Generators

Announcements

Generators

https://www.daleseo.com/python-yield/

Generators and Generator Functions

```
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>
```

```
def evens(start, end):
    even = start + (start % 2)
    while  even < end:
        yield even
        print('just yield')
        even = even + 2
#####
>> t = evens(1, 10)
>> next(t)
2
>> next(t)
just yield
4

#####
>> t = evens(21)
>> next(t)
22
>> next(t)
24
>> next(t)
26
>> t
<generator ~>
```

```
def evens(start, end):
    even = start + (start % 2)
    while  even < end:
        yield even
        even = even + 2
#####
>> evens(1, 10)
<generator
>> t = evens(1, 10)
>> next(t)
2
>> next(t)
4
>> next(t)
6
>> next(t)
8
>> next(t)
StopIteration
>> list(evens(1, 10))
[2, 4, 6, 8]
>> for x in evens(1, 10:
        print(x)
2
4
6
8
```

A *generator function* is a function that **yield**s values instead of **return**ing them

A normal function **return**s once; a *generator function* can **yield** multiple times

A *generator* is an iterator created automatically by calling a *generator function*

When a *generator function* is called, it returns a *generator* that iterates over its yields

(Demo)

Generators & Iterators

>> lsit(countdown(3))
[3, 2, 1, ‘Blast off!’, ‘Blast off!’, ‘Blast off!’, ‘Blast off!’]

Generator Functions can Yield from Iterables

A **yield from** statement yields all values from an iterator or iterable (Python 3.3)

```
def countdown(k):
    if k > 0:
        yield k
        yield from countdown(k-1)
    else:
        yield “Blast off!”
#####
>> blast_off = countdown(10)
>> blast_off
<generator ~ >
>> next(blast_off)
10
>> next(blast_off)
9
>> next(blast_off)
8
>>list(blast_off)
[7, 6, 5, 4, 3, 2, 1]
>> next(blast_off)
7

…

>> next(blast_off)
1
>> next(blast_off)
Blast off!
```

```
>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]

def a_then_b(a, b):
    for x in a:
        yield x
    for x in b:
        yield x
```

```
def a_then_b(a, b):
    yield from a
    yield from b
```

2:33분쯤 내용 놓침.
def countdown 내용 바꿔가면서…

```
def countdown(k):
    if k > 0:
        yield k
        t = countdown(k - 1):
            yield from t
#####
>> blast_off = countdown(10)
>> blast_off
<generator ~ >
>> next(blast_off)
10
>> next(blast_off)
???????
```

```
>>> list(countdown(5))
[5, 4, 3, 2, 1]

def countdown(k):
    if k > 0:
        yield k
        yield from countdown(k-1)
```

(Demo)

```
def countdown(k):
    if k > 0:
        yield k
        for x in countdown(k - 1):
            yield x
#####
>> blast_off = countdown(10)
>> blast_off
<generator ~ >
>> next(blast_off)
10
>> next(blast_off)
???????
```

```
def prefixes(s):
    if s:
        yield from prefixes(s[:len(s) - 1])
        yield s

def substrings(s):
    if s:
        yield from prefixes(s)
        yield from substrings(s[1:])
```

```
>> list(prefixes('both'))
['b', 'bo', 'bot', 'both']
>> list(substrings('both'))
['b', 'bo', 'bot', 'both', 'o', 'ot', 'oth', 't', 'th', 'h']
```

Example: Partitions

Yielding Partitions

A partition of a positive integer n, using parts up to size m, is a way in which n can be expressed as the sum of positive integer parts up to m in increasing order.

partitions(6, 4)

```
2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6
```

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

(Demo)

3/1 2:50pm picture w/o yield