Facultad de Ciencias de la Administración

TECNICATURA UNIVERSITARIA EN DESARROLLO WEB

PROGRAMACIÓN I

Unidad V – Estructuras de Control

Estructuras de control repetitivas/iterativas

Tecnicatura Universitaria en Desarrollo Web

Facultad de Ciencias de la Administración

Universidad Nacional de Entre Ríos

- **Objetivos**
  - Identificar las distintas alternativas de las que se dispone para controlar el flujo de ejecución de programas.
  - Entender como hacer combinaciones de las mismas.
  - Comprender cómo pueden diseñarse/documentarse algoritmos a través de diagramas.
- **Temas a desarrollar:**
  - Ejecución Secuencial de sentencias.
  - Estructuras de control condicionales: if, elif, else, match.
  - Estructuras de control iterativas: while, for, for in range, break, continue. Análisis de eficiencia.
  - Diagramas de flujo.

Repetición/iteración

- Las computadoras comúnmente se utilizan para automatizar **tareas repetitivas** dado que **repetir tareas idénticas** o **similares** sin cometer errores es algo que a las computadoras hacen excelente.
- *En programación, se llama **iteración** a la habilidad de ejecutar un bloque de sentencias de manera repetida.*
- Hay dos tipos de iteraciones:
  - **Iteración indefinida:** el bloque de código se ejecuta hasta que se cumple alguna condición.
    - En **Python** con un bucle `while`
  - **Iteración definida:** el número de repeticiones se especifica explícitamente de antemano.
    - En **Python** con un bucle `for`

Re-asignación e incremento/decremento

- Vimos que es posible **asignar** más de una vez valores a una misma variable.
- Una nueva asignación hace que la misma variable **haga referencia** a un **nuevo valor** y **deje de referenciar** al **viejo**.

```
» x = 5
» y = 7
» x
5
» x = y
» x
7
```

- La primera vez que mostramos **x**, tiene valor **5**, luego su valor es **7**.
- Actualización de variables
  - Un tipo común de re-asignación es la actualización, donde el nuevo valor depende del anterior.

```
» x = x + 1
```

  - Esto significa *"al valor actual de x sumarle uno, y luego actualizar x con el nuevo valor"*.
  - Incrementar en uno a una variable se llama ***incremento*** y restarle uno se llama ***decremento***.

Operadores de asignación

Operador	Descripción
=	Asigna el valor de la expresión derecha
+=	Asigna el valor de la expresión derecha sumada al valor de la variable a la izquierda.
-=	Asigna el valor de la expresión derecha restada al valor de la variable a la izquierda.
/=	Asigna el valor de la expresión derecha dividida al valor de la variable a la izquierda.
*=	Asigna el valor de la expresión derecha multiplicada al valor de la variable a la izquierda
Otros: **=, &=, |=	

Bloque while

- La primer estructura de control iterativa que veremos es el bloque **while**.
- A continuación mostramos como podemos hacer una cuenta descendente de números del 10 al 1.

```
def cuenta_descendente(n):
    while n > 0:
        print (n)
        n = n - 1
    print (”It's the final countdown!!! Tarata taaaaa!!!”)
```

- Casi que se puede leer la instrucción **while** como que si fuese en inglés:
  - *"Mientras n sea mayor que 0 mostrar el valor de n y luego decrementar n. Cuando se llegue a 0 mostrar It's the final countdown"*.

Bloque while (2)

- Más formalmente, aquí el flujo de ejecución del bloque `while`:

  1) Determinar si la condición es `True` o `False` (verdadera o falsa).
  2) Si es `False` (falsa), salir del `while` y continuar con la ejecución de la siguiente sentencia.
  3) Si la condición es `True` (verdadera), ejecutar el cuerpo y luego volver al paso 1.

- El tipo de flujo es llamado ***bucle*** porque el tercer paso vuelve hacia atrás al primer paso.
- El cuerpo del bucle **tiene que cambiar el valor de una o más variables** para que la **condición** se vuelva **falsa** en algún momento y **termine el bucle**.
- En caso que no fuese así, el bucle se repetirá por siempre, lo que se llama **bucle infinito.**

Bloque while (3)

- En el caso de **`cuenta_descendente()`** podemos determinar que el bucle finaliza siempre que **`n`** sea un entero positivo, en cada paso se vuelve cada vez más pequeño hasta llegar a **`0`**.
- Para otros bucles no es fácil saber. Por ejemplo:

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0: # n es par
            n = n / 2
        else: # n es impar
            n = n * 3 + 1
```

Bloque while (4)

- La condición para este bucle es `n != 1`, así que si el bucle continua hasta que `n` es `1`.
- Con cada iteración, el programa imprime el valor de **n** y luego controla si es par o impar. Si es par, **n** es dividido por **2**. Si es impar, el valor de **n** es reemplazado por **n * 3 + 1**. Por ejemplo, si el argumento pasado a `sequence` es **3**, los resultados son **3, 10, 5, 16, 8, 4, 2, 1**.
- Dado que **n** a veces se incrementa y a veces se decrementa, no hay una prueba obvia que **n** en algún momento alcanzará el valor 1 y termine. Para algunos valores particulares de **n**, podemos probar que termina. Por ejemplo, si el valor de inicio es una potencia de dos, entonces el valor de **n** será par cada vez que se pasa a través del bucle, hasta que lleguemos a 1.
- La pregunta difícil de contestar es como podemos probar que termina para todos los valores positivos de **n**.
- Hasta ahora, nadie ha sido capaz de afirmarlo o negarlo: https://en.wikipedia.org/wiki/Collatz_conjecture

Rangos o range

- *Los **rangos** son secuencias de números enteros no modificables predefinidas. Se crean utilizando la función* **`range`***, que también determina el tipo de dato.*

- Así **`range (4)`** genera la siguiente salida:

