Ciclos y Rangos

Unidad 3

Apunte de cátedra

2do Cuatrimestre 2023

Pensamiento computacional (90)
Cátedra: Camejo

.UBAXXI

1. Ciclos

1.1 Introducción

Supongamos que en una fábrica se nos pide hacer un procedimiento para entrenar al personal nuevo. Para comenzar se nos encarga la descripción de uno muy simple: descarga de cajas de material del camión del proveedor y almacenamiento en el depósito. Así que aplicamos lo que venimos aprendiendo hasta ahora sobre algoritmos y describimos la operación para la descarga de 3 cajas:

1 Abrir la puerta del depósito y encender luces
2 Ir al garage o playón donde estacionó el camión
3 Abrir las puertas traseras de la caja del transporte
4 Tomar una caja con ambas manos, asegurándola para no tirarla
5 Caminar sosteniendo la caja hasta el depósito
6 Colocar la caja sobre el piso en el sector correspondiente
7 Ir al garage o playón donde estacionó el camión
8 Tomar OTRA caja con ambas manos, asegurándola para no tirarla
9 Caminar sosteniendo la caja hasta el depósito
10 Colocar la caja sobre la caja anterior
11 Ir al garage o playón donde estacionó el camión
12 Tomar OTRA caja con ambas manos, asegurándola para no tirarla
13 Caminar sosteniendo la caja hasta el depósito
14 Colocar la caja sobre la caja anterior
15 Apagar luces y cerrar puerta del depósito
16 Ir al garage o playón donde estacionó el camión
17 Cerrar y trabar puertas del camión
18 Avisar fin de descarga al transportista

Ya lo tenemos. Ahora el jefe dice que en el camión suelen venir entre 5 y 15 cajas de material y nos pide que definas el mismo procedimiento para todos los casos posibles. Notemos que se repiten las instrucciones 2, 3, 4, 5 y 6 para cada caja ¿Qué hacemos? ¿Vamos a seguir copiando y pegando las instrucciones para cada caja? ¿Y si algún día vienen más de 15 o menos de 5? ¿Vamos a tener una lista de instrucciones distinta para cada cantidad de cajas que puedan venir? Parece ser necesario hacer algo más genérico que le facilite la vida a todos. Una nueva versión:

1 Abrir la puerta del depósito y encender luces
2 Ir al garage o playón donde estacionó el camión
3 Abrir las puertas traseras de la caja del transporte
4 Tomar una caja con ambas manos, asegurándola para no tirarla
5 Caminar sosteniendo la caja hasta el depósito
6 Si es la primera caja, colocarla sobre el piso en el sector correspondiente; si no, apilarla sobre la anterior; salvo que ya haya 3 apiladas, en ese caso colocarla a la derecha sobre el piso

7 Ir al garage o playón donde estacionó el camión
8 Repetir 4,5,6,7 mientras queden cajas para descargar
9 Cerrar y trabar puertas del camión
10 Avisar fin de descarga al transportista
11 Volver a depósito
12 Apagar luces y cerrar puerta del depósito

Esta descripción es bastante más compacta y cubre todas las posibles cantidades de cajas en un envío (habituales y excepcionales), de modo que con una única página en el manual de procedimientos será suficiente.

Sin embargo, los algoritmos de los programas que venimos escribiendo hasta ahora se parecen más al primer procedimiento. Así que para mejorar estos algoritmos vamos a incorporar una nueva herramienta que nos permita escribir, de manera compacta, la ejecución de ciertos pasos de forma repetida.

Esa clase de recurso se denomina genéricamente **ciclo**, **bucle** o **sentencia iterativa**; y es una máquina repetidora de pasos o instrucciones en un programa. Es decir que podemos elegir una parte de nuestro programa, que necesitamos se ejecute repetidamente y colocarla dentro de una sentencia especial con ese objetivo; que permite realizar esa repetición automáticamente, una y otra vez, hasta terminar.

Esta herramienta está dentro del grupo de las **Estructuras de Control Iterativas** ya que es sentencia estructurada, con encabezado y cuerpo; y tienen la capacidad de alterar el **Flujo de Control** Normal de un **Programa** (**FCP**). Es decir, puede hacer que mi código no se ejecute de línea a línea sino que pueda volver y repetir las sentencias que se necesite.

