**Preguntas y respuestas**

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

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

Afortunadamente, las computadoras solo conocen dos tipos de respuestas:

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

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

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

**Comparación: operador de igualdad**

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

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

No olvides esta importante distinción:

* = es un **operador de asignación**, por ejemplo, a = b assigna a la varable a el valor de b.
* == es una pregunta *¿Son estos valores iguales?* así que a == b **compara** a y b.

Es un **operador binario con enlazado del lado izquierdo**. Necesita dos argumentos y **verifica si son iguales**.

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

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

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

var == 0

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

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

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

La pregunta será la siguiente:

black\_sheep == 2 \* white\_sheep

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

black\_sheep == (2 \* white\_sheep)

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

var = 0 # asignando 0 a var

print(var == 0)

var = 1 # asignando 1 a var

print(var == 0)

Ejecuta el código y comprueba si tenías razón.

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

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

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

var = 0 # asignando 0 a var

print(var != 0)

var = 1 # asignando 1 a var

print(var != 0)

Ejecuta el código y comprueba si tenías razón.

# Operadores de comparación: 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:

black\_sheep > white\_sheep # 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 del lado izquierdo**, y su **prioridad es mayor que la mostrada por**==**y**!=.

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

centigrade\_outside ≥ 0.0 # mayor o igual que

## Operadores de comparación: menor o igual que

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

Observa este ejemplo simple:

current\_velocity\_mph < 85 # Menor que

current\_velocity\_mph ≤ 85 # Menor o igual que

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

## Haciendo uso de las respuestas

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

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

answer = number\_of\_lions >= number\_of\_lionesses

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

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

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

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

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

**LABORATORIO**

**Tiempo Estimado**

5 minutos

**Nivel de Dificultad**

Muy Fácil

**Objetivos**

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

**Escenario**

Usando uno de los operadores de comparación en Python, escribe un programa simple de dos líneas que tome el parámetro n como entrada, que es un entero, e imprime False si n es menor que 100, y True si n es mayor o igual que 100.

No debes crear ningún bloque if (hablaremos de ellos muy pronto). Prueba tu código usando los datos que te proporcionamos.

**Datos de Prueba**

Ejemplo de entrada: 55

Resultado esperado: False

Ejemplo de entrada: 99

Resultado esperado: False

Ejemplo de entrada: 100

Resultado esperado: True

Ejemplo de entrada: 101

Resultado esperado: True

Ejemplo de entrada: -5

Resultado esperado: False

Ejemplo de entrada: +123

Resultado esperado: True

* [**Sandbox**](https://edube.org/sandbox)

Code



1

* **Console**

**Condiciones y ejecución condicional**

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

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

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

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

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

if true\_or\_not:

do\_this\_if\_true

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

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

¿Cómo funciona esta sentencia?

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

En la vida real, a menudo expresamos un deseo:

*si el clima es bueno, saldremos a caminar*

*después, almorzaremos*

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

Sabiendo que condiciones influyen en nuestro comportamiento y asumiendo que tenemos las funciones sin parámetros go\_for\_a\_walk() y have\_lunch(), podemos escribir el siguiente fragmento de código:

if the\_weather\_is\_good:

go\_for\_a\_walk()

have\_lunch()

**Ejecución condicional: la sentencia if**

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

if sheep\_counter >= 120: # #evalúa una expresión condicional

sleep\_and\_dream() #se ejecuta si la expresión condicional es True

Puedes leerlo como sigue: si sheep\_counter es mayor o igual que 120, entonces duerme y sueña (es decir, ejecuta la función sleep\_and\_dream).

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

Analiza el siguiente código:

if sheep\_counter >= 120:

make\_a\_bed()

take\_a\_shower()

sleep\_and\_dream()

feed\_the\_sheepdogs()

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

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

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

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

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

Nota: no hay una palabra sobre lo que sucederá si el clima es malo. Solo sabemos que no saldremos al aire libre, pero no sabemos que podríamos hacer. Es posible que también queramos planificar algo en caso de mal tiempo.

Podemos decir, por ejemplo: *Si el clima es bueno, saldremos a caminar, de lo contrario, iremos al cine*.

Ahora sabemos lo que haremos **si se cumplen las condiciones**, y sabemos lo que haremos **si no todo sale como queremos**. En otras palabras, tenemos un "Plan B".

Python nos permite expresar dichos planes alternativos. Esto se hace con una segunda forma, ligeramente mas compleja, de la sentencia condicional, la sentencia *if-else*:

if true\_or\_false\_condition:

perform\_if\_condition\_true

else:

perform\_if\_condition\_false

Por lo tanto, hay una nueva palabra: else - esta es una **palabra clave reservada**.

La parte del código que comienza con else dice que hacer si no se cumple la condición especificada por el if (observa los **dos puntos** después de la palabra).

La ejecución de *if-else* es la siguiente:

* Si la condición se evalúa como **True** (su valor no es igual a cero), la instrucción perform\_if\_condition\_true se ejecuta, y la sentencia condicional llega a su fin.
* Si la condición se evalúa como **False** (es igual a cero), la instrucción perform\_if\_condition\_false se ejecuta, y la sentencia condicional llega a su fin.

[Prev](https://edube.org/learn/python-essentials-1-esp/tomando-decisiones-en-python-11) [Next](https://edube.org/learn/python-essentials-1-esp/tomando-decisiones-en-python-13)

**La sentencia if-else: más sobre ejecución condicional**

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

if the\_weather\_is\_good:

go\_for\_a\_walk()

else:

go\_to\_a\_theater()

have\_lunch()

Si el clima es bueno, saldremos a caminar. De lo contrario, iremos al cine. No importa si el clima es bueno o malo, almorzaremos después (después de la caminata o después de ir al cine).

Todo lo que hemos dicho sobre la sangría funciona de la misma manera dentro de **la rama *else*** :

if the\_weather\_is\_good:

go\_for\_a\_walk()

have\_fun()

else:

go\_to\_a\_theater()

enjoy\_the\_movie()

have\_lunch()

**Sentencias if-else anidadas**

Ahora, analicemos dos casos especiales de la sentencia condicional.

Primero, considera el caso donde la instrucción **colocada después del**if **es otro**if.

Lee lo que hemos planeado para este Domingo. Si hay buen clima, saldremos a caminar. Si encontramos un buen restaurante, almorzaremos allí. De lo contrario, vamos a comer un sandwich. Si hay mal clima, iremos al cine. Si no hay boletos, iremos de compras al centro comercial más cercano.

Escribamos lo mismo en Python. Considera cuidadosamente el código siguiente:

if the\_weather\_is\_good:

if nice\_restaurant\_is\_found:

have\_lunch()

else:

eat\_a\_sandwich()

else:

if tickets\_are\_available:

go\_to\_the\_theater()

else:

go\_shopping()

Aquí hay dos puntos importantes:

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

**La sentencia elif**

El segundo caso especial presenta otra nueva palabra clave de Python: **elif**. Como probablemente sospechas, es una forma más corta de **else-if**.

elif se usa para **verificar más de una condición**, y para **detener** cuando se encuentra la primera sentencia verdadera.

Nuestro siguiente ejemplo se parece a la anidación, pero las similitudes son muy leves. Nuevamente, cambiaremos nuestros planes y los expresaremos de la siguiente manera: si hay buen clima, saldremos a caminar, de lo contrario, si obtenemos entradas, iremos al cine, de lo contrario, si hay mesas libres en el restaurante, vamos a almorzar; si todo falla, regresaremos a casa y jugaremos ajedrez.

¿Has notado cuantas veces hemos usado la palabra *de lo contrario*? Esta es la etapa en la que la palabra clave reservada elif desempeña su función.

Escribamos el mismo escenario empleando Python:

if the\_weather\_is\_good:

go\_for\_a\_walk()

elif tickets\_are\_available:

go\_to\_the\_theater()

elif table\_is\_available:

go\_for\_lunch()

else:

play\_chess\_at\_home()

La forma de ensamblar las siguientes sentencias *if-elif-else* a veces se denomina **cascada**.

Observa de nuevo como la sangría mejora la legibilidad del código.

Se debe prestar atención adicional a este caso:

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

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

[Prev](https://edube.org/learn/python-essentials-1-esp/tomando-decisiones-en-python-12) [Next](https://edube.org/learn/python-essentials-1-esp/tomando-decisiones-en-python-14)

**Analizando ejemplos de código**

Ahora te mostraremos algunos programas simples pero completos. No los explicaremos a detalle, porque consideramos que los comentarios (y los nombres de las variables) dentro del código son guías suficientes.

Todos los programas resuelven el mismo problema: **encuentran el número mayor de una serie de números y lo imprimen**.

