Chapter 5

Iteración

5.1 Actualización de variables

Uno de los usos habituales de las sentencias de asignación consiste en realizar una actualización sobre una variable – en la cual el valor nuevo de esa variable depende del antiguo.

```
x = x + 1
```

Esto quiere decir "'toma el valor actual de `x`, añádele 1, y luego actualiza `x` con el nuevo valor".

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

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

Antes de que puedas actualizar una variable, debes *inicializarla*, normalmente mediante una simple asignación:

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

Actualizar una variable añadiéndole 1 se denomina *incrementar*; restarle 1 recibe el nombre de *decrementar* (o disminuir).

5.2 La sentencia `while`

Los PCs se suelen utilizar a menudo para automatizar tareas repetitivas. Repetir tareas idénticas o muy similares sin cometer errores es algo que a las máquinas se les da bien y en cambio a las personas no. Como las iteraciones resultan tan

habituales, Python proporciona varias características en su lenguaje para hacerlas más sencillas.

Una forma de iteración en Python es la sentencia `while`. He aquí un programa sencillo que cuenta hacia atrás desde cinco y luego dice "¡Despegue!".

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

Casi se puede leer la sentencia `while` como si estuviera escrita en inglés. Significa, "Mientras `n` sea mayor que 0, muestra el valor de `n` y luego reduce el valor de `n` en 1 unidad. Cuando llegues a 0, sal de la sentencia `while` y muestra la palabra `¡Despegue!`"

Éste es el flujo de ejecución de la sentencia `while`, explicado de un modo más formal:

1. Se evalúa la condición, obteniendo `Verdadero` or `Falso`.
2. Si la condición es falsa, se sale de la sentencia `while` y se continúa la ejecución en la siguiente sentencia.
3. Si la condición es verdadera, se ejecuta el cuerpo del `while` y luego se vuelve al paso 1.

Este tipo de flujo recibe el nombre de *bucle*, ya que el tercer paso enlaza de nuevo con el primero. Cada vez que se ejecuta el cuerpo del bucle se dice que realizamos una *iteración*. Para el bucle anterior, podríamos decir que "ha tenido cinco iteraciones", lo que significa que el cuerpo del bucle se ha ejecutado cinco veces.

El cuerpo del bucle debe cambiar el valor de una o más variables, de modo que la condición pueda en algún momento evaluarse como falsa y el bucle termine. La variable que cambia cada vez que el bucle se ejecuta y controla cuándo termina éste, recibe el nombre de *variable de iteración*. Si no hay variable de iteración, el bucle se repetirá para siempre, resultando así un *bucle infinito*.

5.3 Bucles infinitos

Una fuente de diversión sin fin para los programadores es la constatación de que las instrucciones del champú: "Enjabone, aclare, repita", son un bucle infinito, ya que no hay una *variable de iteración* que diga cuántas veces debe ejecutarse el proceso.

En el caso de una `cuenta atrás`, podemos verificar que el bucle termina, ya que sabemos que el valor de `n` es finito, y podemos ver que ese valor se va haciendo más pequeño cada vez que se repite el bucle, de modo que en algún momento llegará a 0. Otras veces un bucle es obviamente infinito, porque no tiene ninguna variable de iteración.

5.4 "Bucles infinitos" y break

A veces no se sabe si hay que terminar un bucle hasta que se ha recorrido la mitad del cuerpo del mismo. En ese caso se puede crear un bucle infinito a propósito y usar la sentencia `break` para salir fuera de él cuando se desee.

El bucle siguiente es, obviamente, un *bucle infinito*, porque la expresión lógica de la sentencia `while` es simplemente la constante lógica `True (verdadero)`;

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

Si cometes el error de ejecutar este código, aprenderás rápidamente cómo detener un proceso de Python bloqueado en el sistema, o tendrás que localizar dónde se encuentra el botón de apagado de tu equipo. Este programa funcionará para siempre, o hasta que la batería del equipo se termine, ya que la expresión lógica al principio del bucle es siempre cierta, en virtud del hecho de que esa expresión es precisamente el valor constante `True`.

A pesar de que en este caso se trata de un bucle infinito inútil, se puede usar ese diseño para construir bucles útiles, siempre que se tenga la precaución de añadir código en el cuerpo del bucle para salir explícitamente, usando `break` cuando se haya alcanzado la condición de salida.

