03 - Iteration

Bucles `while`

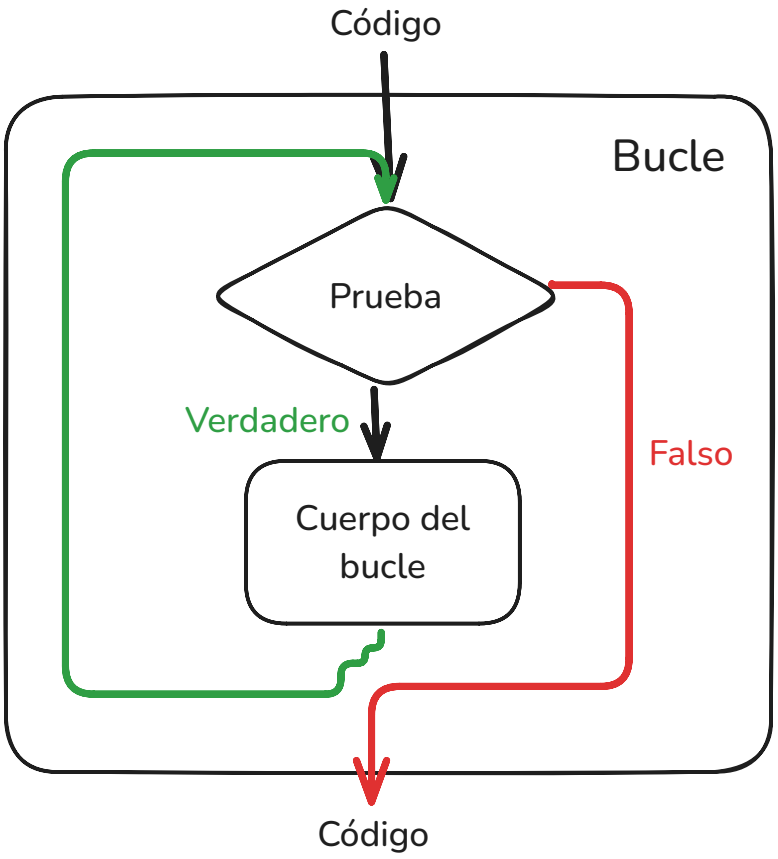
La mayoría de las tareas computacionales no pueden realizarse utilizando programas de bifurcación. Considere, por ejemplo, escribir un programa que pregunte imprimir la palabra `X` cierta cantidad de veces. Podría pensar en escribir algo como

```
num_x = int(input('How many times should I print the letter X? '))
to_print = ''
if num_x == 1:
    to_print = 'X'
elif num_x == 2:
    to_print = 'XX'
elif num_x == 3:
    to_print = 'XXX'
#…
print(to_print)
```

Pero rápidamente se haría evidente que necesitarías tantas condicionales como enteros positivos hay, y hay un número infinito de ellos. Lo que quieres escribir es un programa que se parezca a (lo siguiente es pseudocódigo, no Python)

```
num_x = int(input('How many times should I print the letter X? '))
to_print = ''
# concatenate X to to_print num_x times
print(to_print)
```

Cuando queremos que un programa haga lo mismo muchas veces, podemos utilizar la iteración.



Como una sentencia condicional, **comienza con una prueba**. Si la prueba se evalúa como `True`, el programa *ejecuta el cuerpo del bucle una vez, y luego vuelve a reevaluar la prueba*. Este proceso se repite hasta que el resultado de la prueba es `False`, tras lo cual el control pasa al código que sigue a la sentencia de iteración.

Podemos escribir el tipo de bucle representado en la figura utilizando una sentencia `while`.

```
x = 3
ans = 0
num_iterations = 0
while (num_iterations < x):
    ans = ans + x
    num_iterations = num_iterations + 1
print(f'{x}*{x} = {ans}')
```

El código comienza vinculando la variable `x` al número entero `3`. A continuación, procede a elevar `x` al cuadrado utilizando la suma repetitiva. Construimos la tabla simulando a mano el código, es decir, fingimos ser un intérprete de Python y ejecutamos el programa utilizando lápiz y papel. Utilizar lápiz y papel puede parecer pintoresco, pero es una forma excelente de entender cómo se comporta un programa.

Test #	x	ans	num_iterations
1	3	0	0
2	3	3	1
3	3	6	2
4	3	9	3

La cuarta vez que se alcanza la prueba, se evalúa como `False` y el flujo de control procede a la sentencia `print` que sigue al bucle. ¿Para qué valores de `x` terminará este programa? Hay tres casos a considerar: `x == 0`, `x > 0`, y `x < 0`.

