Curso de Desarrollo en Lenguaje Python para Inteligencia Artificial (Málaga)

M.374.001.001

24 de marzo 2021 09:30-13:30

Modulo 1 – Tema 5

Spiros Michalakopoulos

Gobierno de España · Ministerio de Industria, Comercio y Turismo · Escuela de organización industrial

Unión Europea · Fondo Social Europeo · El FSE invierte en tu futuro

polo de contenidos digitales

Ayuntamiento de Málaga

Módulo 1: Introducción a la programación de Python

Desarrollo en Lenguaje Python para Inteligencia Artificial

Año de realización: 2021

PROFESOR
Spiros Michalakopoulos

spiros.eoi@gmail.com
https://www.linkedin.com/in/spiros-michalakopoulos

www.eoi.es

Índice

1. Introducción
2. Datos y Operadores
3. Estructuras de Datos Compuestas
4. Sentencias de Control
5. Funciones
6. Iteradores

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

Tema 5
Iteradores

Repaso datos mutables e inmutables

- Tipos de datos **mutables**: son los que pueden ser modificados directamente. Es el caso de las listas y los diccionarios. Cuando se hace una modificación, ésta se hace sobre el objeto original.

- Tipos de datos **inmutables**: son aquellos que no pueden ser modificados. Cada modificación es, en realidad, un objeto nuevo.

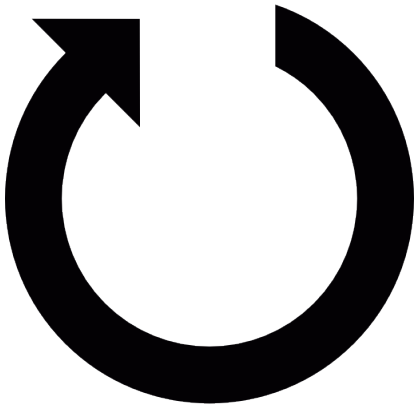
Utilizar adecuadamente cada tipo

Utiliza los *tipos mutables* para cosas que **son mutables en el mundo real** y los *tipos inmutables* para las cosas **inmutables en el mundo real.**

Iterando en Python

Los objetos con formato de secuencia en Python son **iterables.** Podemos acceder a ellos sucesivamente, de uno en uno, recorriendo la secuencia de forma ordenada.

Bucle for



- sets
- lists
- dictionaries
- strings
- ...

Las clases pueden ser iterables

- Se pueden definir las clases como iterables si se adhieren al “iterator protocol” de Python
- Es necesario incluir el método especial **__iter__**
- Cualquier objeto correspondiente a esa clase será iterable, aunque todavía tendremos que generar un iterador y tendrá que admitir el método **next**.

“Iterator Protocol”

- ¿Qué eran los métodos especiales?
- Python (a partir de versión 2.2) incluye un protocolo que define el proceso para la iteración sobre elementos en un contenedor.
- __iter__
- __next__
- Permite usar un bucle for para recorrer los elementos de una secuencia.

Detrás de escena

Al ejecutar

```
secuencia = [‘elemento_1’, ‘elemento_2’]
for x in secuencia:
    print(x)
```

```
elemento_1
elemento_2
```

En realidad, el intérprete de Python lo pasa a...

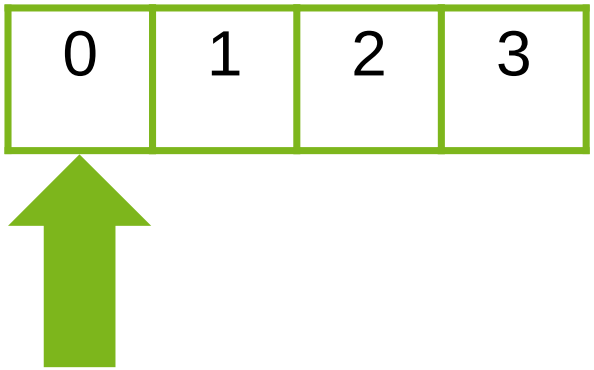
```
iterator = iter(secuencia)
while True:
    try:
        x = next(iterator)
        print(x)
    except StopIteration as e:
        break
```

elemento_1
elemento_2

El "Iterator Protocol" en acción

- Utilizamos directamente los métodos *iter* y *next* en vez de recurrir al bucle for
- A partir de un iterable, creamos un iterador con *iter(iterable)*
- Se van recorriendo los elementos del iterable con *next(iterable)*
- La instancia *next()* "recuerda" en qué elemento se ha quedado y muestra el siguiente elemento cada vez que se ejecute.
- Incluyen la excepción *StopIteration*, que se muestra después de haber devuelto el último elemento del iterable si se vuelve a ejecutar el método *next*

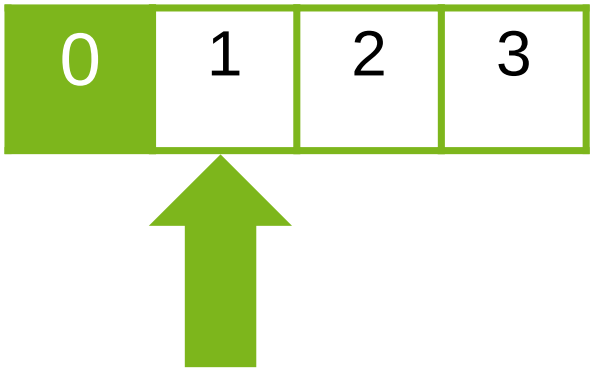
¿Qué es realmente el iterador?



