Lesson 13 - Generators

July 14, 2020

1 Agenda

- What is a generator
- Generator functions in the standard library

2 References

- PEP 255 – Simple Generators
- itertools — Functions creating iterators for efficient looping
- Itertools Recipes

3 What is a generator

> **Generator** - a function which returns a generator iterator. It looks like a normal function except that it contains ***yield*** expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the ***next()*** function.
>
> **Generator iterator** - an object created by a generator function.
>
> Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator iterator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).
>
> Python Glossary

Generators are iterators that produce the values of an expression passed to *yield*. A **generator function** is a generator factory.

!! Iterators deeply embeded in Python; they are an integral part of the language.

[1]:
```
# %load letter_printer.py
def letter_printer():
    print('letter_printer has been called ...')

    print('Yielding A ...')
```

```
    yield 'A'

    print('Yielding B ...')
    yield 'B'

    print('Yielding c ...')
    yield 'C'

    print('letter_printer has called it a day!')
```

[2]:
```
print(letter_printer)

lprinter = letter_printer()
print(lprinter)
```

```
<function letter_printer at 0x7f03ac2a0320>
<generator object letter_printer at 0x7f03ac22e0d0>
```

[3]:
```
lprinter()
```

```
␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-3-0ff1c5b8761c> in <module>
   ----> 1 lprinter()


       TypeError: 'generator' object is not callable
```

[ ]:
```
next(lprinter)
```

[ ]:
```
letter_b = next(lprinter)
print("letter_b equals to: {}".format(letter_b))
```

[ ]:
```
next(lprinter)
```

[ ]:
```
next(lprinter)
```

[ ]:
```
print(lprinter)
```

[ ]:
```
next(lprinter)
```

[ ]:
```
new_lprinter = letter_printer()
for letter in new_lprinter:
    print(">>> {}\n".format(letter))
```

[ ]:
```
letter_list = [letter for letter in letter_printer()]
```

[ ]:
```
print(letter_list)
```

[ ]:
```
# %load number_printer.py
def number_printer(max_number=10):
    print('number_printer has been called ...')
    for number in range(1, max_number):
        print(number)

    print('number_printer has called it a day!')
```

[ ]:
```
numbers = number_printer()
```

```
> pip install pympler

Collecting pympler
  Downloading Pympler-0.8.tar.gz (175 kB)
     |                  | 175 kB 1.2 MB/s eta 0:00:01
Building wheels for collected packages: pympler
  Building wheel for pympler (setup.py) ... done
  ...
```

[ ]:
```
from pympler import asizeof

numbers = [x for x in range(10)]

print(asizeof.asized(numbers, detail=2).format())
```

[ ]:
```
# %load number_printer_generator.py
def number_printer_generator(max_number=10):
    print('number_printer has been called ...')
    for number in range(1, max_number):
        print(number)
        yield number

    print('number_printer has called it a day!')
```

[ ]:
```
from pympler import asizeof

gen = number_printer_generator()
```

```
print(asizeof.asized(gen, detail=2).format())
```

3.1 Generator expressions

[ ]:
```
comprehension = [n for n in number_printer_generator()]

print(comprehension)
```

[ ]:
```
generator_expression = (n for n in number_printer_generator())

print(generator_expression)
```

[ ]:
```
numbers = (n for n in range(10000))

print(generator_expression)
```

Arithmetic progression

[ ]:
```
# %load arithmetic_progression.py
def arithmetic_progression(begin, step, end=None):
    result = type(begin + step)(begin)

    forever = True if end is None else False
    index = 0

    while forever or result < end:
        yield result

        index += 1
        result = begin + step * index
```

[ ]:
```
from arithmetic_progression import arithmetic_progression as arith_prog

integers = arith_prog(0, 1, 5)
print(integers)
```

[ ]:
```
print(list(integers))
```

[ ]:
```
from arithmetic_progression import arithmetic_progression as arith_prog

decimals = arith_prog(0, 0.25, 5)
```

[ ]:
```
next(decimals)
```

[ ]:
```
next(decimals)
```

[ ]:
```
sample = [next(decimals) for _ in range(5)]
print(sample)
```

[ ]:
```
# %load filter_even.py
def is_even(number):
    return number % 2 == 0


def filter_even(iterable):
    for item in iterable:
        if is_even(item):
            yield item
```

[ ]:
```
numbers = list(range(10))
print(numbers)

even = filter_even(numbers)
print(even)
```