Por ejemplo, supón que quieres recoger entradas de texto del usuario hasta que éste escriba `fin`. Podrías escribir:

```
while True:
    linea = input('> ')
    if linea == 'fin':
        break
    print(linea)
print('¡Terminado!')

# Código: https://es.py4e.com/code3/copytildone1.py
```

La condición del bucle es `True`, lo cual es verdadero siempre, así que el bucle se repetirá hasta que se ejecute la sentencia break.

Cada vez que se entre en el bucle, se pedirá una entrada al usuario. Si el usuario escribe `fin`, la sentencia `break` hará que se salga del bucle. En cualquier otro caso, el programa repetirá cualquier cosa que el usuario escriba y volverá al principio del bucle. Éste es un ejemplo de su funcionamiento:

```
> hola a todos
hola a todos
> he terminado
he terminado
> fin
¡Terminado!
```

Este modo de escribir bucles `while` es habitual, ya que así se puede comprobar la condición en cualquier punto del bucle (no sólo al principio), y se puede expresar la condición de parada afirmativamente ("detente cuando ocurra..."), en vez de tener que hacerlo con lógica negativa ("sigue haciéndolo hasta que ocurra...").

5.5 Finalizar iteraciones con `continue`

Algunas veces, estando dentro de un bucle se necesita terminar con la iteración actual y saltar a la siguiente de forma inmediata. En ese caso se puede utilizar la sentencia `continue` para pasar a la siguiente iteración sin terminar la ejecución del cuerpo del bucle para la actual.

A continuación se muestra un ejemplo de un bucle que repite lo que recibe como entrada hasta que el usuario escribe "fin", pero trata las líneas que empiezan por el carácter almohadilla como líneas que no deben mostrarse en pantalla (algo parecido a lo que hace Python con los comentarios).

```
while True:
    linea = input('> ')
    if linea[0] == '#' :
        continue
    if linea == 'fin':
        break
    print(linea)
print('¡Terminado!')

# Código: https://es.py4e.com/code3/copytildone2.py
```

He aquí una ejecución de ejemplo de ese nuevo programa con la sentencia `continue` añadida.

```
> hola a todos
hola a todos
> # no imprimas esto
> ¡imprime esto!
¡imprime esto!
> fin
¡Terminado!
```

Todas las líneas se imprimen en pantalla, excepto la que comienza con el símbolo de almohadilla, ya que en ese caso se ejecuta `continue`, finaliza la iteración actual y salta de vuelta a la sentencia `while` para comenzar la siguiente iteración, de modo que que se omite la sentencia `print`.

5.6 Bucles definidos usando `for`

A veces se desea repetir un bucle a través de un *conjunto* de cosas, como una lista de palabras, las líneas de un archivo, o una lista de números. Cuando se

tiene una lista de cosas para recorrer, se puede construir un bucle *definido* usando una sentencia `for`. A la sentencia `while` se la llama un bucle *indefinido*, porque simplemente se repite hasta que cierta condición se hace `Falsa`, mientras que el bucle `for` se repite a través de un conjunto conocido de elementos, de modo que ejecuta tantas iteraciones como elementos hay en el conjunto.

La sintaxis de un bucle `for` es similar a la del bucle `while`, en ella hay una sentencia `for` y un cuerpo que se repite:

```
amigos = ['Joseph', 'Glenn', 'Sally']
for amigo in amigos:
    print('Feliz año nuevo:', amigo)
print('¡Terminado!')
```

En términos de Python, la variable `amigos` es una lista[1] de tres cadenas y el bucle `for` se mueve recorriendo la lista y ejecuta su cuerpo una vez para cada una de las tres cadenas en la lista, produciendo esta salida:

```
Feliz año nuevo: Joseph
Feliz año nuevo: Glenn
Feliz año nuevo: Sally
¡Terminado!
```

La traducción de este bucle `for` al español no es tan directa como en el caso del `while`, pero si piensas en los amigos como un *conjunto*, sería algo así como: "Ejecuta las sentencias en el cuerpo del bucle una vez *para (for)* cada amigo que esté *en (in)* el conjunto llamado amigos."