**Ejemplo 1:**

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

#Se leen dos números

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

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

# Elige el número más grande

if number1 > number2:

larger\_number = number1

else:

larger\_number = number2

# Imprime el resultado

print("El número más grande es:", larger\_number)

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

**Ejemplo 2:**

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

#Se leen dos números

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

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

# Elige el número más grande

if number1 > number2: larger\_number = number1

else: larger\_number = number2

# Imprime el resultado

print("El número más grande es:", larger\_number)

Nota: si alguna de las ramas de *if-elif-else* contiene una sola instrucción, puedes codificarla de forma más completa (no es necesario que aparezca una línea con sangría después de la palabra clave), pero solo continúa la línea después de los dos puntos).

Sin embargo, este estilo puede ser engañoso, y no lo vamos a usar en nuestros programas futuros, pero definitivamente vale la pena saber si quieres leer y entender los programas de otra persona.

No hay otras diferencias en el código.

**Ejemplo 3:**

Es hora de complicar el código: encontremos el mayor de los tres números. ¿Se ampliará el código? Un poco.

Suponemos que el primer valor es el más grande. Luego verificamos esta hipótesis con los dos valores restantes.

Observa el siguiente código:

# Se leen tres números

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

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

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

# Asumimos temporalmente que el primer número

# es el más grande.

# Lo verificaremos pronto.

largest\_number = number1

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

# y actualiza el número más grande si es necesario.

if number2 > largest\_number:

largest\_number = number2

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

# y actualiza el número más grande si es necesario.

if number3 > largest\_number:

largest\_number = number3

# Imprime el resultado.

print("El número más grande es:", largest\_number)

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

* [**Sandbox**](https://edube.org/sandbox)

Code



1

* **Console**

# Pseudocódigo e introducción a los bucles (ciclos)

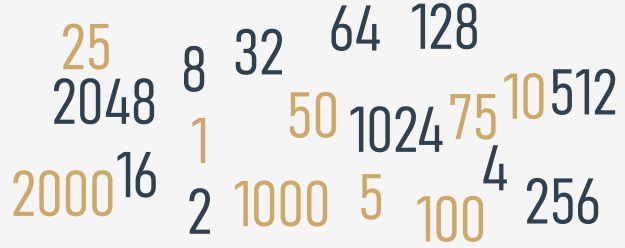
Ahora deberías poder escribir un programa que encuentre el mayor de cuatro, cinco, seis o incluso diez números.

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

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

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

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



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

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

Veamos nuestro pseudocódigo a continuación:

largest\_number = -999999999

number = int(input())

if number == -1:

print(largest\_number)

exit()

if number > largest\_number:

largest\_number = number

# Ir a la línea 02

¿Qué está pasando en él?

En primer lugar, podemos simplificar el programa si, al principio del código, le asignamos a la variable largest\_number un valor que será más pequeño que cualquiera de los números ingresados. Usaremos -999999999 para ese propósito.

En segundo lugar, asumimos que nuestro algoritmo no sabrá por adelantado cuántos números se entregarán al programa. Esperamos que el usuario ingrese todos los números que desee; el algoritmo funcionará bien con cien y con mil números. ¿Cómo hacemos eso?

Hacemos un trato con el usuario: cuando se ingresa el valor-1, será una señal de que no hay más datos y que el programa debe finalizar su trabajo.

De lo contrario, si el valor ingresado no es igual a -1, el programa leerá otro número, y así sucesivamente.

El truco se basa en la suposición de que cualquier parte del código se puede realizar más de una vez, precisamente, tantas veces como sea necesario.

La ejecución de una determinada parte del código más de una vez se denomina **bucle**. El significado de este término es probablemente obvio para ti.

Las líneas 02 a 08 forman un bucle. Los **pasaremos tantas veces como sea necesario**para revisar todos los valores ingresados.

¿Puedes usar una estructura similar en un programa escrito en Python? Si, si puedes.

**Información Adicional**

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

# Se leen tres números.

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

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

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

# Verifica cuál de los números es el mayor

# y pásalo a la variable largest\_number

largest\_number = max(number1, number2, number3)

# Imprime el resultado.

print("El número más grande es:", largest\_number)

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

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

**LABORATORIO**

**Tiempo Estimado**

5-10 minutos

**Nivel de Dificultad**

Fácil

**Objetivos**

* Familiarizarse con la función input().
* Familiarizarse con los operadores de comparación en Python.
* Familiarizarse con el concepto de ejecución condicional.

**Escenario**

[Espatifilo](https://upload.wikimedia.org/wikipedia/commons/b/bd/Spathiphyllum_cochlearispathum_RTBG.jpg), más comúnmente conocida como la planta de cuna de Moisés o flor de la paz, es una de las plantas para interiores más populares que filtra las toxinas dañinas del aire. Algunas de las toxinas que neutraliza incluyen benceno, formaldehído y amoníaco.

Imagina que tu programa de computadora ama estas plantas. Cada vez que recibe una entrada en forma de la palabra Espatifilo, grita involuntariamente a la consola la siguiente cadena: "¡Espatifilo es la mejor planta de todas!"

Escribe un programa que utilice el concepto de ejecución condicional, tome una cadena como entrada y que:

* Imprima el enunciado "Si, ¡El ESPATIFILIO! es la mejor planta de todos los tiempos!" en la pantalla si la cadena ingresada es "ESPATIFILIO".
* Imprima "No, ¡quiero un gran ESPATIFILIO!" si la cadena ingresada es "espatifilo".
* Imprima "¡ESPATIFILIO!, ¡No [entrada]!" de lo contrario. Nota: [entrada] es la cadena que se toma como entrada.

Prueba tu código con los datos que te proporcionamos. ¡Y hazte de un ESPATIFILIO también!

**Datos de Prueba**

Entrada de muestra: espatifilo

Resultado esperado: No, ¡quiero un gran ESPATIFILIO!

Entrada de ejemplo: pelargonio

Resultado esperado: !ESPATIFILIO!, ¡No pelargonio!

Entrada de muestra: ESPATIFILIO

Resultado esperado: Si, ¡El ESPATIFILIO es la mejor planta de todos los tiempos!

* [**Sandbox**](https://edube.org/sandbox)

Code



1

* **Console**

**LABORATORIO**

**Tiempo Estimado**

10-15 minutos

**Nivel de Dificultad**

Fácil/Medio

**Objetivos**

Familiarizar al estudiante con:

* Utilizar la sentencia *if-else* para ramificar la ruta de control.
* Construir un programa completo que resuelva problemas simples de la vida real.

**Escenario**

Érase una vez una tierra de leche y miel, habitada por gente feliz y próspera. La gente pagaba impuestos, por supuesto, su felicidad tenía límites. El impuesto más importante, denominado *Impuesto Personal de Ingresos*(*IPI*, para abreviar) tenía que pagarse una vez al año y se evaluó utilizando la siguiente regla:

* Si el ingreso del ciudadano no era superior a 85,528 pesos, el impuesto era igual al 18% del ingreso menos 556 pesos y 2 centavos (esta fue la llamada *exención fiscal*).
* Si el ingreso era superior a esta cantidad, el impuesto era igual a 14,839 pesos y 2 centavos, más el 32% del excedente sobre 85,528 pesos.

Tu tarea es escribir una **calculadora de impuestos**.

* Debe aceptar un valor de punto flotante: el ingreso.
* A continuación, debe imprimir el impuesto calculado, redondeado a pesos totales. Hay una función llamada round() que hará el redondeo por ti, la encontrarás en el código de esqueleto del editor.

Nota: Este país feliz nunca devuelve dinero a sus ciudadanos. Si el impuesto calculado es menor que cero, solo significa que no hay impuesto (el impuesto es igual a cero). Ten esto en cuenta durante tus cálculos.

Observa el código en el editor: solo lee un valor de entrada y genera un resultado, por lo que debes completarlo con algunos cálculos inteligentes.

Prueba tu código con los datos que hemos proporcionado.

**Datos de Prueba**

Entrada de muestra: 10000

Resultado esperado: El impuesto es: 1244.0 pesos

Entrada de muestra: 100000

Resultado esperado: El impuesto es: 19470.0 pesos

Entrada de muestra: 1000

Resultado esperado: El impuesto es: 0.0 pesos

Entrada de muestra: -100

Resultado esperado: El impuesto es: 0.0 pesos

* [**Sandbox**](https://edube.org/sandbox)

Code

income = float(input("Introduce el ingreso anual:"))  
  
#  
# Escribe tu código aquí.  
#  
  
tax = round(tax, 0)  
print("El impuesto es:", tax, "pesos")



1

2

3

4

5

6

7

8

9

income = float(input("Introduce el ingreso anual:"))

#

# Escribe tu código aquí.

#

tax = round(tax, 0)

print("El impuesto es:", tax, "pesos")

* **Console**

**LABORATORIO**

**Tiempo Estimado**

10-15 minutos

**Nivel de Dificultad**

Fácil/Medio

**Objetivos**

Familiarizar al estudiante con:

* Utilizar la sentencia if-elif-else.
* Encontrar la implementación adecuada de las reglas definidas verbalmente.
* Emplear el código de prueba empleando entradas y salidas de muestra.

**Escenario**

Como seguramente sabrás, debido a algunas razones astronómicas, el año pueden ser *bisiesto* o *común*. Los primeros tienen una duración de 366 días, mientras que los últimos tienen una duración de 365 días.

