CS1302 - Lecture 6 More Functions – (b1) Generator
(Dr Helena WONG)
==

[Contents]

[Helena's supplements - Exercises]

[Helena's supplements (Generator) for **Generator and Decorator.ipynb]**

==

[Helena's supplements - Exercises]

Starter Exercise 1. **Introducing generator (a method is to call a function that has yield)**

We will learn *generator*.
Try the following programs in *online python tutor*:

Program 1 : **learn yield**

```
def winter_days():
    for i in range(1,31): yield("Nov-"+str(i))
    for i in range(1,32): yield("Dec-"+str(i))
    for i in range(1,32): yield("Jan-"+str(i))

for d in winter_days():
    print(d)
# shows 92 days: Nov-1, Nov-2, Nov-3, .. Jan-30, Jan-31
```

Program 1b : lean **next()**

```
def winter_days():
    for i in range(1,31): yield("Nov-"+str(i))
    for i in range(1,32): yield("Dec-"+str(i))
    for i in range(1,32): yield("Jan-"+str(i))

g = winter_days()
print(next(g), next(g), next(g)) # output: Nov-1 Nov-2 Nov-3
for i in range(50):
    print(next(g),end=" ") # output: Nov-4 Nov-5 ...
```

Program 1c: learn *** (the unpack operator)** and ***StopIteration***

```
def winter_days():
    for i in range(1,31): yield("Nov-"+str(i))
    for i in range(1,32): yield("Dec-"+str(i))
    for i in range(1,32): yield("Jan-"+str(i))

g = winter_days()
print(next(g))      # output: Nov-1
print(*g)           # *: the unpack operator; output: Nov-2, ..
print(next(g))      # No more items to extract: StopIteration is raised
```

Starter Exercise 2. Introducing generator (another method is to use generator expression)

Program 2

```
g = (x for x in range(5))
print(next(g), next(g), next(g)) # output: 0 1 2

print(*g)
print(*g)
```

* Try using next() and * like Program 1b and 1c!
* Also have *StopIteration* ? How?

Generator vs Comprehension

```
g = (x for x in range(5))    # generator
print(*g)
print(*g)

g2 = [x for x in range(5)]    # list (via comprehension), will learn later
print(*g2)
print(*g2)

print(type(g),type(g2))
```

Quick note:

Generator:

- Allows us to generate arbitrarily-many items in a series, without having to store them all in memory at once.
- A generator can be iterated over once, after which it is exhausted and must be re-defined in order to be iterated over again.

https://www.pythonlikeyoumeanit.com/Module2_EssentialsOfPython/Generators_and_Comprehensions.html

Try!

```
def f():
    return 0
    yield 1

for i in f():
    print(i)
```

```
def f():
    return 0
    yield 1

for i in f():
    print(i)
```

What if "yield 1" is removed?

```
def f():
    return 0

for i in f():
    print(i)
```

==

Generator

A generator object (2 methods)
How to obtain items from a generator
yield vs return
yield and send
fibonacci_generator

==

A generator is a programming object that generates a sequence of values.

How to create a generator?

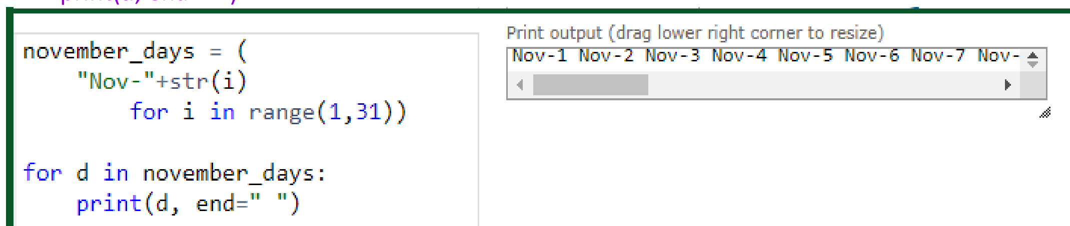
Method 1: **generator expression**: use generator_name = **(***the rule***)** # must have parentheses ()

```
x = (i*i for i in range(1,4))
print(type(x))
print(next(x))
print(next(x))
print(next(x))
.. # raises the StopIterationException at the end.

november_days = ("Nov-"+str(i) for i in range(1,31))
print(type(november_days))
print(next(november_days))
print(next(november_days))
print(next(november_days))
```

A generator object is iterable: We can use a for __ in __ -loop to extract the values

```
november_days = ("Nov-"+str(i) for i in range(1,31))
for d in november_days:
    print(d, end=" ")
```



