5 Bucles

Actualizando variables

Un patrón común en las asignaciones es una instrucción de asignación que actualiza una variable, donde el nuevo valor de la variable depende de la antigua.

```
python
x = x + 1
```

Esto significa que "obtenga el valor actual de `x`, agregue 1 y luego actualice `x` con el nuevo valor".

Si intentas actualizar una variable que no existe, obtienes un error, porque Python evalúa el lado derecho antes de asignar un valor a `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Antes de poder actualizar una variable, debes **inicializarla**:

```
>>> x = 0
>>> x = x + 1
```

Actualizar una variable agregando 1 se llama **incrementar**.

La declaración 'while'

Los ordenadores se utilizan a menudo para automatizar tareas repetitivas. Repetir tareas idénticas o similares sin cometer errores es algo que a los ordenadores se les da muy bien y a las personas se les da muy mal. Debido a que la iteración es tan común, Python proporciona varias

características de lenguaje para que sea más fácil.

Una forma de iteración en Python es la instrucción `while`. Aquí hay un programa simple que cuenta a partir de cinco y luego dice "¡Explosión!".

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

Casi se puede leer la instrucción `while` como si fuera inglés. Significa que "Mientras `n` es mayor que 0, muestra el valor de `n` y luego reduce el valor de `n` en 1. Cuando llegues a 0, sal de la instrucción `while` y muestra la palabra `Blastoff !`"

Más formalmente, aquí está el flujo de ejecución para una instrucción `while`:

1. Evalúa la condición, obteniendo "Verdadero" o "Falso".
2. Si la condición es falsa, salga de la instrucción `while` y continúa con la ejecución en la siguiente instrucción.
3. Si la condición es verdadera, ejecuta el cuerpo y luego vuelve al paso 1.

Este tipo de flujo se llama un bucle porque el tercer paso vuelve a la parte superior. Cada vuelta del bucle la llamamos una **iteración**. Para el bucle anterior, diríamos que "tenía cinco iteraciones", lo que significa que el cuerpo del bucle se ejecutó cinco veces.

El cuerpo del bucle debe cambiar el valor de una o más variables para que finalmente la condición se vuelva falsa y el bucle finalice. Llamamos a la variable que cambia cada vez que el bucle se ejecuta y controla cuando el bucle termina la **variable de iteración**. Si no hay una variable de iteración, el bucle se repetirá para siempre, dando como resultado un **bucle infinito**.

Bucles infinitos

Una fuente inagotable de diversión para los programadores es la observación de que las instrucciones en el champú, "Enjabonar, enjuagar, repetir", son un bucle infinito porque no hay una **variable de iteración** que indique cuántas veces se debe ejecutar el bucle.

En el caso de `countdown`, podemos probar que el bucle termina porque sabemos que el valor de `n` es finito, y podemos ver que el valor de `n` se reduce cada vez que pasa por el bucle, por lo que eventualmente tiene que llegar a 0. Otras veces, un bucle es obviamente infinito porque no tiene ninguna variable de iteración.

"Infinite loops" y `break`

A veces no sabes que es hora de terminar un ciclo hasta que lleguas a la mitad del cuerpo. En ese caso, puedes escribir un bucle infinito a propósito y luego usar la instrucción `break` para saltar fuera del bucle.

Este bucle es obviamente un **bucle infinito** porque la expresión lógica en la instrucción `while` es simplemente la constante lógica `True`:

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('Done!')
```

Si cometes el error y ejecuta este código, aprenderás rápidamente cómo detener un proceso de Python fuera de control en su sistema o dónde encontrarás el botón de apagado en su ordenador. Este programa se ejecutará para siempre o hasta que la batería se agote porque la expresión lógica en la parte superior del bucle siempre es verdadera.

Si bien este es un bucle infinito disfuncional, aún podemos usar este patrón para crear bucles útiles, siempre y cuando agreguemos cuidadosamente `break` cuando hayamos alcanzado la condición de salida.

Por ejemplo, supongamos que desea recibir información del usuario hasta que escriban `done`. Podrías escribir:

https://trinket.io/embed/python3/7a1dd00756

La condición del bucle es `True`, que siempre es verdadera, por lo que el bucle se ejecuta repetidamente hasta que llega a la instrucción break.

Cada vez que itera, le pide al usuario que de un input. Si el usuario escribe `done`, la instrucción `break` sale del bucle. De lo contrario, el programa hace eco de lo que sea que el usuario escriba y vuelve a la parte superior del bucle. Aquí hay una muestra de ejecución:

