 0

contador = 0

for calif in grupo[nombre]:

sum += calif

contador += 1

print(nombre, ":", sum / contador)

Ahora se analizará línea por línea:

* **Línea 1**: crea un diccionario vacío para ingresar los datos: el nombre del alumno es empleado como clave, mientras que todas las calificaciones asociadas son almacenadas en una tupla (la tupla puede ser el valor de un diccionario, esto no es un problema).
* **Línea 3**: se ingresa a un bucle "infinito" (no te preocupes, saldrémos de el en el momento indicado).
* **Línea 4**: se lee el nombre del alumno.
* **Línea 5-6**: si el nombre es exit, nos salimos del bucle.
* **Línea 8**: se pide la calificación del alumno (un valor entero en el rango del 1-10).
* **Línea 10-11**: si el nombre del estudiante ya se encuentra en el diccionario, se alarga la tupla asociada con la nueva calificación (observa el operador +=).
* **Línea 12-13**: si el estudiante es nuevo (desconocido para el diccionario), se crea una entrada nueva, su valor es una tupla de un solo elemento la cual contiene la calificación ingresada.
* **Línea 15**: se itera a través de los nombres ordenados de los estudiantes.
* **Línea 16-17**: inicializa los datos necesarios para calcular el promedio (sumador y contador).
* **Línea 18-20**: Se itera a través de la tupla, tomado todas las calificaciones subsecuentes y actualizando la suma junto con el contador.
* **Línea 21**: se calcula e imprime el promedio del alumno junto con su nombre.

Este es un ejemplo del programa:

Ingresa el nombre del estudiante (o exit para detenerse): Bob

Ingresa la calificación del alumno (0-10): 7

Ingresa el nombre del estudiante (o exit para detenerse): Andy

Ingresa la calificación del alumno (0-10): 3

Ingresa el nombre del estudiante (o exit para detenerse): Bob

Ingresa la calificación del alumno (0-10): 2

Ingresa el nombre del estudiante (o exit para detenerse): Andy

Ingresa la calificación del alumno (0-10): 10

Ingresa el nombre del estudiante (o exit para detenerse): Andy

Ingresa la calificación del alumno (0-10): 3

Ingresa el nombre del estudiante (o exit para detenerse): Bob

Ingresa la calificación del alumno (0-10): 9

Ingresa el nombre del estudiante (o exit para detenerse): exit

Andy : 5.333333333333333

Bob : 6.0

# Puntos Clave: Tuplas

1. Las **Tuplas** son colecciones de datos ordenadas e inmutables. Se puede pensar en ellas como listas inmutables. Se definen con paréntesis:

miTupla = (1, 2, True, "una cadena", (3, 4), [5, 6], None)

print(miTupla)

miLista = [1, 2, True, "una cadena", (3, 4), [5, 6], None]

print(miLista)

Cada elemento de la tupla puede ser de un tipo de dato diferente (por ejemplo, enteros, cadenas, boleanos, etc.). Las tuplas pueden contener otras tuplas o listas (y viceversa).

2. Se puede crear una tupla vacía de la siguiente manera:

tuplaVacia = ()

print(type(tuplaVacia)) # salida: <class 'tuple'>

3. La tupla de un solo elemento se define de la siguiente manera:

tuplaUnElemento = ("uno", ) # paréntesis y coma

tuplaUnElemento2 = "uno", # sin paréntesis, solo la coma

Si se elimina la coma, Python creará una variable no una tupla:

miTup1 = 1,

print(type(miTup1)) # salida: <class 'tuple'>

miTup2 = 1

print(type(miTup2)) # salida: <class 'int'>

4. Se pueden acceder los elementos de la tupla al indexarlos:

miTupla = (1, 2.0, "cadena", [3, 4], (5, ), True)

print(miTupla[3]) # salida: [3, 4]

5. Las tuplas son inmutables, lo que significa que no se puede agregar, modificar, cambiar o quitar elementos. El siguiente fragmento de código provocará una excepción:

miTupla = (1, 2.0, "cadena", [3, 4], (5, ), True)

miTupla[2] = "guitarra" # se levanta una excepción TypeError

Sin embargo, se puede eliminar la tupla completa:

miTupla = 1, 2, 3,

del miTupla

print(miTupla) # NameError: name 'miTupla' is not defined

6. Puedes navegar a través de los elementos de una tupla con un bucle (Ejemplo 1), verificar si un elemento o no esta presente en la tupla (Ejemplo 2), emplear la función len() para verificar cuantos elementos existen en la tupla (Ejemplo 3), o incluso unir o multiplicar tuplas (Ejemplo 4):

# Ejemplo 1

t1 = (1, 2, 3)

for elem in t1:

print(elem)

# Ejemplo 2

t2 = (1, 2, 3, 4)

print(5 in t2)

print(5 not in t2)

# Ejemplo 3

t3 = (1, 2, 3, 5)

print(len(t3))

# Ejemplo 4

t4 = t1 + t2

t5 = t3 \* 2

print(t4)

print(t5)

**EXTRA**

También se puede crear una tupla utilizando la función integrada de Python tuple(). Esto es particularmente útil cuando se desea convertir un iterable (por ejemplo, una lista, rango, cadena, etcétera) en una tupla:

miTup = tuple((1, 2, "cadena"))

print(miTup)

lst = [2, 4, 6]

print(lst) # salida: [2, 4, 6]

print(type(lst)) # salida: <class 'list'>

tup = tuple(lst)

print(tup) # outputs: (2, 4, 6)

print(type(tup)) # salida: <class 'tuple'>

De la misma manera, cuando se desea convertir un iterable en una liste, se puede emplear la función integrada de Python denominada list():

tup = 1, 2, 3,

lst = list(tup)

print(type(lst)) # outputs: <class 'list'>

# Puntos Clave: diccionarios

1. Los diccionarios son **\***colecciones indexadas de datos, mutables y desordenadas. (**\***En Python 3.6x los diccionarios están ordenados de manera predeterminada.

Cada diccionario es un par de *clave : valor*. Se puede crear empleado la siguiente sintaxis:

miDictionario = {

clave1 : valor1,

clave2 : valor2,

clave3 : valor3,

}

2. Si se desea acceder a un elemento del diccionario, se puede hacer haciendo referencia a su clave colocándola dentro de corchetes (ejemplo 1) o utilizando el método get() (ejemplo 2):

polEspDict = {

"kwiat" : "flor",

"woda" : "agua",

"gleba" : "tierra"

}

elemento1 = polEspDict["gleba"] # ejemplo 1

print(elmento1) # salida: tierra

elemento2 = polEspDict.get("woda")

print(elemento2) # salida: agua

3. Si se desea cambiar el valor asociado a una clave específica, se puede hacer haciendo referencia a la clave del elemento, a continuación se muestra un ejemplo:

polEspDict = {

"zamek" : "castillo",

"woda" : "agua",

"gleba" : "tierra"

}

polEspDict["zamek"] = "cerradura"

item = polEspDict["zamek"] # salida: cerradura

4. Para agregar o eliminar una clave (junto con su valor asociado), emplea la siguiente sintaxis:

miDirectorioTelefonico = {} # un diccionario vacio

miDirectorioTelefonico ["Adan"] = 3456783958 # crear o añadir un par clave-valor

print(miDirectorioTelefonico) # salida: {'Adan': 3456783958}

del miDirectorioTelefonico ["Adan"]

print(miDirectorioTelefonico) # salida: {}

Además, se puede insertar un elemento a un diccionario utilizando el método update(), y eliminar el último elemento con el método popitem(), por ejemplo:

polEspDict = {"kwiat" : "flor"}

polEspDict = update("gleba" : "tierra")

print(polEspDict) # salida: {'kwiat' : 'flor', 'gleba' : 'tierra'}

polEspDict.popitem()

print(polEspDict) # outputs: {'kwiat' : 'flor'}

5. Se puede emplear el bucle for para iterar a través del diccionario, por ejemplo:

polEspDict = {

"zamek" : "castillo",

"woda" : "agua",

"gleba" : "tierra"

}

for item in polEspDict:

print(item) # salida: zamek

# woda

# gleba

6. Si deseas examinar los elementos (claves y valores) del diccionario, puedes emplear el método items() por ejemplo:

polEspDict = {

"zamek" : "castillo",

"woda" : "agua",

"gleba" : "tierra"