Desde la introducción del calendario Gregoriano (en 1582), se utiliza la siguiente regla para determinar el tipo de año:

* Si el número del año no es divisible entre cuatro, es un *año común*.
* De lo contrario, si el número del año no es divisible entre 100, es un *año bisiesto*.
* De lo contrario, si el número del año no es divisible entre 400, es un *año común*.
* De lo contrario, es un *año bisiesto*.

Observa el código en el editor: solo lee un número de año y debe completarse con las instrucciones que implementan la prueba que acabamos de describir.

El código debe mostrar uno de los dos mensajes posibles, que son Año Bisiesto o Año Común, según el valor ingresado.

Sería bueno verificar si el año ingresado cae en la era Gregoriana y emitir una advertencia de lo contrario: No dentro del período del calendario Gregoriano. Consejo: utiliza los operadores != y %.

Prueba tu código con los datos que hemos proporcionado.

**Datos de Prueba**

Entrada de muestra: 2000

Resultado esperado: Año Bisiesto

Entrada de muestra: 2015

Resultado esperado: Año Común

Entrada de muestra: 1999

Resultado esperado: Año Común

Entrada de muestra: 1996

Resultado esperado: Año Bisiesto

Entrada de muestra: 1580

Resultado esperado: No esta dentro del período del calendario Gregoriano

year = int(input("Introduce un año:"))

#

# Escribe tu código aquí.

#

**Puntos Clave**

1. Los operadores de **comparación** (o también denominados operadores *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 |
| > | Devuelve True 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 deseas ejecutar algún código solo si se cumple una determinada condición, puedes usar una **sentencia condicional**:

* Una única sentencia if, por ejemplo:

x = 10

if x == 10: # condición

print("x es igual a 10") # Ejecutado si la condición es Verdadera.

* Una serie de sentencias if, por ejemplo:

x = 10

if x > 5: # primera condición

print("x es mayor que 5") # Ejecutado si la primera condición es Verdadera.

if x < 10: # segunda condición

print("x is less than 10") # Ejecutado si la segunda condición es Verdadera.

if x == 10: # tercera condición

print("x is equal to 10") # Ejecutado si la tercera condición es Verdadera.

Cada sentencia if se prueba por separado.

* Una sentencia 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 Falsa.

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

x = 10

if x > 5: # True

print("x > 5")

if x > 8: # True

print("x > 8")

if x > 10: # False

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 sentencia if-elif-else, por ejemplo:

x = 10

if x == 10: # True

print("x == 10")

if x > 15: # False

print("x > 15")

elif x > 10: # False

print("x > 10")

elif x > 5: # True

print("x > 5")

else:

print("else no será ejecutado")

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.

* Sentencias condicionales anidadas, ejemplo:

x = 10

if x > 5: # True

if x == 6: # False

print("anidado: x == 6")

elif x == 10: # True

print("anidado: x == 10")

else:

print("anidado: else")

else:

print("else")

# Puntos Clave: continuación

**Ejercicio 1**

¿Cuál es la salida del siguiente fragmento de código?

x = 5

y = 10

z = 8

print(x > y)

print(y > z)

Revisar

**Ejercicio 2**

¿Cuál es la salida del siguiente fragmento de código?

x, y, z = 5, 10, 8

print(x > z)

print((y - 5) == x)

Revisar

**Ejercicio 3**

¿Cuál es la salida del siguiente fragmento de código?

x, y, z = 5, 10, 8

x, y, z = z, y, x

print(x > z)

print((y - 5) == x)

Revisar

**Ejercicio 4**

¿Cuál es la salida del siguiente fragmento de código?

x = 10

if x == 10:

print(x == 10)

if x > 5:

print(x > 5)

if x < 10:

print(x < 10)

else:

print("else")

Revisar

**Ejercicio 5**

¿Cuál es la salida del siguiente fragmento de código?

x = "1"

if x == 1:

print("uno")

elif x == "1":

if int(x) > 1:

print("dos")

elif int(x) < 1:

print("tres")

else:

print("cuatro")

if int(x) == 1:

print("cinco")

else:

print("seis")

Revisar

**Ejercicio 6**

¿Cuál es la salida del siguiente fragmento de código?

x = 1

y = 1.0

z = "1"

if x == y:

print("uno")

if y == int(z):

print("dos")

elif x == y:

print("tres")

else:

print("cuatro")

**Bucles (ciclos) en el código con while**

¿Estás de acuerdo con la sentencia presentada a continuación?

mientras haya algo que hacer

hazlo

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

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

while conditional\_expression:

instruction

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

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

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

Observa el algoritmo a continuación:

while conditional\_expression:

instruction\_one

instruction\_two

instruction\_three

:

:

instruction\_n

Ahora, es importante recordar que:

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

**Un bucle infinito**

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

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

while True:

print("Estoy atrapado dentro de un bucle.")

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

**NOTA**

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

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

Analiza el programa cuidadosamente. Localiza donde comienza el bucle (línea 8) y descubre **como se sale del cuerpo del bucle**:

# Almacena el actual número más grande aquí.

largest\_number = -999999999

# Ingresa el primer valor.

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

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

while number != -1:

# ¿Es el número más grande que el valor de largest\_number?

if number > largest\_number:

# Sí si, se actualiza largest\_number.

largest\_number = number

# Ingresa el siguiente número.

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

# Imprime el número más grande

print("El número más grande es:", largest\_number)

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

**El bucle while: más ejemplos**

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

# Un programa que lee una secuencia de números

# y cuenta cuántos números son pares y cuántos son impares.

# El programa termina cuando se ingresa un cero.

odd\_numbers = 0

even\_numbers = 0

# Lee el primer número.

number = int(input("Introduce un número o escribe 0 para detener: "))

# 0 termina la ejecución.

while number != 0:

# Verificar si el número es impar.

if number % 2 == 1:

# Incrementar el contador de números impares odd\_numbers.

odd\_numbers += 1

else:

# Incrementar el contador de números pares even\_numbers.

even\_numbers += 1

# Leer el siguiente número.

number = int(input("Introduce un número o escribe 0 para detener: "))

# Imprimir resultados.

print("Cuenta de números impares:", odd\_numbers)

print("Cuenta de números pares:", even\_numbers)

Ciertas expresiones se pueden simplificar sin cambiar el comportamiento del programa.

Intenta recordar cómo Python interpreta la verdad de una condición y ten en cuenta que estas dos formas son equivalentes:

while number != 0: y while number:

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

if number % 2 == 1: y if number % 2:

**Empleando una variable counter para salir del bucle**

Observa el fragmento de código a continuación:

counter = 5

while counter != 0:

print("Dentro del bucle.", counter)

counter -= 1

print("Fuera del bucle.", counter)

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

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

¿Puedes ver la diferencia?

counter = 5

while counter:

print("Dentro del bucle.", counter)

counter -= 1

print("Fuera del bucle.", counter)

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

**RECUERDA**

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

* [**Sandbox**](https://edube.org/sandbox)

Code



1

* **Console**

**LABORATORIO**

**Tiempo Estimado**

15 minutos

**Nivel de Dificultad**

Fácil

**Objetivos**

Familiarizar al estudiante con:

* Utilizar el bucle while.
* Reflejar situaciones de la vida real en código de computadora.

**Escenario**

Un mago junior ha elegido un número secreto. Lo ha escondido en una variable llamada secret\_number. Quiere que todos los que ejecutan su programa jueguen el juego *Adivina el número secreto*, y adivina qué número ha elegido para ellos. ¡Quiénes no adivinen el número quedarán atrapados en un bucle sin fin para siempre! Desafortunadamente, él no sabe cómo completar el código.

Tu tarea es ayudar al mago a completar el código en el editor de tal manera que el código:

* Pedirá al usuario que ingrese un número entero.
* Utilizará un bucle while.
* Comprobará si el número ingresado por el usuario es el mismo que el número escogido por el mago. Si el número elegido por el usuario es diferente al número secreto del mago, el usuario debería ver el mensaje "¡Ja, ja! ¡Estás atrapado en mi bucle!"  y se le solicitará que ingrese un número nuevamente. Si el número ingresado por el usuario coincide con el número escogido por el mago, el número debe imprimirse en la pantalla, y el mago debe decir las siguientes palabras: "¡Bien hecho, muggle! Eres libre ahora".

¡El mago está contando contigo! No lo decepciones.

**INFO EXTRA**

Por cierto, observa la función print(). La forma en que lo hemos utilizado aquí se llama *impresión multilínea*. Puede utilizar **comillas triples**para imprimir cadenas en varias líneas para facilitar la lectura del texto o crear un diseño especial basado en texto. Experimenta con ello.

secret\_number = 777

print(

"""

+==================================+

| ¡Bienvenido a mi juego, muggle! |

| Introduce un número entero |

| y adivina qué número he |

| elegido para ti. |

| Entonces, |

| ¿Cuál es el número secreto? |

+==================================+

""")

**Bucles en tu código con for**

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

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

i = 0

while i < 100:

# do\_something()

i += 1

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

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

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

Echa un vistazo al fragmento:

for i in range(100):

# do\_something()

pass

Existen algunos elementos nuevos. Déjanos contarte sobre ellos:

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

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

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

for i in range(10):

print("El valor de i es actualmente", i)

Ejecuta el código para verificar si tenías razón.

