04. Estructuras Repetitivas

Índice

1. Fundamentos de las estructuras repetitivas

En programación, un ciclo es una instrucción compuesta que permite la repetición controlada de la ejecución de cierto conjunto de instrucciones en un programa, determinando si la repetición debe detenerse o continuar de acuerdo al valor de cierta condición que se incluye en el ciclo, conociéndose también como estructuras repetitivas, o bien, como instrucciones repetitivas.

Un ciclo consta de dos partes:

- El bloque de acciones, conocido como cuerpo del ciclo, siendo este el conjunto de instrucciones cuya ejecución se pide repetir.
- La cabecera del ciclo, la cual incluye una condición y|o elementos adicionales en base a los que se determina si el ciclo continúa o se detiene.

a. Expresiones de conteo y acumulación

En la mayoría de los programas que requieren repetición de acciones será necesario eventualmente llevar a cabo procesos de *conteo* (se resuelve añadiendo una variable de conteo o contador), o *acumulación* (se resuelve incorporando una variable de acumulación o acumulador).

En ambos casos, se trata de variables que en una expresión de asignación aparecen en ambos miembros: la misma variable se usa para hacer un cálculo y para recibir la asignación del resultado de este cálculo.

Un ***contador*** es una variable que sirve para contar ciertos eventos que ocurren durante la ejecución de un programa. Se trata de una variable la cual se va incrementando el valor en 1 ***(a = a + 1)*** cada vez que se ejecuta la expresión. Por lo tanto, un contador es una variable que actualiza su valor en términos de su propio valor anterior y de una constante.

Del mismo modo que en la expresión ***(a = a + 1)*** la variable a funciona como contador, puede funcionar de forma similar como decrementador en la expresión ***(a = a – 1)***, restando de a uno a partir del valor original de la variable ***a***.

Un ***acumulador*** es una variable que permite sumar los valores que va asumiendo otra variable o bien otra expresión en un proceso cualquiera. Un acumulador es una variable que básicamente, actualiza su valor en términos de su propio valor anterior y el valor de otra variable u otra expresión, siendo ***(a = a + x)***.

Del mismo modo que en la expresión ***(a = a + x)*** la variable ***a*** funciona como acumulador de ***a + x***, puede funcionar de forma similar como decrementador en la expresión ***(a = a – x)***, restando el valor de la variable o expresión x al valor anterior de la variable ***a***.

En Python, cualquier expresión de conteo o de acumulación responde a la forma general siguiente: ***variable = variable operador expresión***

Cualquier expresión que venga escrita en la forma general anterior, se puede escribir en la forma resumida siguiente:

Forma general	Forma resumida
a = a + 1	a += 1
a = a – 4	a -= 4
a = a + 7	a += 7
a = a * 3	a *= 3
a = a /23	a /= 23
a = a + x	a += x
a = a * x	a *= x
a = a – x	a -= x
a = a / x	a /= x
a = a + 2 * x	a += 2*x

Tabla - Forma resumida expresiones

2. El ciclo While

En muchas ocasiones se necesita plantear un ciclo que ejecute en forma repetida un bloque de acciones, pero sin conocer previamente la cantidad de vueltas a realizar. En Python, se provee del ciclo ***While*** para este tipo de casos, aunque puede ser aplicado sin ningún problema en situaciones en las que la cantidad de repeticiones a realizar es conocida.

Un ciclo ***While*** está formado por una cabecera y un bloque o cuerpo de acciones. La cabecera del ciclo contiene una expresión lógica que es evaluada en la misma forma en que lo hace una instrucción condicional if, pero con la diferencia de que el ciclo While ejecuta su bloque de acciones en forma repetida siempre que la expresión lógica arroje un valor verdadero.

El ciclo ***While*** realiza múltiples evaluaciones: cada vez que termina de ejecutar el bloque de acciones, vuelve a evaluar la expresión lógica, y si nuevamente se obtiene un valor verdadero, repite la ejecución del bloque, y así, hasta que se obtenga un falso en la evaluación de la condición.

Entonces, el ciclo While responde a la siguiente estructura:

```
while expresión_lógica:
        Bloque de acciones
```

Se puede utilizar el ciclo While para resolver problemas donde si se conoce de antemano la cantidad de iteraciones a realizar. Para ello, en el bloque de acciones se debería incluir un contador que sea utilizado en la evaluación de la expresión lógica para controlar el número de repeticiones que se lleva. Un ejemplo sería:

```
while i <= 10 :
        print(i)
        i += 1
```

3. El ciclo For

El ciclo ***For*** suele ser el más práctico cuando se conoce previamente la cantidad de repeticiones a realizar. En Python, está formulado para hacer mucho más que simplemente ejecutar un bloque de acciones.