El iterador es un nuevo objeto

```
r =
range(4)
it = iter(r)
next(it)
next(it)
next(it)
next(it)
next(it)
```

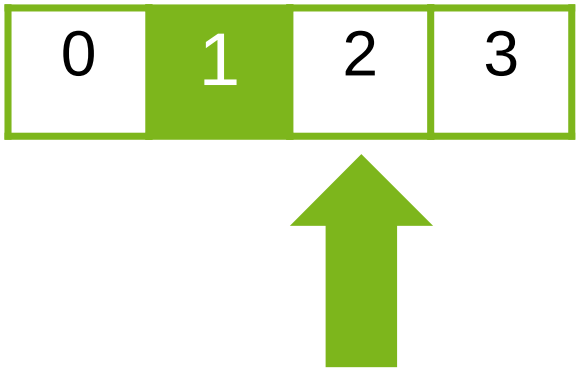
¿Qué es realmente el iterador?



0

```
r =
range(4)
it = iter(r)
next(it)
next(it)
next(it)
next(it)
next(it)
```

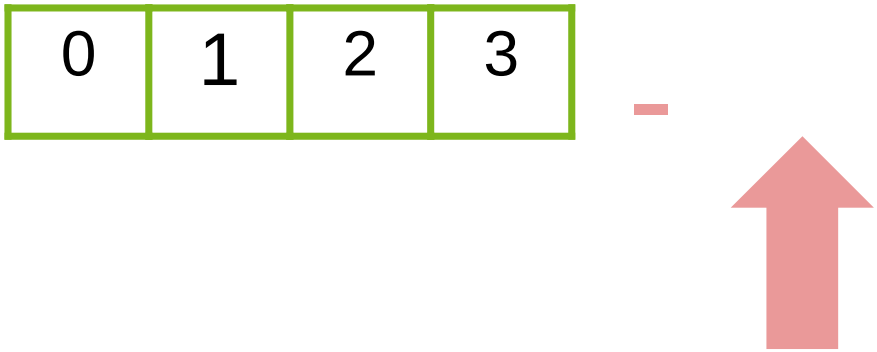
¿Qué es realmente el iterador?



1

```
r =
range(4)
it = iter(r)
next(it)
next(it)
next(it)
next(it)
next(it)
```

¿Y cuando llegamos al final?




```
r = range(4)
it = iter(r)
next(it)  #0
next(it)  #1
next(it)  #2
next(it)  #3
next(it) #-
```

StopIteration

¡Ojo! Range() no es una función

- A partir de la versión 3 de Python, *range()* ya no es una función y pasa a comportarse como un tipo de dato estructurado.
- Para ciertas instrucciones, presenta más similitudes con las tuplas y las listas.
- Es inmutable, pues, al utilizarla, se reserva un espacio en memoria, pero está vacío, es una lista de números abstracta, que solo puede concretarse al crearla y, sobretodo, al pedirle al "intérprete" que la almacene en una variable

Iteradores vs. iterables

- Un iterable es todo objeto con elementos contables
- A partir de un iterable, podemos crear un iterador, utilizando iter()
- En un iterador, podemos recorrer elemento a elemento con el método next, sin tener que escribir más código
- Un iterable no admite next(). Lo admite el iterador que creamos a partir de él.

Los iteradores son de un solo uso

- Los iteradores son una entidad basada en estados
- Solo son capaces de recorrer la secuencia una vez
- Cuando se llega al final de la secuencia, el trabajo del iterador ha terminado. Se dice, entonces, que está *exhausto*
- El iterable tiene la capacidad de crear iteradores cada vez que sea necesario, al inicio de cada iteración de un bucle, si es necesario.
- Cada vez que se crea el iterador, éste sabe cómo debe recorrer la secuencia.

Creando un iterador

```
class iterador(object):
    def __init__(self, ... ):
        ...
        ...
    def __iter__(self):
        return self

    def __next__ (self):
        if ...
        raise StopIteration
```

Crear un iterable en vez de un iterador

- Un iterable no tiene el problema de agotamiento que tiene una clase creada ya como iterador. Cada vez que es ejecutado en un bucle for, se crea un nuevo objeto que es el iterador.
- Para ello, al crear la clase, solo definimos el método iter.
- De esta forma, podremos usar la clase en bucles anidados.