Supongamos que `x == 0`. El valor inicial de `num_iterations` también será `0`, y el cuerpo del bucle nunca se ejecutará.

Supongamos que `x > 0`. El valor inicial de `num_iterations` será menor que `x`, y el cuerpo del bucle se ejecutará al menos una vez. Cada vez que se ejecuta el cuerpo del bucle, el valor de `num_iterations` se incrementa exactamente en 1. Esto significa que como `num_iterations` empezó siendo menor que `x`, después de un número finito de iteraciones del bucle, `num_iterations` será igual a `x`. En este punto la prueba del bucle se evalúa como `False`, y el control pasa al código que sigue a la sentencia `while`.

Supongamos que `x < 0`. Algo muy malo sucede. El control entrará en el bucle, y cada iteración moverá `num_iterations` más lejos de `x` en lugar de acercarse a ella. Por lo tanto, *el programa continuará ejecutando el bucle para siempre* (o hasta que ocurra algo malo, por ejemplo, un error de desbordamiento). ¿Cómo podríamos eliminar este fallo en el programa? Cambiar la prueba a `num_iteraciones < abs(x)` casi funciona. El bucle termina, pero imprime un valor negativo. Si la sentencia de asignación dentro del bucle también se cambia, a `ans = ans + abs(x)`, el código funciona correctamente.

Ejercicio manual

Reemplaza el comentario en el siguiente código con un bucle while.

```
num_x = int(input('How many times should I print the letter X? '))
to_print = ''
#concatenate X to to_print num_x times
print(to_print)
```

Mi aproximación al ejercicio:

```
num_x = int(input('How many times should I print the letter X? '))
to_print = ''
while len(to_print) < num_x:
    to_print = to_print + "X"
print(to_print)
```

Con el bucle `while` itero el código hasta que la longitud de `to_print` sea igual o mayor a `num_x`, por cada iteración se concatena un "X" a `to_print`, una vez que se termina el bucle imprime la variable `to_print`.

A veces es conveniente salir de un bucle sin comprobar la condición del bucle. La ejecución de una sentencia `break` termina el bucle en el que está contenida y transfiere el control al código que sigue inmediatamente al bucle. Por ejemplo, el código

```
#Find a positive integer that is divisible by both 11 and 12
x = 1
while True:
    if x % 11 == 0 and x % 12 == 0:
        break
    x = x + 1
print(x, 'is divisible by 11 and 12')
```

Imprime

```
132 is divisible by 11 and 12
```

Si se ejecuta una sentencia `break` dentro de un bucle anidado (un bucle dentro de otro bucle), el `break` terminará el bucle interno.

> **Ejercicio manual**
>
> Escriba un programa que pida al usuario que introduzca 10 números enteros y luego imprima el mayor número impar que se haya introducido. Si no se introdujo ningún número impar, debería imprimir un mensaje a tal efecto.

Mi aproximación al ejercicio:

```
number = 0
ans = 0
while True:
    n = int(input("Introduce un numero: "))
    if n % 2 != 0 and ans < n:
        ans = n
    number = number + 1
    if number >= 10:
        break
print(f'El mayor numero impar introducido es {ans}')
```

Primero inicialice dos variables, `number` encargada de contabilizar cuantos números ingreso el usuario y `ans` la cual va a contener el numero impar mas alto ingresado.
Dentro del bucle `while` el cual solo se va a detener si llega a la sentencia `break`, inicializo una variable `n` la cual contendrá el numero ingresado por el usuario, un condicional que verificara si se cumple la condición de si no es un numero par y si el numero en la variable `ans` en esa iteración es menor que el numero `n` actual, si se cumplen las condiciones `ans` ahora será `n`.
Luego por cada iteración `number` aumentara en 1.
Por ultimo tenemos otro condicional donde verificamos que la cantidad de entradas por el usuario sea igual a 10 ejecutando la palabra reservada `break`.
Finalmente se imprime el mayor numero impar.

Bucles `for` y `range()`

Los bucles `while` que hemos utilizado hasta ahora son muy estilizados, a menudo iterando sobre una secuencia de enteros. Python proporciona un mecanismo de lenguaje, el bucle `for`, que se puede utilizar para simplificar los programas que contienen este tipo de iteración.

La forma general de una sentencia `for` es:

```
for variable in sequence:
    code block
```

La `variable` que sigue a `for` se asocia al primer valor de la `sequence` y se ejecuta el `code block`. A continuación, se asigna a la variable el segundo valor de la secuencia y se vuelve a ejecutar el bloque de código.

