1 Iteraciones

2 Iteración usando `while`

2.1 Ejemplo de aplicación

Suponga que tenemos que ir a la casa de un/a amigo/a pero se nos olvidó traer la dirección completa y solo tenemos el nombre de la calle en la que vive. Al llegar, nos damos cuenta que es un pasaje con 4 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/a previamente, lo mejor sería simplemente ir y golpear en la primera casa, y preguntar si nuestro/a amigo/a vive ahí. En caso de que la respuesta sea negativa, tendríamos que **repetir** el proceso con las demás casas hasta encontrar la de nuestro/a amigo/a.

Si quisiéramos escribir un programa en Python para realizar este procedimiento, con las herramientas que hasta el momento hemos visto, lo primero que deberíamos hacer sería el siguiente algoritmo:

```
Ir a la primera casa
Golpear la puerta y esperar que alguien abra
Si es la casa de nuestro/a amigo/a entonces:
  Nos quedamos y dejamos de buscar
Sino:
  Ir la segunda casa
  Golpear la puerta y esperar que alguien abra
  Si es la casa de nuestro/a amigo/a entonces:
    Nos quedamos y dejamos de buscar
  Sino:
    Ir la tercera casa
    Golpear la puerta y esperar que alguien abra
    Si es la casa de nuestro/a amigo/a entonces:
      Nos quedamos y dejamos de buscar
    Sino:
      Ir la cuarta casa
      Golpear la puerta y esperar que alguien abra
      Si es la casa de nuestro/a amigo/a entonces:
        Nos quedamos y dejamos de buscar
      Sino:
        Ir la quinta casa
        Golpear la puerta y esperar que alguien abra
        Si es la casa de nuestro/a amigo/a entonces:
          Nos quedamos y dejamos de buscar
        Sino:
          Ir la sexta casa
          Golpear la puerta y esperar que alguien abra
          Si es la casa de nuestro/a amigo/a entonces:
            Nos quedamos y dejamos de buscar
```

```
Sino:
  Ir la séptima casa
  Golpear la puerta y esperar que alguien abra
  Si es la casa de nuestro/a amigo/a entonces:
    Nos quedamos y dejamos de buscar
  Sino:
    Ir la octava casa
    Esta debe ser la casa de nuestro/a amigo/a
```

Como se puede observar, escribir el procedimiento del ejemplo anterior es bastante largo, se vuelve tedioso y poco práctico, sin mencionar que solo funciona si hay ocho casas o menos.

El algoritmo presentado anteriormente podría resumirse de la siguiente manera:

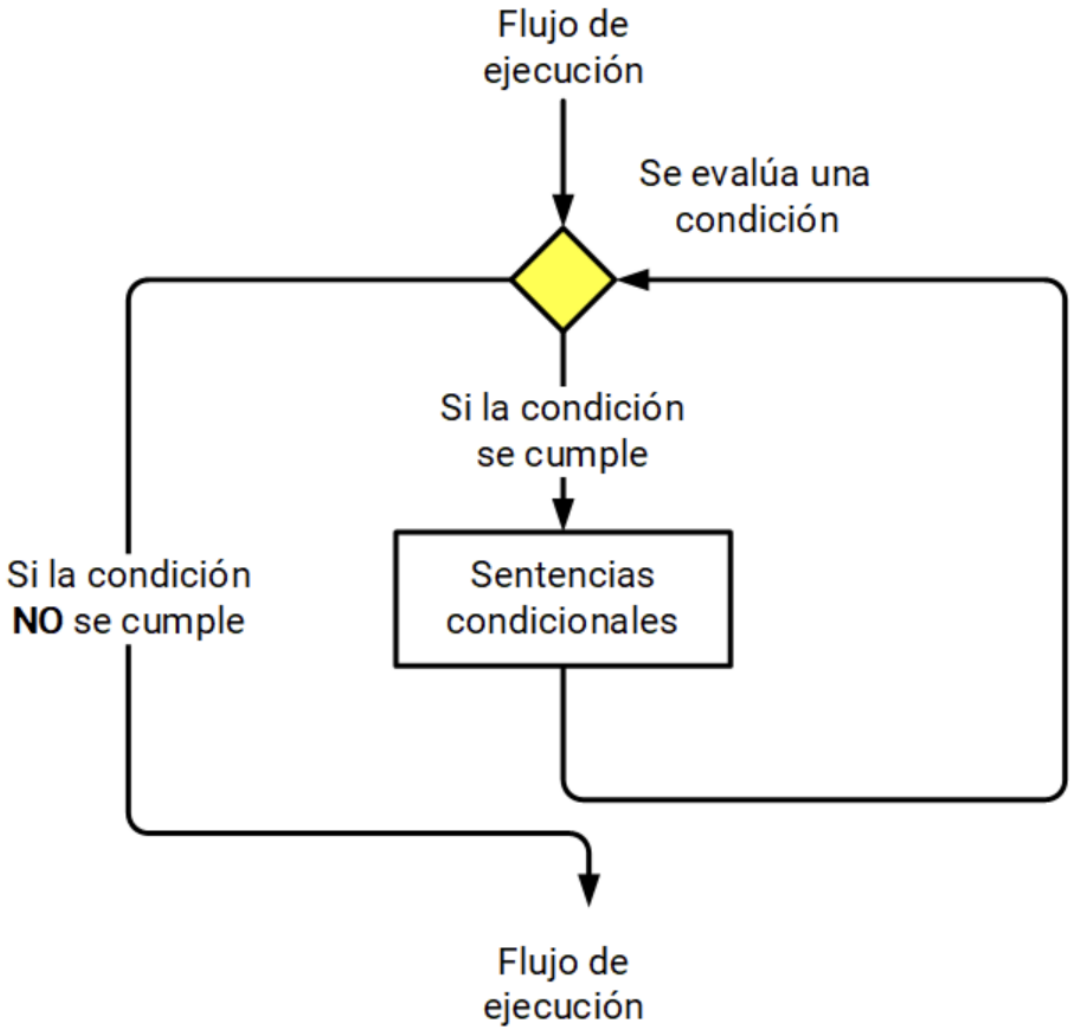
```
Mientras no encuentre a mi amigo/a:
    buscar en la siguiente casa.
    Si lo encuentro, nos quedamos y dejamos de buscar
```

Para poder resolver este y otros problemas de similares características, se presenta una herramienta que permite **repetir una y otra vez la misma sentencia**. Esta repetición recibe el nombre de **iteración**.

2.2 Sentencia iterativa `while`

En Python, al igual que en la mayoría de los lenguajes de programación, existen variados mecanismos y sentencias para conseguir que un programa realice procesos repetitivos (iterativos). En este caso en particular, utilizaremos la sentencia `while`, cuya traducción al español sería **mientras**. En particular, la idea detrás de esta sentencia de control es "*Mientras la condición se cumpla, repetir las sentencias condicionadas*"

El siguiente diagrama de flujo ilustra la ejecución de una estructura de iterativa con el uso de `while`.



Al observar el diagrama, se puede encontrar una gran similitud en la estructura del ciclo `while` y la sentencia `if`. La gran diferencia que existe es que, mientras el `if` continúa con el flujo del programa, el `while` se devuelve a la condición para evaluarla de nuevo. Esto significa que, cuando la condición del ciclo `if` se cumple, **la sentencia se ejecuta solo 1 vez**, mientras que las sentencias del ciclo `while` **se ejecutan tantas veces como sea necesario hasta dejar de cumplir la condición dada**.

Esto implica que, una vez que ejecuto todas las sentencias condicionadas dentro de un `while`, en vez de seguir con el programa, **vuelvo a verificar si la condición se cumple**.

La **sintaxis** (forma de escribir) que Python utiliza para un ciclo `while` es la siguiente:

```
<Sentencias previas>
while <condición>:
  # Se ejecuta si la condición se cumple
  <Bloque de sentencias a repetir>
# Se ejecuta al momento de dejar de cumplir la condición dada
<Bloque de sentencias fuera del ciclo>
```

Para comprender cómo funciona la sentencia `while`, se propone crear un programa en Python que muestre por pantalla la tabla de multiplicar de un número dado por el usuario (1 a 10):

Para ello, se presentarán 2 opciones para obtener lo solicitado: 1. Forma manual. 2. Mediante uso de ciclo while.