Nota:

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

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

for i in range(2, 8):

print("El valor de i es actualmente", i)

En este caso, el primer argumento determina el valor inicial (primero) de la variable de control.

El último argumento muestra el primer valor que no se asignará a la variable de control.

Nota: la función range() **solo acepta enteros como argumentos** y genera secuencias de enteros.

¿Puedes adivinar la salida del programa? Ejecútalo para comprobar si ahora también estabas en lo cierto.

El primer valor mostrado es 2 (tomado del primer argumento de range()).

El último es 7 (aunque el segundo argumento de range() es 8).

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

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

El tercer argumento es un **incremento**: es un valor agregado para controlar la variable en cada giro del bucle (como puedes sospechar, el valor predeterminado del incremento **es 1**).

¿Puedes decirnos cuántas líneas aparecerán en la consola y qué valores contendrán?

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

**salida**

¿Sabes por qué? El primer argumento pasado a la función range() nos dice cual es el número **de inicio** de la secuencia (por lo tanto, 2 en la salida). El segundo argumento le dice a la función dónde **detener** la secuencia (la función genera números hasta el número indicado por el segundo argumento, pero no lo incluye). Finalmente, el tercer argumento indica el **paso**, que en realidad significa la diferencia entre cada número en la secuencia de números generados por la función.

2 (número inicial) → 5 (2 incremento por 3 es igual a 5 - el número está dentro del rango de 2 a 8) → 8 (5 incremento por 3 es igual a 8 - el número no está dentro del rango de 2 a 8, porque el parámetro de parada no está incluido en la secuencia de números generados por la función).

Nota: si el conjunto generado por la función range() está vacío, el bucle no ejecutará su cuerpo en absoluto.

Al igual que aquí, no habrá salida:

for i in range(1, 1):

print("El valor de i es actualmente", i)

Nota: el conjunto generado por range() debe ordenarse en **un orden ascendente**. No hay forma de forzar el range() para crear un conjunto en una forma diferente. Esto significa que el segundo argumento de range() debe ser mayor que el primero.

Por lo tanto, tampoco habrá salida aquí:

for i in range(2, 1):

print("El valor de i es actualmente", i)

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

power = 1

for expo in range(16):

print("2 a la potencia de", expo, "es", power)

power \*= 2

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

for i in range(2, 8, 3):

print("El valor de i es actualmente", i)

**Tiempo Estimado**

5 minutos

**Nivel de dificultad**

Muy fácil

**Objetivos**

Familiarizar al estudiante con:

* Utilizar el bucle for.
* Reflejar situaciones de la vida real en código de computadora.

**Escenario**

¿Sabes lo que es Mississippi? Bueno, es el nombre de uno de los estados y ríos en los Estados Unidos. El río Mississippi tiene aproximadamente 2,340 millas de largo, lo que lo convierte en el segundo río más largo de los Estados Unidos (el más largo es el río Missouri). ¡Es tan largo que una sola gota de agua necesita 90 días para recorrer toda su longitud!

La palabra *Mississippi* también se usa para un propósito ligeramente diferente: para *contar mississippily (mississippimente)*.

Si no estás familiarizado con la frase, estamos aquí para explicarte lo que significa: se utiliza para contar segundos.

La idea detrás de esto es que agregar la palabra *Mississippi* a un número al contar los segundos en voz alta hace que suene más cercano al reloj, y por lo tanto "un Mississippi, dos Mississippi, tres Mississippi" tomará aproximadamente unos tres segundos reales de tiempo. A menudo lo usan los niños que juegan al escondite para asegurarse de que el buscador haga un conteo honesto.

Tu tarea es muy simple aquí: escribe un programa que use un bucle for para "contar de forma mississippi" hasta cinco. Habiendo contado hasta cinco, el programa debería imprimir en la pantalla el mensaje final "¡Listos o no, ahí voy!"

Utiliza el esqueleto que hemos proporcionado en el editor.

**INFO EXTRA**

Ten en cuenta que el código en el editor contiene dos elementos que pueden no ser del todo claros en este momento: la sentencia import time y el método sleep(). Vamos a hablar de ellos pronto.

Por el momento, nos gustaría que supieras que hemos importado el módulo time y hemos utilizado el método sleep() para suspender la ejecución de cada función posterior de print() dentro del bucle for durante un segundo, de modo que el mensaje enviado a la consola se parezca a un conteo real. No te preocupes, pronto aprenderás más sobre módulos y métodos.

**Salida esperada**

1 Mississippi

2 Mississippi

3 Mississippi

4 Mississippi

5 Mississippi

import time

# Escribe un bucle for que cuente hasta cinco.

# Cuerpo del bucle: imprime el número de iteración del bucle y la palabra "Mississippi".

# Cuerpo del bucle - usar: time.sleep (1)

# Escribe una función de impresión con el mensaje final.

**Las sentencias break y continue**

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

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

Python proporciona dos instrucciones especiales para la implementación de estas dos tareas. Digamos por razones de precisión que su existencia en el lenguaje no es necesaria: un programador experimentado puede codificar cualquier algoritmo sin estas instrucciones. Tales adiciones, que no mejoran el poder expresivo del lenguaje, sino que solo simplifican el trabajo del desarrollador, a veces se denominan **dulces sintácticos**o azúcar sintáctica.

Estas dos instrucciones son:

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

Ambas palabras son **palabras clave reservadas**.

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

# break - ejemplo

print("La instrucción break:")

for i in range(1, 6):

if i == 3:

break

print("Dentro del bucle.", i)

print("Fuera del bucle.")

# continue - ejemplo

print("\nLa instrucción continue:")

for i in range(1, 6):

if i == 3:

continue

print("Dentro del bucle.", i)

print("Fuera del bucle.")

# Las sentencias break y continue: más ejemplos

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

Analiza el código y determina como las usarías.

La variante empleando break es la siguiente:

largest\_number = -99999999

counter = 0

while True:

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

if number == -1:

break

counter += 1

if number > largest\_number:

largest\_number = number

if counter != 0:

print("El número más grande es", largest\_number)

else:

print("No has ingresado ningún número.")

Ejecútalo, pruébalo y experimenta con él.

Y ahora la variante con continue:

largest\_number = -99999999

counter = 0

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

while number != -1:

if number == -1:

continue

counter += 1

if number > largest\_number:

largest\_number = number

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

if counter:

print("El número más grande es", largest\_number)

else:

print("No has ingresado ningún número.")

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

De nuevo: ejecútalo, pruébalo y experimenta con él.

**LABORATORIO**

**Tiempo Estimado**

10 minutos

**Nivel de Dificultad**

Fácil

**Objetivos**

Familiarizar al estudiante con:

* Utilizar la instrucción break en los bucles.
* Reflejar situaciones de la vida real en código de computadora.

**Escenario**

La instrucción break se implementa para salir/terminar un bucle.

Diseña un programa que use un bucle while y le pida continuamente al usuario que ingrese una palabra a menos que ingrese "chupacabra" como la palabra de salida secreta, en cuyo caso el mensaje "¡Has dejado el bucle con éxito". Debe imprimirse en la pantalla y el bucle debe terminar.

No imprimas ninguna de las palabras ingresadas por el usuario. Utiliza el concepto de ejecución condicional y la sentencia break.

**Tiempo Estimado**

10-15 minutos

**Nivel de Dificultad**

Fácil

**Objectives**

Familiarizar al estudiante con:

* Utilizar la instrucción continue en los bucles.
* Reflejar situaciones de la vida real en código de computadora.

**Escenario**

La sentencia continue se usa para omitir el bloque actual y avanzar a la siguiente iteración, sin ejecutar las sentencias dentro del bucle.

Se puede usar tanto con while y for.

Tu tarea aquí es muy especial: ¡Debes diseñar un devorador de vocales! Escribe un programa que use:

* Un bucle for.
* El concepto de ejecución condicional (*if-elif-else*).
* La sentencia continue.

Tu programa debe:

* Pedir al usuario que ingrese una palabra.
* Utiliza user\_word = user\_word.upper() para convertir la palabra ingresada por el usuario a mayúsculas; hablaremos sobre los llamados **métodos de cadena**y el upper() muy pronto, no te preocupes.
* Utiliza la ejecución condicional y la instrucción continue para "comer" las siguientes vocales *A*, *E*, *I*, *O*, *U*de la palabra ingresada.
* Imprime las letras no consumidas en la pantalla, cada una de ellas en una línea separada.

Prueba tu programa con los datos que le proporcionamos.