```
print(list(range(4)) # después vemos que es list
0
1
2
3
```

- Un **rango** genera una secuencia de números que van desde **`0`** por defecto hasta el número que se pasa como parámetro menos **`1`**.
  - En realidad, se pueden pasar **hasta tres parámetros** separados por coma, donde el primero es el **inicio** de la secuencia, el segundo el **final** y el tercero el **salto** que se desea entre números.
  - Por defecto se empieza en **`0`** y el salto es de **`1`**. Cabe destacar que los índices pueden ser negativos y tan grandes como se desee.
- Por lo tanto, si hacemos **`range(5,20,2)`**, se generarán números de **`5`** a **`19`** (inclusive) de dos en dos.

Rangos o range (2)

- Siendo **`rang1`** un **rango** y **`num`** un entero estas son las operaciones soportadas por los rangos:
  - **`rang1.count(num)`**: indica cantidad de ocurrencias de **`num`** en el rango. Siempre el resultado es **`1`** o **`0`**.
  - **`rang1.index(num)`**: ubicación de **`num`** en el rango
  - **`rang1.start`**: valor de inicio del rango
  - **`rang1.step`**: permite consultar el paso numérico usado en el rango
  - **`rang1.stop`**: permite consultar el número en el que el rango dejará de generar nuevos números.
- Los **rangos** son inmutables y presentan la gran ventaja de que son iteradores (concepto que veremos más adelante en profundidad), por lo que, cuando se crean, no guardan la información que representan, sino el procedimiento necesario para generar la secuencia de números. Esto hace que ocupen muy poco espacio y permitan controlar la memoria utilizada en los programas, ya que van generando los valores de uno en uno y no todos a la vez.

Rangos o range (3)

- Se puede acceder a un elemento en concreto del rango utilizando un índice y usar operaciones como **len()** o **in** e incluso usar selecciones de subsecuencias.
  - » rango = range (0, 24, 2)
  - » rango
  - range (0, 24, 2)
  - » 8 in rango
  - True
  - » 7 in rango
  - False
  - » rango[3]
  - 6

Bucle for

- El bucle **for** de **Python** tiene algunas particularidades comparado con otros lenguajes de comparación.
- El **for** es un tipo de bucle, similar al **while** pero con ciertas diferencias.
  - La principal es que el número de iteraciones de un **for** esta definido de antemano, mientras que en un **while** no.
  - Mientras que en el **while** la condición es evaluada en cada iteración para decidir si volver a ejecutar o no el código, en el **for** no existe tal condición, sino un iterable que define las veces que se ejecutará el código.
- En el siguiente ejemplo vemos un bucle **for** que se ejecuta 5 veces, y donde la variable i incrementa en cada iteración su valor en 1.

```
» for i in range(0, 5):
    » print(i)
```

```
0
1
2
3
4
```

Bucle for (2)

- Es preferible utilizar un bucle **for** (en vez de **while**) cuando la cantidad de iteraciones se conoce de antemano o está definida por las dimensiones de un tipo de datos que soporte operaciones de iteración.
- En **Python** se puede iterar prácticamente todo, como por ejemplo una cadena. En el siguiente ejemplo vemos como la **i** va tomando los valores de cada letra.

```
» for i in "Python":
    » print(i)
P
y
t
h
o
n
```

Break

- A veces se ejecutan algunas iteraciones de un bucle y nos damos cuenta que no es necesario seguir con las siguientes. En estas situaciones la sentencia **break** hace que la ejecución continúe inmediatamente después del bucle.
- Por ejemplo, queremos capturar los datos ingresados por el usuario hasta que escriba listo. Podríamos escribir:

```
while True:
    linea = input('Ingrese un texto o "listo" para terminar')
    if linea == 'listo':
        break
    print(linea)
print('Listo!')
```

- La condición del bucle es **True**, que siempre se va a cumplir, así que el bucle se ejecuta hasta que llega la sentencia **break**.
- Esta forma de programar los bucles **while** es común porque podemos controlar si una condición se cumple en cualquier lugar dentro del bucle (no solo al principio) y podemos expresar que se detenga afirmativamente **("Frená si pasa esto!")** en vez de negativamente **("Hacé esto hasta que pase aquello")**.

Continue

- La palabra reservada **continue** nos permite terminar la iteración actual y continuar con la siguiente.
- Por ejemplo:

```
for i in "Python":
    if i == "t":
        continue
    print(i)
```

```
P
Y
h
o
n
```

Bibliografía

- Óscar Ramírez Jiménez: ***"Python a fondo"*** 1era Edición. Ed. Marcombo S.L.. 2021.
- Allen Downey. ***"Think Python"***. 2Da Edición. Green Tea Press. 2015.
- Bill Lubanovic. ***"Introducing Python"***. 2Da Edición. O' Reilly. 2020.
- Eirc Matthes: ***"Python Crash Course"***. 1era Edición. Ed. No Starch Press. 2016.
- Zed A. Shaw: ***"Learn Python 3 the Hard Way"***. 1era Edición. Ed. Addison-Wesley. 2017.
- Web - **John Sturtz: Python "for" Loops (Definite Iteration) https://realpython.com/python-for-loop/.**