L2: Iterator and Generator

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

January 9th, 2020

Architecture Research Group

SSE, TGGS

Your First Task

Pylint

- Please make sure you can run python on your machine
- Then, please install pylint
  - https://www.pylint.org/
- Pylint is a handy tool for you to check your python code

- You will need to run things on python today

Other Things to Install

- Please install Scala and Rust
  - https://www.scala-lang.org/download/
  - https://www.rust-lang.org/tools/install

Iterator

What is an Iterator?

- Give you an illusion of a linear sequence
- Allow you to traverse a collection of items

- Simple interface
  - Is there a next element? (i.e., boolean hasNext())
  - Get to the next element (i.e., E next())
- Notion of iterators are similar in multiple languages

Iterator in Python

- Iterable object: an object can be used with a for loop
  - There is a way to get the next element
  - There is a way to determine if there are no more element
- Creating an iterator from an iterable object
  - In Python, you can use *iter* to get an iterator

```
it = iter([4, 1, 5])
print(next(it))
print(next(it))
print(next(it))
print(next(it))   # Here you will get an StopIterator exception
```

Let's Build an Iterator from Scratch

- What are the requirements?

- Fundamentally we need:
  - One internal index (think of a pointer to the current element)
  - One internal way to keep track of the number of elements
  - Two methods
    - A method that return an iterator
    - A method that return the current element, and move the index to the next element
  - And in Python, a constructor

So, how do we actually build this iterator?

A Python Iterator from Scratch

- Constructor

```
def __init__(self, n):
        self.n, self.index = n, 0
```

- A method that return the iteration

```
def __iter__(self):
        return self
```

- A method that return the current element and move on

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

Sample Usage

- Using in conjunction with a for loop

```
for num in my_range(5):
    print(num)
```

- Using in conjunction with a list

```
print(list(my_range(4))
```

- Using an iterator manually

```
it = iter(my_range(2))
print(next(it))
print(next(it))
```

Generator

Iterator Overhead

- You need to keep track of where the iterator is
- Need to write certain specific methods

- Generator can address these problems

Generator Overview

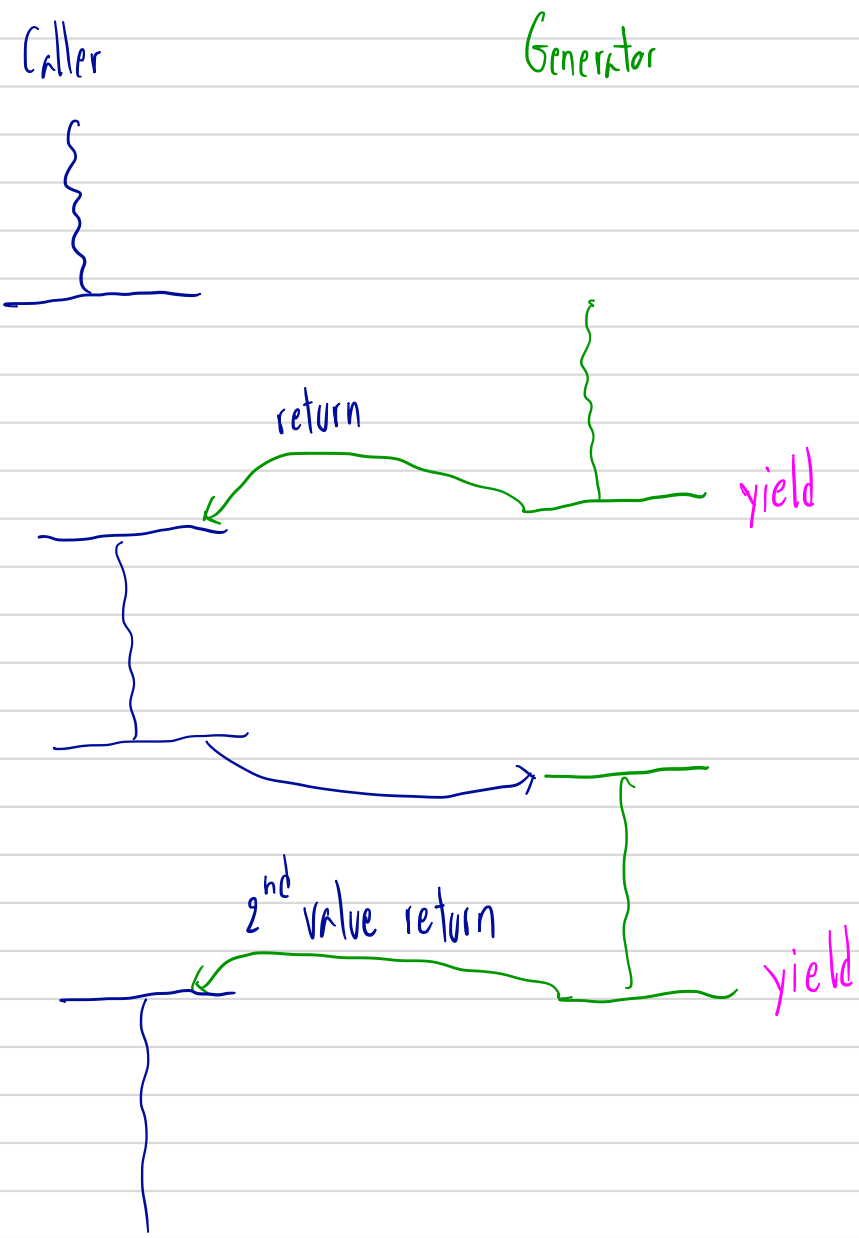
- What is the type of the following expressions?