**Datos de Prueba**

Entrada de muestra: Gregory

Salida esperada:

G

R

G

R

Y

Entrada de muestra: abstemious

Salida esperada:

B

S

T

M

S

Entrada de muestra: IOUEA

Salida esperada:

# Indicar al usuario que ingrese una palabra

# y asignarlo a la variable user\_word.

for letter in user\_word:

# Completa el cuerpo del bucle for.

**LABORATORIO**

**Tiempo Estimado**

5-10 minutos

**Nivel de Dificultad**

Fácil

**Objetivos**

Familiarizar al estudiante con:

* Utilizar la instrucción continue en los bucles.
* Modificar y actualizar el código existente.
* Reflejar situaciones de la vida real en código de computadora.

**Escenario**

Tu tarea aquí es aún más especial que antes: ¡Debes rediseñar el devorador de vocales (feo) del laboratorio anterior (3.1.2.10) y crear un mejor devorador de vocales (bonito) mejorado! Escribe un programa que use:

* Un bucle for.
* El concepto de ejecución condicional (*if-elif-else*).
* La instrucción continue.

Tu programa debe:

* Pedir al usuario que ingrese una palabra.
* Utilizar user\_word = user\_word.upper() para convertir la palabra ingresada por el usuario a mayúsculas; hablaremos sobre los llamados **métodos de cadena**y el upper() muy pronto, no te preocupes.
* Emplea la ejecución condicional y la instrucción continue para "comer" las siguientes vocales *A*, *E*, *I*, *O*, *U*de la palabra ingresada.
* Asigne las letras no consumidas a la variable word\_without\_vowels e imprime la variable en la pantalla.

Analiza el código en el editor. Hemos creado word\_without\_vowels y le hemos asignado una cadena vacía. Utiliza la operación de concatenación para pedirle a Python que combine las letras seleccionadas en una cadena más larga durante los siguientes giros de bucle, y asignarlo a la variable word\_without\_vowels.

Prueba tu programa con los datos que le proporcionamos.

**Datos de Prueba**

Entrada de muestra: Gregory

Salida esperada:

GRGRY

Entrada de muestra: abstemious

Salida esperada:

BSTMS

Entrada de muestra: IOUEA

Salida esperada:

word\_without\_vowels = ""

# Indicar al usuario que ingrese una palabra

# y asignarla a la variable user\_word.

for letter in user\_word:

# Completa el cuerpo del bucle.

# Imprimir la palabra asignada a word\_without\_vowels.

# El bucle while y la rama else

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

Te mostraremos como funciona: intenta juzgar por ti mismo si es utilizable.

En otras palabras, trata de convencerte si la función es valiosa y útil, o solo es azúcar sintáctica.

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

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

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

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

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

i = 5

while i < 5:

print(i)

i += 1

else:

print("else:", i)

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

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

i = 1

while i < 5:

print(i)

i += 1

else:

print("else:", i)

# El bucle for y la rama else

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

La salida puede ser un poco sorprendente.

La variable i conserva su último valor.

Modifica el código un poco para realizar un experimento más.

i = 111

for i in range(2, 1):

print(i)

else:

print("else:", i)

¿Puedes adivinar la salida?

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

Ejecuta el programa y verifica su salida.

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

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

¿Cómo te sientes acerca de esta variante de else?

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

for i in range(5):

print(i)

else:

print("else:", i)

**LABORATORIO**

**Tiempo Estimado**

20-30 minutos

**Nivel de Dificultad**

Medio

**Objetivos**

Familiarizar al estudiante con:

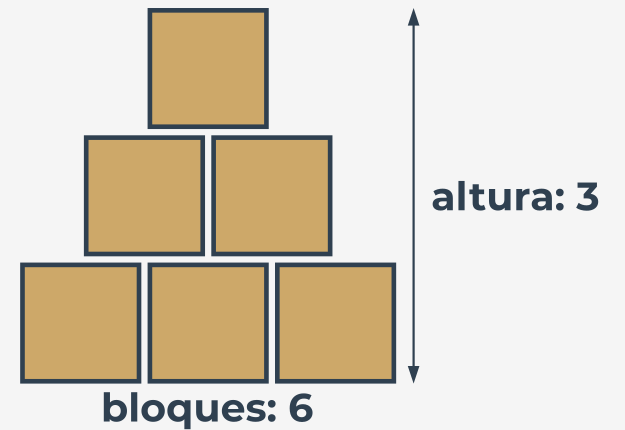
* Utilizar el bucle while.
* Encontrar la implementación adecuada de reglas definidas verbalmente.
* Reflejar situaciones de la vida real en código de computadora.

**Escenario**

Escucha esta historia: Un niño y su padre, un programador de computadoras, juegan con bloques de madera. Están construyendo una pirámide.

Su pirámide es un poco rara, ya que en realidad es una pared en forma de pirámide, es plana. La pirámide se apila de acuerdo con un principio simple: cada capa inferior contiene un bloque más que la capa superior.

La figura ilustra la regla utilizada por los constructores:



Tu tarea es escribir un programa que lea la cantidad de bloques que tienen los constructores, y generar la altura de la pirámide que se puede construir utilizando estos bloques.

Nota: La altura se mide por el número de **capas completas**: si los constructores no tienen la cantidad suficiente de bloques y no pueden completar la siguiente capa, terminan su trabajo inmediatamente.

Prueba tu código con los datos que hemos proporcionado.

**Datos de Prueba**

Entrada de muestra: 6

Salida esperada: La altura de la pirámide es: 3

Entrada de muestra: 20

Salida esperada: La altura de la pirámide es: 5

Entrada de muestra: 1000

Salida esperada: La altura de la pirámide es: 44

Entrada de muestra: 2

Salida esperada: La altura de la pirámide es: 1

blocks = int(input("Ingresa el número de bloques: "))

#

# Escribe tu código aquí.

#

print("La altura de la pirámide:", height)

**LABORATORIO**

**Tiempo Estimado**

20 minutos

**Nivel de Dificultad**

Media

**Objetivos**

Familiarizar al estudiante con:

* Utilizar el bucle while.
* Convertir bucles definidos verbalmente en código real de Python.

**Escenario**

En 1937, un matemático alemán llamado Lothar Collatz formuló una hipótesis intrigante (aún no se ha comprobado) que se puede describir de la siguiente manera:

1. Toma cualquier número entero que no sea negativo y que no sea cero y asígnale el nombre c0.
2. Si es par, evalúa un nuevo c0 como c0 Ã· 2.
3. De lo contrario, si es impar, evalúe un nuevo  c0  como 3 Ã— c0 + 1.
4. Si c0 â‰  1, salta al punto 2.

La hipótesis dice que, independientemente del valor inicial de c0, el valor siempre tiende a 1.

Por supuesto, es una tarea extremadamente compleja usar una computadora para probar la hipótesis de cualquier número natural (incluso puede requerir inteligencia artificial), pero puede usar Python para verificar algunos números individuales. Tal vez incluso encuentres el que refutaría la hipótesis.

Escribe un programa que lea un número natural y ejecute los pasos anteriores siempre que c0 sea diferente de 1. También queremos que cuente los pasos necesarios para lograr el objetivo. Tu código también debe mostrar todos los valores intermedios de c0.

Sugerencia: la parte más importante del problema es como transformar la idea de Collatz en un bucle while- esta es la clave del éxito.

Prueba tu código con los datos que hemos proporcionado.

**Test Data**

Entrada de muestra: 15

Salida esperada:

46

23

70

35

106

53

160

80

40

20

10

5

16

8

4

2

1

pasos = 17

Entrada de muestra: 16

Salida esperada:

8

4

2

1

pasos = 4

Entrada de muestra: 1023

Salida esperada:

3070

1535

4606

2303

6910

3455

10366

5183

15550

7775

23326

11663

34990

17495

52486

26243

78730

39365

118096

59048

29524

14762

7381

22144

11072

5536

2768

1384

692

346

173

520

260

130

65

196

98

49

148

74

37

112

56

28

14

7

22

11

34

17

52

26

13

40

20

10

5

16

8

4

2

1

pasos = 62

**Puntos Clave**

1. Existen dos tipos de bucles en Python: while y for:

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

# Ejemplo 1

while True:

print("Atascado en un bucle infinito.")

# Ejemplo 2

counter = 5

while counter > 2:

print(counter)

counter -= 1

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

# Ejemplo 1

word = "Python"

for letter in word:

print(letter, end="\*")

# Ejemplo 2

for i in range(1, 10):

if i % 2 == 0:

print(i)

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

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

text = "OpenEDG Python Institute"

for letter in text:

if letter == "P":

break

print(letter, end="")

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

text = "pyxpyxpyx"

for letter in text:

if letter == "x":

continue

print(letter, end="")

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

n = 0

while n != 3:

print(n)

n += 1

else:

print(n, "else")

print()

for i in range(0, 3):

print(i)

else:

print(i, "else")

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

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

Código de ejemplo:

for i in range(3):

print(i, end=" ") # Salidas: 0 1 2

for i in range(6, 1, -2):

print(i, end=" ") # Salidas: 6, 4, 2

# Puntos Clave: continuación

**Ejercicio 1**

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

for i in range(1, 11):

# Línea de código.

# Línea de código.

Revisar

**Ejercicio 2**

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

x = 1

while x < 11:

# Línea de código.

# Línea de código.

# Línea de código.

Revisar

**Ejercicio 3**

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

for ch in "john.smith@pythoninstitute.org":

if ch == "@":

# Línea de código.

# Línea de código.

Revisar

**Ejercicio 4**

Crea un programa con un bucle for y una sentenciacontinue. 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

**Ejercicio 5**

¿Cuál es la salida del siguiente código?

n = 3

while n > 0:

print(n + 1)

n -= 1

else:

print(n)

Revisar

**Ejercicio 5**

¿Cuál es la salida del siguiente código?

n = range(4)

for num in n:

print(num - 1)

else:

print(num)

Revisar

**Ejercicio 7**

¿Cuál es la salida del siguiente código?

