Estructuras cíclicas

March 29, 2022

Ejecuta el siguiente bloque de código siempre antes de ejecutar el resto del notebook.

[1]:
```
from IPython.core.magic import  Magics, magics_class, cell_magic, line_magic

@magics_class
class Helper(Magics):

  def __init__(self, shell=None,  **kwargs):
    super().__init__(shell=shell, **kwargs)

  @cell_magic
  def debug_cell_with_pytutor(self, line, cell):
    import urllib.parse
    url_src = urllib.parse.quote(cell)
    str_begin = '<iframe width="1000" height="500" frameborder="0" src="https://
↪pythontutor.com/iframe-embed.html#code='
    str_end   = '&cumulative=false&py=3&curInstr=0"></iframe>'
    import IPython
    from google.colab import output
    display(IPython.display.HTML(str_begin+url_src+str_end))

get_ipython().register_magics(Helper)
```

1 Estructuras de control cíclicas

Los algoritmos que hemos planteado hasta ahora son un conjunto de instrucciones que se ejecutan una sola vez, pero el computador está diseñado para desarrollar aplicaciones en las que una operación se pueda repetir cuantas veces queramos. En la vida cotidiana es común encontrar tareas donde es necesario que un grupo de instrucciones se repitan muchas veces, por ejemplo el ensamblaje de un producto dentro de una cadena de producción. En la computación, podemos aprovechar la velocidad de respuesta de las máquinas, para que realicen este tipo de tareas y las desarrolle más rápido que como lo haría un ser humano, esto se logra gracias a las estructuras de control repetitivas o ciclos.

Estas estructuras están conformadas por una condición y un grupo de instrucciones que se desean repetir. La condición, como vimos en la sección anterior, es una expresión lógica que determina un valor de verdad: `True` o `False`. Si la condición es `True`, el ciclo se ejecuta; de lo contrario, el computador avanza a la siguiente instrucción fuera del bloque. Dentro del conjunto de instrucciones

que se desean repetir es posible usar cualquiera de las instrucciones vistas hasta ahora, además de poder añadir otros ciclos de forma anidada.

La ejecución repetida de un conjunto de sentencias se llama **iteración**. Python tiene dos estructuras para la iteración: la sentencia `while` y la sentencia `for`. Antes de estudiar las estructuras cíclicas, definamos algunos términos importantes.

- **Contador**: como su nombre lo dice, cuenta el número de ocurrencias de un evento dentro de un ciclo. Por ejemplo: en el programa para determinar el nuevo salario de una persona, nos interesaría conocer el número de personas a las que se les hizo el cálculo. Ahí se requiere un contador. Un contador, por lo general, se inicializa en cero.
- **Acumulador**: es una variable en la cual se lleva el total de un concepto específico en un ciclo. Por ejemplo: en el programa para determinar el nuevo salario de una persona, nos interesaría conocer el total de salarios anteriores, el total de aumentos y el total de nuevos salarios. Para cada uno de esos totales se requiere un acumulador.
- **Promedio**: es el valor medio de un evento cuantificable. Se obtiene dividiendo un acumulador entre su respectivo contador

1.1 Ciclo While

Esta es la estructura cíclica más básica que existe, permite repetir instrucciones mientras se cumpla una condición determinada, lo cual permite diseñar ciclos en los cuales no se conoce de antemano la cantidad de veces que se accederá al bloque de código a repetir.

```
while condicion_de_parada:
    instruccion
#fin ciclo while
```

La sintaxis del ciclo while es muy similar a la estructura del `if` que analizamos en la sección anterior; tenemos la palabra reservada `while`, que le indica a Python que el bloque de instrucciones a continuación se debe repetir si se cumple una condición lógica predefinida (comúnmente la denominamos condición de parada), que determinará cuándo se termina el ciclo, se debe añadir dos puntos `:` para abrir un nuevo bloque en Python.

Si conocemos la cantidad de iteraciones a realizar dentro del ciclo `while`, podemos comparar un contador con dicha cantidad para construir la condición del ciclo, ej: `n > 10`, `cont <= 15`.

Como ejemplo, creemos un programa que imprima las potencias de 2 hasta un número n dado por el usuario (2^n) es decir $2, 4, 8, 16, 32, ...$ Como no sabemos de entrada cuantas repeticiones se van a realizar, debemos usar un ciclo.