```
> hello there
hello there
> finished
finished
> done
Done!
```

Esta forma de escribir bucles `while` es común porque puede verificar la condición en cualquier parte del bucle (no solo en la parte superior) y puede expresar la condición de detención afirmativamente ("detenerse cuando esto sucede") en lugar de negativamente ("seguir adelante hasta que eso suceda").

Finalización de iteraciones con `continue`

A veces te encuentras en una iteración de un bucle y deseas finalizar la iteración actual y pasar de inmediato a la siguiente iteración. En ese caso, puedes usar la instrucción `continue` para saltar a la siguiente iteración sin terminar el cuerpo del bucle para la iteración actual.

Aquí hay un ejemplo de un bucle que copia su entrada hasta que el usuario escribe "hecho", pero trata las líneas que comienzan con el carácter de hash como líneas que no se imprimen (como los comentarios de Python).

https://trinket.io/embed/python3/f1f5dc61e1

Aquí hay una muestra de ejecución de este nuevo programa con `continue` agregado.

```
> hello there
hello there
> # don't print this
```

```
> print this!
print this!
> done
Done!
```

Todas las líneas se imprimen, excepto las que comienza con el signo hash porque cuando se ejecuta `continue`, finaliza la iteración actual y vuelve a la instrucción `while` para iniciar la siguiente iteración, omitiendo la instrucción `print`.

Bucles definidos usando `for`

A veces queremos recorrer un **conjunto** de cosas como una lista de palabras, las líneas de un archivo o una lista de números. Cuando tenemos una lista de cosas para recorrer, podemos construir un bucle **definido** usando una instrucción `for`. Llamamos a la frase `while` un bucle **indefinido** porque simplemente se repite hasta que alguna condición se convierte en `Falsa`, mientras que el bucle `for` se repite en un conjunto conocido de elementos, por lo que se ejecuta a través de tantas iteraciones como Elementos del conjunto.

La sintaxis de un bucle `for` es similar al bucle `while` en que hay una sentencia `for` y un cuerpo de bucle:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print('Happy New Year:', friend)
print('Done!')
```

En términos de Python, la variable `friends` es una lista [1] de tres cadenas y el bucle `for` recorre la lista y ejecuta el cuerpo una vez para cada una de las tres cadenas en la lista que da como resultado esta salida:

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

La traducción de este bucle `for` al inglés no es tan directa como el `while`, pero si piensas en los amigos como un **conjunto**, quedaría así: "Ejecute las declaraciones en el cuerpo del bucle `for` una

vez para cada amigo **en** el conjunto llamado amigos ".

Mirando el bucle `for`, **for** y **in** son palabras clave reservadas de Python, y `friend` y `friends` son variables.

```
for friend in friends:
    print('Happy New Year:', friend)
```

En particular, `friend` es la **variable de iteración** para el bucle for. La variable `friend` cambia para cada iteración del bucle y se controla cuando se completa el bucle `for`. La **variable de iteración** pasa sucesivamente por las tres cadenas almacenadas en la variable `friends`.

Patrones de bucle

A menudo, utilizamos un bucle `for` o `while` para revisar una lista de elementos o el contenido de un archivo y estamos buscando algo como el valor más grande o más pequeño de los datos que escaneamos.

Estos bucles generalmente se componen de:

- Una o más variables inicializadas antes de que comience el ciclo.
- Realización de algunos cálculos en cada elemento del cuerpo del bucle, posiblemente cambiando las variables en el cuerpo del bucle
- Observación las variables resultantes cuando se completa el bucle.

Usaremos una lista de números para demostrar los conceptos y la construcción de estos patrones de bucle.

Bucles de suma y recuento

Por ejemplo, para contar el número de elementos en una lista, escribiríamos el siguiente bucle `for`:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print('Count: ', count)
```

Establecemos la variable `count` en cero antes de que comience el bucle, luego escribimos un bucle `for` para ejecutar la lista de números. Nuestra variable **iterativa** se llama `itervar` y, aunque no usamos `itervar` en el bucle, controla el bucle y hace que el cuerpo se ejecute una vez para cada uno de los valores de la lista.

En el cuerpo del bucle, agregamos 1 al valor actual de `count` para cada uno de los valores en la lista. Mientras el bucle se está ejecutando, el valor de `count` es el número de valores que hemos visto "hasta ahora".

Una vez que el bucle se completa, el valor de `count` es el número total de elementos. El número total "cae en nuestro regazo" al final del bucle. Construimos el bucle para que tengamos lo deseado cuando el bucle finalice.

Otro bucle similar que calcula el total de un conjunto de números es el siguiente:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Total: ', total)
```

En este bucle nosotros **sí** usamos la **variable de iteración**. En lugar de simplemente agregar uno al `recuento` como en el bucle anterior, agregamos el número real (3, 41, 12, etc.) al total acumulado durante cada iteración de bucle. Antes de que el bucle comience, `total` es cero porque aún no hemos visto ningún valor. Durante el bucle, `total` es el total acumulado, y al final del bucle `total` es el total general de todos los valores en el lista.