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

print(i)

Revisar

# Lógica de computadora

¿Te has dado cuenta de que las condiciones que hemos usado hasta ahora han sido muy simples, por no decir, bastante primitivas? Las condiciones que utilizamos en la vida real son mucho más complejas. Veamos este enunciado:

*Si tenemos tiempo libre, y el clima es bueno, saldremos a caminar.*

Hemos utilizado la conjunción and (y), lo que significa que salir a caminar depende del cumplimiento simultáneo de estas dos condiciones. En el lenguaje de la lógica, tal conexión de condiciones se denomina **conjunción**. Y ahora otro ejemplo:

*Si tu estás en el centro comercial o yo estoy en el centro comercial, uno de nosotros le comprará un regalo a mamá.*

La aparición de la palabra or (o) significa que la compra depende de al menos una de estas condiciones. En lógica, este compuesto se llama una **disyunción**.

Está claro que Python debe tener operadores para construir conjunciones y disyunciones. Sin ellos, el poder expresivo del lenguaje se debilitaría sustancialmente. Se llaman **operadores lógicos**.

## and

Un operador de conjunción lógica en Python es la palabra *and*. Es un operador binario **con una prioridad inferior a la expresada por los operadores de comparación**. Nos permite codificar condiciones complejas sin el uso de paréntesis como este:

counter > 0 and value == 100

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

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

|  |  |  |
| --- | --- | --- |
| **Argumento**A | **Argumento**B | A and B |
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

## or

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

|  |  |  |
| --- | --- | --- |
| **Argumento**A | **Argumento**B | A or B |
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

## not

Además, hay otro operador que se puede aplicar para condiciones de construcción. Es un **operador unario que realiza una negación lógica**. Su funcionamiento es simple: convierte la verdad en falso y lo falso en verdad

Este operador se escribe como la palabra not, y su **prioridad es muy alta: igual que el unario**+**y**-. Su tabla de verdad es simple:

|  |  |
| --- | --- |
| **Argumento** | not**Argumento** |
| False | True |
| True | False |

**Expresiones lógicas**

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

# Ejemplo 1:

print(var > 0)

print(not (var <= 0))

# Ejemplo 2:

print(var != 0)

print(not (var == 0))

Puedes estar familiarizado con las leyes de De Morgan. Dicen que:

*La negación de una conjunción es la separación de las negaciones.*

*La negación de una disyunción es la conjunción de las negaciones.*

Escribamos lo mismo usando Python:

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

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

Observa como se han utilizado los paréntesis para codificar las expresiones: las colocamos allí para mejorar la legibilidad.

Deberíamos agregar que ninguno de estos operadores de dos argumentos se puede usar en la forma abreviada conocida como op=. Vale la pena recordar esta excepción.

**Valores lógicos frente a bits individuales**

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

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

i = 1

j = not not i

**Operadadores bit a bit**

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

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

Aquí están todos ellos:

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operaciones bit a bit (**&**,**|**, y**^**)** | | | | |
| **Argumento**A | **Argumento**B | A & B | A | B | A ^ B |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

|  |  |
| --- | --- |
| **Operaciones bit a bit (~)** | |
| **Argumento** | ~**Argumento** |
| 0 | 1 |
| 1 | 0 |

Hagámoslo más fácil:

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

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

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

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

# Operaciones lógicas frente a operaciones de bit: continuación

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

i = 15

j = 22

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

i: 00000000000000000000000000001111

j: 00000000000000000000000000010110

Se ejecuta la asignación:

log = i and j

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

log: True

Ahora la operación a nivel de bits - aquí está:

bit = i & j

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

|  |  |
| --- | --- |
| i | 00000000000000000000000000001111 |
| j | 00000000000000000000000000010110 |
| bit = i & j | 00000000000000000000000000000110 |

Estos bits corresponden al valor entero de seis.

Veamos ahora los operadores de negación. Primero el lógico:

logneg = not i

La variable logneg se establecerá en False: no es necesario hacer nada más.

La negación a nivel de bits es así:

bitneg = ~i

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

|  |  |
| --- | --- |
| i | 00000000000000000000000000001111 |
| bitneg = ~i | 11111111111111111111111111110000 |

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

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

**¿Cómo tratamos los bits individuales?**

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

flag\_register = 0x1234

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

flag\_register = 0000000000000000000000000000x000

Es posible que tengas que hacer frente a las siguientes tareas:

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

x & 1 = x

x & 0 = 0

Si aplicas la operación & a la variable flagRegister junto con la siguiente imagen de bits:

00000000000000000000000000001000

(observa el 1 en la posición de tu bit) como resultado, obtendrás una de las siguientes cadenas de bits:

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

Dicha secuencia de ceros y unos, cuya tarea es tomar el valor o cambiar los bits seleccionados, se denomina **máscara de bits**.

Construyamos una máscara de bits para detectar el estado de tus bits. Debería apuntar a **el tercer bit**. Ese bit tiene el peso de 23=8. Se podría crear una máscara adecuada mediante la siguiente sentencia:

the\_mask = 8

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

if flag\_register & the\_mask:

# Mi bit se estableció en 1.

else:

# Mi bit se restableció a 0.

2. **Reinicia tu bit**: asigna un cero al bit, mientras que todos los otros bits deben permanecer sin cambios; usemos la misma propiedad de la conjunción que antes, pero usemos una máscara ligeramente diferente, exactamente como se muestra a continuación:

11111111111111111111111111110111

Tenga en cuenta que la máscara se creó como resultado de la negación de todos los bits de la variable the\_mask. Restablecer el bit es simple, y se ve así (elige el que más te guste):

flag\_register = flag\_register & ~the\_mask

flag\_register &= ~the\_mask

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

x | 1 = 1

x | 0 = x

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

flag\_register = flag\_register | the\_mask

flag\_register |= the\_mask

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

x ^ 1 = ~x

x ^ 0 = x

Niega tu bit con las siguientes instrucciones:

flag\_register = flag\_register ^ the\_mask

flag\_register ^= the\_mask

* 3.3.1.4 Operaciones lógicas y de bits en Python | Operadores bit a bit

# Desplazamiento izquierdo binario y desplazamiento derecho binario

Python ofrece otra operación relacionada con los bits individuales: **shifting**. Esto se aplica solo a los valores de **número entero**, y no debe usar flotantes como argumentos para ello

Ya aplicas esta operación muy a menudo y muy inconscientemente. ¿Cómo multiplicas cualquier número por diez? Echa un vistazo:

12345 × 10 = 123450

Como puede ver, **multiplicar por diez es de hecho un desplazamiento** de todos los dígitos a la izquierda y llenar el vacío resultante con cero.

¿División entre diez? Echa un vistazo:

12340 ÷ 10 = 1234

Dividir entre diez no es más que desplazar los dígitos a la derecha.

La computadora realiza el mismo tipo de operación, pero con una diferencia: como dos es la base para los números binarios (no 10), **desplazar un valor un bit a la izquierda corresponde a multiplicarlo por dos**; respectivamente, **desplazar un bit a la derecha es como dividir entre dos**(observe que se pierde el bit más a la derecha).

Los **operadores de cambio**en Python son un par de **dígrafos**: < < y > >, sugiriendo claramente en qué dirección actuará el cambio.

value << bits

value >> bits

**El argumento izquierdo de estos operadores es un valor entero cuyos bits se desplazan. El argumento correcto determina el tamaño del turno.**

Esto demuestra que esta operación ciertamente no es conmutativa

La prioridad de estos operadores es muy alta. Los verás en la tabla de prioridades actualizada, que te mostraremos al final de esta sección.

Echa un vistazo a los cambios en la ventana del editor.

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

17 68 8

**salida**

Nota:

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

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

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

* [**Sandbox**](https://edube.org/sandbox)

## Code

var = 17  
var\_right = var >> 1  
var\_left = var << 2  
print(var, var\_left, var\_right)



1

2

3

4

5

var = 17

var\_right = var >> 1

var\_left = var << 2

print(var, var\_left, var\_right)

* **Console**

# Puntos Clave

1. Python es compatible con los siguientes operadores lógicos:

* and → si ambos operandos son verdaderos, la condición es verdadera, por ejemplo, (True and True) es True.
* or → si alguno de los operandos es verdadero, la condición es verdadera, por ejemplo, (True or False) es True.
* not → devuelve False si el resultado es verdadero y devuelve True si es falso, por ejemplo, not True es False.

