Iterators

November 15, 2020

1 Iterators and Generators

1.1 Iterators

Iterators are objects that produce successive items or values from an associated ***iterable***. They: - Hold the state (position) of the iteration - Allow looping just once and must be reinitialized to loop again - Implement the ***__next__*** method that... - returns the next item in the sequence - raises the ***StopIteration*** exception if there is nothing to return - can also be invoked using the ***next(iterable)*** function

An ***iterable*** is an object that can be iterated over. - Must be capable of returning an iterator - Must implement the ***__iter__*** method, callable using the *iter* function

Simple iterables and iterators examples Lets define an *iterable* and an *iterator*...

```
In [1]: class Iterable:
            def __iter__(self):
                """
                Called by iter(Iterable())
                """
                return Iterator()

        class Iterator:
            def __init__(self):
                self.x = -1

            def __next__(self):
                """
                Called by next(iterator)
                """
                self.x += 1
                return self.x
```

... and instantiate them.

```
In [2]: iterable = Iterable()

        print(iterable)
```

```
<__main__.Iterable object at 0x7f474c781160>
```

```
In [3]: iterator = iter(iterable)
        # iter(iterable) ==> iterable.__iter__()

        print(iterator)
```

```
<__main__.Iterator object at 0x7f474c7814e0>
```

Let's call the *next()* function on the iterator.

```
In [4]: # iterator.__next__()
        print(next(iterator))
```

```
0
```

An object can also define both *next* and *iter* methods.

```
In [5]: class SimpleIterable:
            def __iter__(self):
                self.x = -1
                return self

        #     def __next__(self):
        #         self.x += 1
        #         return self.x

            def __next__(self):
                if self.x <= 3:
                    self.x += 1
                else:
                    raise StopIteration
                return self.x
```

```
In [6]: iterable = SimpleIterable()
        print(type(iterable))
        # iterable.x
```

```
<class '__main__.SimpleIterable'>
```

```
In [7]: iterator = iter(iterable)
        type(iterator)
        # iterable.x
```

```
Out[7]: __main__.SimpleIterable
```

```
In [8]: # Call next 5 times
        next(iterator)
```

```
Out[8]: 0
```

1.2 How to iterate on iterators

Iterators from containers (e.g. lists)

```
In [9]: a = list(range(4))
        print(a)
```

```
[0, 1, 2, 3]
```

Calling *next()* on a list won't work.

```
In [10]: next(a)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-10-15841f3f11d4> in <module>
----> 1 next(a)


TypeError: 'list' object is not an iterator
```

But getting an *iterator* from a list and iterating on it will.

```
In [11]: a = iter(a)
         # iter(iter) = iter

         next(a)
```

```
Out[11]: 0
```

The foreach construct

- Built-in in the language with the ***for*** ... ***in*** construct
- It allows looping on all elements of an iterable.
- Automatically calls the *iter(...)* function before starting looping

```
In [12]: print("With iter()")
         iterator = iter(iterable)
         for item in iterator:
             print(item)
```

```
With iter()
0
1
2
3
4
```

```
In [13]: print("Without iter()")
         for item in SimpleIterable():
             print(item)

Without iter()
0
1
2
3
4
```

1.3 Generators

- Functions containing the keyword ***yield***
- *yield* :
  - works similarly to *return* and returns an object when called...
  - ... but **state of the function is saved**
- When *next()* is called again on the generator function, execution resumes where it was left off
- Note that generators **do not return** values when initialized.

1.3.1 Examples

Trivial generator

```
In [14]: def f():
           print("-- start --")
           yield 3

           print("-- middle --")
           yield 4

           print("-- finished --")

In [15]: generator = f()
         generator

Out[15]: <generator object f at 0x7f47500215e8>

In [16]: next(generator)

-- start --


Out[16]: 3
```

Counter

```
In [17]: def counter():
             x : int = 0

             while True:
                 yield x
                 x += 1

         generator = counter()
         generator
```

```
Out[17]: <generator object counter at 0x7f4750021930>
```

```
In [18]: next(generator)
```

```
Out[18]: 0
```

Generators can also be defined inline!

```
In [19]: generator = (x for x in range(10))
         print("generator type:", type(generator))

         list(generator)
```

```
generator type: <class 'generator'>


Out[19]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Mind the difference with list comprehensions!