A medida que se ejecuta el bucle, `total` acumula la suma de los elementos; una variable utilizada de esta manera a veces se llama un **acumulador**.

Ni el bucle de conteo ni el bucle de suma son particularmente útiles en la práctica porque hay funciones incorporadas `len()` y `sum()` que computan el número de elementos en una lista y el total de los elementos en la lista respectivamente .

Bucles máximos y mínimos

Para encontrar el valor más grande en una lista o secuencia, construimos el siguiente bucle:

```
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
```

```
    if largest is None or itervar > largest :
        largest = itervar
    print('Loop:', itervar, largest)
print('Largest:', largest)
```

Cuando el programa se ejecuta, la salida es la siguiente:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

La variable `largest` se comprende mejor como el "valor más grande que hemos visto hasta ahora". Antes del bucle, asignamos a `largest` la constante `None`. `None` es un valor constante especial que podemos almacenar en una variable para marcar la variable como "vacía".

Antes de que comience el bucle, el valor más grande que hemos visto hasta ahora es 'Ninguno', ya que todavía no hemos visto ningún valor. Mientras el bucle se está ejecutando, si `largest` es `None` entonces tomamos el primer valor que vemos como el más grande hasta ahora.

Después de la primera iteración, `largest` ya no es `None`, así que la segunda parte de la expresión lógica compuesta que comprueba, `itervar > largest`, se dispara solo cuando vemos un valor que es más grande que el "más grande hasta ahora". Cuando vemos un nuevo valor "aún más grande", tomamos ese nuevo valor para "más grande". Puede ver en la salida del programa que "más grande" progresa de 3 a 41 a 74.

Al final del ciclo, hemos escaneado todos los valores y la variable `largest` ahora contiene el valor más grande en la lista.

Para calcular el número más pequeño, el código es muy similar con un pequeño cambio:

```
smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
```

```
        smallest = itervar
    print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

Nuevamente, 'smallest' es el "más pequeño hasta ahora" antes, durante y después de que se ejecute el bucle. Cuando el bucle se ha completado, `smallest` contiene el valor mínimo en la lista.

De nuevo, al contar y sumar, las funciones incorporadas `max()` y `min()` hacen que la escritura de estos bucles exactos sea innecesaria.

La siguiente es una versión simple de la función `min()` incorporada en Python:

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

En la versión de la función anterior, eliminamos todas las declaraciones `print` para que sean equivalentes a la función `min` que ya está incorporada en Python.

Depurando

A medida que comienzas a escribir programas más grandes, puedes pasar más tiempo depurando. Más código significa más posibilidades de cometer un error y más lugares para que los errores se oculten.

Una forma de reducir el tiempo de depuración es "depuración por bisección". Por ejemplo, si hay 100 líneas en tu programa y las verificas una cada vez, te tomaría 100 pasos.

En su lugar, intenta romper el problema por la mitad. Busca en la mitad del programa, o cerca de él, un valor intermedio que puedas verificar. Agrega una declaración `print` (o algo que puedas verificar) y ejecuta el programa.

Si la verificación del punto medio es incorrecta, el problema debe estar en la primera mitad del programa. Si es correcto, el problema está en la segunda mitad.

Cada vez que realizas una verificación como esta, reduces a la mitad el número de líneas en las que tiene que buscar. Después de seis pasos (que es mucho menos que 100), se reduciría a una o dos líneas de código, al menos en teoría.

En la práctica, no siempre está claro cuál es la "mitad del programa" y no siempre es posible verificarlo. No tiene sentido contar líneas y encontrar el punto medio exacto. En su lugar, piensa en los lugares del programa en los que podría haber errores y en los lugares donde es fácil poner un chequeo. Luego, elije un lugar donde pienses que las probabilidades son casi las mismas de que el error esté antes o después de la comprobación.

Ejercicios

Ejercicio 1 escribe un programa que lea números repetidamente hasta que el usuario ingrese "listo". Una vez que se ingrese "listo", imprima el total, el recuento y el promedio de los números. Si el usuario ingresa algo que no sea un número, detecta su error usando `try` y `except` e imprime un mensaje de error y salta al siguiente número.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.333333333333333
```

Ejercicio 2 escribe otro programa que solicite una lista de números como arriba y al final imprima el máximo y el mínimo de los números en lugar del promedio.

> [1]. Examinaremos las listas con más detalle en un capítulo posterior. ↩

Revision #1
Created 5 April 2025 09:28:06 by Javier Quintana
Updated 5 April 2025 09:28:27 by Javier Quintana