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

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

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

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

\* -16 (decimal del complemento a 2 con signo) -- lee más acerca de la operación [Complemento a dos](https://es.wikipedia.org/wiki/Complemento_a_dos).

**Ejercicio 1**

¿Cuál es la salida del siguiente fragmento de código?

x = 1

y = 0

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

print(not(z))

Revisar

**Ejercicio 2**

¿Cuál es la salida del siguiente fragmento de código?

x = 4

y = 1

a = x & y

b = x | y

c = ~x # !difícil!

d = x ^ 5

e = x >> 2

f = x << 2

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

Revisar 0 5 -5 1 1 16 el -5 de obtiene cambiando ceros por unos y sumado uno sabiendo que ya es negativo.

Complemento a dos: <https://es.wikipedia.org/wiki/Complemento_a_dos>

# ¿Por qué necesitamos listas?

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

var1 = int(input())

var2 = int(input())

var3 = int(input())

var4 = int(input())

var5 = int(input())

var6 = int(input())

:

:

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

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

Debes percatarte que ni siquiera tienes suficiente papel para completar la tarea.

Hasta ahora, has aprendido como declarar variables que pueden almacenar exactamente un valor dado a la vez. Tales variables a veces se denominan **escalares** por analogía con las matemáticas. Todas las variables que has usado hasta ahora son realmente escalares.

Piensa en lo conveniente que sería declarar una variable que podría **almacenar más de un valor**. Por ejemplo, cien, o mil o incluso diez mil. Todavía sería una y la misma variable, pero muy amplia y espaciosa. ¿Suena atractivo? Quizás, pero ¿cómo manejarías un contenedor así lleno de valores diferentes? ¿Cómo elegirías solo el que necesitas?

¿Y si solo pudieras numerarlos? Y luego di: *dame el valor número 2; asigna el valor número 15; aumenta el número del valor 10000*.

Te mostraremos como declarar tales **variables de múltiples valores**. Haremos esto con el ejemplo que acabamos de sugerir. Escribiremos un programa **que ordene una secuencia de números**. No seremos particularmente ambiciosos: asumiremos que hay exactamente cinco números.

Vamos a crear una variable llamada numeros; se le asigna no solo un número, sino que se llena con una lista que consta de cinco valores (nota: la lista **comienza con un corchete abierto y termina con un corchete cerrado**; el espacio entre los corchetes es llenado con cinco números separados por comas).

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

Digamos lo mismo utilizando una terminología adecuada: numeros**es una lista que consta de cinco valores, todos ellos números**. También podemos decir que esta sentencia crea una lista de longitud igual a cinco (ya que contiene cinco elementos).

Los elementos dentro de una lista **pueden tener diferentes tipos**. Algunos de ellos pueden ser enteros, otros son flotantes y otros pueden ser listas.

Python ha adoptado una convención que indica que los elementos de una lista están **siempre numerados desde cero**. Esto significa que el elemento almacenado al principio de la lista tendrá el número cero. Como hay cinco elementos en nuestra lista, al último de ellos se le asigna el número cuatro. No olvides esto.

Pronto te acostumbrarás y se convertirá en algo natural.

Antes de continuar con nuestra discusión, debemos indicar lo siguiente: nuestra lista **es una colección de elementos, pero cada elemento es un escalar**.

# Indexando Listas

¿Cómo cambias el valor de un elemento elegido en la lista?

Vamos a **asignar un nuevo valor de**111**al primer elemento**en la lista. Lo hacemos de esta manera:

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

print("Contenido de la lista original:", numbers) # Imprimiendo contenido de la lista original.

numbers[0] = 111

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

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

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

print("Contenido de la lista original:", numbers) # Imprimiendo contenido de la lista original.

numbers[0] = 111

print("\nPrevio contenido de la lista:", numbers) # Imprimiendo contenido de la lista anterior.

numbers[1] = numbers[4] # Copiando el valor del quinto elemento al segundo elemento.

print("Nuevo contenido de la lista:", numbers) # Imprimiendo el contenido de la lista actual.

El valor dentro de los corchetes que selecciona un elemento de la lista se llama un **índice**, mientras que la operación de seleccionar un elemento de la lista se conoce como **indexación**.

Vamos a utilizar la función print() para imprimir el contenido de la lista cada vez que realicemos los cambios. Esto nos ayudará a seguir cada paso con más cuidado y ver que sucede después de una modificación de la lista en particular.

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

* [**Sandbox**](https://edube.org/sandbox)

## Code

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



1

2

3

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

print("Contenido de la lista:", numbers) # Imprimiendo contenido de la lista original.

* **Console**

Solución

**Accediendo al contenido de la lista**

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

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

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

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

print(numbers) # Imprimiendo la lista completa.

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

[111, 1, 7, 2, 1]

**output**

**La función len()**

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

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

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

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

# Eliminando elementos de una lista

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

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

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

del numbers[1]

print(len(numbers))

print(numbers)

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

print(numbers[4])

numbers[4] = 1

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

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

* [**Sandbox**](https://edube.org/sandbox)

## Code

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



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

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

print("Contenido de la lista original:", numbers) # Imprimiendo el contenido de la lista original.

numbers[0] = 111

print("\nContenido de la lista con cambio:", numbers) # Imprimiendo contenido de la lista con 111.

numbers[1] = numbers[4] # Copiando el valor del quinto elemento al segundo elemento.

print("Contenido de la lista con intercambio:", numbers) # Imprimiendo contenido de la lista con intercambio.

print("\nLongitud de la lista:", len(numbers)) # Imprimiendo la longitud de la lista.

###

del numbers[1] # Eliminando el segundo elemento de la lista.

print("Longitud de la nueva lista:", len(numbers)) # Imprimiendo nueva longitud de la lista.

print("\nNuevo contenido de la lista:", numbers) # Imprimiendo el contenido de la lista actual.

###

* **Console**

# 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(numbers[-1])

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

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

print(numbers[-2])

El fragmento de ejemplo dará como salida un 2.

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

* [**Sandbox**](https://edube.org/sandbox)

## Code

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



1

2

3

4

numbers = [111, 7, 2, 1]

print(numbers[-1])

print(numbers[-2])

* **Console**

**LABORATORIO**

## Tiempo Estimado

5 minutos

## Nivel de Dificultad

Muy fácil

## Objetivos

Familiarizar al estudiante con:

* Usar instrucciones básicas relacionadas con listas.
* Crear y modificar listas.

## Escenario

Había una vez un sombrero. El sombrero no contenía conejo, sino una lista de cinco números: 1, 2, 3, 4 y 5.

Tu tarea es:

* Escribir una línea de código que solicite al usuario que reemplace el número central en la lista con un número entero ingresado por el usuario (Paso 1).
* Escribir una línea de código que elimine el último elemento de la lista (Paso 2).
* Escribir una línea de código que imprima la longitud de la lista existente (Paso 3).

¿Listo para este desafío?

Solución:

hat\_list = [1, 2, 3, 4, 5] # Esta es una lista existente de números ocultos en el sombrero.

# Paso 1: escribe una línea de código que solicite al usuario

# reemplazar el número de en medio con un número entero ingresado por el usuario.

hat\_list[2]=int(input("ingrese un numero para reemplazar: "))

# Paso 2: escribe aquí una línea de código que elimine el último elemento de la lista.

del hat\_list[4]

# Paso 3: escribe aquí una línea de código que imprima la longitud de la lista existente.

print("\nLongitud de la lista:", len(hat\_list))

print(hat\_list)

# Funciones frente a métodos

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

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

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

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

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

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

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

result = function(arg)

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

Una invocación de un método típico usualmente se ve así:

result = data.method(arg)

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

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

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

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

[Pr](https://edube.org/learn/python-essentials-1-esp/laboratorio-lo-b-aacute-sico-de-las-listas-1)

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

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

list.append(value)

Dicha operación se realiza mediante un método llamado append(). Toma el valor de su argumento y lo coloca **al final de la lista** que posee el método.

La longitud de la lista aumenta en uno.

El método insert() es un poco más inteligente: puede agregar un nuevo elemento **en cualquier lugar de la lista**, no solo al final.

list.insert(location, value)

Toma dos argumentos:

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

Observa el código en el editor. Ve como usamos los métodos append() e insert(). Presta atención a lo que sucede después de usar insert(): el primer elemento anterior ahora es el segundo, el segundo el tercero, y así sucesivamente.

Agrega el siguiente fragmento después de la última línea de código en el editor:

numbers.insert(1, 333)

Imprime el contenido de la lista final en la pantalla y ve que sucede. El fragmento de código sobre el fragmento de código inserta 333 en la lista, por lo que es el segundo elemento. El segundo elemento anterior se convierte en el tercero, el tercero en el cuarto, y así sucesivamente.

numbers = [111, 7, 2, 1]

print(len(numbers))

print(numbers)

###

numbers.append(4)

print(len(numbers))

print(numbers)

###

numbers.insert(0, 222)

print(len(numbers))

print(numbers)

#

**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.

Será una secuencia de números enteros consecutivos del 1 (luego agrega uno a todos los valores agregados) hasta 5.