[2]:
```
%%debug_cell_with_pytutor

n = int(input("digita la potencia máxima a imprimir: "))
i = 0
while i <= n:
    print(f'2^{i} = {2**i}')
    i += 1
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-2-105ca8feddba> in <module>
----> 1 get_ipython().run_cell_magic('debug_cell_with_pytutor', '', '\nn =␣
 ↪int(input("digita la potencia máxima a imprimir: "))\ni = 0\nwhile i <= n:\n␣
 ↪  print(f\'2^{i} = {2**i}\')\n    i += 1\n\n')

~\anaconda3\lib\site-packages\IPython\core\interactiveshell.py in␣
 ↪run_cell_magic(self, magic_name, line, cell)
   2380             with self.builtin_trap:
   2381                 args = (magic_arg_s, cell)
-> 2382                 result = fn(*args, **kwargs)
   2383             return result
   2384

<decorator-gen-118> in debug_cell_with_pytutor(self, line, cell)

~\anaconda3\lib\site-packages\IPython\core\magic.py in <lambda>(f, *a, **k)
    185     # but it's overkill for just that one bit of state.
    186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
    188
    189         if callable(arg):

<ipython-input-1-b1ec388e0f0e> in debug_cell_with_pytutor(self, line, cell)
     14     str_end   = '&cumulative=false&py=3&curInstr=0"></iframe>'
     15     import IPython
---> 16     from google.colab import output
     17     display(IPython.display.HTML(str_begin+url_src+str_end))
     18

ModuleNotFoundError: No module named 'google'
```

Cuando usamos un contador dentro de la condición de parada de un ciclo, es muy importante actualizar su valor internamente dentro del bloque de código, de lo contrario, el programa se quedará atorado en un ciclo infinito.

Un ejemplo de aplicación de los ciclos `while` cuando no conocemos la cantidad de iteraciones a realizar es un menú, puesto que este debe estar disponible para el usuario hasta que él mismo decida salir. Usemos el ejemplo de calculadora de la sección anterior:

[ ]:
```
%%debug_cell_with_pytutor
# a = float(input('diginte el número 1: '))
# b = float(input('diginte el número 2: '))
a = 10
b = 5
salida = False
```

```
while salida == False:
    print(
        """Menú:
        1. Sumar
        2. Restar
        3. Multiplicar
        4. Dividir
        5. Módulo
        6. Potencia
        0. salir
    """)
    opcion = int(input('Digite la opción: '))
    if(opcion == 1):
        print(f"{a} + {b} = ",a+b)
    elif(opcion == 2):
        print(f"{a} - {b} = ",a-b)
    elif(opcion == 3):
        print(f"{a} * {b} = ",a*b)
    elif(opcion == 4):
        print(f"{a} / {b} = ",a/b)
    elif(opcion == 5):
        print(f"{a} % {b} = ",a%b)
    elif(opcion == 6):
        print(f"{a} ^ {b} = ",a**b)
    elif(opcion == 0):
        salida = True
    else:
        print("opción invalida")
```

En este menú, solo cuando ingresamos la opción 0 la variable `salida` cambia de `False` a `True` por lo que el ciclo se repetirá cuantas veces queramos.

Cuando conocemos la cantidad de iteraciones deseadas tenemos otra alternativa mucho más simple de implementar, ya que no debemos controlar manualmente el estado de las variables, ni garantizar que la condición de parada se cumpla en algún momento (Para evitar los ciclos infinitos) Esta alternativa es la sentencia `for`.

1.2 Ciclo For

En todo problema que necesite ser solucionado usando ciclos se puede implementar la estructura `while`, que es la estructura general para la solución de problemas cíclicos. Sin embargo, cuando conocemos la cantidad de iteraciones que requiere el programa, es más conveniente usar la estructura `for`, su uso es más limitado, pero cuando se puede implementar permite que se ahorren dos instrucciones que incluye ciclo `while`.

Esta estructura permite que una o más instrucciones se repitan cero o más veces mientras el valor de una variable crece o decrece a razón de un valor constante, en este caso, la variable contador. La diferencia entre esta clase de ciclos y la estructura `while` es que `for` maneja la inicialización del

contador y su incremento en forma automática, como veremos en su sintaxis:

```
for contador in range(inicio,fin,variación):
    instrucciones
#fin ciclo for
```

las instrucciones `in range` se encargan de asignarle un valor al contador cada que se inicia una iteración, a su vez, `range` es la función encargada de crear el conjunto de valores que necesitamos dentro del ciclo, estos serán siempre números enteros, e irán desde el valor que definamos en `inicio` hasta `fin` **sin incluir directamente este último** y con una separación entre cada valor dada por `variación`, cuyo valor por defecto es de `1`, es decir que el ciclo aumentará uno a uno los valores del contador (ej: $0, 1, 2, 3, 4, 5, ...$), en caso de no especificar este dato, `range` asumirá este valor por defecto.

Si el valor de `fin`es menor a `inicio` el ciclo será decreciente, sin embargo, se debe definir un incremento negativo en el parámetro `variación`, por ejemplo `-1`. La variable `contador` no necesita ser inicializada en ningún valor antes del ciclo y una vez finalice el ciclo, esta conservará el último valor que recibe de `range`; si utilizamos la misma variable en otro ciclo, su valor interno se reiniciará con el primer dato que le asigne la función `range`.