Es un ciclo especialmente diseñado para recorrer secuencias como tuplas, cadenas de caracteres, rangos, listas, etc. Recuperando de a un elemento por vez, es decir, un elemento por cada vuelta que da en el ciclo o bucle.

Por ejemplo, si se define una tupla con 3 nombres, es posible plantear un ciclo for de manera que use una variable para recuperar de a un nombre por vez en cada vuelta:

```
nombres = ('Sergio', 'Pedro', 'Antonio')
for nom in nombres :
        print(nom)
```

En cada repetición, el propio ciclo se encarga de colocar en la variable nom un elemento de la tupla, en el orden en que han sido alojados en ella, y finaliza cuando ya no tiene elementos que tomar de la tupla.

La primera línea de la instrucción for se conoce como la cabecera del ciclo. Incluye la definición de una variable que será usada para ir almacenando uno a uno los valores que se recuperen automáticamente desde la estructura de datos que se pretende recorrer.

Cuando se tiene una estructura de datos cualquiera, la operación de recorrer esa estructura y procesar de a uno sus elementos se designa en forma general como iterar la estructura. La variable que se use para ir tomando uno a uno los valores de la estructura a medida que se itera sobre ella, se conoce como variable iteradora.

El ciclo for en Python esta esencialmente previsto para la iteración de estructuras de datos. Un ciclo for iterando sobre todos los caracteres de una cadena contenida en la variable usa una variable iteradora para almacenar una copia de los caracteres de la cadena a razón de uno por vuelta del ciclo en el orden en que aparecen.

Se puede invocar a la función ***range(a,b)*** con un solo parámetro, asumiendo que ese es el valor final, sin incluirlo (***n-1***), de la secuencia a generar, y toma como valor inicial el cero.

La función ***range()*** acepta un tercer valor de ser necesario, el cual indica el incremento a usar para pasar de un valor a otro en la secuencia generada, pudiendo ser tanto positivo como negativo. Por defecto, el incremento es 1. En ese caso, la función ***range()*** sería ***range(inicio, final, incremento)***, como, por ejemplo:

```
for i in range(1, 5, 1) :
        print(i)
```

Un range r creado con la instrucción (***r = range(start, stop, incremento)***) representa una secuencia numérica inmutable que contiene sólo a los números dados por las siguientes dos expresiones:

- Si incremento > 0, entonces, el contenido de cada elemento r[i] se determina como i >= 0 y r[i] < stop.
- Si incremento < 0, entonces, el contenido de cada elemento r[i] se determina como i >= 0 y r[i] > stop.

4. Estructuras repetitivas variantes

a. Elementos de control adicionales

Un bloque de acciones de un ciclo puede incluir una instrucción ***break*** para cortar el ciclo de inmediato sin retornar a la cabecera para evaluar la expresión lógica de control.

El bloque de acciones del ciclo while carga por teclado y chequea con un if (por ejemplo) si ese número es cero o negativo, en caso de serlo, se ejecutaría la instrucción ***break***, si sólo pudiesen introducir números positivos, y su efecto será cancelativo con respecto al ciclo: el ciclo while se interrumpiría y el programa continuaría su ejecución.

Del mismo modo, pero de forma inversa, un ciclo puede incluir una instrucción ***continue***, la cual fuerce una repetición del ciclo sin terminar de ejecutar las instrucciones que queden por debajo de su invocación.

Al ejecutarse la instrucción ***continue***, el ciclo no cortará su ejecución, si no que volverá a la cabecera para volver a chequear la expresión lógica de control, pero no ejecutará en ese caso las instrucciones del bloque.

En ocasiones, el programador necesita dejar vacío el bloque de acciones de un ciclo o una rama de una condición, para estos casos, Python hace uso de la instrucción ***pass***.

La instrucción ***pass*** sirve para indicar al intérprete que simplemente, considere vacío el bloque que la contiene. También puede usarse para dejar vacía una rama de una condición.

A diferencia de otros lenguajes, en Python, un ciclo puede llevar opcionalmente una cláusula ***else*** en forma similar a una instrucción condicional, aunque el efecto de este else en un ciclo es bastante diferente a lo que ocurre con una condición común:

- En un ciclo ***while***, las instrucciones de la rama ***else*** se ejecutan en el momento en que la expresión de control del ciclo se evalúa en ***False***.
- En un ciclo ***for***, las instrucciones de la rama ***else*** se ejecutan cuando el for termina de iterar sobre todos los elementos de la colección o secuencia dada.
- En ambos casos, la rama ***else*** no será ejecutada si el ciclo terminó por acción de una instrucción ***break***.

5. Tratamiento de números complejos en Python

En álgebra y análisis matemático, el campo de los números complejos, permite trabajar con raíces de orden par de números negativos, introduciendo un elemento llamado unidad imaginaria, cuyo valor se asume como ***j*** o raíz cuadrada de (-1).