}

for clave, valor in polEspDict.items():

print("Pol/Esp ->", clave, ":", valor)

7. Para comprobar si una clave existe en un diccionario, se puede emplear la palabra reservada in:

polEspDict = {

"zamek" : "castillo",

"woda" : "agua",

"gleba" : "tierra"

}

if "zamek" in polEspDict:

print("SI")

else:

print("NO")

8. Se puede emplear la palabra reservada del para eliminar un elemento, o un diccionario entero. Para eliminar todos los elementos de un diccionario se debe emplear el método clear():

polEspDict = {

"zamek" : "castillo",

"woda" : "agua",

"gleba" : "tierra"

}

print(len(polEspDict)) # salida: 3

del polEspDict["zamek"] # elimina un elemento

print(len(polEspDict)) # salida: 2

polEspDict.clear() # elimina todos los elementos

print(len(polEspDict)) # salida: 0

del polEspDict # elimina el diccionario

9. Para copiar un diccionario, emplea el método copy():

polEspDict = {

"zamek" : "castillo",

"woda" : "agua",

"gleba" : "tierra"

}

copyDict = polEspDict.copy()

# Puntos Claves: Tuplas y diccionarios

**Ejercicio 1**

¿Que ocurrirá cuando se intente ejecutar el siguiente código?

miTup = (1, 2, 3)

print(miTup[2])

Revisar

El programa imprimirá 3 en pantalla.

**Ejercicio 2**

¿Cuál es la salida del siguiente fragmento de código?

tup = 1, 2, 3

a, b, c = tup

print(a \* b \* c)

Revisar

El programa imprimirá 6 en pantalla. Los elementos de la tupla tup han sido "desempaquetados" en las variables a, b, y c.

**Ejercicio 3**

Completa el código para emplear correctamente el método count() para encontrar la cantidad de 2 duplicados en la tupla siguiente.

tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9

duplicados = # tu código

print(duplicados) # salida: 4

Revisar

tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9

duplicates = tup.count(2)

print(duplicado) # salida: 4

**Ejercicio 4**

Escribe un programa que "una" los dos diccionarios (d1 y d2) para crear uno nuevo (d3).

d1 = {'Adam Smith':'A', 'Judy Paxton':'B+'}

d2 = {'Mary Louis':'A', 'Patrick White':'C'}

d3 = {}

for elemento in (d1, d2):

# tu código

print(d3)

Revisar

Solución Muestra:  
d1 = {'Adam Smith':'A', 'Judy Paxton':'B+'}

d2 = {'Mary Louis':'A', 'Patrick White':'C'}

d3 = {}

for elemento in (d1, d2):

d3.update(elemento)

print(d3)

**Ejercicio 5**

Escribe un programa que convierta la lista l en una tupla.

l = ["carro", "Ford", "flor", "Tulipán"]

t = # tu código

print(t)

Revisar

Solución muestra:  
l = ["carro", "Ford", "flor", "Tulipán"]

t = tuple(l)

print(t)

**Ejercicio 6**

Escribe un programa que convierta la tupla colores en un diccionario.

colores = (("verde", "#008000"), ("azul", "#0000FF"))

# tu código

print(colDict)

Revisar

colores = (("verde", "#008000"), ("azul", "#0000FF"))

colDict = dict(colores)

print(colDict)

**Ejercicio 7**

¿Que ocurrirá cuando se ejecute el siguiente código?

miDict = {"A":1, "B":2}

copyMiDict = myDict.copy()

miDict.clear()

print(copyMiDict)

Revisar

El programa mostrará {'A': 1, 'B': 2} en pantalla

**Ejercicio 8**

¿Cuál es la salida del siguiente programa?

colores = {

"blanco" : (255, 255, 255),

"gris" : (128, 128, 128),

"rojo" : (255, 0, 0),

"verde" : (0, 128, 0)

}

for col, rgb in colores.items():

print(col, ":", rgb)

Revisar

blanco : (255, 255, 255)

gris : (128, 128, 128)

rojo : (255, 0, 0)

verde : (0, 128, 0)

**Módulo 5 Módulos, paquetes, cadenas, métodos de listas y excepciones**

**En este módulo, aprenderás acerca de:**

* Módulos de Python: su lógica, su función, cómo importarlos de diferentes maneras y presentar el contenido de algunos módulos estándar proporcionados por Python.
* La forma en que los módulos se unen para formar paquetes.
* El concepto de una excepción y su implementación en Python, incluida la instrucción try-except, con sus aplicaciones y la instrucción raise.
* Cadenas y sus métodos específicos, junto con sus similitudes y diferencias en comparación con las listas.

**¿Qué es un módulo?**

El código de computadora tiene una tendencia a crecer. Podemos decir que el código que no crece probablemente sea completamente inutilizable o este abandonado. Un código real, deseado y ampliamente utilizado se desarrolla continuamente, ya que tanto las demandas de los usuarios como las expectativas de los usuarios se desarrollan a su propio ritmo.

Un código que no puede responder a las necesidades de los usuarios se olvidará rápidamente y se reemplazará instantáneamente con un código nuevo, mejor y más flexible. Se debe estar preparado para esto, y nunca pienses que tus programas están terminados por completo. La finalización es un estado de transición y generalmente pasa rápidamente, después del primer informe de error. Python en sí es un buen ejemplo de cómo actúa esta regla.

El código creciente es, de hecho, un problema creciente. Un código más grande siempre significa un mantenimiento más difícil. La búsqueda de errores siempre es más fácil cuando el código es más pequeño (al igual que encontrar una rotura mecánica es más simple cuando la maquinaria es más simple y más pequeña).

Además, cuando se espera que el código que se está creando sea realmente grande (puedes usar el número total de líneas de código como una medida útil, pero no muy precisa, del tamaño del código) entonces, se deseará, o más bien, habrá la necesidad de dividirlo en muchas partes, implementado en paralelo por unos cuantos, una docena, varias docenas o incluso varios cientos de desarrolladores.

Por supuesto, esto no se puede hacer usando un archivo fuente grande, el cual esta siendo editado por todos los programadores al mismo tiempo. Esto seguramente conducirá a un desastre.

Si se desea que dicho proyecto de software se complete con éxito, se deben tener los medios que le permitan:

* Se deben dividir todas las tareas entre los desarrolladores.
* Despues, unir todas las partes creadas en un todo funcional.

Por ejemplo, un determinado proyecto se puede dividir en dos partes principales:

* La interfaz de usuario (la parte que se comunica con el usuario mediante widgets y una pantalla gráfica).
* La lógica (la parte que procesa los datos y produce resultados).

Cada una de estas partes se puede (muy probablemente) dividir en otras más pequeñas, y así sucesivamente. Tal proceso a menudo se denomina **descomposición**.

Por ejemplo, si te pidieran organizar una boda, no harías todo tu mismo: encontrarías una serie de profesionales y dividirías la tarea entre todos.

¿Cómo se divide una pieza de software en partes separadas pero cooperantes? Esta es la pregunta. **Módulos** son la respuesta.

**¿Cómo hacer uso de un módulo?**

El manejo de los módulos consta de dos cuestiones diferentes:

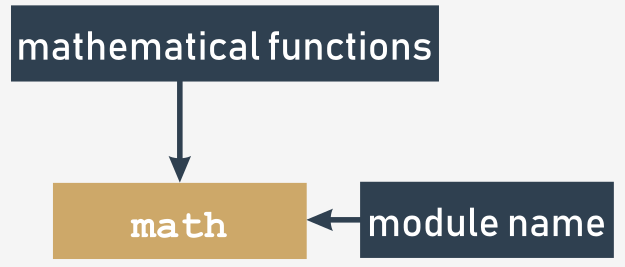
* El primero (probablemente el más común) ocurre cuando se desea utilizar un módulo ya existente, escrito por otra persona o creado por el programador mismo en algún proyecto complejo: en este caso, se considera al programador como el **usuario** del módulo.
* El segundo ocurre cuando se desea crear un nuevo módulo, ya sea para uso propio o para facilitar la vida de otros programadores: aquí eres el **proveedor** del módulo.

Discutamos ambas por separado.