1.2 Definiciones

Los **bucles** son primitivas sencillas, normalmente tienen una sintaxis simple. Su dificultad no está en cómo se escriben, ni en entender su funcionamiento, sino en la manera en que se ajustan sus parámetros. Esto es lo que nos garantizará que haga exactamente lo que esperamos. Tenemos que prestar atención tanto al código que debe repetir como a la cantidad de veces exacta que se va a repetir.

La cantidad de repeticiones que se van a realizar el **ciclo** puede ser una cantidad **constante** (fija) o una cantidad **variable**, es decir que no se sabe la cantidad de veces que se va a repetir hasta que se ejecute el código. Esto último puede pasar por ejemplo cuando queremos que se repita algo la cantidad de veces que ingrese el usuario por consola.

Al escribir un **bucle** necesitamos tener muy en claro 4 aspectos relevantes. Si nos equivocamos en cualquiera de ellos el **ciclo** no funcionará como esperamos. Estos 4 aspectos son los siguientes:

1.2.1. Cuerpo

Es la parte del **bucle** que se va a ejecutar repetidamente. Son un conjunto de sentencias válidas que se ejecutan una y otra vez (**como conjunto, siguiendo el flujo interno**) hasta que la evaluación de la **condición** (que es la llave del **bucle**) o **invariante** diga lo contrario.

1.2.2. Condición o Invariante

Es la parte de la sentencia que permite decidir si se ejecutará el **cuerpo** de ese **bucle** una vez más o se abandonará el mismo para proseguir con la sentencia siguiente al **ciclo**. Hay que tener en cuenta que posee doble función, determina cuándo empieza el ciclo para que el **cuerpo** sea ejecutado y cuándo termina. Es la puerta de entrada/salida al **ciclo.**

1.2.3. Estado Previo

Son los valores de las variables de la **condición** antes del **bucle.**

1.2.4. Paso

Es la posibilidad de cambio de valor en alguna de las variables de la **condición**.

1.3 ¿Cómo funciona un Ciclo?

Cuando el **FCP** del programa llega a un **Ciclo**, lo primero que hace es evaluar la **condición**: si esta da *Verdadero* se ejecuta completamente el **cuerpo**; y cuando se termina de ejecutar el mismo, el flujo vuelve a la **condición** (dibujando el lazo, rulo o bucle) y esta se evalúa nuevamente. Si vuelve a dar *Verdadero* se repite todo el proceso anterior; si en cambio da *Falso*, se abandona el **ciclo**.

Es decir, el programa llega, entra (si se dan las condiciones) y se queda iterando allí (repitiendo el **cuerpo**) todas las veces que haga falta. Cuando las condiciones determinan el fin de las repeticiones se va del **Ciclo**, y no vuelve.

2. While

El **while** es una sentencia de ciclo que nos permite ejecutar el cuerpo del ciclo *mientras* la condición sea verdadera, es decir:

```
while condición:
      instrucción 1
      instrucción 2
       …
```

⚠️ **¡Atención!**

Es muy importante respetar poner un TAB (un espacio con la tecla TAB del teclado) para la sintaxis del while, ya que esta *indentación* es lo que le permite a Python distinguir el cuerpo del ciclo del resto de las instrucciones del programa.

2.1 ¿Cómo se ejecuta while de Python?

Cuando el **FCP** llega a una sentencia **while** lo primero que se realiza es la evaluación de la **condición**. Si la **condición** da *VERDADERO* se ejecutará completamente el **cuerpo** del **while**. Cuando termina la ejecución del se evalúa nuevamente la condición: si da VERDADERO repite lo anterior; si da FALSO sale del **while** y ejecuta la sentencia siguiente al mismo. Si la primera vez que el **FCP** llega al **while** y evalúa la **condición** da FALSO, el **FCP** sigue de largo a la próxima sentencia y saltea todo el **cuerpo**.

