Iteradores y Generadores

Yábir G. Benchakhtir

18 de noviembre de 2017

LibreIM

Contenido

Iteradores

¿Qué es un iterador en python?

Definición

Un **objeto** representando un flujo de datos.

Tiene las siguientes propiedades:

- Reiteradas llamadas a *__next__()* devuelve elementos del iterador.
- Devuelve **StopIteration** cuando no hay más elementos que devolver.
- Necesitan un método *__iter__()* que devuelve el propio objeto. Esto hace a todo iterador iterable.

Iterables en Python

Definición

Un objeto capaz capaz de devolver uno de sus miembros cada vez.

Algunos ejemplos de iterables son los tipos *list*, *str*, *tuple* y algunos no secuenciales como los diccionarios o los archivos.

Los objetos iterables se pueden usar en con todas las estructuras que usen secuencias. Un bucle *for* es el más tipico pero también se puede usar con *map*, *zip*...

```
Python 3.6.3 (default, Oct  3 2017, 21:45:48)
[GCC 7.2.0] on linux
>>> lst = [0,1,2,3,4,5]
>>> lst.__iter__
<method-wrapper '__iter__' of list object at 0x7fba41c74b88>
>>> iter(lst)
<list_iterator object at 0x7fba41c7dbe0>
>>> lstIterator = iter(lst)
>>> lstIterator.__next__()
0
>>> next(lstIterator)
1
```

Iterando

Para iterar sobre un *iterador* podemos usar un bucle **for**.

```
>>> lst = ["Python", "Ruby", "JS", "Haskell", "Go"]
>>> for lang in lst:
...     print(lang)
...
Python
Ruby
JS
Haskell
Go
>>>
```

Lo que nunca hay que hacer es:

```
>>> for i in range(len(lst)):
...     print(lst[i])
...
Python
Ruby
JS
Haskell
Go
```

Algo mejor en caso de necesitar los índices sería:

```
>>> for pos, elem in enumerate(lst):
...     print(pos, " ", elem)
...
0   Python
1   Ruby
2   JS
3   Haskell
4   Go
```

Generadores

Definición

Un generador es una función que usa **yield** para producir una serie de valores. La función devuelve un generador iterador.

Un primer ejemplo de generador

```
>>> def count(start,num):
...     i = start
...     while i <= num:
...         yield i
...         i += 1
...
>>> for i in count(4,9):
...     print(i)
...
4
5
6
7
8
9
```

También podemos crear generadores con la sintaxis para crear listas por comprehesion.

```
>>> gen = (x for x in range(10))
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> gen = (x for x in range(2))
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Itertools

Count

Sintaxis:

```
itertools.repeat(object[, times])
```

Ejemplo:

```
>>> from itertools import repeat
>>> a = repeat(2)
>>> next(a)
2
>>> next(a)
2
>>> next(a)
2
>>> next(a)
2
>>> from itertools import repeat
>>> b = repeat(3,2)
>>> next(b)
3
>>> next(b)
```

¿Cuándo es útil?

Si queremos repetir algo n veces.

```
for _ in itertools.repeat(None, n):
    repetir()
```

es más rápido que

```
for i in range(n):
    repetir()
```

También cuando trabajamos con *map* y *zip*

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> list(map(pow, range(10), 2))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

filter

Sintaxis:

```
itertools.filterfalse(predicate, iterable)
```

Funcionamiento:

Devuelve un iterador de elementos del iterable para los que el predicado se evalua como falso.

while

Sintaxis:

```
itertools.takewhile(predicate, iterable)
itertools.dropwhile(predicate, iterable)
```

Funcionamiento:

- *takewhile* devuelve los elementos del iterable hasta que el predicado se evalua como falso para algún elemento.
- *dropwhile* ignora los elementos del iterable hasta que el predicado se evalua como falso para algún elemento.

Sintaxis:

```
itertools.islice(iterable, stop)
itertools.islice(iterable, start, stop[, step])
```

Funcionamiento:

Toma una porción finita de un generador.

accumulate

Sintaxis:

```
itertools.accumulate(iterable[, func])
```

Funcionamiento:

Devuelve un iterador resultado de aplicar una operación binaria de manera acumulada. Por defecto usa *operator.sum*

Ejemplo de uso

```
>>> s = [5,4,2,8,7,6,3,0,9,1]
>>> list(accumulate(s))
[5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
>>> list(accumulate(s,min))
[5, 4, 2, 2, 2, 2, 2, 0, 0, 0]
>>> from operator import mul
>>> list(accumulate(s,mul))
[5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
```

Ejemplo

```
iterate(f, x) -> x, f(x), f(f(x)), ...
```

```
def iterate(f, x):
    yield x
    yield from iterate(f, f(x))
```

Una versión que no se quedará sin memoria:

```
def iterate(f, x):
    while True:
        yield x
        x = f(x)
```

La versión:

```
def iterate(f, x):
    return accumulate(repeat(x), lambda fx, _: f(fx))
```

Ejemplo de @joelgrus.

starmap

Sintaxis:

```
itertools.starmap(function, iterable)
```

Funcionamiento:

Mismo funcionamiento que map pero los argumentos han sido agrupados de manera previa.

```
>>> list(starmap(mul, enumerate("Dark Souls",1)))
['D', 'aa', 'rrr', 'kkkk', '     ', 'SSSSSS', 'ooooooo',
'uuuuuuuu', 'lllllllll', 'sssssssss']
```

Algunas funciones interesantes son

```
itertools.chain(*iterables)
itertools.product(*iterables, repeat=1)
zip(*iterables)
itertools.groupby(iterable, key=None)
itertools.permutations(iterable, r=None)
```

Extras

Algunas funciones extra

```
>>> all([1,2,3])
True
>>> all([1,2,0])
False
>>> any([1,2,0])
True
>>> any([1,2,3])
True
>>> sum([1,2,3])
6
>>> from functools import reduce
>>> from operator import mul
>>> reduce(mul,[1,2,3,4])
24
```

Referencias

Glosario de Python

Referencia de Itertools