En primer lugar, un módulo se identifica por su **nombre**. Si se desea utilizar cualquier módulo, se necesita saber su nombre. Se entrega una cantidad (bastante grande) de módulos junto con Python. Se puede pensar en ellos como una especie de "equipo extra de Python".

Todos estos módulos, junto con las funciones integradas, forman la **Biblioteca estándar de Python** - un tipo especial de biblioteca donde los módulos desempeñan el papel de libros (incluso podemos decir que las carpetas desempeñan el papel de estanterías). Si deseas ver la lista completa de todos los "volúmenes" recopilados en esa biblioteca, se puede encontrar aquí: https://docs.python.org/3/library/index.html.

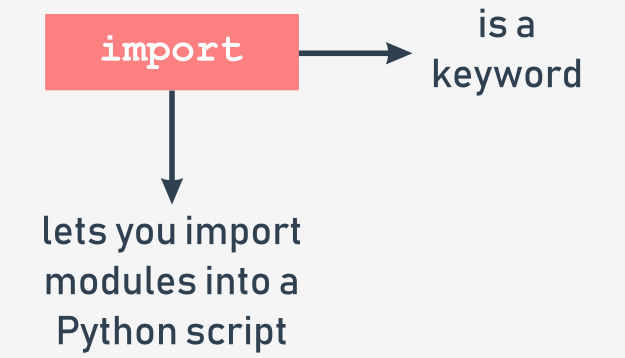
Cada módulo consta de entidades (como un libro consta de capítulos). Estas entidades pueden ser funciones, variables, constantes, clases y objetos. Si se sabe cómo acceder a un módulo en particular, se puede utilizar cualquiera de las entidades que almacena.



Comencemos la discusión con uno de los módulos más utilizados, el que lleva por nombre math. Su nombre habla por sí mismo: el módulo contiene una rica colección de entidades (no solo funciones) que permiten a un programador implementar efectivamente cálculos que exigen el uso de funciones matemáticas, como *sen()* o *log()*.

**Importando un módulo**

Para que un módulo sea utilizable, hay que **importarlo** (piensa en ello como sacar un libro del estante). La importación de un módulo se realiza mediante una instrucción llamada import. Nota: import es también una palabra reservada (con todas sus implicaciones).



Supongamos que deseas utilizar dos entidades proporcionadas por el módulo math:

* Un símbolo (constante) que representa un valor preciso (tan preciso como sea posible usando aritmética de punto flotante doble) de π (aunque usar una letra griega para nombrar una variable es totalmente posible en Python, el símbolo se llama **pi**: es una solución más conveniente, especialmente para esa parte del mundo que ni tiene ni va a usar un teclado griego).
* Una función llamada sin() (el equivalente informático de la función matemática *sine*).

Ambas entidades están disponibles a través del módulo math, pero la forma en que se pueden usar depende en gran medida de cómo se haya realizado la importación.

La forma más sencilla de importar un módulo en particular es usar la instrucción de importación de la siguiente manera:

import math

La cláusula contiene:

* La palabra reservada import.
* El **nombre del módulo** que se va a importar.

La instrucción puede colocarse en cualquier parte del código, pero debe colocarse **antes del primer uso de cualquiera de las entidades del módulo**.

Si se desea (o se tiene que) importar más de un módulo, se puede hacer repitiendo la cláusula import, o listando los módulos despues de la palabra reservada import, por ejemplo:

import math, sys

La instrucción importa dos módulos, primero uno llamado math y luego un segundo llamado sys.

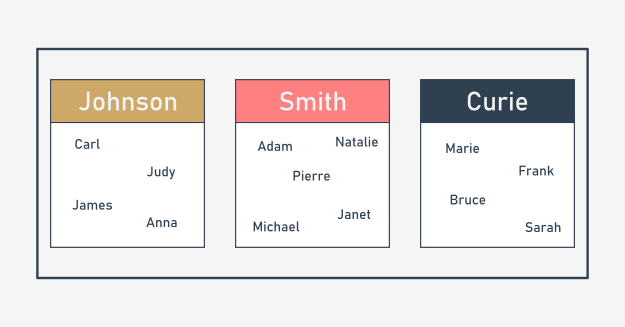
La lista de módulos puede ser arbitrariamente larga.

# Importando un módulo: continuación

Para continuar, debes familiarizarte con un término importante: **namespace**.

No te preocupes, no entraremos en detalles: esta explicación será lo más breve posible.

Un **namespace** es un espacio (entendido en un contexto no físico) en el que existen algunos nombres y los nombres no entran en conflicto entre sí (es decir, no hay dos objetos diferentes con el mismo nombre). Podemos decir que cada grupo social es un namespace - el grupo tiende a nombrar a cada uno de sus miembros de una manera única (por ejemplo, los padres no darán a sus hijos los mismos nombres).

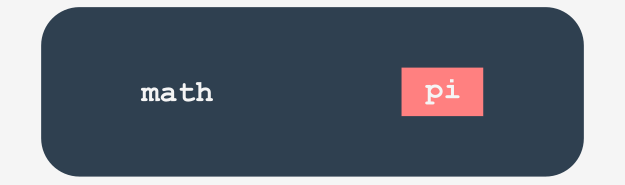


Esta singularidad se puede lograr de muchas maneras, por ejemplo, mediante el uso de apodos junto con los nombres (funcionará dentro de un grupo pequeño como una clase en una escuela) o asignando identificadores especiales a todos los miembros del grupo (el Seguro Social de EE. UU. El número es un buen ejemplo de tal práctica).

**Dentro de un determinado namespace, cada nombre debe permanecer único**. Esto puede significar que algunos nombres pueden desaparecer cuando cualquier otra entidad de un nombre ya conocido ingresa al namespace. Mostraremos cómo funciona y cómo controlarlo, pero primero, volvamos a las importaciones.

Si el módulo de un nombre especificado **existe y es accesible** (un módulo es de hecho un **archivo fuente de Python**), Python importa su contenido, **se hacen conocidos todos los nombres definidos en el módulo**, pero no ingresan al namespace del código.

Esto significa que puede tener sus propias entidades llamadas sin o pi y no serán afectadas en alguna manera por el import.



En este punto, es posible que te estes preguntando cómo acceder al pi el cual viene del módulo math.

Para hacer esto, se debe de mandar llamar el pi con el su nombre en el módulo original.

**Importando un módulo: continuación**

Observa el fragmento a continuación, esta es la forma en que se habilitan los nombres de pi y sin con el nombre de su módulo de origen:

math.pi

math.sin

Es sencillo, se pone:

* El **nombre del módulo** (math).
* Un **punto**.
* Y el **nombre de la entidad** (pi).

Tal forma indica claramente el namespace en el que existe el nombre.

Nota: el uso de esto es **obligatorio** si un módulo ha sido importado con la instrucción import. No importa si alguno de los nombres del código y del namespace del módulo están en conflicto o no.

Este primer ejemplo no será muy avanzado: solo se desea imprimir el valor de **sin(1/2π)**.

Observa el código en el editor. Así es como se prueba.

import math

print(math.sin(math.pi/2))

El código genera el valor esperado: 1.0.

Nota: el eliminar cualquiera de las dos indicaciones hará que el código sea erróneo. No hay otra forma de entrar al namespace de math si se hizo lo siguiente:

import math

# Importando un módulo: continuación

Ahora te mostraremos como pueden dos namespaces (el tuyo y el del módulo) coexistir.

Echa un vistazo al ejemplo en la ventana del editor.

import math

def sin(x):

if 2 \* x == pi:

return 0.99999999

else:

return None

pi = 3.14

Se ha definido una variable y función propia para pi y sin respectivamente, y se emplean junto con los de la librería math.

Ejecuta el programa. El código debe producir la siguiente salida:

0.99999999

1.0

Como puedes ver, las entidades no se afectan entre sí.

**Importando un módulo: continuación**

En el segundo método, la sintaxis del import señala con precisión qué entidad (o entidades) del módulo son aceptables en el código:

from math import pi

La instrucción consta de los siguientes elementos:

* La palabra reservada from.
* El **nombre del módulo** a ser (selectivamente) importado.
* La palabra reservada import.
* El **nombre o lista de nombres de la entidad o entidades** las cuales estan siendo importadas al namespace.