Por ejemplo, hagamos un programa que pida 5 números y cuente cuántos múltiplos de 3 vinieron.

```
# Contemos cuántos múltiplos de 3 ingresan en un lote de 5 números
print('Ingresá 5 números enteros')
total_mult = 0
num = int(input('Número: '))
if num % 3 == 0:     #identifico si es múltiplo de 5
    total_mult+=1
num = int(input('Número: '))
if num % 3 == 0:
    total_mult+=1
num = int(input('Número: '))
if num % 3 == 0:
    total_mult+=1
num = int(input('Número: '))
if num % 3 == 0:
    total_mult+=1
num = int(input('Número: '))
if num % 3 == 0:
```

```
    total_mult+=1
print('Vinieron:',total_mult,'múltiplos de 3')
```

Este programa puede ser reescrito usando un **bucle**: hay un conjunto de instrucciones que se repiten de manera contigua. Además por ejemplo si quisiéramos extender la cantidad de números sobre los que va a trabajar, deberíamos hacer algunos copy-paste del grupo de sentencias. Y si decidimos dejarle al usuario la decisión de la cantidad de números que va a ingresar, no podríamos hacerlo de otra forma que no sea con una sentencia iterativa, ya que no sabremos la cantidad de veces que vamos a tener que repetir este código.

Así que reconstruyamos este código usando ciclos. Teníamos que prestar atención a 4 aspectos para armar y afinar correctamente una Iteración:

- **Cuerpo**
- **Condición**
- **Estado Previo**
- **Paso**

Primero vamos a pensar cuál sería el **cuerpo**, o al menos, el mínimo **cuerpo**. Este será, sin dudas, la porción del código que se repetirá de forma contigua. En nuestro ejemplo marcamos el código que repetimos una y otra vez: **α**.

```
# Contemos cuántos múltiplos de 3 ingresan en un lote de 5 números
print('Ingresá 5 números enteros')
total_mult = 0
num = int(input('Número: '))
if num % 3 == 0:     #identifico si es múltiplo de 5
    total_mult+=1
num = int(input('Número: '))
if num % 3 == 0:
    total_mult+=1
num = int(input('Número: '))
if num % 3 == 0:
    total_mult+=1
num = int(input('Número: '))
if num % 3 == 0:
    total_mult+=1    α.
num = int(input('Número: '))
if num % 3 == 0:
    total_mult+=1
print('Vinieron:',total_mult,'múltiplos de 3')
```

Así que empezaremos por reescribir el programa compactándolo:

```
# Contemos cuántos múltiplos de 3 ingresan en un lote de 5 números
print('Ingresá 5 números enteros')
total_mult = 0
    num = int(input('Número: '))
    if num % 3 == 0:
```

```
        total_mult+=1
print('Vinieron:',total_mult,'múltiplos de 3')
```

Nuestro **cuerpo(α)** se posiciona desplazado a la derecha, ya que debe ir indentado, como fue aclarado antes. Pero nos falta la **condición** que sirva de puerta: que se abra para ingresar a ejecutar el **cuerpo** todas las veces que haga falta y se cierre definitivamente cuando ya repetimos el mismo las veces necesarias.

Entonces, debemos pedir un número y averiguar si es múltiplo de 3 cinco veces. O, lo que es lo mismo, hasta que no hayamos hecho 5 veces esto (5 vueltas) no paramos. Miremos esta posible condición:

```
# Contemos cuántos múltiplos de 3 ingresan en un lote de 5 números
print('Ingresá 5 números enteros')
total_mult = 0
while veces <= 5:
    num = int(input('Número: '))
    if num % 3 == 0:
        total_mult+=1
print('Vinieron:',total_mult,'múltiplos de 3')
```

Ahora si ejecutamos este programa, vamos a ver que Python nos da un error. ¿Qué pasó? No aseguramos el **estado previo**. Es decir, ***no preparamos con algún valor controlado cada una de las variables que forman parte de la condición antes de comenzar a ejecutar el Bucle***. En nuestro caso, la **condición** tiene una única variable "**veces"** y como no estuvo previamente inicializada, Python no la reconoce y da error.