Bonus: **list(), set(), etc..**

```
d = list(november_days) # pass the generator to the list constructor
```

Method 2: Another way to define the generator is to use the keyword **yield** in a function;
then, running the function will return a generator.

```
def winter_days():
    for i in range(1,31): yield("Nov-"+str(i))
    for i in range(1,32): yield("Dec-"+str(i))
    for i in range(1,32): yield("Jan-"+str(i))

for d in winter_days(): print(d, end=" ")
# output: Nov-1 Nov-2 Nov-3 Nov-4 Nov-5 .. Jan-30 Jan-31

g = winter_days()
print(next(g), next(g), next(g))
for i in range(50): print(next(g),end=" ")
```

```
def winter_days():
    for i in range(1,31): yield("Nov-"+str(i))
    for i in range(1,32): yield("Dec-"+str(i))
    for i in range(1,32): yield("Jan-"+str(i))

g = winter_days()
print(next(g), next(g), next(g))
for i in range(50): print(next(g),end=" ")
```

Print output (drag lower right corner to resize)

```
Nov-1 Nov-2 Nov-3
Nov-4 Nov-5 Nov-6 Nov-7 Nov-8 Nov-9 Nov
```

Frames Objects

Global frame

winter_days → function winter_days

g → generator insta

i 49

return vs yield

return will return the data and terminate a function immediately

yield just pauses the function, and will continue from there in next call

fibonacci_sequence: An efficient version of fibonacci generator (c.f. the recursive version is slow !!)

```
# F0, F1, F2, ... are 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 ..
def fibonacci_sequence(Fn, Fn1, stop):
    while Fn < stop:
        yield Fn    # return Fn and pause execution
        Fn, Fn1 = Fn1, Fn1 + Fn

g = fibonacci_sequence(13, 21, 500)
print(next(g), next(g), next(g)) # 13 21 34
print(next(g), next(g), next(g)) # 55 89 144
print(*g) # 233 377
```

yield and send

The **send** (value) method sends a value into the generator function.

The value argument becomes the result of the current yield expression

What's the output?

```
def spend_money():
    money = 100
    while True:
        money -= 5
        obtain = yield money
        money += obtain or 0

g = spend_money()
print(next(g)) # output: _________
print(next(g)) # output: _________
print(next(g)) # output: _________
print(g.send(50)) # output: _________
print(g.send(500)) # output: _________
print(next(g)) # output: _________
```

```python
def spend_money():
    money = 100
    while True:
        money -= 5
        obtain = yield money # by default obtain gets None
        money += obtain or 0 # i.e., "obtain if obtain else 0"

g = spend_money()
print(next(g)) # run line 2 to 5 (up to yield money only)
                # output: 95
print(next(g)) # finish line 5 (obtain gets None), line 6,3,4,5 (up to yield money only)
                # output: 90
print(next(g)) # finish line 5 (obtain gets None), line 6,3,4,5 (up to yield money only)
                # output: 85
print(g.send(50)) # finish line 5 (obtain gets 50!!), line 6,3,4,5 (up to yield money only)
                # output: 85+50-5 = 130

print(g.send(500)) # finish line 5 (obtain gets 500!!), line 6,3,4,5 (up to yield money only)
                # output: 130+500-5 = 625

print(next(g)) # # finish line 5 (obtain gets None), line 6,3,4,5 (up to yield money only)
                # output: 625-5 = 620
```

```
95
90
85
130
625
620
```

* For extra send() or next(), where there is no more yield expression to run, you'll get *StopIteration*.
* When send() is called to start the generator, it must be called with None as the argument (e.g. **g.send(None)**)
because there is no yield expression that could receive the value.

fibonacci_sequence + send
* This version is an improvement of the one inside the lecture .ipynb

```python
# F0, F1, F2, ... are 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 ..

def fibonacci_sequence(Fn, Fnn, stop):
    while Fn < stop:
        value = yield Fn
        if value is not None:
            Fn, Fnn = value[0], value[1]
        else:
            Fn, Fnn = Fnn, Fnn + Fn

g = fibonacci_sequence(13, 21, 500)
print(next(g), next(g), next(g)) # 13 21 34
print(next(g), next(g), next(g)) # 55 89 144
print(g.send((55, 89))) # 55
print(*g) # 89 144 233 377

# tell g that, "in the coming result I want start from 55, 89"
# g.send((55,89))     # "inject" (55,89) into g, and also get one outcome
```

==

See additional contents from **Generator and Decorator.ipynb**