La instrucción tiene este efecto:

* Las entidades listadas son las unicas que son **importadas del módulo indicado**.
* Los nombres de las entidades importadas pueden **ser accedidas dentro del programa**.

Nota: no se importan otras entidades, únicamente las especificadas. Además, no se pueden importar entidades adicionales utilizando una línea como esta:

print(math.e)

Esto ocasionará un error, (e es la constante de Euler: 2.71828...).

Reescribamos el código anterior para incorporar esta nueva técnica.

Aquí esta:

from math import sin, pi

print(sin(pi/2))

El resultado debe de ser el mismo que el anterior, se han empleado las mismas entidades: 1.0. Copia y pega el código en el editor, y ejecuta el programa.

¿El código parece más simple? Quizás, pero el aspecto no es el único efecto de este tipo de importación. Veamos mas a detalle esto.

**Importando un módulo: continuación**

Observa el código en el editor. Analízalo cuidadosamente:

from math import sin, pi #Linea 1

print(sin(pi/2)) #Linea 3

pi = 3.14 #Linea 5

def sin(x):

if 2 \* x == pi:

return 0.99999999

else:

return None #Linea 11

print(sin(pi/2)) #Linea 13

* La línea 01: lleva a cabo la importación selectiva.
* La línea 03: hace uso de las entidades importadas y obtiene el resultado esperado (1.0).
* La línea 05 a la 11: redefine el significado de pi y sin - en efecto, reemplazan las definiciones originales (importadas) dentro del namespace del código.
* La línea 13: retorna 0.99999999, lo cual confirma nuestras conclusiones.

Hagamos otra prueba. Observa el código a continuación:

pi = 3.14 # linea 01

def sin(x):

if 2 \* x == pi:

return 0.99999999

else:

return None # linea 07

print(sin(pi/2)) # linea 09

from math import sin, pi # linea 12

print(sin(pi/2)) # linea 14

Aquí, se ha invertido la secuencia de las operaciones del código:

* Las líneas del 01 al 07: definen nuestro propio pi y sin.
* La línea 09: hace uso de ellas ( 0.99999999 aparece en pantalla).
* La línea 12: lleva a cabo la importación - los símbolos importados reemplazan sus definiciones anteriores dentro del namespace.
* La línea 14: retorna 1.0 como resultado.

# Importando un Módulo: \*

En el tercer método, la sintaxis del import es una forma más agresiva que la presentada anteriormente:

from module import \*

Como puedes ver, el nombre de una entidad (o la lista de nombres de entidades) se reemplaza con un solo asterisco (\*).

Tal instrucción **importa todas las entidades del módulo indicado**.

¿Es conveniente? Sí, lo es, ya que libera del deber de enumerar todos los nombres que se necesiten.

¿Es inseguro? Sí, a menos que conozca todos los nombres proporcionados por el módulo, **es posible que no puedas evitar conflictos de nombres**. Trata esto como una solución temporal e intenta no usarlo en un código regular.

# Importando un módulo: la palabra reservada as

Si se importa un módulo y no se esta conforme con el nombre del módulo en particular (por ejemplo, sí es el mismo que el de una de sus entidades ya definidas) puede darsele otro nombre: esto se llama **aliasing** o renombrado.

Aliasing (renombrado) hace que el módulo se identifique con un nombre diferente al original.

La creación de un alias se realiza junto con la importación del módulo, y exige la siguiente forma de la instrucción import:

import module as alias

el "module" identifica el nombre del módulo original mientras que el "alias" es el nombre que se desea usar en lugar del original.

Nota: as es una palabra reservada.

# Importando un Módulo: continuación

Si necesitas cambiar la palabra math, puedes introducir tu propio nombre, como en el ejemplo:

import math as m

print(m.sin(m.pi/2))

Nota: después de la ejecución exitosa de una importación con alias, **el nombre original del módulo se vuelve inaccesible** y no debe ser utilizado.

A su vez, cuando usa la variante from module import name y se necesita cambiar el nombre de la entidad, se crea un alias para la entidad. Esto hará que el nombre sea reemplazado por el alias que se elija.

Así es como se puede hacer:

from module import nombre as alias

Como anteriormente, el nombre original (sin alias) se vuelve inaccesible.

La frase nombre as alias puede repetirse: emplea comas para separar las frases, como a continuación:

from module import n as a, m as b, o as c

El ejemplo puede parecer un poco extraño, pero funciona:

from math import pi as PI, sin as sine

print(sine(PI/2))

Ahora estás familiarizado con los conceptos básicos del uso de módulos. Permítenos mostrarte algunos módulos y algunas de sus entidades útiles.

# Trabajando con módulos estándar

Antes de comenzar a revisar algunos módulos estándar de Python, veamos la función dir(). No tiene nada que ver con el comando dir de las terminales de Windows o Unix. El comando dir() no muestra el contenido de un directorio o carpeta de disco, pero no se puede negar que hace algo similar: puede revelar todos los nombres proporcionados a través de un módulo en particular.

Hay una condición: el módulo debe haberse importado previamente como un todo (es decir, utilizar la instrucción import module - from module no es suficiente).

La función devuelve una **lista ordenada alfabéticamente** la cual contiene todos los nombres de las entidades disponibles en el módulo:

dir(module)

Nota: Si el nombre del módulo tiene un alias, debe usar el alias, no el nombre original.

Usar la función dentro de un script normal no tiene mucho sentido, pero aún así, es posible.

Por ejemplo, se puede ejecutar el siguiente código para imprimir los nombres de todas las entidades dentro del módulo math:

import math

for name in dir(math):

print(name, end="\t")

El código de ejemplo debería producir el siguiente resultado:

\_\_doc\_\_ \_\_loader\_\_ \_\_name\_\_ \_\_package\_\_ \_\_spec\_\_ acos acosh asin asinh atan atan2 atanh ceil copysign cos cosh degrees e erf erfc exp expm1 fabs factorial floor fmod frexp fsum gamma hypot isfinite isinf isnan ldexp lgamma log log10 log1p log2 modf pi pow radians sin sinh sqrt tan tanh trunc

¿Has notado los nombres extraños que comienzan con \_\_ al inicio de la lista? Se hablará más sobre ellos cuando hablemos sobre los problemas relacionados con la escritura de módulos propios.

Algunos de los nombres pueden traer recuerdos de las lecciones de matemáticas, y probablemente no tendrás ningún problema en adivinar su significado.

El emplear la función dir() dentro de un código puede no parecer muy útil; por lo general, se desea conocer el contenido de un módulo en particular antes de escribir y ejecutar el código.

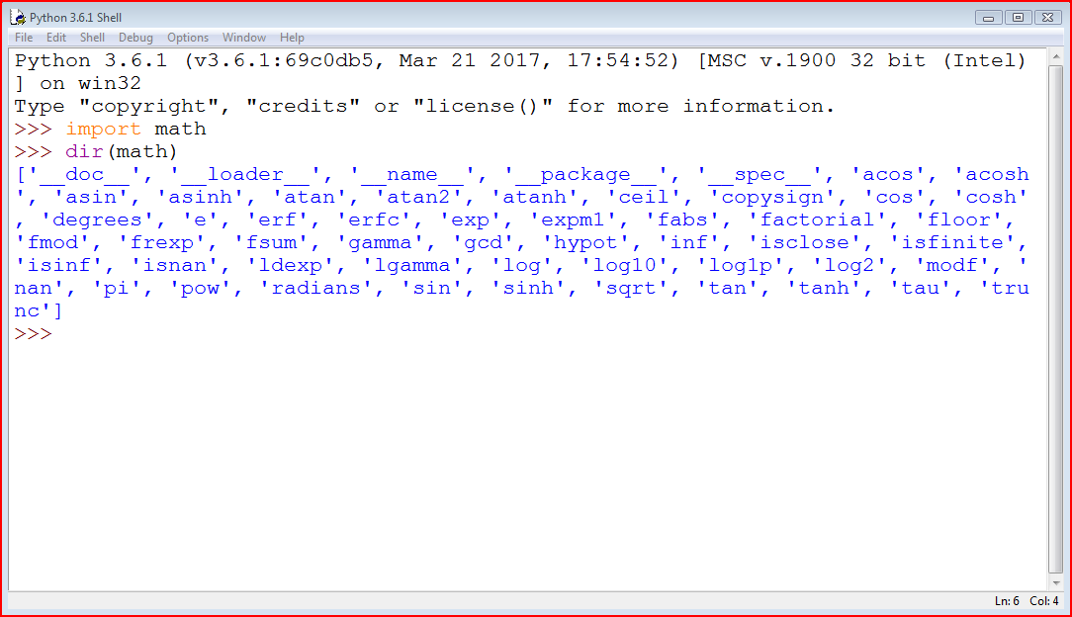
Afortunadamente, se puede ejecutar la función **directamente en la consola de Python** (IDLE), sin necesidad de escribir y ejecutar un script por separado.