Hemos modificado un poco el fragmento:

my\_list = [] # Creando una lista vacía.

for i in range(5):

my\_list.insert(0, i + 1)

print(my\_list)

¿Qué pasará ahora? Ejecuta el programa y comprueba si esta vez también tenías razón.

Deberías obtener la misma secuencia, pero en **orden inverso**(este es el mérito de usar el método insert()).

**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.

Será una secuencia de números enteros consecutivos del 1 (luego agrega uno a todos los valores agregados) hasta 5.

Hemos modificado un poco el fragmento:

my\_list = [] # Creando una lista vacía.

for i in range(5):

my\_list.insert(0, i + 1)

print(my\_list)

¿Qué pasará ahora? Ejecuta el programa y comprueba si esta vez también tenías razón.

Deberías obtener la misma secuencia, pero en **orden inverso**(este es el mérito de usar el método insert()).

my\_list = [] # Creando una lista vacía.

for i in range(5):

my\_list.append(i + 1)

print(my\_list)

resultado [1, 2, 3, 4, 5]

# 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 deseas **calcular la suma de todos los valores almacenados en la lista**my\_list.

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

Comentemos este ejemplo:

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

## El segundo aspecto del bucle for

Pero el bucle for puede hacer mucho más. Puede ocultar todas las acciones conectadas a la indexación de la lista y entregar todos los elementos de la lista de manera práctica.

Este fragmento modificado muestra como funciona:

my\_list = [10, 1, 8, 3, 5]

total = 0

for i in my\_list:

total += i

print(total)

¿Qué sucede aquí?

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

my\_list = [10, 1, 8, 3, 5]

total = 0

for i in range(len(my\_list)):

total += my\_list[i]

print(total)

# Listas en acción

Dejemos de lado las listas por un breve momento y veamos un tema intrigante.

Imagina que necesitas reorganizar los elementos de una lista, es decir, revertir el orden de los elementos: el primero y el quinto, así como el segundo y cuarto elementos serán intercambiados. El tercero permanecerá intacto.

Pregunta: ¿Cómo se pueden intercambiar los valores de dos variables?

Echa un vistazo al fragmento:

variable\_1 = 1

variable\_2 = 2

variable\_2 = variable\_1

variable\_1 = variable\_2

Si haces algo como esto, **perderás el valor previamente almacenado**en variable\_2. Cambiar el orden de las tareas no ayudará. Necesitas una tercera variable **que sirva como almacenamiento auxiliar**.

Así es como puedes hacerlo:

variable\_1 = 1

variable\_2 = 2

auxiliary = variable\_1

variable\_1 = variable\_2

variable\_2 = auxiliary

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

variable\_1 = 1

variable\_2 = 2

variable\_1, variable\_2 = variable\_2, variable\_1

Claro, efectivo y elegante, ¿no?

**Listas en acción**

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

my\_list = [10, 1, 8, 3, 5]

my\_list[0], my\_list[4] = my\_list[4], my\_list[0]

my\_list[1], my\_list[3] = my\_list[3], my\_list[1]

print(my\_list)

Ejecuta el fragmento. Su salida debería verse así:

[5, 3, 8, 1, 10]

**salida**

Se ve bien con cinco elementos.

¿Seguirá siendo aceptable con una lista que contenga 100 elementos? No, no lo hará.

¿Puedes usar el bucle for para hacer lo mismo automáticamente, independientemente de la longitud de la lista? Si, si puedes.

Así es como lo hemos hecho:

my\_list = [10, 1, 8, 3, 5]

length = len(my\_list)

for i in range(length // 2):

my\_list[i], my\_list[length - i - 1] = my\_list[length - i - 1], my\_list[i]

print(my\_list)

Nota:

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

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

**LABORATORIO**

## Tiempo Estimado

10-15 minutos

## Nivel de Dificultad

Fácil

## Objetivos

Familiarizar al estudiante con:

* Crear y modificar listas simples.
* Utilizar métodos para modificar listas.

## Escenario

Los Beatles fueron uno de los grupos de música más populares de la década de 1960 y la banda más vendida en la historia. Algunas personas los consideran el acto más influyente de la era del rock. De hecho, se incluyeron en la compilación de la revista *Time* de las 100 personas más influyentes del siglo XX.

a banda sufrió muchos cambios de formación, que culminaron en 1962 con la formación de John Lennon, Paul McCartney, George Harrison y Richard Starkey (mejor conocido como Ringo Starr).

Escribe un programa que refleje estos cambios y le permita practicar con el concepto de listas. Tu tarea es:

* Paso 1: Crea una lista vacía llamada beatles.
* Paso 2: Emplea el método append() para agregar los siguientes miembros de la banda a la lista: John Lennon, Paul McCartney y George Harrison.
* Paso 3: Emplea el buclefor y el append() para pedirle al usuario que agregue los siguientes miembros de la banda a la lista: Stu Sutcliffe, y Pete Best.
* Paso 4: Usa la instrucción del para eliminar a Stu Sutcliffe y Pete Best de la lista.
* Paso 5: Usa el método insert() para agregar a Ringo Starr al principio de la lista.

Por cierto, ¿eres fan de los Beatles? (Los Beatles son una de las bandas favoritas de Greg. Pero espera...¿Quién es Greg?)

* [**Sandbox**](https://edube.org/sandbox)

## Code

# paso 1  
print("Paso 1:", Beatles)  
  
# paso 2  
print("Paso 2:", Beatles)  
  
# paso 3  
print("Paso 3:", Beatles)  
  
# paso 4  
print("Paso 4:", Beatles)  
  
# paso 5  
print("Paso 5:", Beatles)  
  
  
# probando la longitud de la lista  
print("Los Fav", len(Beatles))



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

# paso 1

print("Paso 1:", Beatles)

# paso 2

print("Paso 2:", Beatles)

# paso 3

print("Paso 3:", Beatles)

# paso 4

print("Paso 4:", Beatles)

# paso 5

print("Paso 5:", Beatles)

# probando la longitud de la lista

print("Los Fav", len(Beatles))

* **Console**

Solución

# paso 1

Beatles=[]

# paso

print("Paso 1:", Beatles)

Beatles.append("john lennon")

Beatles.append("paul mccarney")

Beatles.append("george harrinton")

print("Paso 2:", Beatles)

# paso 3

print ("Agruegue a Stu Sutcliffe y a Pete Best a la lista")

for i in range (2):

Beatles.append(input())

print("Paso 3:", Beatles)

# paso 4

del Beatles[3:4]

print("Paso 4:", Beatles)

# paso 5

Beatles.insert(0, 'Ringo Starr')

print("Paso 5:", Beatles)

# probando la longitud de la lista

print("Los Fav", len(Beatles))

# Puntos Clave

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

my\_list = [1, None, True, "Soy una cadena", 256, 0]

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

my\_list = [1, None, True, 'Soy una cadena', 256, 0]

print(my\_list[3]) # salida: Soy una cadena

print(my\_list[-1]) # salida: 0

my\_list[1] = '?'

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

my\_list.insert(0, "primero")

my\_list.append("último")

print(my\_list) # outputs: ['primero', 1, '?', True, 'Soy una cadena', 256, 0, 'último']

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

my\_list = [1, 'a', ["lista", 64, [0, 1], False]]

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

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

my\_list = [1, 2, 3, 4]

del my\_list[2]

print(my\_list) # salida: [1, 2, 4]

del my\_list # borra la lista entera

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

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

my\_list = ["blanco", "purpura", "azul", "amarillo", "verde"]

for color in my\_list:

print(color)

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

my\_list = ["blanco", "purpura", "azul", "amarillo", "verde"]

print(len(my\_list)) # salida 5

del my\_list[2]

print(len(my\_list)) # salida 4

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

**Ejercicio 1**

¿Cuál es la salida del siguiente fragmento de código?

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

lst.insert(1, 6)

del lst[0]

lst.append(1)

print(lst)

Revisar

**Ejercicio 2**

¿Cuál es la salida del siguiente fragmento de código?

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

lst\_2 = []

add = 0

for number in lst:

add += number

lst\_2.append(add)

print(lst\_2)

Revisar

**Ejercicio 3**

¿Cuál es la salida del siguiente fragmento de código?

lst = []

del lst

print(lst)

# 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.

[Prev](https://edube.org/learn/python-essentials-1-esp/resumen-de-la-secci-oacute-n-15) [Next](https://edube.org/learn/python-essentials-1-esp/ordenando-listas-simples-el-ordenamiento-de-burbuja-4)

Revisar

**Ejercicio 4**

¿Cuál es la salida del siguiente fragmento de código?

lst = [1, [2, 3], 4]

print(lst[1])

print(len(lst))