Python IV: Bucles

Los bucles son conjuntos de instrucciones que se ejecutan repetidamente. Como en la mayoría de los lenguajes, existen dos tipos de bucles: **`for`** y **`while`**, ambos con sintaxis muy sencilla. Veamos dos ejemplos.

```
x = (1, 2, 3)

for i in x:

     print(i, 2**i)

> 1 2

> 2 4

> 3 8
```

```
i = 1

while i < 4:

     print(i, 2**i)

     i += 1

> 1 2

> 2 4

> 3 8
```

Como ya dijimos, se puede iterar sobre los strings también. En el ejemplo de arriba, x podría perfectamente ser un string (¡excepto por la operación matemática!). Existen distintas maneras de iterar sobre diccionarios:

```
for key in diccionario:

     print(key, diccionario[key])

> nombre Andrea

> apellido Zapata

> nota 7.0

> ranking 1

for key, value in diccionario.items():
```

```
        print(key, val)
```

```
> nombre Andrea
```

```
> apellido Zapata
```

```
> nota 7.0
```

```
> ranking 1
```

(Recuerden usar la tecla TAB para explorar las posibilidades, tanto en IPython como en Jupyter, así como en cualquier IDE). Fíjense que en el segundo caso, en cada iteración el iterable nos entrega *dos* variables. Esta es otra expresión del hecho que una variable en python puede contener elementos de distinto tipo.

Tal como en Bash, podemos usar declaraciones break y continue para escapar un bucle o saltar una iteración (pueden comparar este ejemplo con el equivalente que vimos en Bash):

```
var = 0
while var < 100:
        var += 5
        if var == 10:
                continue
        print(var)
        if var in (10, 25):
                break
```

```
> 5
```

```
> 15
```

```
> 20
```

```
> 25
```

En Python podemos anidar cláusulas simplemente indentando progresivamente. Además, podemos usar **`else`** como un bloque de “término exitoso” de un bucle, que se ejecuta solo si el ciclo **no** terminó con **`break`**:

```
x = [0, 1, 2, 3, 4]
for i in x:
        if i > 5:
```

```
            print(f'Encontré {i} > 5')
            break
    else:
        print('No hay números mayores que 5')
    > No hay números mayores que 5
```

Vamos a aprovechar de introducir algunos objetos útiles para bucles:

- **`range`** es un *generador* que produce un rango entre números enteros

```
for i in range(3):
    print(i)
> 0
> 1
> 2
for i in range(2, 20, 5):
    print(i)
> 2
> 7
> 12
> 17
```

Podemos aprovechar `range` para obtener una lista de números ordenados:

```
list(range(5))
> [0, 1, 2, 3, 4]
```

- **`enumerate`** entrega un índice para cada elemento de nuestro iterable:

```
ciudades = ('Valparaiso', 'Concepcion')
for i, ciudad in enumerate(ciudades):
    print(i, ciudad)
> 0 Valparaiso
> 1 Concepcion
```

- Finalmente, **`zip`** nos permite iterar sobre más de un iterable al mismo tiempo:

```
regiones = ('Valparaiso', 'Bio Bio', 'Los Rios')

for region, ciudad in zip(regiones, ciudades):

     print(f'{region:10s} {ciudad}')

> Valparaiso Valparaiso

> Bio Bio    Concepcion
```

Noten que al iterar sobre más de un iterable a la vez, el bucle se detiene al terminarse el iterable más corto.

Comprensión de iterables

Hay una manera más compacta de realizar bucles sencillos en Python: la comprensión. Podemos crear una lista así:

```
x = []

for i in range(5):

     x.append(i)
```

o, equivalentemente:

```
x = [i for i in range(5)]

type(x)

> list

x[1]

> 1
```

que puede combinarse con condicionales:

```
x = [i for i in range(5) if i%3 == 0]

x

> [0, 3]
```

Las operaciones de modificación de una lista son particularmente lentas, pero aún en general la comprensión de listas puede ser significativamente más rápido que la iteración dentro de un ciclo. En todo caso, tal como sugiere el Zen de Python, las

comprensiones deben usarse cuando resulta suficientemente sencillo leerlas. Podemos anidar comprensiones reemplazando cada valor por una comprensión:

```
x = [[10*i + j for j in range(3)]
     for i in range(5)]
x
> [[0, 1, 2], [10, 11, 12], [20, 21, 22], [30, 31, 32], [40, 41, 42]]
```

Generadores

Dijimos que **`range`** es un *generador*. Un generador es un tipo especial de objeto que se genera al vuelo, y sus elementos sólo pueden usarse una vez y luego se pierden. Esto los hace mucho más eficientes que las listas o las tuplas, pero hay que tener en cuenta la limitación anterior. Podemos crear generadores con comprensión:

```
x = (i for i in range(5))
type(x)
> generator
x[1]
> TypeError: 'generator' object is not subscriptable
```

En cambio, podemos acceder a cada elemento del generador **una vez**, ya sea en un bucle o usando el operador `next`:

```
next(x)
> 0
for i in x:
     print(i)
> 1
> 2
> 3
> 4
```

Los generadores son, quizá, algo difíciles de incorporar en nuestro subconsciente de programadores, pero en algunos casos pueden ahorrarnos mucho tiempo y memoria.

Por último, podemos usar comprensión de diccionarios:

```
capitales = {key: val for key, val
             in zip(regiones, ciudades)}
print(capitales)
```

```
> {'Valparaiso': 'Valparaiso', 'Bio Bio': 'Concepcion'}
```