Así es como se puede hacer:

import math

dir(math)

Deberías de ver algo similar a esto:



**Funciones seleccionadas del módulo math**

Comencemos con una vista previa de algunas de las funciones proporcionadas por el módulo math.

Se han elegido algunas arbitrariamente, pero esto no significa que las funciones no mencionadas aquí sean menos significativas. Tomate el tiempo para revisar las demás por ti mismo: no tenemos el espacio ni el tiempo para hablar de todas a detalle.

El primer grupo de funciones de módulo math están relacionadas con **trigonometría**:

* sin(x) → el seno de x.
* cos(x) → el coseno de x.
* tan(x) → la tangente de x.

Todas estas funciones toman un argumento (una medida de ángulo expresada en radianes) y devuelven el resultado apropiado (ten cuidado con tan() - no todos los argumentos son aceptados).

También están sus versiones inversas:

* asin(x) → el arcoseno de x.
* acos(x) → el arcocoseno de x.
* atan(x) → el arcotangente de x.

Estas funciones toman un argumento (verifica que sea correcto) y devuelven una medida de un ángulo en radianes.

Para trabajar eficazmente con mediciones de ángulos, el módulo math proporciona las siguientes entidades:

* pi → una constante con un valor que es una aproximación de π.
* radians(x) → una función que convierte x de grados a radianes.
* degrees(x) → actuando en el otro sentido (de radianes a grados).

Ahora observa el código en el editor. El programa de ejemplo no es muy sofisticado, pero ¿puedes predecir sus resultados?

from math import pi, radians, degrees, sin, cos, tan, asin

ad = 90

ar = radians(ad)

ad = degrees(ar)

print(ad == 90.)

print(ar == pi / 2.)

print(sin(ar) / cos(ar) == tan(ar))

print(asin(sin(ar)) == ar)

Además de las funciones circulares (enumeradas anteriormente), el módulo math también contiene un conjunto de sus **análogos hiperbólicos**:

* sinh(x) → el seno hiperbólico.
* cosh(x) → el coseno hiperbólico.
* tanh(x) → la tangente hiperbólico.
* asinh(x) → el arcoseno hiperbólico.
* acosh(x) → el arcocoseno hiperbólico.
* atanh(x) → el arcotangente hiperbólico.

**Funciones seleccionadas del módulo math: continuación**

Existe otro grupo de las funciones math relacionadas con la **exponenciación**:

* e → una constante con un valor que es una aproximación del número de Euler (e).
* exp(x) → encontrar el valor de ex.
* log(x) → el logaritmo natural de x.
* log(x, b) → el logaritmo de x con base b.
* log10(x) → el logaritmo decimal de x (más preciso que log(x, 10)).
* log2(x) → el logaritmo binario de x (más preciso que log(x, 2)).

Nota: la función pow():

* pow(x, y) → encontrar el valor de xy (toma en cuenta los dominios).

Esta es una función incorporada y no se tiene que importar.

Observa el código en el editor. ¿Puedes predecir su salida?

from math import e, exp, log

print(pow(e, 1) == exp(log(e)))

print(pow(2, 2) == exp(2 \* log(2)))

print(log(e, e) == exp(0))

SALIDA: False

True

True

**Funciones seleccionadas del módulo math: continuación**

El último grupo consta de algunas funciones de propósito general como:

* ceil(x) → devuelve el entero más pequeño mayor o igual que x.
* floor(x) → el entero más grande menor o igual que x.
* trunc(x) → el valor de x truncado a un entero (ten cuidado, no es equivalente a ceil o floor).
* factorial(x) → devuelve x! (x tiene que ser un valor entero y no negativo).
* hypot(x, y) → devuelve la longitud de la hipotenusa de un triángulo rectángulo con las longitudes de los catetos iguales a x e y (lo mismo que sqrt(pow(x, 2) + pow(y, 2)) pero más preciso).

Mira el código en el editor. Analiza el programa cuidadosamente.

Demuestra las diferencias fundamentales entre ceil(), floor() y trunc().

Ejecuta el programa y verifica su salida.

from math import ceil, floor, trunc

x = 1.4

y = 2.6

print(floor(x), floor(y))

print(floor(-x), floor(-y))

print(ceil(x), ceil(y))

print(ceil(-x), ceil(-y))

print(trunc(x), trunc(y))

SALIDA:

1 2

-2 -3

2 3

# ¿Existe aleatoriedad real en las computadoras?

Otro módulo que vale la pena mencionar es el que se llama random.

Ofrece algunos mecanismos que permiten operar con **números pseudoaleatorios**.

Toma en cuenta el prefijo **pseudo** - los números generados por los módulos pueden parecer aleatorios en el sentido de que no se pueden predecir, pero no hay que olvidar que todos se calculan utilizando algoritmos muy refinados.

Los algoritmos no son aleatorios, son deterministas y predecibles. Solo aquellos procesos físicos que se salgan completamente de nuestro control (como la intensidad de la radiación cósmica) pueden usarse como fuente de datos aleatorios reales. Los datos producidos por computadoras deterministas no pueden ser aleatorios de ninguna manera.

Un generador de números aleatorios toma un valor llamado **semilla**, lo trata como un valor de entrada, calcula un número "aleatorio" basado en él (el método depende de un algoritmo elegido) y produce una **nueva semilla**.

La duración de un ciclo en el que todos los valores semilla son únicos puede ser muy largo, pero no es infinito: tarde o temprano los valores iniciales comenzarán a repetirse y los valores generadores también se repetirán. Esto es normal. Es una característica, no un error.

El valor de la semilla inicial, establecido durante el inicio del programa, determina el orden en que aparecerán los valores generados.

El factor aleatorio del proceso puede ser **aumentado al establecer la semilla tomando un número de la hora actual** - esto puede garantizar que cada lanzamiento del programa comience desde un valor semilla diferente (por lo tanto, usará diferentes números aleatorios).

Afortunadamente, Python realiza dicha inicialización al importar el módulo.

**Funciones seleccionadas del módulo random**

La función general llamada random() (no debe confundirse con el nombre del módulo) **produce un número flotante**x**entre el rango**(0.0, 1.0) - en otras palabras: (0.0 <= x < 1.0).

El programa de ejemplo en el editor producirá cinco valores pseudoaleatorios, ya que sus valores están determinados por el valor semilla (un valor impredecible) actual, no se pueden adivinar. Ejecuta el programa.

from random import random

for i in range(5):

print(random())

SALIDA:

0.8246541110114589

0.0696713849412749

0.44423723876953003

La función seed() es capaz de **establecer la semilla del generador**. Te mostraremos dos de sus variantes:

* seed() - establece la semilla con la hora actual.
* seed(int\_value) - establece la semilla con el valor entero int\_value.

Hemos modificado el programa anterior; de hecho, hemos eliminado cualquier rastro de aleatoriedad del código:

from random import random, seed

seed(0)

for i in range(5):

print(random())

Debido al hecho de que la semilla siempre se establece con el mismo valor, la secuencia de valores generados siempre se ve igual.

Ejecuta el programa. Esto es lo que tenemos:

0.844421851525

0.75795440294

0.420571580831

0.258916750293

0.511274721369

¿Y tú?

Nota: sus valores pueden ser ligeramente diferentes si tu sistema utiliza aritmética de punto flotante más precisa o menos precisa, pero la diferencia se verá bastante lejos del punto decimal.

