

Repeticiones en Python

Supongamos que vamos a la casa de un amigo pero se nos olvidó traer la dirección completa y sólo tenemos el nombre de la calle en la que vive. Al llegar nos damos cuenta que es un pasaje con 8 casas a cada lado ¿Qué podemos hacer para no perder el viaje? Asumiendo que los vecinos no se conocen entre sí, y que no tenemos forma de contactarlo previamente, lo mejor sería simplemente ir y golpear en la primera casa, y preguntar si nuestro amigo vive ahí. En caso de que la respuesta sea no, tendríamos que **repetir** el proceso con las siguientes casas, hasta encontrar la de nuestro amigo.

Si quisiéramos escribir un programa en Python para realizar este procedimiento, con las herramientas que hasta el momento hemos aprendido, tan sólo podríamos realizar algo parecido a lo mostrado en el ejemplo 1.

Ejemplo 1

Buscando a mi amigo

```
1. Ir a la primera casa
2. Golpear la puerta y esperar que alguien abra
3. Si es la casa de nuestro amigo entonces:
4.     Nos quedamos y dejamos de buscar
5. Sino:
6.     Ir a la segunda casa
7.     Golpear la puerta y esperar que alguien abra
8.     Si es la casa de nuestro amigo entonces:
9.         Nos quedamos y dejamos de buscar
10.    Sino:
11.        Ir a la tercera casa
12.        Golpear la puerta y esperar que alguien abra
13.        Si es la casa de nuestro amigo entonces:
14.            Nos quedamos y dejamos de buscar
15.        Sino:
16.            Ir a la cuarta casa
17.            :
75.                ... Ir a la décima sexta casa
76.                ... Ésta debe ser la casa de nuestro amigo
```

Como podemos ver, escribir el procedimiento del ejemplo 1 es bastante largo y se vuelve tedioso. Además, ésta **no es la forma natural** en que daríamos las instrucciones a una persona. Peor aún, este procedimiento **sólo es posible** porque sabemos que hay 16 casas en el pasaje, y **no podríamos hacerlo** sin saber *a priori* la cantidad de veces que debemos realizar la búsqueda.

Pregunta 1

Junto a tu grupo, responde ahora la pregunta 1 de la actividad.

En la respuesta a la pregunta 1 probablemente tuvimos que usar palabras como **repetir**, **mientras**, **hasta**, **para cada**, **durante**, **entretanto**, etc; pues queremos decirle al computador que repita las operaciones de búsqueda hasta que encontremos a nuestro amigo. Esta idea de **repetir un procedimiento** se conoce como **iteración**. Una iteración en Python no es más que la instrucción de **repetir un bloque de sentencias**.

Estructura de ciclos while

En Python, al igual que en la mayoría de los lenguajes hay **variados mecanismos** y **sentencias** para conseguir que un programa itere, sin embargo, hoy nos centraremos en aprender una en particular: la sentencia **while**, cuya traducción al español sería “**mientras**”. La sintaxis en Python para un ciclo while es la siguiente:

Importante

```
while <condición> :  
    <Bloque de sentencias a repetir>
```

Para comenzar a entender cómo funciona la sentencia **while**, miremos los programas `TablaMultSimple1.py` y `TablaMultSimple2.py`. Ambos programas entregan la **misma salida**:

```
>>> ===== RESTART =====  
>>>  
Ingresa un valor cuya tabla de multiplicar es requerida: 3  
1 * 3 = 3  
2 * 3 = 6  
3 * 3 = 9  
4 * 3 = 12  
5 * 3 = 15  
6 * 3 = 18  
7 * 3 = 21  
8 * 3 = 24  
9 * 3 = 27  
10 * 3 = 30  
11 * 3 = 33  
12 * 3 = 36  
>>>
```

TablaMultSimple1.py

```
1. #
2. # Bloque principal
3. #
4.
5. # Entrada de datos
6. factor = input("Ingresa un valor cuya tabla de multiplicar es requerida: ")
7.
8. # Salida
9. print "1 *", factor, "=", 1 * factor
10. print "2 *", factor, "=", 2 * factor
11. print "3 *", factor, "=", 3 * factor
12. print "4 *", factor, "=", 4 * factor
13. print "5 *", factor, "=", 5 * factor
14. print "6 *", factor, "=", 6 * factor
15. print "7 *", factor, "=", 7 * factor
16. print "8 *", factor, "=", 8 * factor
17. print "9 *", factor, "=", 9 * factor
18. print "10 *", factor, "=", 10 * factor
19. print "11 *", factor, "=", 11 * factor
20. print "12 *", factor, "=", 12 * factor
```

TablaMultSimple2.py

```
1. #
2. # Bloque principal
3. #
4.
5. # Entrada de datos
6. factor = input("Ingresa un valor cuya tabla de multiplicar es requerida: ")
7.
8. # Salida
9. otroFactor = 1
10. while otroFactor <= 12:
11.     print otroFactor, "*", factor, "=", otroFactor * factor
12.     otroFactor = otroFactor + 1
13.
```

Lo primero que debemos recordar, es que la función `input()` que inicia el bloque principal es la sentencia primera en ejecutarse, por lo que el programa queda **esperando** a que el usuario interactúe con él, es decir, solicita que el usuario entregue un **valor de entrada** al programa. La salida mostrada se obtiene cuando el usuario ingresa un valor 3.