2.2.1 Forma manual

[ ]:
```
# ENTRADA
# Se solicita el número de la tabla de multiplicar a generar
factor = input("Favor ingresar un valor cuya tabla de multiplicar es requerida:␣
 ↪")

# Se realiza el cambio de tidpo de dato, en este caso en particular se␣
 ↪transforma a número entero
factor = int(factor)

# SALIDA
# Se genera la tabla de multiplicar del 1 al 10
print("1  * ", factor, "=", 1*factor)
print("2  * ", factor, "=", 2*factor)
print("3  * ", factor, "=", 3*factor)
print("4  * ", factor, "=", 4*factor)
print("5  * ", factor, "=", 5*factor)
print("6  * ", factor, "=", 6*factor)
print("7  * ", factor, "=", 7*factor)
print("8  * ", factor, "=", 8*factor)
print("9  * ", factor, "=", 9*factor)
print("10 * ", factor, "=", 10*factor)
```

```
Favor ingresar un valor cuya tabla de multiplicar es requerida: 2
1  *  2 = 2
2  *  2 = 4
3  *  2 = 6
4  *  2 = 8
5  *  2 = 10
6  *  2 = 12
7  *  2 = 14
8  *  2 = 16
9  *  2 = 18
10 *  2 = 20
```

2.2.2 Forma ciclo while

[ ]:
```
# ENTRADA
# Se solicita el número de la tabla de multiplicar a generar
factor = input("Favor ingresar un valor cuya tabla de multiplicar es requerida:␣
 ↪")
```

```
# Se realiza el cambio de tidpo de dato, en este caso en particular se␣
 ↪transforma a número entero
factor = int(factor)

# SALIDA
otro_factor = 1
while otro_factor <= 10:
  print(otro_factor, "*", factor, "=", otro_factor*factor)
  otro_factor = otro_factor + 1
```

```
Favor ingresar 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
```

Estos dos programas resuelven el problema solicitado. No obstante, la primera solución usa la sentencia `print` tantas veces como se requiere la información por pantalla, en este caso en particular son 10 veces. En contraparte, el programa desarrollado con el ciclo `while` solo usa la sentencia `print` una vez.

Lo que sucede es que la diferencia mencionada se debe a la **sintaxis utilizada**; **semánticamente** ambos programas **ejecutan** la sentencia `print` 10 veces. En la primera resolución, las 10 sentencias `print` aparecen explícitamente en el programa y en la segunda resolución, aparece una sola vez la sentencia explícita, pero **se repite 10 veces**. La repetición de las sentencias está **condicionada** al cumplimiento de la condición del `while`.

La condición que acompaña al ciclo es la siguiente:

```
while otro_factor <=10:
```

Siendo el valor inicial de la variable `otro_factor` igual a 1

Si se observa la última línea (dentro `while`), se puede notar que la variable `otro_factor` aumenta en una unidad cada vez el ciclo se cumple.

2.2.3 Ejercicio `while`

Para poder ejercitar y comprender de mejor manera el funcionamiento del ciclo `while`, se propone realizar un programa que resuelva la suma de los primeros **n** números naturales, utilizando en la resolución ciclos iterativos.

Para poder desarrollar lo solicitado, se debe realizar un algoritmo de solución. Para facilitar el trabajo, se considera que $\mathbf{n} = \mathbf{5}$:

1. Se deben sumar los 2 primeros números (1+2)
2. Luego, la suma obtenida se debe agregar 3 (1+2+3)
3. Luego, la suma obtenida se debe agregar 4 (1+2+3+4)
4. Luego, la suma obtenida se debe agregar 5 (1+2+3+4+5)

Con lo anterior se pueden sacar las siguientes conclusiones:

- Es necesario definir una variable que indique el número final a la que se desea llegar la sumatoria
- Es necesario definir una variable que indique el número actual que se requiere sumar
- Es necesario definir una variable que guarde el valor de las sumas que se vayan realizando

Dicho de otro modo, el código quedaría expresado de la siguiente manera:

[ ]:
```
# ENTRADA
# Se define la variable que toma el valor final de la sumatoria (se solicita el␣
 ↪valor por teclado)
numero_final = input("Favor ingresar valor final de la sumatoria a realizar: ")

# PROCESAMIENTO
# Se realiza el cambio de tipo de dato, en este caso se transforma a número␣
 ↪entero
numero_final = int(numero_final)

# Se define la variable con el número actual que se desea sumar (irá cambiando␣
 ↪en cada paso del ciclo)
i = 1

# Se define la variable que guardará las sumas parciales que se realizarán.␣
 ↪Siempre se debe definir con
# el neutro de la operación a trabajar
suma_parcial = 0

# Se plantea la condición del ciclo iterativo con el que se desea trabajar␣
 ↪(recordar que debe estar
# la variable que modificará su valor)
# En este caso, la variable "i" irá cambiando su valor hasta llegar al valor␣
 ↪final
while i <=numero_final:
  suma_parcial = suma_parcial + i
  i = i + 1

# SALIDA
# Se genera el mensaje con lo solicitado en el enunciado
print("La sumatoria de los primeros", numero_final, "números naturales es",␣
 ↪suma_parcial)
```

```
Favor ingresar valor final de la sumatoria a realizar: 6
La sumatoria de los primeros 6 números naturales es 21
```

El programa desarrollado anteriormente puede ser explicado de la siguiente manera:

La variable `numero_final` recibe como entrada el valor ingresado por el usuario (`numero_final = 5`)

Se define la variable `i`, la cual irá tomando los valores de 1 hasta n (en este caso la variable `i` tomaría los valores de 1, 2, 3, 4, 5)

Se define la variable `suma_parcial`, la cual se utiliza como **acumulador**. Su función es ir guardando los valores de las sumas parciales, hasta llegar a obtener el valor esperado.

A continuación, se muestra una tabla con los valores de las variables utilizadas en el programa anterior:

Variable/Valor	Iteración					
	Inicio	1	2	3	4	5
numero_final	5	5	5	5	5	5
suma_parcial	0	1	3	6	10	15
i	1	2	3	4	5	5

La tabla anterior muestra las variables y los valores que van tomando durante la ejecución del programa. Se puede ver que hay valores que se mantienen en cada iteración y otros que cambian debido al flujo de ejecución del programa.

Esto se conoce como la **traza** de un programa y es donde se muestran los valores que van tomando las variables durante la ejecución del código. Es de mucha importancia al momento de programar, puesto que es la mejor forma para saber si el programa realizado va cumpliendo con lo requerido durente cada paso.

La realización de las trazas es fundamental para lograr un programa cumpla con todo lo estipulado, pero no solo al finalizar el programa, sino que se recomienda realizarla durante el avance del desarrollo del código. De esa manera se puede trabajar con mayor seguridad y no se revisan los errores solo al finalizar el trabajo.

Una marca de un buen programador es que este puede predecir con certeza el comportamiento de su código sin ejecutarlo. Pues esto implica que: * Es capaz de identificar el funcionamiento de cada sentencia individual dentro del código. * Es capaz de combinar el flujo del programa para identificar cuál es la salida coherente con las entradas dadas.

2.2.4 Tautología

Cuando se trabaja con ciclos iterativos del tipo `while`, es necesario que exista alguna instrucción dentro del ciclo que modifique de alguna manera la condición a la que está sujeta la iteración, de manera que en algún momento, cuando corresponda, la condición sea **Falsa** (`False`) o **deje de cumplirse**.

A continuación se propone mostrar por pantalla los primeros 10 números naturales:

[ ]:
```
i = 1
while i <= 10:
  print(i)
```

En el ejemplo anterior se dio una condición que nunca cambia, puesto que, dentro de las sentencias a repetir, la variable `i` no cambia de valor. Por consiguiente, estamos en presencia de un **ciclo infinito**.

Para poder resolver lo solicitado, de forma correcta, se añade la siguiente línea al final del ciclo:

```
i = i + 1
```

[ ]:
```
i = 1
while i <= 10:
  print(i)
  # Nótese que se añadió *dentro* del ciclo, no posterior a este
  i = i + 1
```

```
1
2
3
4
5
6
7
8
9
10
```

Solo se modifica una línea en el desarrollo del código, pero esto basta para poder resolver lo solicitado.

Se debe tener en cuenta que 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 deje de cumplirse en algún momento determinado.

Cuando un programa queda en un ciclo infinito, la única forma de que termine es forzarlo a que lo haga. Esto se puede hacer presionando en la consola la combinación de teclas `Ctrl + c`.

3 Iteración usando `for-in`

Existen diversas herramientas para realizar repeticiones en Python. Previamente conocimos la estructura de control `while`, la que nos permite realiza repeticiones del código considerando el cumplimiento de una condición.