**Funciones seleccionadas del módulo random: continuación**

Si deseas valores aleatorios enteros, una de las siguientes funciones encajaría mejor:

* randrange(fin)x
* randrange(inico, fin)
* randrange(inicio, fin, incremento)
* randint(izquierda, derecha)

Las primeras tres invocaciones generarán un número entero tomado (pseudoaleatoriamente) del rango:

* range(fin)
* range(inicio, fin)
* range(inicio, fin, incremento)

Toma en cuenta la **exclusión implícita del lado derecho**.

La última función es equivalente a randrange(izquierda, derecha+1) - genera el valor entero i, el cual cae en el rango [izquierda, derecha] (sin exclusión en el lado derecho).

Observa el código en el editor. Este programa generará una línea que consta de tres ceros y un cero o un uno en el cuarto lugar.

from random import randrange, randint

print(randrange(1), end=' ')

print(randrange(0, 1), end=' ')

print(randrange(0, 1, 1), end=' ')

print(randint(0, 1))

SALIDA:

0 0 0 0

**Funciones seleccionadas del módulo random: continuación**

Las funciones anteriores tienen una desventaja importante: pueden producir valores repetidos incluso si el número de invocaciones posteriores no es mayor que el rango especificado.

Observa el código en el editor. Es muy probable que el programa genere un conjunto de números en el que algunos elementos no sean únicos.

Esto es lo que se obtiene al ejecutarlo:

9,4,5,4,5,8,9,4,8,4,

Como puedes ver, esta no es una buena herramienta para generar números para la lotería. Afortunadamente, existe una mejor solución que escribir tu propio código para verificar la singularidad de los números "sorteados".

Es una función con el nombre de - choice:

* choice(secuencia)
* sample(secuencia, elementos\_a\_elegir=1)

La primera variante elige un elemento "aleatorio" de la secuencia de entrada y lo devuelve.

El segundo crea una lista (una muestra) que consta del elemento elementos\_a\_elegir (que por defecto es 1) "sorteado" de la secuencia de entrada.

En otras palabras, la función elige algunos de los elementos de entrada, devolviendo una lista con la elección. Los elementos de la muestra se colocan en orden aleatorio. Nota que elementos\_a\_elegir no debe ser mayor que la longitud de la secuencia de entrada.

Observa el código a continuación:

from random import choice, sample

lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(choice(lst))

print(sample(lst, 5))

print(sample(lst, 10))

Nuevamente, la salida del programa no es predecible. Nuestros resultados se ven así:

4

[3, 1, 8, 9, 10]

[10, 8, 5, 1, 6, 4, 3, 9, 7, 2]

Código en el editor:

from random import randint

for i in range(10):

print(randint(1, 10), end=',')

SALIDA:

6,8,1,1,3,6,5,3,9,6,

**¿Cómo saber dónde estás?**

A veces, puede ser necesario encontrar información no relacionada con Python. Por ejemplo, es posible que necesites conocer la ubicación de tu programa dentro del entorno de la computadora.

Imagina el entorno de tu programa como una pirámide que consta de varias capas o plataformas.



Las capas son:

* El código (en ejecución) se encuentra en la parte superior.
* Python (mejor dicho, su entorno de ejecución) se encuentra directamente debajo de él.
* La siguiente capa de la pirámide se llena con el SO (sistema operativo): el entorno de Python proporciona algunas de sus funcionalidades utilizando los servicios del sistema operativo. Python, aunque es muy potente, no es omnipotente: se ve obligado a usar muchos ayudantes si va a procesar archivos o comunicarse con dispositivos físicos.
* La capa más inferior es el hardware: el procesador (o procesadores), las interfaces de red, los dispositivos de interfaz humana (ratones, teclados, etc.) y toda otra maquinaria necesaria para hacer funcionar la computadora: el sistema operativo sabe cómo emplearlos y utiliza muchos trucos para trabajar con todas las partes en un ritmo constante.

Esto significa que algunas de las acciones del programa tienen que recorrer un largo camino para ejecutarse con éxito, imagina que:

* **Tu código** quiere crear un archivo, por lo que invoca una de las funciones de Python.
* **Python**acepta la orden, la reorganiza para cumplir con los requisitos del sistema operativo local (es como poner el sello "aprobado" en una solicitud) y lo envía.
* El **SO** comprueba si la solicitud es razonable y válida (por ejemplo, si el nombre del archivo se ajusta a algunas reglas de sintaxis) e intenta crear el archivo. Tal operación, aparentemente es muy simple, no es atómica: consiste de muchos pasos menores tomados por...
* El **hardware**, el cual es responsable de activar los dispositivos de almacenamiento (disco duro, dispositivos de estado sólido, etc.) para satisfacer las necesidades del sistema operativo.

Por lo general, no eres consciente de todo ese alboroto: quieres que se cree el archivo y eso es todo.

Pero a veces quieres saber más, por ejemplo, el nombre del sistema operativo que aloja Python y algunas características que describen el hardware que aloja el sistema operativo.

Hay un módulo que proporciona algunos medios para permitir saber dónde se encuentra y qué componentes funcionan. El módulo se llama platform. Veamos algunas de las funciones que brinda.

**Funciones seleccionadas del módulo platform**

El módulo platform permite acceder a los datos de la plataforma subyacente, es decir, hardware, sistema operativo e información sobre la versión del intérprete.

Existe también una función que puede mostrar todas las capas subyacentes en un solo vistazo, llamada platform. Simplemente devuelve una cadena que describe el entorno; por lo tanto, su salida está más dirigida a los humanos que al procesamiento automatizado (lo veremos pronto).

Así es como se puede invocar: platform(aliased = False, terse = False).

Ahora:

* aliased → cuando se establece a True (o cualquier valor distinto de cero) puede hacer que la función presente los nombres de capa subyacentes alternativos en lugar de los comunes.
* terse → cuando se establece a True (o cualquier valor distinto de cero) puede convencer a la función de presentar una forma más breve del resultado (si lo fuera posible).

from platform import platform

print(platform())

print(platform(1))

print(platform(0, 1))

Ejecutamos el programa usando tres plataformas diferentes: esto es lo que se obtuvo:

**Intel x86 + Windows ® Vista (32 bit)**:

Windows-Vista-6.0.6002-SP2

Windows-Vista-6.0.6002-SP2

Windows-Vista

**Intel x86 + Gentoo Linux (64 bit)**:

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz-with-gentoo-2.3

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz-with-gentoo-2.3

Linux-3.18.62-g6-x86\_64-Intel-R-\_Core-TM-\_i3-2330M\_CPU\_@\_2.20GHz-with-glibc2.3.4

**Raspberry PI2 + Raspbian Linux (32 bit)**:

Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0

Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0

Linux-4.4.0-1-rpi2-armv7l-with-glibc2.9

También puedes ejecutar el programa en el IDLE de tu máquina local para verificar qué salida tendrá.

# Funciones seleccionadas del módulo platform: continuación

A veces, es posible que solo se desee conocer el nombre genérico del procesador que ejecuta el sistema operativo junto con Python y el código, una función llamada machine() te lo dirá. Como anteriormente, la función devuelve una cadena.

from platform import machine

print(machine())

Nuevamente, ejecutamos el programa en tres plataformas diferentes:

**Intel x86 + Windows ® Vista (32 bit)**:

x86

**Intel x86 + Gentoo Linux (64 bit)**:

x86\_64

**Raspberry PI2 + Raspbian Linux (32 bit)**:

armv7l

# Funciones seleccionadas del módulo platform: continuación

La función processor() devuelve una cadena con el nombre real del procesador (si lo fuese posible).

from platform import processor

print(processor())

Una vez más, ejecutamos el programa en tres plataformas diferentes:

**Intel x86 + Windows ® Vista (32 bit)**:

x86

**Intel x86 + Gentoo Linux (64 bit)**:

Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz

**Raspberry PI2 + Raspbian Linux (32 bit)**:

armv7l

# Funciones seleccionadas del módulo platform: continuación

Una función llamada system() devuelve el nombre genérico del sistema operativo en una cadena.

from platform import system

print(system())

Nuestras plataformas de ejemplo se presentaron así:

**Intel x86 + Windows ® Vista (32 bit)**:

Windows

**Intel x86 + Gentoo Linux (64 bit)**:

Linux

**Raspberry PI2 + Raspbian Linux (32 bit)**:

Linux

# Funciones seleccionadas del módulo platform: continuación

La versión del sistema operativo se proporciona como una cadena por la función version().