Estos programas obviamente **muestran la tabla de multiplicar** del factor ingresado por el usuario. Pero el programa `TablaMultSimple1.py` usa doce sentencias `print` para calcular y mostrar la tabla, mientras que `TablaMultSimple2.py` utiliza solamente una. ¿Cómo es esto posible?

Lo que sucede es que esta diferencia es meramente **sintáctica**; **semánticamente**, ambos programas **ejecutan** doce sentencias `print`. En `TablaMultSimple1.py` las doce sentencias `print` aparecen explícitamente **en secuencia**, y en `TablaMultSimple2.py` aparece una sola sentencia explícita pero en el **cuerpo de un ciclo** que **se repite** doce veces. La ejecución del cuerpo del ciclo está **condicionada** al cumplimiento de la **condición** de la sentencia `while` (línea 10), y si observamos cómo cambia el valor de la variable `otroFactor` (línea 12), podremos darnos cuenta que el ciclo se ejecuta doce veces.

Entonces en Python (al igual que en todos los lenguajes de programación imperativos) el ciclo `while` tiene la siguiente semántica: Se define un bloque de código, que se conoce como el **cuerpo del ciclo**, que se ejecuta repetidamente **mientras** la **condición del ciclo** se evalúe con valor **verdadero**. Como consecuencia el cuerpo puede ejecutarse **cero o más veces**. La figura 1 muestra una representación gráfica de la semántica de ciclo `while`.



Figura 1: Diagrama de flujo de la ejecución de la estructura `while`

Si nos fijamos en la figura 1, podemos darnos cuenta que el flujo de ejecución **sale de un ciclo while únicamente** cuando la condición **no se cumple**; así, si la condición **nunca** deja de cumplirse, el ciclo quedará operando hasta que el usuario **detenga la ejecución** (`<ctrl>+c`), **cierre** el intérprete de Python o se produzca **el agotamiento** de algún recurso (memoria del computador, espacio para representación de un número, etc.) El programa `AlInfinitoYMasAlla.py` muestra un ejemplo de un ciclo que **nunca se detiene**. No lo ejecutemos, sólo analicémoslo y veamos por qué nunca se detiene.

AlInfinitoYMasAlla.py

```
i = 0
while i <= 10 :
    print "Este mensaje se imprimirá hasta el fin de los tiempos"
```

El programa `AlInfinitoYMasAlla.py` condiciona la escritura del mensaje al que el valor de la variable `i` sea menor que diez. Probablemente la idea era imprimir el mensaje diez veces. Pero el cuerpo del ciclo **nunca altera** el valor de la variable `i`. Luego la condición no deja de cumplirse y el ciclo se repite eternamente.

Para asegurar que un ciclo termina, es **necesaria una instrucción dentro de él** que **modifique** de alguna forma la condición a la que está sujeta la iteración, de manera que en algún momento, cuando corresponda, la condición se evalúe con **valor falso**. Si nos fijamos en la línea 12 del programa `TablaMultSimple2.py`, esta asignación va incrementando el valor de la variable `otroFactor`, que es la variable usada en la condición del ciclo, por lo que después de doce iteraciones, la iteración se rompe.

Importante

Crear ciclos infinitos es un error común cuando se está aprendiendo a programar. Estos ciclos usualmente ocurren porque no hay una sentencia dentro del cuerpo del ciclo que permita que la condición no se cumpla en algún momento, pero a veces se producen porque accidentalmente se **re-inicia la iteración** al utilizar, dentro del ciclo, la misma variable que se usa en la condición, o por utilizar casualmente una **tautología** como condición.

Pregunta 2 y 3

Trabaja ahora con tu grupo en responder las preguntas 2 y 3 de la actividad.

Otra utilidad del ciclo `while` está a la hora de crear menús, pues nos sirve para **volver al inicio** en caso de que el usuario seleccione una **opción inválida**. Esto se hace

agregando todo el menú dentro de un ciclo para controlar que el flujo de ejecución no salga del menú, hasta que se haya seleccionado una **opción válida**.