Así que será necesario darle un valor inicial en cualquier punto previo al encabezado del **while**. Así que volvamos a escribir el código con este arreglo:

```
# Contemos cuántos múltiplos de 3 ingresan en un lote de 5 números
print('Ingresá 5 números enteros')
total_mult = 0
veces = 0
while veces <= 5:
    num = int(input('Número: '))
    if num % 3 == 0:
        total_mult+=1
print('Vinieron:',total_mult,'múltiplos de 3')
```

Ahora Python va a ejecutar correctamente el código, pero vemos que no para de pedirnos números… parece que el programa no va a terminar nunca y sólo nos queda terminar la ejecución abruptamente ¿Qué sucedió? Entramos en lo que se conoce como un **loop infinito**, un ciclo que no para nunca.

Esto sucedió porque no cuidamos el **paso** o **avance**, es decir tenemos que asegurarnos que dentro del **cuerpo**, **por lo menos para una de las variables de la condición, haya al menos una sentencia donde se le pueda cambiar el valor**. Es decir, al menos una de las variables de la condición tiene que ser cambiada dentro

del **cuerpo**. De ese modo, eventualmente el cambio en el valor de la variable puede producir un valor ***False*** de la **condición**.

En nuestro problema, tenemos que ir contando cada número ingresado aumentando la variable “veces”, y de esta forma en algún momento la cuenta nos dará un valor mayor que 5, lo cual va a ocasionar que la condición nos de Falsa. Esto queda de la siguiente manera:

```
# Contemos cuántos múltiplos de 3 ingresan en un lote de 5 números
print('Ingresá 5 números enteros')
total_mult = 0
veces = 0
while veces <= 5:
    num = int(input('Número: '))
    if num % 3 == 0:
        total_mult+=1
    veces += 1
print('Vinieron:',total_mult,'múltiplos de 3')
```

Ahora al programa le falta un pequeño detalle, si lo probamos observaremos que en realidad nos pide 6 números en lugar de 5. Esto es porque el cuerpo se ejecuta cuando “veces” vale 0, 1, 2, 3, 4 y 5, ya que la condición solamente va a dar False cuando “veces” valga 6.

Este tipo de desajustes son comunes, incluso si sos un programador experimentado; pero se salvan fácilmente. Lo que hay que hacer es ajustar el **estado previo**, la **condición**, el **paso**, o cualquier combinación de ellos, hasta que el combo funcione. Por ejemplo, sólo tocando la inicialización de **veces** podemos hacer que funcione correctamente:

```
# Contemos cuántos múltiplos de 3 ingresan en un lote de 5 números
print('Ingresá 5 números enteros')
total_mult = 0
veces = 1
while veces <= 5:
    num = int(input('Número: '))
    if num % 3 == 0:
        total_mult+=1
    veces += 1
print('Vinieron:',total_mult,'múltiplos de 3')
```

De esta forma, se va a ejecutar el cuerpo con “veces” valiendo 1, 2, 3, 4 y 5, ejecutándose correctamente 5 veces.

Otra manera de arreglar el problema sería dejar la inicialización `veces = 0` y solamente cambiando la condición por `veces < 5`, lo cual va a ocasionar que cuando “veces” valga 5 la condición dé Falsa y no se ejecute el código del cuerpo (porque 5 no es menor que 5). Con esta forma se va a ejecutar con “veces” valiendo 0, 1, 2, 3 y 4.

Otro ejemplo:

```
# Contemos cuántos múltiplos de 3 ingresan en un lote de 5 números
print('Ingresá 5 números enteros')
total_mult = 0
veces = 5
while veces <= 5:
    num = int(input('Número: '))
    if num % 3 == 0:
        total_mult+=1
    veces -= 1
print('Vinieron:',total_mult,'múltiplos de 3')
```

Hay muchas formas de hacer que el cuerpo se ejecute 5 veces, lo importante es cuidar la condición, el paso y el estado previo.

Sin un **estado previo** controlado existen serias posibilidades de que el **bucle** nunca se ejecute. Sin una **condición** correcta es probable que la cantidad de vueltas no sea la adecuada. Sin un **paso** seguramente entraríamos en un loop infinito, es decir, nunca podemos salir del **bucle**. Sin una adecuada combinación de todos ellos probablemente la cantidad de vueltas no sea apropiada.