Ejecuta el código y verifica su salida.

from platform import version

print(version())

**Funciones seleccionadas del módulo platform: continuación**

Si necesitas saber qué versión de Python está ejecutando tu código, puedes verificarlo utilizando una serie de funciones dedicadas, aquí hay dos de ellas:

* python\_implementation() → devuelve una cadena que denota la implementación de Python (espera CPython aquí, a menos que decidas utilizar cualquier rama de Python no canónica).
* python\_version\_tuple() → devuelve una tupla de tres elementos la cual contiene:
  + la parte **mayor** de la versión de Python.
  + la parte **menor**,
  + el número de nivel del **patch**.

CÓDIGO:

from platform import python\_implementation, python\_version\_tuple

print(python\_implementation())

for atr in python\_version\_tuple():

print(atr)

# Índice del Módulo de Python

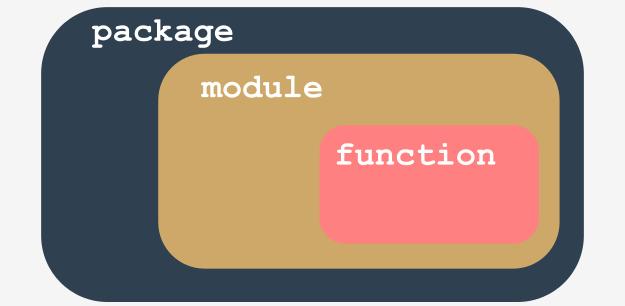
Puedes leer sobre todos los módulos estándar de Python aquí: <https://docs.python.org/3/py-modindex.html>.

Todo lo que se necesita hacer es encontrar los módulos que se desean y aprender a cómo usarlos. Es fácil.

**¿Qué es un paquete?**

Escribir tus propios módulos no difiere mucho de escribir scripts comunes.

Existen algunos aspectos específicos que se deben tomar en cuenta, pero definitivamente no es algo complicado. Lo verás pronto.



Resumamos algunos aspectos importantes:

* Un **módulo es un contenedor lleno de funciones** - puedes empaquetar tantas funciones como desees en un módulo y distribuirlo por todo el mundo.
* Por supuesto, no es una buena idea mezclar funciones con diferentes áreas de aplicación dentro de un módulo (al igual que en una biblioteca: nadie espera que los trabajos científicos se incluyan entre los cómics), así que se deben agrupar las funciones cuidadosamente y asignar un nombre claro e intuitivo al módulo que las contiene (por ejemplo, no le des el nombre videojuegos a un módulo que contiene funciones destinadas a particionar y formatear discos duros).
* Crear muchos módulos puede causar desorden: tarde que temprano querrás **agrupar tus módulos** de la misma manera que previamente has agrupado funciones: ¿Existe un contenedor más general que un módulo?.
* Sí lo hay, es un **paquete**: en el mundo de los módulos, un paquete juega un papel similar al de una carpeta o directorio en el mundo de los archivos.

# Tu primer módulo

En esta sección, trabajarás localmente en tu máquina. Comencemos desde cero, de la siguiente manera:

  
Se necesitan dos archivos para realizar estos experimentos. Uno de ellos será el módulo en sí. Está vacío ahora. No te preocupes, lo vas a llenar con el código real.

El archivo lleva por nombre module.py. No muy creativo, pero es simple y claro.

El segundo archivo contiene el código usando el nuevo módulo. Su nombre es main.py.

Su contenido es muy breve hasta ahora:



Nota: **ambos archivos deben estar ubicados en la misma carpeta**. Te recomendamos crear una carpeta nueva y vacía para ambos archivos. Esto hará que las cosas sean más fáciles.

Inicia el IDLE y ejecuta el archivo main.py.¿Que ves?

No deberías ver nada. Esto significa que Python ha importado con éxito el contenido del archivo module.py. No importa que el módulo esté vacío por ahora. El primer paso ya está hecho, pero antes de dar el siguiente paso, queremos que eches un vistazo a la carpeta en la que se encuentran ambos archivos.

¿Notas algo interesante?

Ha aparecido una nueva subcarpeta, ¿puedes verla? Su nombre es \_\_pycache\_\_. Echa un vistazo adentro. ¿Que ves?

Hay un archivo llamado (más o menos) module.cpython-xy.pyc donde x y y son dígitos derivados de tu versión de Python (por ejemplo, serán 3 y 4 si utilizas Python 3.4).

El nombre del archivo es el mismo que el de tu módulo. La parte posterior al primer punto dice qué implementación de Python ha creado el archivo (CPython) y su número de versión. La ultima parte (pyc) viene de las palabras *Python* y *compilado*.

Puedes mirar dentro del archivo: el contenido es completamente ilegible para los humanos. Tiene que ser así, ya que el archivo está destinado solo para uso de Python.

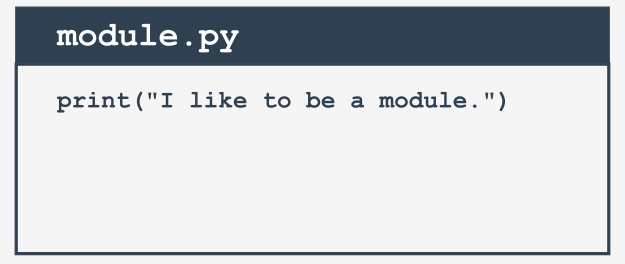
Cuando Python importa un módulo por primera vez, **traduce el contenido a una forma "semi" compilada**. El archivo no contiene código en lenguaje máquina: es **código semi-compilado** interno de Python, listo para ser ejecutado por el intérprete de Python. Como tal archivo no requiere tantas comprobaciones como las de un archivo fuente, la ejecución comienza más rápido y también se ejecuta más rápido.

Gracias a eso, cada importación posterior será más rápida que interpretar el código fuente desde cero.

Python puede verificar si el archivo fuente del módulo ha sido modificado (en este caso, el archivo pyc será reconstruido) o no (cuando el archivo pyc pueda ser ejecutado al instante). Este proceso es completamente automático y transparente, no se tiene que estar tomando en cuenta.

# Tu primer módulo: continuación

Ahora hemos puesto algo en el archivo del módulo:



¿Puedes notar alguna diferencia entre un módulo y un script ordinario? No hay ninguna hasta ahora.

Es posible ejecutar este archivo como cualquier otro script. Pruébalo por ti mismo.

¿Que es lo que pasa? Deberías de ver la siguiente línea dentro de tu consola:

Me gusta ser un módulo.

Volvamos al archivo main.py:



Ejecuta el archivo. ¿Que ves? Con suerte, verás algo como esto:

Me gusta ser un módulo.

¿Qué significa realmente?

Cuando un módulo es importado, su contenido es **ejecutado implícitamente por Python**. Le da al módulo la oportunidad de inicializar algunos de sus aspectos internos (por ejemplo, puede asignar a algunas variables valores útiles). Nota: la **inicialización se realiza solo una vez**, cuando se produce la primera importación, por lo que las asignaciones realizadas por el módulo no se repiten innecesariamente.

Imagina el siguiente contexto:

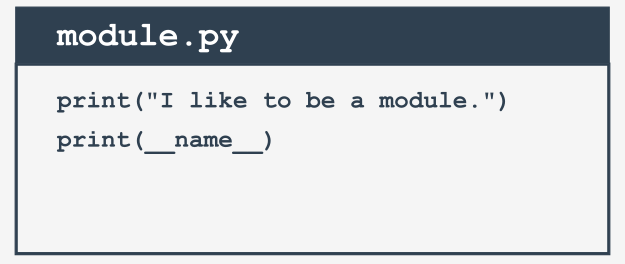
* Existe un módulo llamado mod1.
* Existe un módulo llamado mod2 el cual contiene la instrucción import mod1.
* Hay un archivo principal que contiene las instrucciones import mod1 y import mod2.

A primera vista, se puede pensar que mod1 será importado dos veces - afortunadamente, **solo se produce la primera importación**. Python recuerda los módulos importados y omite silenciosamente todas las importaciones posteriores.

Python puede hacer mucho más. También crea una variable llamada \_\_name\_\_.