El proceso continúa hasta que se agota la secuencia o se ejecuta una sentencia `break` dentro del bloque de código. Por ejemplo, el código:

```
total = 0
for num in (77, 11, 3):
    total = total + num
print(total)
```

imprimirá 91. La expresión (77, 11, 3) es una **tupla**. Por ahora, piense en una tupla como una *secuencia de valores*.

La secuencia de valores ligados a la variable se genera normalmente utilizando la función incorporada `range`, que devuelve una serie de enteros. La función `range` toma tres argumentos enteros: `start`, `stop` y `step`. Produce la progresión inicio, inicio + paso, `start + 2 * step`, etc. Si el paso es positivo, el último elemento es el mayor número entero tal que (`start + i * step`) es estrictamente menor que stop. Si el paso es negativo, el último elemento es el menor número entero tal que (`start + i * step`) es mayor que stop. Por ejemplo, la expresión `range(5, 40, 10)` da como resultado la secuencia 5, 15, 25, 35, y la expresión `range(40, 5, -10)` da como resultado la secuencia 40, 30, 20, 10.

Si se omite el primer argumento de `range`, por defecto es 0, y si se omite el último argumento (el tamaño del paso), por defecto es 1. Por ejemplo, tanto `range(0, 3)` como `range(3)` producen la secuencia 0, 1, 2. Los números de la progresión se generan «según sea necesario», por lo que incluso expresiones como `range(1000000)` consumen poca memoria.

Considere el código:

```
x = 4
for i in range(x):
    print(i)
```

Imprimirá:

```
0
1
2
3
```

Re implementemos el algoritmo que usamos en la sección del bucle `while` para elevar al cuadrado un entero (corregido para que funcione

con números negativos).

```
x = 3
ans = 0
for num_iterations in range(abs(x)):
    ans = ans + abs(x)
print(f'{x}*{x} = {ans}')
```

Observe que, a diferencia de la implementación del bucle `while`, el número de iteraciones no se controla mediante una prueba explícita, y la variable índice `num_iterations` no se incrementa explícitamente.

Observe que el código no cambia el valor de `num_iterations` dentro del cuerpo del bucle `for`. Esto es típico, pero no necesario, lo que plantea la cuestión de qué ocurre si la variable índice se modifica dentro del bucle `for`. Considere

```
for i in range(2):
    print(i)
    i = 0
    print(i)
```

¿Crees que imprimirá 0, 0, 1, 0, y luego se detendrá? ¿O crees que imprimirá 0 una y otra vez? La respuesta es 0, 0, 1, 0. Antes de la primera iteración del bucle `for`, se evalúa la función `range` y el primer valor de la secuencia que produce se asigna a la variable índice, `i`. Al comienzo de cada iteración posterior del bucle, se asigna a `i` el siguiente valor de la secuencia. Cuando se agota la secuencia, el bucle termina. El bucle `for` anterior es equivalente al código

```
index = 0
last_index = 1
while index <= last_index:
    i = index
    print(i)
    i = 0
    print(i)
    index = index + 1
```

Observe, por cierto, que el código con el bucle `while` es considerablemente más engorroso que el bucle `for`. El bucle `for` es un mecanismo lingüístico conveniente.

Ahora bien, ¿Qué le parece que imprime el siguiente código?

```
x = 1
for i in range(x):
    print(i)
    x = 4
```

¿imprime? Sólo 0, porque *los argumentos de la función range en la línea con for se evalúan justo antes de la primera iteración del bucle, y no se reevalúan en las iteraciones siguientes*.

Veamos ahora con qué frecuencia se evalúan las cosas cuando anidamos bucles.

Consideremos:

```
x = 4
for j in range(x):
    for i in range(x):
        x = 2
```

¿Cuántas veces se ejecuta cada uno de los dos bucles? Ya vimos que el `range(x)` que controla el bucle exterior se evalúa la primera vez que se alcanza y no se reevalúa en cada iteración, por lo que hay cuatro iteraciones del bucle exterior. Esto implica que el bucle `for` interior se alcanza cuatro veces. La primera vez que se alcanza, la variable `x = 4`, por lo que habrá cuatro iteraciones. Sin embargo, las siguientes tres veces que se alcanza, `x = 2`, por lo que habrá dos iteraciones cada vez. En consecuencia, si ejecuta

```
x = 3
for j in range(x):
    print('Iteration of outer loop')
    for i in range(x):
        print(' Iteration of inner loop')
        x = 2
```

Imprime:

```
Iteration of outer loop
    Iteration of inner loop
    Iteration of inner loop
    Iteration of inner loop
Iteration of outer loop
    Iteration of inner loop
    Iteration of inner loop
Iteration of outer loop
```

```
    Iteration of inner loop
    Iteration of inner loop
```

La sentencia `for` puede utilizarse junto con el operador `in` para iterar cómodamente sobre los caracteres de una cadena. Por ejemplo,

```
total = 0
for c in '12345678':
    total = total + int(c)
print(total)
```

Suma los dígitos de la cadena denotada por el literal '12345678' e imprime el total.

