Iterables_and_iterators

May 7, 2021

0.1 Iterable

In Python an Iterable is any object that implement the **Iterable protocol**. The requirement to comply with this protocol is to implement the `__iter__()` method and return an **Iterator**.

0.2 Iterator

All **Iterators** must implement the **Iterable protocol** in addition to implement the `__next__()` method to retrieve elements from the **Iterator**. When there are no more elements available `next()` will raise the `StopIteration` exception

As an alternative, the **Iterator protocol** can be implemented with only the `__getitem__()` method that receives an index as parameter. It must return values for consecutive integers, starting from zero, as indexes. When the index is out of range of the data, it will raise the `IndexError` exception.

[1]:
```
class ExampleIterator:
    def __init__(self, data):
        self._index = 0
        self._data = data

    def __iter__(self):
        return self

    def __next__(self):
        if self._index >= len(self._data):
            raise StopIteration()

        result = self._data[self._index]
        self._index += 1
        return result

class ExampleIterable:
    def __init__(self, data):
        self._data = data

    def __iter__(self):
        return ExampleIterator(self._data)
```

[3]:
```
sequence = ExampleIterable([1, 2, 3, 4, 5])
for i in sequence:
    print(i)
```

```
1
2
3
4
5
```

[3]:
```
[i * 2 for i in sequence]
```

[3]:
```
[2, 4, 6, 8, 10]
```

[6]:
```
class AlternateIterable:
    def __init__(self, data):
        self._data = data

    def __getitem__(self, index):
        return self._data[index]
```

[5]:
```
sequence = AlternateIterable([1, 2, 3, 4, 5])
for i in sequence:
    print(i)
```

```
1
2
3
4
5
```

[5]:
```
[i * 3 for i in AlternateIterable([1, 2, 3, 4, 5])]
```

[5]:
```
[3, 6, 9, 12, 15]
```

0.3 `iter()` function

This function is used implement the **Iterator protocol** for the **callable** that is passed as a parameter.

`iter(callable, sentinel)`

- callable: is an object that takes zero arguments
- sentinel: it's the value used to stop the iteration

This is often used for creating **infite sequences** from existing functions

[6]:
```
from datetime import datetime as dt
```

```
it = iter(dt.now, None)

for i in range(10):
    print(next(it))
```

```
2019-11-29 08:34:49.384085
2019-11-29 08:34:49.384311
2019-11-29 08:34:49.384351
2019-11-29 08:34:49.384387
2019-11-29 08:34:49.384422
2019-11-29 08:34:49.384456
2019-11-29 08:34:49.384490
2019-11-29 08:34:49.384525
2019-11-29 08:34:49.384559
2019-11-29 08:34:49.384594
```

0.4 Building-block functions

The idea behind this functions was develop in the **functional programming** paradigm. All these functions implement the **Iterator protocol**.

0.4.1 Map

Apply a function to every element in a sequence. It returns a new sequence with the result.

In Python 3 `Map` has a **lazy** implementation, but in Python 2 has an **eager** implementation.

It can accept **any number** of input sequences. The number of input sequences **must match** the number of function arguments

[7]:
```
def combine(size, colour, animal):
    return '{}, {}, {}'.format(size, colour, animal)

sizes = ['small', 'medium', 'large']
colours = ['red','yellow','blue']
animals = ['dog','cat','duck']

list(map(combine, sizes, colours, animals))
```

[7]:
```
['small, red, dog', 'medium, yellow, cat', 'large, blue, duck']
```

[8]:
```
import itertools

def combine2(quantity, size, colour, animal):
    return '{}, {}, {}, {}'.format(quantity, size, colour, animal)

list(map(combine2, itertools.count(), sizes, colours, animals))
```

[8]:
```
['0, small, red, dog', '1, medium, yellow, cat', '2, large, blue, duck']
```

0.4.2 Filter

Apply a function to each element in a sequence. It returns a new sequence with the elements for which the functions returns `True`

In Python 3 `Filter` has a **lazy** implementation, but in Python 2 has an **eager** implementation.

It can only accept a **single** input sequence. The function has to receive a single parameter too.

Passing `None` as the first parameter to `Filter` in will return a new sequence without the elements for which the function evaluates to `False`

[9]:
```
list(filter(lambda x: x > 0, [1, 4, 7, -6, 0, 2, -7, 10, -55]))
```

[9]:
```
[1, 4, 7, 2, 10]
```

[10]:
```
list(filter(None, [1, 4, 7, -6, 0, 2, -7, 10, -55]))
```

[10]:
```
[1, 4, 7, -6, 2, -7, 10, -55]
```

[11]:
```
list(filter(None, [0, 1, False, True, [], [1,2,3], '', 'hello']))
```

[11]:
```
[1, True, [1, 2, 3], 'hello']
```

0.4.3 Reduce

The `Reduce` function is part of the `functools` module. It repeatedly apply a function to the elements of a sequence reducing them to a single value.

The function provided to the `Reduce` function receives two parameters and must return another value, which it will be the first parameter in the following call to the function.

If you pass a sequence with **only one element** to the `Reduce` function, the function provided **will never be called** and it will return the only element in the sequence as a result.

The initial value of the accumulator can be passed as a third parameter to the `Reduce` function. Conceptually it is just added at the beginning of the sequence.

[12]:
```
from functools import reduce
import operator

reduce(operator.add, [1, 2, 3, 4, 5])
```

[12]:
```
15
```

[13]:
```
def mul(x, y):
    print('mul {} * {}'.format(x, y))
    return x * y

reduce(mul, [1, 2, 3, 4, 5])
```

```
mul 1 * 2
mul 2 * 3
mul 6 * 4
mul 24 * 5
```

[13]: 120

[14]:
```
reduce(mul, [])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-afb5e523a920> in <module>
----> 1 reduce(mul, [])

TypeError: reduce() of empty sequence with no initial value
```

[15]:
```
reduce(mul, [1])
```

[15]: 1

[16]:
```
reduce(mul, [1, 2, 3], 0)
```

```
mul 0 * 1
mul 0 * 2
mul 0 * 3
```

[16]: 0