Además, cada archivo fuente usa su propia versión separada de la variable, no se comparte entre módulos.

Te mostraremos cómo usarlo. Modifica el módulo un poco:



[Ver código en Sandbox](https://edube.org/sandbox/796b8690-fe0c-11e8-a471-02426ea0318a)

Ahora ejecuta el archivo module.py. Deberías ver las siguientes líneas:

Me gusta ser un módulo.

\_\_main\_\_

Ahora ejecuta el archivo main.py. ¿Y? ¿Ves lo mismo que nosotros?

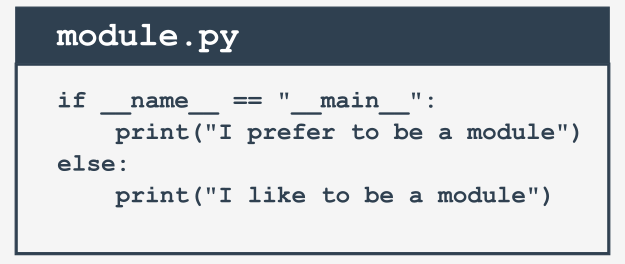
Me gusta ser un módulo.

module

Podemos decir que:

* Cuando se ejecuta un archivo directamente, su variable \_\_name\_\_ se establece a \_\_main\_\_.
* Cuando un archivo se importa como un módulo, su variable \_\_name\_\_ se establece al nombre del archivo (excluyendo a .py).

Así es como puedes hacer uso de la variable \_\_main\_\_ para detectar el contexto en el cual se activó tu código:



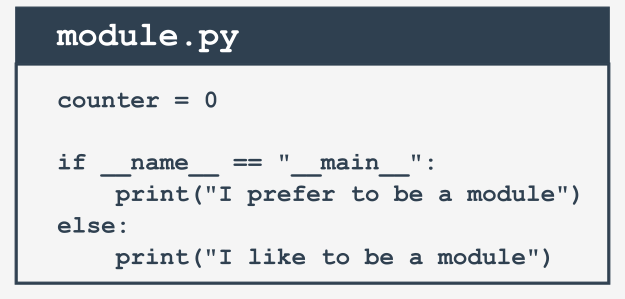
[Ver código en Sandbox](https://edube.org/sandbox/e26ff9aa-fe0c-11e8-a149-02426ea0318a)

Sin embargo, hay una forma más inteligente de utilizar la variable. Si escribes un módulo lleno de varias funciones complejas, puedes usarla para colocar una serie de pruebas para verificar si las funciones trabajan correctamente.

Cada vez que modifiques alguna de estas funciones, simplemente puedes ejecutar el módulo para asegurarte de que sus enmiendas no estropeen el código. Estas pruebas se omitirán cuando el código se importe como un módulo.

# Tu primer módulo: continuación

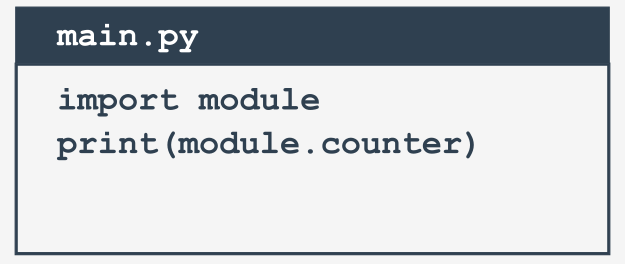
Este módulo contendrá dos funciones simples, y si deseas saber cuántas veces se han invocado las funciones, necesitas un contador inicializado en cero cuando se importa el módulo. Puedes hacerlo de esta manera:



[Ver código en Sandbox](https://edube.org/sandbox/10f128d8-fe0f-11e8-aed6-02426ea0318a)

El introducir tal variable es absolutamente correcto, pero puede causar importantes **efectos secundarios** que debes tener en cuenta.

Analiza el archivo modificado main.py:



[Ver código en Sandbox](https://edube.org/sandbox/afda627a-fe0f-11e8-8b95-02426ea0318a)

Como puedes ver, el archivo principal intenta acceder a la variable de contador del módulo. ¿Es esto legal? Sí lo es. ¿Es utilizable? Claro. ¿Es seguro? Eso depende: si confías en los usuarios de tu módulo, no hay problema; sin embargo, es posible que no desees que el resto del mundo vea tu **variable personal o privada**.

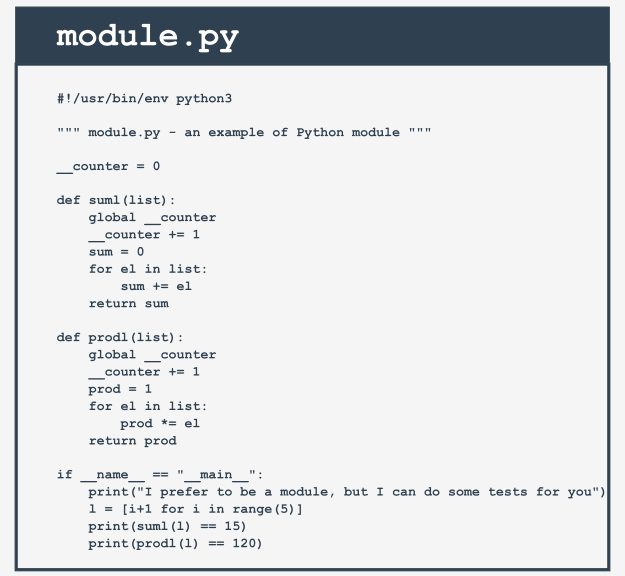
A diferencia de muchos otros lenguajes de programación, Python no tiene medios para permitirte ocultar tales variables a los ojos de los usuarios del módulo. Solo puedes informar a tus usuarios que esta es tu variable, que pueden leerla, pero que no deben modificarla bajo ninguna circunstancia.

Esto se hace anteponiendo al nombre de la variable \_ (un guión bajo) o \_\_ (dos guiones bajos), pero recuerda, es solo un **acuerdo**. Los usuarios de tu módulo pueden obedecerlo o no.

Nosotros por supuesto, lo respetaremos. Ahora pongamos dos funciones en el módulo: evaluarán la suma y el producto de los números recopilados en una lista.

Además, agreguemos algunos adornos allí y eliminemos los restos superfluos.

El módulo está listo:

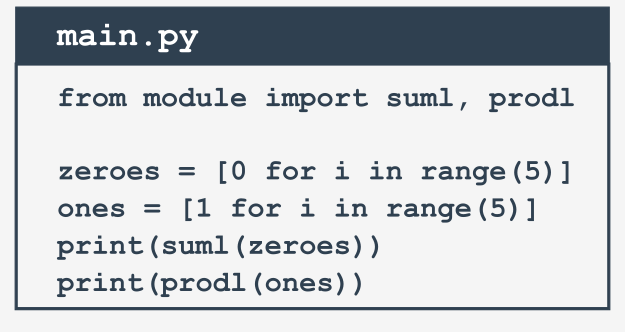


[Ver código en Sandbox](https://edube.org/sandbox/4e50c974-fe12-11e8-bb47-02426ea0318a)

Algunos elementos necesitan explicación:

* La línea que comienza con #! desde el punto de vista de Python, es solo un **comentario** debido a que comienza con #. Para sistemas operativos Unix y similares a Unix (incluido MacOS), dicha línea **indica al sistema operativo cómo ejecutar el contenido del archivo** (en otras palabras, qué programa debe lanzarse para interpretar el texto). En algunos entornos (especialmente aquellos conectados con servidores web) la ausencia de esa línea causará problemas.
* Una cadena (quizás una multilínea) colocada antes de las instrucciones de cualquier módulo (incluidas las importaciones) se denomina **doc-string**, y debe explicar brevemente el propósito y el contenido del módulo.
* Las funciones definidas dentro del módulo (suml() y prodl()) están disponibles para ser importadas.
* Se ha utilizado la variable \_\_name\_\_ para detectar cuándo se ejecuta el archivo de forma independiente.

Ahora es posible usar el nuevo módulo, esta es una forma de hacerlo:



[Ver código en Sandbox](https://edube.org/sandbox/3ebb7eb8-fe13-11e8-be36-02426ea0318a)

5.1.3.5 Módulos y Paqu