g = (x**2 for x in [3, 4, 1, 2, 5, 7])

- The expression produce many values in sequential order

A Generator vs. A Function

- In programming, a function produce one value as an output
  - For multiple outputs, you need to call the function multiple times
- Generator:
  - Does not require multiple function calls
  - Contain one or more yield statement
  - When called, return an iterator object
    - Iter() and next() are implemented automatically
  - Once a function yield, control is paused, caller resume
    - But local variables and states are stored, can be resumed for successive calls
  - StopIteration is raised automatically


Caller
Generator
return
yield
2nd value return
yield

A Generator with Coroutine

- In python, you can hand-off internal state through a coroutine (Throw some work to others)
- This can be used in conjunction with a generator

- A generator can pass its internal state in the background similar to coroutine

Let's try an example

Coroutine

- Assume the following generator

```
def foo():
    print('foo-A') yield 3
    print('foo-B') yield 5
    print('foo-C') yield 2
    print('foo-Done')
```

- What is going to happen here?

```
g = foo()
print(next(g))
print(next(g))
print(next(g))
print(next(g))
h = foo()
print(list(h))
```

Using Generators for Infinite Seq.

- If we assume “the next prime” is a coroutine
  - Finding the next prime is computationally expensive
  - We want to limit how many prime number we should produce
    - Only compute up to N on demand
- Generator can address this problem
  - Values are yielded back, computation is paused
  - No need to compute the actual infinite sequence
  - But generator allows you to keep producing the next element!
- We will implement this as a part of the in-class exercise

Recursive Generators

- Generator can also be used to recursively
- Example: unraveling a nested list

```
def walk_nested_list(lst):
    def do_walk(node_elt, depth=0):
        if isinstance(node_elt, int):
            yield depth, node_elt
        else: for node in node_elt: 3
            yield from do_walk(node, depth=depth+1)
yield from do_walk(lst, depth=0)
```

Pipelining Generators

- Generator can be used to pipeline a series of operations

```
with open('sells.log') as file:
    pizza_col = (line[3] for line in file)
    per_hour = (int(x) for x in pizza_col if x != 'N/A’)
    print("Total pizzas sold = ",sum(per_hour))
```

- **Source:** https://www.programiz.com/python-programming/generator

In-class Exercise

- Using the concept of iterator, implement Fibonacci
- Using the concept of generator, implement all prime after n

- Starter code are on the in-class assignment 2