Ejercicios manuales

Escribe un programa que imprima la suma de los números primos mayores que 2 y menores que 1000. Sugerencia: probablemente quieras usar un bucle for que sea una prueba de primalidad anidada dentro de un bucle for que itere sobre los enteros impares entre 3 y 999.

Mi aproximación al ejercicio:

```
sum = 0
for i in range(2, 1000):
    is_prime = True
    for j in range(2, i):
        if i % j == 0:
            is_prime = False
            break
    if is_prime:
        sum += i
print(sum)
```

Inicializo la variable `sum` la cual va a contener la suma de todos los números primos mayores que 2 y menor que 1000.
El bucle `for` iterara los números en el rango entre 2 a 1000.
Inicializo una variable booleana que me permitirá indicar si el numero es primo o no primo.
Anido otro bucle del rango desde 2 hasta el numero el cual esta en la iteración por encima. Verifico que el numero de la iteración por encima y el numero que esta en la iteración actual el resto de 0, por ende seria un numero no primo ya que si podría ser divisible por otro

numero que no sea el 1 o por si mismo, entonces no tiene sentido seguir con esa iteración y hacer un break, para pasar al siguiente numero. Luego si es primo lo adicionaremos a la variable `sum`. Finalizando el algoritmo imprimiendo la variable `sum`.

El estilo importa

Conocer un lenguaje y saber utilizarlo bien son dos cosas distintas. Considere las dos frases siguientes:

«*Todo el mundo sabe que si un hombre es soltero y tiene mucho dinero, necesita casarse*».
«*Es una verdad universalmente reconocida, que un hombre soltero en posesión de una buena fortuna, debe estar en falta de una esposa*».

Cada una es una frase española correcta, y cada una significa aproximadamente lo mismo. Pero no son igual de convincentes ni fáciles de entender. Al igual que el estilo es importante cuando se escribe en español, también lo es cuando se escribe en Python.

Sin embargo, aunque tener una voz distintiva puede ser una ventaja para un novelista, no lo es para un programador. *Cuanto menos tiempo tengan que pasar los lectores de un programa pensando en cosas irrelevantes para el significado del código, mejor*. Por eso los buenos programadores siguen convenciones de codificación diseñadas para que los programas sean fáciles de entender, más que divertidos de leer.

La mayoría de los programadores de Python siguen las convenciones establecidas en la guía de estilo PEP 8.21 Como todos los conjuntos de convenciones, algunas de sus prescripciones son arbitrarias. Por ejemplo, prescribe el uso de cuatro espacios para las sangrías. ¿Por qué cuatro espacios y no tres o cinco? No hay ninguna buena razón. Pero si todo el mundo utiliza el mismo número de espacios, es más fácil leer (y quizás combinar) código escrito por diferentes personas. En términos más generales, si todo el mundo utiliza el mismo conjunto de convenciones al escribir Python, los lectores pueden concentrarse en comprender la semántica del código en lugar de malgastar ciclos mentales asimilando decisiones estilísticas.

Las convenciones más importantes tienen que ver con la nomenclatura. Ya hemos discutido la importancia de usar nombres de variables que transmitan el significado de la variable. Las frases sustantivas funcionan bien para esto.

Por ejemplo, usamos el nombre `num_iterations` para una variable que denota el número de iteraciones. Cuando un nombre incluye múltiples palabras, la convención en Python es usar un guion bajo (`_`) para separar las palabras. De nuevo, esta convención es arbitraria. Algunos programadores prefieren utilizar lo que a menudo se llama **camelCase**, por ejemplo, `numIterations` - argumentando que es más rápido de escribir y utiliza menos espacio.

También existen algunas convenciones para los nombres de variables de un solo carácter. La más importante es *evitar usar una L minúscula o una I mayúscula* (que se confunden fácilmente con el número uno) *o una O mayúscula* (que se confunde fácilmente con el número cero).

Ejercicio manual de la lectura

Suponga que se le da una variable entera positiva llamada `N`. Escriba un fragmento de código Python que imprima `"hola mundo"` en líneas separadas, `N` veces. Puedes utilizar un bucle `while` o un bucle `for`.

Mi aproximación

```
for times in range(N):
    print("hello world")
```

Simplemente utilizamos un bucle `for` que itera `N` veces, en cada iteración imprimiremos `"hello world"`.