La condición no siempre tiene que ser determinada por una variable numérica. En el siguiente ejemplo, le mostramos por pantalla un saludo al usuario hasta que ingrese una X.

```
# Imprime saludo por pantalla hasta que usuario ingresa X
print('Te saludo hasta que me dejes')
respuesta = ‘S’
while respuesta != ‘X’:
    print(‘¡Hola!')
    respuesta = input(‘¿Querés otro saludo? Ingresá X si la respuesta es no’)
```

2.2. Break y Continue

Break y continue son dos palabras clave en Python que se utilizan en bucles, como el bucle while, para controlar el flujo de ejecución del programa.

2.2.1. Break

La declaración **break** se utiliza para salir inmediatamente de un bucle antes de que se complete su iteración normal. Cuando se encuentra una instrucción **break**, el programa salta fuera del bucle y continúa con la ejecución de las instrucciones que están después del bucle.

Ejemplo:

```
numero = 10
while numero > 0:
      if numero % 3 == 0:
            print("El primer número divisible por 3 es:", numero)
            break
      numero += 1
```

2.2.2. Continue

La declaración **continue** se utiliza para saltar una iteración del bucle y continuar con la siguiente iteración. Cuando se encuentra una instrucción **continue**, el programa se salta el resto del bloque de código del bucle para esa iteración específica y continúa con la siguiente iteración.

```
numero = 1
while numero <= 10:
      if numero % 2 == 0:
         # Si es par, suma 1 y saltar a la siguiente iteración
         numero += 1
         continue
      print(numero)
      numero += 1
```

Notemos que tanto para el uso de break como de continue, si el código se encuentra con uno de ellos, no ejecuta nada posterior y vuelve a comenzar el ciclo, evaluando la condición (en el caso del while) y volviendo a ejecutar el bloque dentro del bucle. Por lo que en ambos ejemplos no tuvimos necesidad de usar un 'else': simplemente al usar el break o el continue sabemos que no se va a ejecutar el código que sigue. Y, si no se cumple la condición del if, entonces ejecuta el código debajo del bloque y listo.

2.3. Consideraciones del while

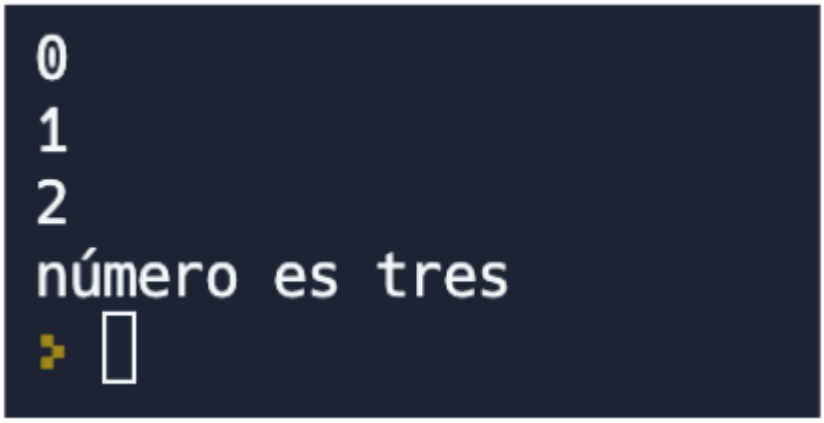
Es importante no ser redundantes con el código y no preguntar cosas que ya sabemos.

```
while condicion:
    <código>

<codigo cuando no se cumple la condicion>
```

Veamos un ejemplo:

```
numero = 0
while numero < 3:
    print(numero)
    numero += 1

if numero == 3:
    print("número es tres")
else:
    print("número no es tres")
```

El output siempre es el mismo:



¿Por qué?
Nuestra condición del while es lo que nos dice "mientras esto se cumpla, yo repito el bloque de código de adentro".
Nuestra condición es que `numero < 3`. En el momento en el que **numero** llega a 3, el while deja de cumplir con la condición, y **la ejecución se corta, termina con el bucle.**

```
numero = 0
while numero < 3:
    print(numero)
    numero += 1

if numero == 3:
    print("número es tres")

else:
    print("número no es tres")
```