Veamos qué imprimen distintos ciclos según su definición de range:

[ ]:
```
print('range(0,10)')
#aquí range generará los números del 0 al 9
for i in range(0,10):
    print(i,end=" ")

print('\n\nrange(15,5,-1)')
#se generarán los números del 15 al 6 en modo decreciente
for i in range(15,5,-1):
    print(i,end=" ")

print('\n\nrange(0,20,2)')
#se generan los núemros de dos en dos, del 0 hasta "dos" antes del 20, es decir␣
 ↪18
for i in range(0,20,2):
    print(i,end=" ")

print('\n\nrange(10,100,10)')
#se generan los núemros de diez en diez, del 10 hasta "diez" antes del 100, es␣
 ↪decir 90
for i in range(10,100,10):
    print(i,end=" ")
```

[ ]:
```
for i in range(0,10):
    print(i)
print("fin ciclo",i)
```

1.3 Instrucciones útiles para ciclos:

1.3.1 `break`

Esta sentencia nos permite interrumpir un ciclo. Veámoslo con un ejemplo, un número es primo si es divisible únicamente por sí mismo y por 1 (recordemos que el 1 no es primo). Podemos implementar esta verificación con un ciclo y una operación de módulo, si el módulo del número por el contador es igual a 0, el número es divisible por el contador, por lo tanto, no hace falta verificar los demás números pues ya sabemos que nos es primo.

[ ]:
```
%%debug_cell_with_pytutor

n = 77 #cambia este valor varias veces y ejecuta el bloque
sw = 0 #esta variable nos permitirá verificar al final si el número es primo
for i in range(2,n):
    if n % i == 0:
        print(f"{n} es divisible por {i}, no es primo")
        sw = 1 #como no es primo, cambiamos el valor
        break
if sw == 0: #si continúa siendo 0 quiere decir que la condición no se cumplió
    print(f"{n} es primo")
```

1.3.2 `continue`

Esta sentencia nos permite saltarnos una iteración completamente, ignorando las instrucciones que pudiera encontrar luego de dicha sentencia, es decir, una vez que el intérprete encuentre la instrucción, dará por terminada la iteración y regresará al inicio del ciclo para verificar si se deben realizar más iteraciones. En el ciclo `while` debemos tener cuidado con esta sentencia, si ponemos el continue antes de actualizar una condición de parada, o el contador, crearemos un ciclo infinito. En los siguientes bloques de código veremos este comportamiento, Python Tutor nos alerta del ciclo infinito en el segundo bloque, ya que esta herramienta no nos permite ejecutar más de 1000 instrucciones.

[ ]:
```
%%debug_cell_with_pytutor

for i in range(0,5):
    if(i == 3):
        continue
    print(i)
```

[ ]:
```
%%debug_cell_with_pytutor
i =  0
while i < 5:
    if(i == 3):
        continue
    print(i)
    i+=1
```

1.4 Ciclos anidados

Un ciclo anidado es un ciclo dentro del cuerpo del ciclo externo. El ciclo interno o externo puede ser de cualquier tipo, como un ciclo `while` o un `for`. Por ejemplo, el ciclo exterior `for` puede contener un `while` y viceversa.

El ciclo externo puede contener más de un ciclo interno. No hay ninguna limitación en el encadenamiento de ciclos. En el ciclo anidado, el número de iteraciones será igual al número de iteraciones del ciclo exterior multiplicado por las iteraciones del ciclo interior.

En cada iteración del ciclo exterior el ciclo interior ejecuta todas sus iteraciones. En cada iteración de un ciclo exterior el ciclo interior vuelve a empezar y completa su ejecución antes de que el ciclo exterior pueda continuar con su siguiente iteración.

Los ciclos anidados se utilizan normalmente para trabajar con estructuras de datos multidimensionales, como la impresión de matrices bidimensionales o la iteración de una lista que contiene una lista anidada (Esto lo veremos más adelante)

```
# ciclo for externo
for contador1 in range(ini1,fin1):
   # ciclo for interno
   for contador2 in range(ini2,fin2):
       cuerpo ciclo interno
   cuerpo ciclo externo
```

Es importante no usar el mismo contador en ambos ciclos, de lo contrario, se experimentarán comportamientos extraños en el código, por ejemplo, te podrías saltar algunas iteraciones en el ciclo u obtendrás resultados imprevistos al operar con el contador internamente.

Usemos el ejemplo anterior para ver este comportamiento, creemos un ciclo que nos entregue los números primos hasta un número dado. > si quieres ver su comportamiento en Python Tutor, agrega la siguiente línea al inicio del bloque:

```
%%debug_cell_with_pytutor
```

[ ]:
```
limite = 100 #cambia este valor varias veces y ejecuta el bloque
for i in range(2,limite):
    sw = 0
    for j in range(2,i):
        if i % j == 0: #nuestro contador i será el numero a verificar si es␣
 ↪primo
            sw = 1
            break
    if sw == 0:
        print(i, end=" ")
```