Revisando el bucle `for`, *for* e *in* son palabras reservadas de Python, mientras que `amigo` y `amigos` son variables.

```
for amigo in amigos:
    print('Feliz año nuevo::', amigo)
```

En concreto, `amigo` es la *variable de iteración* para el bucle for. La variable `amigo` cambia para cada iteración del bucle y controla cuándo se termina el bucle `for`. La *variable de iteracion* se desplaza sucesivamente a través de las tres cadenas almacenadas en la variable `amigos`.

5.7 Diseños de bucles

A menudo se usa un bucle `for` o `while` para movernos a través de una lista de elementos o el contenido de un archivo y se busca algo, como el valor más grande o el más pequeño de los datos que estamos revisando.

Los bucles generalmente se construyen así:

- Se inicializan una o más variables antes de que el bucle comience

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

- Se realiza alguna operación con cada elemento en el cuerpo del bucle, posiblemente cambiando las variables dentro de ese cuerpo.
- Se revisan las variables resultantes cuando el bucle se completa

Usaremos ahora una lista de números para demostrar los conceptos y construcción de estos diseños de bucles.

5.7.1 Bucles de recuento y suma

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

```
contador = 0
for valor in [3, 41, 12, 9, 74, 15]:
    contador = contador + 1
print('Num. elementos: ', contador)
```

Ajustamos la variable `contador` a cero antes de que el bucle comience, después escribimos un bucle `for` para movernos a través de la lista de números. Nuestra variable de *iteración* se llama `valor`, y dado que no usamos `valor` dentro del bucle, lo único que hace es controlar el bucle y hacer que el cuerpo del mismo sea ejecutado una vez para cada uno de los valores de la lista.

En el cuerpo del bucle, añadimos 1 al valor actual de `contador` para cada uno de los valores de la lista. Mientras el bucle se está ejecutando, el valor de `contador` es la cantidad de valores que se hayan visto "hasta ese momento".

Una vez el bucle se completa, el valor de `contador` es el número total de elementos. El número total "cae en nuestro poder" al final del bucle. Se construye el bucle de modo que obtengamos lo que queremos cuando éste termina.

Otro bucle similar, que calcula el total de un conjunto de números, se muestra a continuación:

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

En este bucle, *sí* utilizamos la *variable de iteración*. En vez de añadir simplemente uno a `contador` como en el bucle previo, ahora durante cada iteración del bucle añadimos el número actual (3, 41, 12, etc.) al total en ese momento. Si piensas en la variable `total`, ésta contiene la "suma parcial de valores hasta ese momento". Así que antes de que el bucle comience, `total` es cero, porque aún no se ha examinado ningún valor. Durante el bucle, `total` es la suma parcial, y al final del bucle, `total` es la suma total definitiva de todos los valores de la lista.

Cuando el bucle se ejecuta, `total` acumula la suma de los elementos; una variable que se usa de este modo recibe a veces el nombre de *acumulador*.

Ni el bucle que cuenta los elementos ni el que los suma resultan particularmente útiles en la práctica, dado que existen las funciones internas `len()` y `sum()` que cuentan el número de elementos de una lista y el total de elementos en la misma respectivamente.

5.7.2 Bucles de máximos y mínimos

Para encontrar el valor mayor de una lista o secuencia, construimos el bucle siguiente:

```
mayor = None
print('Antes:', mayor)
for valor in [3, 41, 12, 9, 74, 15]:
    if mayor is None or valor > mayor :
        mayor = valor
    print('Bucle:', valor, mayor)
print('Mayor:', mayor)
```

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

```
Antes: None
Bucle: 3 3
Bucle: 41 41
Bucle: 12 41
Bucle: 9 41
Bucle: 74 74
Bucle: 15 74
Mayor: 74
```

Debemos pensar en la variable `mayor` como el "mayor valor visto hasta ese momento". Antes del bucle, asignamos a `mayor` el valor `None`. `None` es un valor constante especial que se puede almacenar en una variable para indicar que la variable está "vacía".

Antes de que el bucle comience, el mayor valor visto hasta entonces es `None`, dado que no se ha visto aún ningún valor. Durante la ejecución del bucle, si `mayor` es `None`, entonces tomamos el primer valor que tenemos como el mayor hasta entonces. Se puede ver en la primera iteración, cuando el valor de `valor` es 3, mientras que `mayor` es `None`, inmediatamente hacemos que `mayor` pase a ser 3.