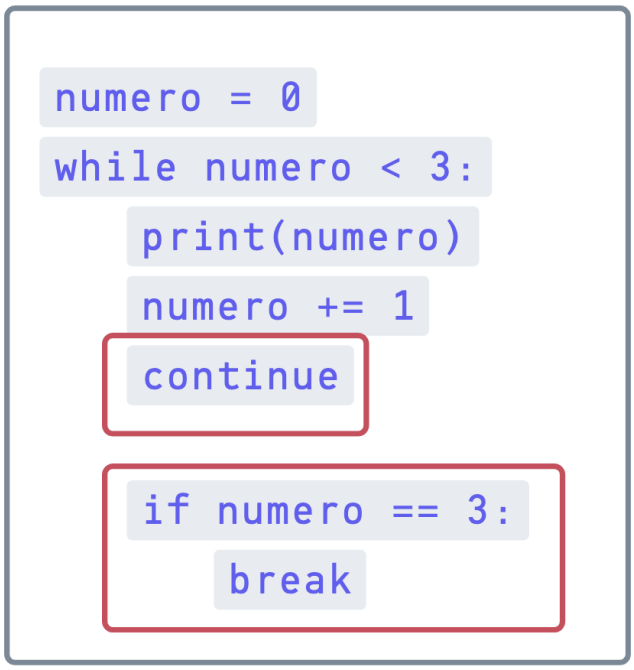
Es decir que el código en verde se ejecuta siempre. Si se ejecuta siempre, ¿necesito poner un if debajo? Yo ya sé que si sale del while, es porque el número es 3.

De la misma forma, el código rojo nunca se ejecuta. Si no se ejecuta nunca, ¿necesito tenerlo? Mejor sacar código que no estoy usando.

La función entonces queda algo así:

```
numero = 0
while numero < 3:
    print(numero)
    numero += 1

print("número es tres")
```

De la misma forma, no tendría sentido hacer algo así:



```
numero = 0
while numero < 3:
    print(numero)
    numero += 1
    continue

    if numero == 3:
        break
```

Por un lado, el continue está de más. Si el bloque dentro del while llega a la última línea, simplemente vuelve a empezar. No hace falta que le aclaremos con el continue que “tiene que volver a empezar”. El continue es para cuando nosotros queremos **forzar** que vuelva a comenzar.
Por el otro, el if dentro del while con el break tampoco tiene sentido. Primero, porque con el continue ahí nunca se ejecutaría. Pero además, como dijimos más arriba, la condición es que `numero < 3`. En el momento en el que **numero** llega a 3, el while deja de cumplir con la condición, y **la ejecución se corta.** Por lo que el if sólo estorba haciendo algo que ya de por sí iba a hacer el while.

3. For

En Python también tenemos a disposición el **bucle for**. Este es un **ciclo** que tiene automatizado, o incorporado al encabezado, el manejo del **estado previo** y del **paso**. Es decir, no tenemos que estar controlando que el paso y el estado previo sean correctos. La sintaxis es la siguiente:

```
for var in iterable:
      instrucción 1
      instrucción 2
       …
```

Un **iterable** llamamos a las estructuras que se pueden recorrer, decir “for var in iterable” es como decir “para cada elemento en esta estructura” se ejecuta el cuerpo. Esto se va a entender mejor cuando veamos los rangos.

3.1 ¿Cómo se ejecuta for de Python?

Cuando el **FCP** llega a una sentencia **for** lo primero que se hace es verificar que en el **iterable** existan valores, si es así, se asigna el primer valor del mismo a la variable de control (**var**) y se procede a ejecutar completamente el **cuerpo**. Una vez finalizada la ejecución completa del **cuerpo** el **FCP** vuelve al encabezado del **for** y asigna el valor siguiente en el **iterable** a la **variable de control**; repitiendo la ejecución del **cuerpo**.

Cuando se acaban los valores en el **iterable** y el **FCP** no dispone de más valores para asignarle a la **variable de control** el **ciclo** finaliza y el **FCP** pasa a ejecutar la sentencia siguiente al **for** (la primera debajo, que estará alineada al encabezado del **for**, o aún más a su izquierda).

Por ahora vamos a usar **rango** o **string** como opción para determinar los valores de la variable de control. Ya que son los únicos **iterables** que manejaremos por el momento.