Si bien `while` es la forma más transparente y fácil de seguir, añadiremos otra herramienta para hacer este tipo de construcciones: el ciclo `for-in`.

Es importante detallar que si bien todas la aplicaciones de `for-in` pueden hacerse con `while`, el caso inverso no es tan fácil. Es decir, traducir `while` a `for-in` no siempre es tan directo.

3.1 Sintaxis `for-in`

El `for-in` esta pensado especialmente para iterar sobre objetos compuestos. Por ello, las iteraciones son realizadas mediante el uso de un **identificador**, el cual es una variable que irá tomando los valores de los elementos del objeto en cada paso.

En cada paso, esta variable toma el valor del **elemento iterable** del objeto. Como siempre, para no generar confusión, es recomendable que el identificador sea definido con un nombre representativo.

La sintaxis que utiliza el `for-in` es la siguiente:

```
for <identificador> in <elemento_iterable>:
    <operaciones_a_realizar>
<sentencias siguientes>
```

Para poder utilizar esta herramienta se debe conocer su estructura:

- **Elemento iterable**: es un tipo de dato que está compuesto por varios elementos. Ya hemos visto que un `string` tiene estas características, pues está compuesto por caracteres. Adicionalmente en Python existen otros que veremos más adelante, como **listas** y **archivos**, y otros que quedan fuera del alcance del curso como tuplas, conjuntos, clases propias, diccionarios, etc.
- **Identificador**: corresponde a una variable que toma el valor de la unidad por la que se recorre el dato iterable. En el caso de los strings, sería un caracter, sin embargo, para listas y archivos, la unidad por la que se itera es el elemento de la lista y la línea de texto respectivamente.

La ventaja de trabajar con `for-in` es que el identificador cambia de manera automática su valor en cada paso del ciclo, tomando ordenadamente los datos desde el primer hasta el último elemento del objeto iterable, en orden.

Para apreciar la diferencia entre ambas iteraciones, se muestra un ejemplo donde se requiere mostrar por pantalla cada caracter de un `string`. El Ejemplo 1 muestra su implementación usando while.

[ ]:
```python
# Ejemplo 01: Implementación con while
texto = 'Esto es Python con iteraciones'

i = 0
while i < len(texto):
  print(texto[i])
  i = i + 1
```

```
E
s
t
o

e
s

P
y
t
```

```
h
o
n

c
o
n

i
t
e
r
a
c
i
o
n
e
s
```

Por otro lado, el Ejemplo 2, muestra otra solución, usando esta vez un `for-in`.

[ ]:
```python
# Ejemplo 02: Implementación con for-in
texto = 'Esto es Python con iteraciones'

for c in texto:
  print(c)
```

```
E
s
t
o

e
s

P
y
t
h
o
n

c
o
n

i
t
```

```
e
r
a
c
i
o
n
e
s
```

Al analizar el código del **Ejemplo 2**, ocurre lo siguiente: * La variable **c** toma el primer valor del string en el primer caso `E` mayúscula. * Luego se imprime el valor almacenado en la variable. * Considerando que no existen más instrucciones dentro del `for-in`, la variable **c** automáticamente cambia su valor y avanza a la posición siguiente de la lista. * Este proceso se continúa realizando hasta que la variable **c** alcance el último valor de la lista.

3.2 Diferencias entre ciclos `while` y `for-in`

La diferencia principal entre ambas construcciones es que: * `while` suele requerir del uso de un **iterador** explícito para ir accediendo a las posiciones del dato iterable, es decir, requiere que **explícitamente** actualicemos una condición. * `while` permite la construcción de iteraciones más complejas (dividiendo el número por 2, esperando una entrada de usuario específica, mientras todavía queden elementos en una lista, entre otras). * `for-in` recorre todos los elementos del dato iterable, lo que permite iterar de forma implícita, es decir, sin indicar abiertamente cuando inicio y cuando paro. * `for-in` tiene problemas si el objeto que estoy actualizando es modificado dentro de la iteración.

3.3 Consideraciones del ciclo `for-in`

Cuando se realizan iteraciones mediante el uso de ciclos `for-in`, el identificador toma todos los valores del objeto iterable, pero no modifica el elemento dentro del objeto iterable, por lo que solamente captura el valor de la variable, pero no puede actualizarla.