Los auto-iteradores se agotan. Si esto va a ser un problema, es preferible crear un iterable.

Iteradores vs listas

- Una función que genere una lista, debe generar todos los datos a devolver cada vez que es llamada.
- Si necesitamos crear una lista suficientemente grande, esto nos va a dar un problema de recursos de memoria.
- Un iterador va generando los resultados según avanza. También puede dar problemas de agilidad si hay muchas llamadas next, pero no arrasará con toda la memoria.
- Un iterador puede crear series infinitas, lo que sería imposible para una lista generada por una función.

De los iteradores a los generadores

- Un generador es una función que devuelve un iterador
- Contiene instrucciones yield para ir generando, uno a uno, valores a utilizar en un bucle for
- Cada instrucción yield suspende temporalmente la ejecución recordando el estado local de ejecución
- Cuando la función vuelve a ser llamada, retoma su ejecución en el estado anterior, en vez de reiniciarse, como ocurre normalmente con las funciones.
- Los generadores también se agotan.

Una Función distinta a las demás

- Cuando el intérprete de Python se encuentra con la palabra *yield* en una función, pasa a tratarla de forma diferente. Se devuelve un objeto “Generator”
- Los generadores no se ejecutan cuando son llamados, sino cuando son iterados
- Los generadores funcionan en un proceso de iteración, mientras que las funciones no.
- Los generadores “congelan” su estado después de *yield* hasta la siguiente vez que son llamados

Infinitos hasta que llega return

- Los generadores también pueden ser infinitos, pues no necesitan generar todos los elementos al ser llamados.
- La instrucción return hace que el generador interrumpa un bucle de iteración.

Generadores con objetos

- Los métodos también pueden actuar como generadores.
- Por un lado, al definir el método __iter__, lo podemos hacer como generador, utilizando *yield*
- Por otro lado, cualquier método que sea un generador, también será un iterador y nos permitirá iterar sobre el objeto.
- Los objetos creados con métodos generadores, no están afectados por el agotamiento típico de los iteradores y las funciones generadoras. Cada vez que el objeto es iterado, se crea una nueva instancia de generador.

Cuándo usar un Generador

- Un generador puede sustituir a cualquier función que devuelva una lista. Pero la cuestión clave es la “acumulación que se da en la lista durante la iteración.
- Cualquier función o método que tenga que actuar como acumulador, es candidato ideal para un generador
- Pero... una función generadora va a sufrir agotamiento, así que no es conveniente usarla en casos de necesitar recurrir sucesivas veces al contenido. En ese caso, mejor usar listas.

range no es un generador

- La clase predefinida en Python range no es una función, y por tanto tampoco es un generador
- A veces, hay quien lo considera un generador y se utiliza como tal. Algunos errores pueden camuflarse detrás de un uso inadecuado que no ha tenido en cuenta la funcionalidad real de range.

Encadenando generadores

- Se pueden usar varios generadores “encadenados” para filtrar o manipular secuencias.
- Al encadenar generadores, hay que tener en cuenta que en alguno de los pasos no se admitirá un iterable, sino un iterador
- Pero como los generadores solo se ejecutan al iterar sobre ellos y no al llamarlos, las excepciones aparecen en la iteración que se aplica al último generador de la cadena.

La instrucción “islice”

- Los generadores no soportan las instrucciones *index* o *slice*
- El módulo *itertools*, en la librería estandar de Python, contiene la función *islice*, que tiene algunas similitudes con cómo funciona *slice* con las listas
- *islice* devuelve un iterable (en vez de listas)
- Pero para ver el resultado, es necesario iterar sobre islice, como ocurre con los generadores.

Mayor tiempo, menor memoria

- La iteración en un generador va a ser más lenta que, por ejemplo, la iteración sobre una lista
- La iteración sobre un generador requiere que el protocolo de iteración vaya llamando a next con cada elemento
- Puedes visualizarlo con el módulo timeit
- Por otro lado, los generadores requieren un menor uso de memoria, lo cual marca una diferencia significativa con las listas cuando tenemos que trabajar con grandes cantidades de items
- Los items de los generadores se crean solo cuando van a ser usados, y después desaparecen, no se guardan.

Comprensión de generadores

- Frente a la comprensión de listas, existe la comprensión de generadores
- Es especialmente útil al filtrar o trabajar con grandes ficheros de datos, ya que no hay que leer todo el fichero

```
>>> text = open("bigdata.txt", "r")
>>> lines = (line for line in text if line.startswith("2020-01-01"))
<generator object <genexpr> at 0x10fe3d550>
```

Comprensión de generadores

```
def average_line_length(lines):
    num_lines = 0
    total_length = 0
    for line in lines:
        num_lines += 1
        total_length += len(line)
    return total_length / num_lines
```

```
>>> result = average_line_length(lines)
24
```