[ ]:
```
print(len(numbers))
```

[ ]:
```
[next(even) for _ in range(4)]
```

[ ]:
```
numbers.extend(list(range(10, 20)))

print(numbers)
```

[ ]:
```
[next(even) for _ in range(6)]
```

[ ]:
```
numbers.extend(list(range(20, 30)))

print(numbers)
```

[ ]:
```
next(even)
```

[ ]:
```
list(filter_even(numbers))
```

[ ]:
```
next(even)
```

4 Generator functions in the standard library

itertools

4.1 Filtering generator functions

[4]:
```
# %load filter_func.py
import itertools


def is_even(number):
    return number % 2 == 0


numbers = list(range(10))

# builtin filter function
print('numbers >> {}'.format(numbers))
builtin_filter = filter(is_even, numbers)
print('builtin_filter >> {}'.format(builtin_filter))
builtin_filter = list(builtin_filter)
print('builtin_filter >> {}\n'.format(builtin_filter))

# filterfalse
print('numbers >> {}'.format(numbers))
filterfalse = itertools.filterfalse(is_even, numbers)
print('filterfalse >> {}'.format(filterfalse))
filterfalse = list(filterfalse)
print('filterfalse >> {}\n'.format(filterfalse))

# dropwhile
print('numbers >> {}'.format(numbers))
dropwhile = list(itertools.dropwhile(is_even, numbers))
print('dropwhile >> {}\n'.format(dropwhile))

print('numbers >> {}'.format(numbers))
dropwhile = list(itertools.dropwhile(lambda x: x > 5, numbers))
print('dropwhile >> {}\n'.format(dropwhile))

# takewhile
print('numbers >> {}'.format(numbers))
takewhile = list(itertools.takewhile(is_even, numbers))
print('takewhile >> {}\n'.format(takewhile))

print('numbers >> {}'.format(numbers))
takewhile = list(itertools.takewhile(lambda x: x > 5, numbers))
print('takewhile >> {}\n'.format(takewhile))
```

```
numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
builtin_filter >> <filter object at 0x7f03ac1b4710>
builtin_filter >> [0, 2, 4, 6, 8]
```

```
numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
filterfalse >> <itertools.filterfalse object at 0x7f03ac1b4ad0>
filterfalse >> [1, 3, 5, 7, 9]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
dropwhile >> [1, 2, 3, 4, 5, 6, 7, 8, 9]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
dropwhile >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
takewhile >> [0]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
takewhile >> []
```

4.2 Mapping generator functions

[5]:
```python
# %load mapping_func.py
import itertools
import operator


numbers = list(range(10))

# builtin enumerate function
sequence = 'Python is cool!'
print('sequence >> {}'.format(sequence))
enum = list(enumerate(sequence, 1))
print('enumerate >> {}\n'.format(enum))

# builtin map function
print('numbers >> {}'.format(numbers))
squared = list(map(lambda x: x * x, numbers))
print('squared >> {}\n'.format(squared))

print('numbers >> {}'.format(numbers))
multiplied = list(
    map(operator.mul, numbers, squared))
print('multiplied >> {}\n'.format(multiplied))

# itertools accumulate
print('numbers >> {}'.format(numbers))
accumulated = list(itertools.accumulate(numbers))
```

```
print('accumulated >> {}\n'.format(accumulated))

print('numbers >> {}'.format(numbers))
accumulated_min = list(itertools.accumulate(numbers, min))
print('accumulated_min >> {}\n'.format(accumulated_min))

print('numbers >> {}'.format(numbers))
accumulated_max = list(itertools.accumulate(numbers, max))
print('accumulated_max >> {}\n'.format(accumulated_max))

print('numbers >> {}'.format(numbers))
accumulated_multi = list(itertools.accumulate(numbers[1:], operator.mul))
print('accumulated_multi >> {}\n'.format(accumulated_multi))
```

```
sequence >> Python is cool!
enumerate >> [(1, 'P'), (2, 'y'), (3, 't'), (4, 'h'), (5, 'o'), (6, 'n'), (7, '
'), (8, 'i'), (9, 's'), (10, ' '), (11, 'c'), (12, 'o'), (13, 'o'), (14, 'l'),
(15, '!')]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
squared >> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
multiplied >> [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
accumulated >> [0, 1, 3, 6, 10, 15, 21, 28, 36, 45]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
accumulated_min >> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
accumulated_max >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
accumulated_multi >> [1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

4.3 Merging generator functions

[6]:
```
# %load merging_func.py
import itertools


numbers = list(range(10))
sequence = 'Python is cool!'
```

```
# built-in zip
print('numbers >> {}'.format(numbers))
zipped = list(zip(sequence, numbers))
print('zipped >> {}\n'.format(zipped))

# itertools chain
print('numbers >> {}'.format(numbers))
print('sequence >> {}'.format(sequence))
chain = list(itertools.chain(sequence, numbers))
print('chain >> {}\n'.format(chain))
```

```
numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
zipped >> [('P', 0), ('y', 1), ('t', 2), ('h', 3), ('o', 4), ('n', 5), (' ', 6),
('i', 7), ('s', 8), (' ', 9)]

numbers >> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sequence >> Python is cool!
chain >> ['P', 'y', 't', 'h', 'o', 'n', ' ', 'i', 's', ' ', 'c', 'o', 'o', 'l',
'!', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```