3.2 Rangos

Un **rango** es un constructor o estructura de control que permite generar una sucesión de números enteros, con un cierto paso o salto regular. La sintaxis es:

```
range(valor_inicial, valor_fin, paso)
```

✏️ **Nota:**

- El paso DEBE ser un entero
- *valor_inicio* está incluído en el rango; *valor_fin* NO
- *valor_inicio* es **opcional**; si no se coloca se asume 0
- *paso* es **opcional**; si no se coloca se asume 1
- Si se desea rango decreciente se debe colocar un paso negativo
- Si se coloca paso, DEBE COLOCARSE *valor_inicio*, aunque este último sea 0

Algunos ejemplos de uso de rango:

`range(1,10)`	1,2,3,4,5,6,7,8,9
`range(6)`	0,1,2,3,4,5
`range(0,8,2)`	0,2,4,6
`range(15,10,-5)`	15
`range(15,10,-1)`	15, 14, 13, 12, 11

Versión del programa de contabilidad de múltiplos de 5 con **for**:

```
# Contemos cuántos múltiplos de 3 ingresan en un lote de 5 números
print('Ingresá 5 Números enteros')
total_mult = 0
for vuelta in range(5):
    num=int(input('Número: '))
    if num % 3 == 0:
        total_mult += 1
print('Vinieron:',total_mult,'múltiplos de 3')
```

En la sentencia `for vuelta in range(5)` tenemos todo junto el estado previo, el paso y la condición.

El ciclo for también puede iterar sobre elementos de una lista (vamos a ver más de esto después). Pero de momento, podemos ver que:

```
# Lista de números
numeros = [11, 24, 32, 4, 15, 6, 17, 38, 94, 110]
for numero in numeros:
     print(numero)
```

Itera sobre todos los números de la lista, en orden, y los imprime.

3.3. Break y Continue

Tanto break como continue funcionan de la misma forma para ciclos while y ciclos for.
Por lo tanto, dejamos unos ejemplos únicamente:

2.2.1. Break

Ejemplo:

```
# Lista de números
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Bucle for para encontrar el primer número divisible por 7 en la lista
for numero in numeros:
    if numero % 7 == 0:
        print("El primer número divisible por 7 es:", numero)
        break
```

2.2.2. Continue

```
# Lista de números
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Bucle for para imprimir los números impares en la lista
for numero in numeros:
```

```
    if numero % 2 == 0:
        # Si es par, saltar a la siguiente iteración
        continue
    print(numero)
```

4. Ciclos anidados

Como ya vimos, un **ciclo** repite un bloque de sentencias (al que llamamos **cuerpo** del **ciclo**). Por supuesto que ese **cuerpo** puede estar conformado con cualquier combinación y cantidad de sentencias válidas de Python. Por lo tanto, resulta obvio suponer que dentro de un **ciclo** pueda haber otro. Cuando se da esto, decimos que los **ciclos** están **anidados**.

El **FCP** de un programa ejecuta un **bucle** hasta salir y recién sigue por la sentencia siguiente al mismo. Por lo tanto, cuando el **FCP** ingrese a un **ciclo** interno (que está dentro de otro más externo), se quedará repitiendo este hasta finalizar y recién proseguirá ejecutando el **cuerpo** del más externo.

Puede haber tantos niveles de anidamiento como queramos o necesitemos.

✏️ **Nota:** cuando hay varios ciclos anidados hay que tener mucho cuidado de no alterar dentro de uno interno la variable de control de un bucle externo. Para ser ordenados y evitar problemas, cada **bucle con su variable de control.**

Hagamos un ejemplo en el que se usan ciclos. En primer término haremos un programa que cuente cuántos divisores tiene un número natural (entero positivo). Para esto tenemos en cuenta las siguientes notas:

- Todo número natural es divisible por sí mismo y por 1, ergo, no los contamos.
- Un número dividido por sí mismo produce cociente 1. El próximo cociente entero (2) lo obtenemos al dividir cualquier número por su mitad.
- De las dos consideraciones anteriores deducimos que la búsqueda de potenciales divisores de un número natural debería estar dentro del rango de enteros entre 2 y la mitad del número, inclusive.

```
num = int(input('Ingresá un número entero positivo: '))
while num <= 0:
    num = int(input('Ingresá un número entero positivo: '))
cant_divisores = 0
for d in range(2, num//2+1):
    if num % d == 0:
        cant_divisores += 1
print(num,'tiene',cant_divisores,'divisores')
```

En este ejemplo, lo que hacemos es pedirle al usuario un número, y para asegurarnos que lo que ingrese sea un entero positivo, usamos el **while** para crear un bucle en el cual se le pide al usuario el número una y otra vez mientras el número ingresado sea menor o igual a cero. Es decir, que el número será válido cuando la condición del while dé Falso (NO va a ser menor o igual a 0, por lo tanto es positivo) y permita seguir con la ejecución del código.