Es decir, **cualquier cambio a algún elemento recorrido no será transferido al objeto.**

Por ejemplo, consideremos el ejemplo 3.

[ ]:
```python
# Ejemplo 03:
texto = 'Esto es Python con iteraciones'

for c in texto:
  c = c.upper()
  print(c)

print(texto)
```

```
E
S
T
O
```

```
E
S

P
Y
T
H
O
N

C
O
N

I
T
E
R
A
C
I
O
N
E
S
Esto es Python con iteraciones
```

Si bien en el código explícitamente existe un cambio del valor de c en la línea `c = c.upper()`, podemos ver que este jamás afectó a la variable texto. Esto es porque cualquier cambio que hagamos en c, no se devuelve jamás a la variable texto.

Esto aplica tanto para objetos inmutables como strings y tuplas, como para aquellos que si pueden modificarse, como las listas.

Adicionalmente, cuál será la unidad de iteración depende única y exclusivamente del tipo de dato; el identificador que escojamos para iterar y el nombre que le pongamos no modifica el elemento por el cuál se iterará. Por ejemplo:

[ ]:
```python
# Ejemplo 04:
texto = 'Esto es Python con iteraciones'

for palabra in texto:
  print(palabra)
```

```
E
s
t
o
```

```
e
s

P
y
t
h
o
n

c
o
n

i
t
e
r
a
c
i
o
n
e
s
```

En este caso, el nombre `palabra` del identificador nos hubiese hecho sospechar que el iterador tomaría cada palabra como elemento y nos entregararía:

```
Esto
es
Python
con
Iteraciones
```

Sin embargo, en la práctica, el iterar sobre un `string` siempre se hará caracter a caracter. Si quisiera que itere sobre palabras, sílabas o cualquier unidad distinta, estaría obligado a usar una solución más compleja.

Para evitar este tipo de confusiones respecto a la naturaleza del dato con el cual se está trabajando en el `for-in`, se recomienda que el **iterador**, sea lo más representativo posible de la unidad base sobre la cuál se está iterando.

Por ejemplo: * Objeto iterable **string** –> iterador = **caracter** * Objeto iterable **lista** –> iterador = **elemento** * Objeto iterable **archivo** –> iterador = **línea**

Cabe señalar que lo anterior es solo recomendación para no generar confusiones, puesto que independiente del nombre del iterador, Python siempre considerará la **unidad** respectiva según el tipo de dato.

3.4 Función `range()`

Una de las formas más comunes para usar el `for-in`, aparece cuando conocemos la función nativa `range()`. Esta función crea un **objeto** iterable que podemos recorrer según sus parámetros:

- `range(inicio, fin, salto)`: Es el uso más general.
  - `inicio`: Corresponde al primer valor del objeto iterable.
  - `fin`: Corresponde al valor hasta el que esperamos llegar. Este parámetro no puede omitirse y `range()` creará objetos iterables hasta `fin - 1` (en general, `fin` siempre es estrictamente mayor que el último elemento). Es decir, si invoco `range(1, 10)`, crearé un objeto iterable con los valores `1, 2, 3, 4, 5, 6, 7, 8 y 9`, sin incluir el 10.
  - `salto`: Indica de cuánto es el salto entre cada elemento del iterable. Este parámetro es opcional y si se omite, se asume que el valor de inicio será 1.
- `range(fin)`: Esta versión, que omite los parámetros `inicio` y `salto`, es equivalente a usar `range(0, fin, 1)`.

Los ejemplos 5, 6 y 7 muestran distintos escenarios en los que podríamos usar la función `range()`.

[ ]:
```
# Ejemplo 05: Iterando solo con el operador de fin

for e in range(15):
  aux = e ** 2
  print(e,'** 2 :', aux)
```

```
0 ** 2 : 0
1 ** 2 : 1
2 ** 2 : 4
3 ** 2 : 9
4 ** 2 : 16
5 ** 2 : 25
6 ** 2 : 36
7 ** 2 : 49
8 ** 2 : 64
9 ** 2 : 81
10 ** 2 : 100
11 ** 2 : 121
12 ** 2 : 144
13 ** 2 : 169
14 ** 2 : 196
```