EjemploMenu.py

```
#
# Función que despliega en pantalla el menú y
# solicita al usuario elegir una opción.
# Entrada:
# Salida: opción ingresada por el usuario (valor entero);
#         se asegura que la opción elegida es válida
#
def eligeopcionMenu():
    opcion = 0
    while opcion != 1 and opcion != 2 and opcion != 3:
        print
        print "MENU"
        print "===="
        print
        print "Opción 1: Saludar"
        print "Opción 2: Despedir"
        print "Opción 3: Salir del programa"
        print

        opcion = input("Elija una opción: ")

    return int(opcion)
```

Es importante destacar que al igual que las estructuras de decisión, el cuerpo de un ciclo **while** puede ser tan complejo como se desee, y puede contener entrada de datos, salida de datos, expresiones con variables e invocación de funciones, estructuras de decisión e incluso **otros ciclos anidados** (ciclos dentro de ciclos). El programa `PresentaNumero.py` ejemplifica esta observación e introduce algunas ideas nuevas para dar formato a los mensajes de salida.

Pregunta 4

Responde con tu grupo la pregunta 4 de la actividad.

Trazas

La introducción de ciclos en nuestros programas hace **más difícil entenderlos**: ya no basta con una lectura de arriba abajo, sino que es necesario considerar **las veces que se**

repite algunos bloques de sentencias y **cómo se van alterando** el valor de las variables a medida que se itera.

PresentaNumero.py

```
1. #
2. # Función que presenta a un número indicando si es par o impar.
3. # Además muestra los pares/impares que le preceden respectivamente.
4. # Entrada: el número a presentar (n, entero)
5. # Salida: mensajes al usuario presentando el número y sus precededores
6. #
7. def presentaNumero(n):
8.     n = int(n)
9.     if n % 2 == 0:
10.         print "Soy", n, "y soy un número par"
11.         print "Mis predecesores pares son: "
12.     else:
13.         print "Soy", n, "y soy un número impar"
14.         print "Mis predecesores impares son: "
15.
16.     # Muestra los predecesores de n mayores a cero
17.     n = n - 2
18.     while n > 0:
19.         print "\t",
20.         i = 0
21.         # Muestra hasta 10 números por línea
22.         while n > 0 and i < 10:
23.             print n,
24.             n = n - 2
25.             i = i + 1
26.         print
27.
```

Para evitar confusiones, los programadores revisan un programa utilizando **trazas**, que es un **análisis de la ejecución** del programa con el fin de ir revisando, paso a paso, las sentencias que se están ejecutando hasta que sea posible **predecir** o **verificar la salida entregada**, o revisar dónde puede generarse un **error** al momento de ejecutar.

Una forma de representar una traza es mediante dos **tablas**: la primera representa lo que el programa mantiene en memoria, y la segunda sigue los resultados del programa línea a línea. Por ejemplo, las tablas 1 y 2 muestran la traza del programa `TablaMultSimple2.py` cuando el usuario ingresa el valor 5.

Pregunta 5

Trabaja con tu grupo en responder la pregunta 5 de la actividad.

Tabla 1: Estados en memoria durante la ejecución del programa TablaMultiSimple2.py

Memoria		
Contexto	Variable	Valores
Global	factor	5
Global	otroFactor	4 2 3 4 5 6 7 8 9 10 11 12 13

Tabla 2: Ejecución del programa TablaMultiSimple2.py cuando el usuario ingresa el valor 5

Programa	
Línea	Resultado
6	Crea variable factor con valor 5 dado por el usuario
9	Crea variable otroFactor con valor 1
10	¿1 <= 12? → True
11	Mensaje a pantalla: "1 * 5 = 5"
12	Incrementa el valor de otroFactor a 2
10	¿2 <= 12? → True
11	Mensaje a pantalla: "2 * 5 = 10"
12	Incrementa el valor de otroFactor a 3
10	¿3 <= 12? → True
11	Mensaje a pantalla: "3 * 5 = 15"
12	Incrementa el valor de otroFactor a 4
10	¿4 <= 12? → True
11	Mensaje a pantalla: "4 * 5 = 20"
12	Incrementa el valor de otroFactor a 5
10	¿5 <= 12? → True
11	Mensaje a pantalla: "5 * 5 = 25"
12	Incrementa el valor de otroFactor a 6
10	¿6 <= 12? → True
11	Mensaje a pantalla: "6 * 5 = 30"
12	Incrementa el valor de otroFactor a 7
10	¿7 <= 12? → True
11	Mensaje a pantalla: "7 * 5 = 35"
12	Incrementa el valor de otroFactor a 8
10	¿8 <= 12? → True
11	Mensaje a pantalla: "8 * 5 = 40"
12	Incrementa el valor de otroFactor a 9
10	¿9 <= 12? → True
11	Mensaje a pantalla: "9 * 5 = 45"
12	Incrementa el valor de otroFactor a 10
10	¿10 <= 12? → True
11	Mensaje a pantalla: "10 * 5 = 50"
12	Incrementa el valor de otroFactor a 11
10	¿11 <= 12? → True
11	Mensaje a pantalla: "11 * 5 = 55"
12	Incrementa el valor de otroFactor a 12
10	¿12 <= 12? → True
11	Mensaje a pantalla: "12 * 5 = 60"
12	Incrementa el valor de otroFactor a 13
10	¿13 <= 12? → False
13	Fin del programa