Después creamos un bucle con **for** que se va a ejecutar tantas veces como números enteros haya entre el rango desde 2 hasta la mitad del número ingresado (por las notas que vimos anteriormente). En el cuerpo del for nos fijamos para cada número dentro del rango, que se va guardando en la variable de control d, si el número ingresado por el usuario es divisible por d (que el resto de la división dé cero) los vamos contando en la variable `cant_divisores`.

⚠️ **¡Atención!**

Usamos un bucle while para validar el ingreso y descartar números que no sean naturales. No basta con pedirle al usuario que haga algo, debemos asegurarnos que lo cumpla. La regla es la siguiente: nunca confíes en que el usuario obedezca; y dale todas las oportunidades que necesite para corregirse.

Ahora averiguaremos la cantidad de divisores (a parte de sí mismo y 1) que tienen 10 números ingresados por el usuario. Ya tenemos resuelto cómo contar la cantidad de divisores de un número en el programa de arriba. Si tenemos que hacer eso 10 veces en lugar de 1 lo que necesitamos es repetir todo el programa anterior 10 veces, lo cual ya vimos que lo podemos hacer con otro bucle, en este caso usamos **for**.

```
print('Ingresá 10 números y te contamos cuántos divisores tiene cada uno')
for n in range(10):
    num = int(input('Ingresá un número entero positivo: '))
    while num <= 0:
        num=int(input('Ingresá un número entero positivo: '))
    cant_divisores = 0
    for d in range(2,num//2+1):
        if num % d == 0:
            cant_divisores += 1
    print(num,'tiene',cant_divisores,'divisores')
```

Como observarás, ahora tenemos 3 **bucles**; uno externo (**for** con variable de control **n**), y dos internos que son disjuntos entre sí (el **while** de validación y el **for** con variable de control **d**). Por cada vuelta del externo puede girar el **while** varias veces (eso dependerá de lo obediente que sea el usuario) y también girará algunas veces el **for** de búsqueda de divisores (eso dependerá del tamaño del número ingresado). Por último recalcamos que si tenemos **anidamiento** de **bucles for**, cada cual con su variable de control.

Ya aprendimos los bucles **for** y **while** y los vimos usados en diferentes programas o combinados en el mismo para hacer un programa más complejo. Ahora, ¿cómo sabemos cuándo usar for y cuándo while? Aunque ambos pueden ser útiles para las mismas cosas, hay ocasiones en las que es más cómodo usar uno por sobre el otro.

En general, cuando queremos hacer algo una x cantidad de veces, por ejemplo como vimos el programa de arriba, que se quería repetir el proceso de búsqueda de divisores para 10 números, o cuando nos fijamos si el número ingresado era divisible por los números que están entre 2 y la mitad del número, conviene usar el bucle **for**. Esto es porque con esta estructura no tenemos que estar atentos al paso y al estado previo de las variables, sino que todo está junto en la sentencia del for.

Cuando en cambio, no sabemos la cantidad de veces que se va a repetir el bucle o cuando la condición es más compleja, es preferible usar el **while**. Esto se puede ver por ejemplo cuando no sabíamos qué cantidad de veces el usuario iba a ingresar un número inválido (que no sea entero positivo) en el programa de arriba.

✏️ **Nota:**

Es importante aclarar que lo dicho anteriormente no se cumple siempre, sino que es algo general para que puedan distinguir y elegir cuándo usar for y cuándo while, pero ambos bucles pueden usarse como les parezca más cómodo. Ustedes eligen cuándo les conviene usar uno o el otro según las condiciones del problema que tengan que resolver.

Bibliografía Adicional

Wachenchauzer, R. (2018). *Aprendiendo a programar usando Python como herramienta*. Capítulo 5.