Tras la primera iteración, `mayor` ya no es `None`, así que la segunda parte de la expresión lógica compuesta que comprueba si `valor > mayor` se activará sólo cuando encontremos un valor que sea mayor que el "mayor hasta ese momento". Cuando encontramos un nuevo valor "mayor aún", tomamos ese nuevo valor para `mayor`. Se puede ver en la salida del programa que `mayor` pasa desde 3 a 41 y luego a 74.

Al final del bucle, se habrán revisado todos los valores y la variable `mayor` contendrá entonces el mayor valor de la lista.

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

```
print('Antes:', menor)
for valor in [3, 41, 12, 9, 74, 15]:
    if menor is None or valor < menor:
        menor = valor
    print('Bucle:', valor, menor)
print('Menor:', menor)
```

De nuevo, `menor` es el "menor hasta ese momento" antes, durante y después de que el bucle se ejecute. Cuando el bucle se ha completado, `menor` contendrá el valor mínimo de la lista

También como en el caso del número de elementos y de la suma, las funciones internas `max()` y `min()` convierten la escritura de este tipo de bucles en innecesaria.

Lo siguiente es una versión simple de la función interna de Python `min()`:

```
def min(valores):
    menor = None
    for valor in valores:
        if menor is None or valor < menor:
            menor = valor
    return menor
```

En esta versión de la función para calcular el mínimo, hemos eliminado las sentencias `print`, de modo que sea equivalente a la función `min`, que ya está incorporada dentro de Python.

5.8 Depuración

A medida que vayas escribiendo programas más grandes, puede que notes que vas necesitando emplear cada vez más tiempo en depurarlos. Más código significa más oportunidades de cometer un error y más lugares donde los bugs pueden esconderse.

Un método para acortar el tiempo de depuración es "depurar por bisección". Por ejemplo, si hay 100 líneas en tu programa y las compruebas de una en una, te llevará 100 pasos.

En lugar de eso, intenta partir el problema por la mitad. Busca en medio del programa, o cerca de ahí, un valor intermedio que puedas comprobar. Añade una sentencia `print` (o alguna otra cosa que tenga un efecto verificable), y haz funcionar el programa.

Si en el punto medio la verificación es incorrecta, el problema debería estar en la primera mitad del programa. Si ésta es correcta, el problema estará en la segunda mitad.

Cada vez que realices una comprobación como esta, reduces a la mitad el número de líneas en las que buscar. Después de seis pasos (que son muchos menos de 100), lo habrás reducido a una o dos líneas de código, al menos en teoría.

En la práctica no siempre está claro qué es "en medio del programa", y no siempre es posible colocar ahí una verificación. No tiene sentido contar las líneas y encontrar

el punto medio exacto. En lugar de eso, piensa en lugares del programa en los cuales pueda haber errores y en lugares donde resulte fácil colocar una comprobación. Luego elige un sitio donde estimes que las oportunidades de que el bug esté por delante y las de que esté por detrás de esa comprobación son más o menos las mismas.

5.9 Glosario

acumulador Una variable usada en un bucle para sumar o acumular un resultado.

bucle infinito Un bucle en el cual la condición de terminación no se satisface nunca o para el cual no existe dicha condición de terminación.

contador Una variable usada en un bucle para contar el número de veces que algo sucede. Inicializamos el contador a cero y luego lo vamos incrementando cada vez que queramos que "cuente" algo.

decremento Una actualización que disminuye el valor de una variable.

inicializar Una asignación que da un valor inicial a una variable que va a ser después actualizada.

incremento Una actualización que aumenta el valor de una variable (a menudo en una unidad).

iteración Ejecución repetida de una serie de sentencias usando bien una función que se llama a si misma o bien un bucle.

5.10 Ejercicios

Ejercicio 1: Escribe un programa que lea repetidamente números hasta que el usuario introduzca "fin". Una vez se haya introducido "fin", muestra por pantalla el total, la cantidad de números y la media de esos números. Si el usuario introduce cualquier otra cosa que no sea un número, detecta su fallo usando `try` y `except`, muestra un mensaje de error y pasa al número siguiente.

```
Introduzca un número: 4
Introduzca un número: 5
Introduzca un número: dato erróneo
Entrada inválida
Introduzca un número: 7
Introduzca un número: fin
16 3 5.33333333333
```

Ejercicio 2: Escribe otro programa que pida una lista de números como la anterior y al final muestre por pantalla el máximo y mínimo de los números, en vez de la media.