[ ]:
```
# Ejemplo 06: Iterando con operador de inicio y fin

for e in range(20, 30):
  aux = e ** 2
  print(e,'** 2 :', aux)
```

```
20 ** 2 : 400
21 ** 2 : 441
22 ** 2 : 484
23 ** 2 : 529
```

```
24 ** 2 : 576
25 ** 2 : 625
26 ** 2 : 676
27 ** 2 : 729
28 ** 2 : 784
29 ** 2 : 841
```

[ ]:
```python
# Ejemplo 07: Iterando con operador de inicio, fin y salto

for e in range(0, 100, 5):
  aux = e ** 2
  print(e,'** 2 :', aux)
```

```
0 ** 2 : 0
5 ** 2 : 25
10 ** 2 : 100
15 ** 2 : 225
20 ** 2 : 400
25 ** 2 : 625
30 ** 2 : 900
35 ** 2 : 1225
40 ** 2 : 1600
45 ** 2 : 2025
50 ** 2 : 2500
55 ** 2 : 3025
60 ** 2 : 3600
65 ** 2 : 4225
70 ** 2 : 4900
75 ** 2 : 5625
80 ** 2 : 6400
85 ** 2 : 7225
90 ** 2 : 8100
95 ** 2 : 9025
```

3.4.1 Consideraciones del `range()`

Al usar **`range()`**, vale la pena considerar: * La función está definida solo para valores enteros, por lo que tanto los valores de **`inicio`**, **`fin`** y **`salto`** siempre deben ser de tipo **`int`**. * La función acepta valores negativos, por lo que podría hacer llamados del tipo **`range(100, -100, -10)`**. * Dependiendo de la situación, es posible que range() entregue un iterador vacío. Por ejemplo **`range(100, -100, 10)`** no produce ningún elemento, pues para Python es imposible alcanzar el -100 de **`fin`** con saltos positivos de 10, si es que el **`inicio`** es 100.

Finalmente, vale la pena señalar que **`range()`** no es la única función generadora que existe. Existen múltiples utilidades para generar elementos iterables, algunos nativos de Python, como los del módulo **`itertools`** o algunas funciones nativas, y otros alojados en módulos especializados como **`numpy`**.

En particular, pese a que no veremos numpy en este curso, siempre es bueno conocer las funciones

generadoras `np.linspace()`, `np.arange()`, `np.zeros()` y `np.ones()`.

```
[ ]: list(range(100,-100,-10))
```

```
[ ]: [100,
      90,
      80,
      70,
      60,
      50,
      40,
      30,
      20,
      10,
      0,
      -10,
      -20,
      -30,
      -40,
      -50,
      -60,
      -70,
      -80,
      -90]
```

4 Bibliografía

4.1 `while`

GeeksforGeeks. (2021, 25 agosto). Python While Loop. Recuperado 4 de agosto de 2022, de https://www.geeksforgeeks.org/python-while-loop/

Shaw, Z. (2017). While loops. En *Learn Python 3 the Hard Way* (p. 116). Addison-Wesley.

4.2 `for-in`

Ceder, V. L. (2010). The for loop. En *The Quick Python Book* (2.a ed., p. 92). Manning Publications.

GeeksforGeeks. (2022, 14 julio). Python For Loops. Recuperado 4 de agosto de 2022, de https://www.geeksforgeeks.org/python-for-loops/

Python Software Foundation. (2022, 4 agosto). *More Control Flow Tools*. Python 3.10.6 documentation. Recuperado 4 de agosto de 2022, de https://docs.python.org/3/tutorial/controlflow.html#for-statements

Shaw, Z. (2017). Loops and Lists. En Learn Python 3 the Hard Way (p. 112). Addison-Wesley.

4.3 `range()`

W3Schools. (2022). Python range() Function. Recuperado el 20 de marzo de 2023, de: https://www.w3schools.com/python/ref_func_range.asp

GeeksforGeeks (2022, 20 octubre). Python range() function. Recuperado el 20 de marzo de 2023, de https://www.geeksforgeeks.org/python-range-function/

4.4 Formato y buenas prácticas

Van Rossum, G., Warsaw, B., & Coghlan, N. (2001, 21 julio). *PEP 8 – Style Guide for Python Code*. Python Enhancement Proposals. Recuperado 1 de agosto de 2022, de https://peps.python.org/pep-0008/

4.5 Trazas

Beecher, K. (2017). Anticipating and Dealing with Errors. En *Computational Thinking* (p. 87). BCS.