```
In [20]: not_a_generator = [x for x in range(10)]
         print("not_a_generator type:", type(not_a_generator))
```

```
not_a_generator type: <class 'list'>
```

1.3.2 Generators are lazy iterators

- They are used to **generate values dynamically**
  - Very useful to cope, for instance, with Out of Memory issues
- As iterators, they don't implement the *__len__* method
  - i.e., *len()* function will cause an exception
- Generators support **bidirectional communication**.
  - You can pass values to the generator **after** its initialization
- **Concurrent** and **recursive invocations** are **allowed**...
  - ... even though they are **not thread safe** out of the box.

1.3.3 Dynamic value generation example

```
In [21]: from datetime import datetime
         print("MS output format:", datetime.now().microsecond)

         def very_unsafe_prng(max_value):
             while True:
                 yield datetime.now().microsecond % max_value

         generator = very_unsafe_prng(10)
         generator
```

```
MS output format: 445995
```

```
Out[21]: <generator object very_unsafe_prng at 0x7f4750021750>
```

```
In [22]: # No len!
         len(generator)
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-22-95dd6a62a607> in <module>
          1 # No len!
    ----> 2 len(generator)


        TypeError: object of type 'generator' has no len()
```

```
In [23]: next(generator)
```

```
Out[23]: 1
```

1.3.4 Bidirectional communication

Bidirectional communication allows to send values to the generator. Relies on three methods: - ***.send(...)***: sends the value to the generator and, like **next()**, returns the next value - ***.throw(...)***: throws the passed exception after resuming the generator that will handle it - ***.close()***: stops the generator. Equivalent to ***.throw(GeneratorExit())*** - *yield* can be used in expressions to assign values to generator's variables - Values will be assigned when the generator resumes from *yield*

Example

```
In [24]: from random import choice

         # Define allowed values ([1, 7])
```

```
values = list(range(1, 8))

# Define a generator
def seven_and_half():
    values_sum = 0
    results = []

    player = 1

    # Keep going until generator is closed
    while True:
        # Pick a card (pseudo) randomly
        value = choice(values)

        # Accumulate the values
        values_sum += value

        #
        try:
            response = yield value, values_sum
        except GeneratorExit:
            results.append((player, values_sum),)

            for player, score in results:
                print("Player {} scored {}".format(player, score))
            break

        if response is False or response is None:
            results.append((player, values_sum),)
            values_sum = 0
            player += 1

    print("Exiting")
```

In [25]:
```
# Init the generator
generator = seven_and_half()
```

In [26]:
```
keep_playing = None

# Keep in mind that
# generator.send(None) === next(generator)

while True:
    if keep_playing is False:
        break

    print("Picking a card.")
    value, values_sum = generator.send(keep_playing)
```

```
            print("Picked {}. Total: {}".format(value, values_sum))

            if values_sum > 7:
                print("You lost!")
                keep_playing = False
            else:
                keep_playing : bool = (input("Keep picking? ") in ["y", "Y", True] )
            # print(output)
```

```
Picking a card.
Picked 5. Total: 5
Keep picking? n
```

```
In [27]: generator.close()
```

```
Player 1 scored 5
Exiting
```

1.4 Exercise

Intro CSV (Comma-Separated Values) files are text files where **each row is a** data **record** and **columns** are **separated by commas** (or some other character).

The first row is (usually) the header (i.e., the name of the corresponding column).

A CSV file looks like:

```
id,name,surname
0,Mickey,Mouse
```

Request You have to **process a *CSV* file**. Lets assume it is **too large to fit in RAM**.

You should process it in small pieces, e.g., by reading each line sequentially using a generator.

Specifically, after reading the header, for each line of the file create a dictionary with the column-value associations.

A few tips

- *Pathlib* module offers classes representing filesystem paths with semantics appropriate for different operating systems
- Use the _**zip**(*iterables)_ builtin function. From the docs:
  - Builds an iterator that aggregates elements from each of the iterables
  - That is, it returns an iterator of tuples, i.e., The i-th element of the tuple contains the i-th element from each of the argument sequences or iterables.
  - The iterator stops when the shortest input iterable is exhausted
- Use the ***with*** statement. From the docs: the with statement is used to wrap the execution of a block with methods defined by a context manager. This allows common *try…except…finally* usage patterns to be encapsulated for convenient reuse.
- As an example CSV, download *as raw* the Rolling Stone Magazine's list of "The 500 Greatest Albums of All Time." from GitHub.

1.4.1 Solution

In [28]:
```
from pathlib2 import Path


        def dataset_reader(file):
            # Lets use pathlib instead of using the open() function,
            # with open(file, "r+") as f:

            # Creating a Path instance.
            file = Path(file)

            # Not actually needed, just showing some functionality
            if not file.absolute():
                file = file.resolve()

            print("Opening", file.name, "in folder", file.parent)

            if not file.exists():
                raise FileNotFoundError("File doesn't exist!")


            with file.open("r+", encoding="ISO-8859-15") as f:
                header = f.readline()
                columns = header.strip().split(',')

                print("Found columns:", columns)
                for line in f:
                    values = line.strip().split(',')
        #             print(values)

                    try:
                        yield dict(zip(columns, values))
                    except GeneratorExit:
                        print("Closing the generator!")
                        break
```

In [29]:
```
file = "albumlist.csv"

        generator = dataset_reader(file)
```

In [30]:
```
next(generator)
```

```
Opening albumlist.csv in folder .
Found columns: ['Number', 'Year', 'Album', 'Artist', 'Genre', 'Subgenre']
```

Out[30]:
```
{'Number': '1',
 'Year': '1967',
```