En muchos lenguajes de programación, se producirá un error en tiempo de ejecución si se intenta calcular la raíz cuadrada de un número negativo. En Python, el resultado será efectivamente un número complejo, pues Python provee un tipo especial de dato numérico llamado *complex*, que permite representar números complejos de la forma (***a + bj***). El factor *j* representa la unidad imaginaria y es automáticamente gestionado por Python.

Si un programador necesita asignar directamente un número complejo en una variable, recurrirá a la clase complex.

La función ***complex(a,b)*** toma dos parámetros, el primero, será la parte real del complejo a crear, el segundo, será la parte imaginaria. De esta forma, la función complex como ejemplo sería (***c = complex(2.25, 5)***), siendo 2.25 la parte real, y el 5, la parte imaginaria.

Un ejemplo de uso de números complejos es la resolución de ecuaciones de segundo grado.

6. Medición del tiempo de ejecución de un proceso en Python

Muchos programas tendrán a demorar cada vez más su tiempo de ejecución a medida que el tamaño del conjunto de datos crezca. Medir directamente el tiempo de ejecución de un proceso en Python es necesario en esos casos.

Todo lenguaje provee funciones en sus librerías estándar que, de una forma u otra, permiten medir el tiempo de demora en ejecutarse un proceso. Estas funciones, están provistas en el módulo time de las librerías Python.

Para tener acceso a dichas funciones, se deberá importar este módulo con la siguiente instrucción: ***import time***

La función ***time.clock()*** retorna un número en coma flotante que indica la cantidad de segundos transcurridos desde cierto momento arbitrariamente seleccionado (momento de origen), hasta el momento en que se invocó la función.

La función ***time.gmtime(t)*** retorna una colección de valores que representan la fecha dada por la cantidad de segundos t tomada como parámetro. Si t = 0, entonces la fecha pedida coincide con el momento ***epoch*** (momento de origen del tiempo inicial).

7. Ejercicios propuestos

Ejercicio 1: Crear el fichero cnt1.py en el cual pida introducir 'S' o 's', contar el número de veces que se introduce hasta que se introduzca un valor 'N' o 'n'.

Ejercicio2: Crear un fichero llamado acum.py en el cual se pide ir introduciendo números por teclado mientras se pregunta si se quiere introducir más números, en caso de que la respuesta sea 'N' o 'n', se cortará la ejecución y se mostrará el número de valores introducidos por teclado y la suma de todos ellos.

Ejercicio 3: Crear un fichero llamado decr.py en el cual se pida introducir números hasta que se introduzcan 3 números negativos, cuando esto se produzca, mostrar el número de valores positivos introducidos, así como la suma de los valores positivos y negativos.

Ejercicio 4: Crear un fichero llamado factorial.py en el cual se calcule la factorial de un número introducido por teclado y muestre el resultado por pantalla.

Ejercicio 5: Crear un fichero llamado hipotenusa.py en el cual se pide el valor de los dos catetos del triángulo y muestre por pantalla el valor de la hipotenusa del mismo.

Ejercicio 6: Crear un fichero llamado ecuaciones.py en el cual se pida el valor de a, b y c de una ecuación de segundo grado y que muestre por pantalla el valor de x en ambos casos, real e imaginario.

Ejercicio 7: Crear el fichero negFor.py en el cual se introduzcan 3 números y muestre cuantos han sido negativos. Usar FOR.

Ejercicio 8: Crear el fichero contNeg.py en el cual se pida la cantidad de números a introducir por teclado y que cuente cuantos números negativos se han introducido en esa serie. Usar FOR.

Ejercicio 9: Crear un fichero numSecreto.py en el cual se genere un número entero aleatorio entre 1-10, el usuario, tendrá 3 intentos para acertar el número generado aleatoriamente, cada vez que falle, se le mostrará el rango en el que se comprende el número, es decir, si por ejemplo el número secreto es 7 y se ha introducido el 3, se mostrará el mensaje "Error! Le quedan X intentos, el número secreto está entre 3 y 10".

Ejercicio 10: Crear un fichero llamado tiempos.py en el cual se mida el tiempo de ejecución desde que se lanza hasta que se ingresa por teclado el valor, y que mida también el tiempo que tarda en realizar las operaciones factoriales de ese valor introducido por teclado.

Ejercicio 11: Crear un fichero llamado piePapTij.py en el cual se juegue contra la máquina a piedra, papel o tijera hasta que uno de los 2 llegue a 3 rondas ganadas. EL usuario usará 1 para piedra, 2 para papel, 3 para tijeras.

Soluciones a los ejercicios propuestos
→*(https://mega.nz/#!Z35giCAQ!C1bqHVEZccYTK9CQWguAI9SjOWSrGPrRIhnqMsPaPzQ)*