Generators

Some simple examples and reminders (we saw the equivalents of these in lecture)

In [1]:

```
# Generator of integers from start through end-1

def a_range(start, end):
    # as a while loop
    while start < end:
        yield start
        start += 1

def rec_range(start, end):
    # a recursive version
    if start >= end:
        return
    yield start
    for n in rec_range(start+1, end):
        yield n

# cleaner: yield from vs. yield
def my_range(start, end):
    # a recursive version
    if start >= end:
        return
    yield start
    # continue with more yields, code
    yield from my_range(start+1, end)

r = rec_range(1,5)
print(r)
print("next element:", next(r)) #rarely used this way
for i in r: #usually used like this
    print(i)
print("done with first loop using generator. Trying another loop now...")
for i in r: #already yielded all elements from r previously!
    print(i)
```

```
<generator object rec_range at 0x0000028A18AC7780>
next element: 1
2
3
4
done with first loop using generator. Trying another loop now...
```

In [2]:

```
# We can "yield from" any iterable...
def test():
    yield from {1,3,2} #can yield from any iterable
    yield [1, 4, 5]

print(list(test()))
#note that yielding from a SET may yield elements in any order
```

```
[1, 2, 3, [1, 4, 5]]
```

What does "return" in a generator do? It signals that the generator is done (has no more elements to yield). Interestingly, the "value" returned by a return statement is *irrelevant* (not used):

In [3]:

```python
def my_range(start, end):
    # a recursive version
    if start >= end:
        return "Some other thing that gets ignored"
    yield(start)
    # continue with more yields, code
    yield from my_range(start+1, end)

print(list(my_range(1,5)))
```

```
[1, 2, 3, 4]
```

Reminder -- generators get used up!

In [4]:

```python
nums_1_to_4 = my_range(1,5)
x = list(nums_1_to_4)
y = list(nums_1_to_4)
print("x:", x)
print("y:", y)
```

```
x: [1, 2, 3, 4]
y: []
```

Converting a 'builder' function to an iterator

A good way to think about, and write, a generator is to:

1. write a function that creates a *list* of the all items you want to yield. This typically appends new values onto a "results" list
2. replace the append/add operation with an appropriate yield

For example:

In [5]:

```python
def list_range(start, end):
    result = []
    if start >= end:
        return result #becomes just a return, assuming already yielded all vals
    return [start] + list_range(start+1, end) #becomes a yield, and a yield from

def another_list_range(start, end, result=None):
    if not result:
        result = []
    if start >= end:
        return result #becomes just a return, assuming already yielded all vals
    result.append(start) #becomes a yield
    return another_list_range(start+1, end, result) #becomes a yield from

# And a generator version...
def gen_range(start, end):
    if start >= end:
        return
    yield start
    yield from gen_range(start+1, end)

print("list_range:", list_range(1,5))
print("another_list_range:", another_list_range(1,5))
print("list of gen_range:", list(gen_range(1,5)))
```

```
list_range: [1, 2, 3, 4]
another_list_range: [1, 2, 3, 4]
list of gen_range: [1, 2, 3, 4]
```

In [6]:
```
# Bug: what if we "yield gen_range" at the end instead of "yield from gen_range"?
def bug_range(start, end):
    if start >= end:
        return
    yield start
    yield bug_range(start+1, end)

print("list of bug_range:", list(bug_range(1,5)))
```

```
list of bug_range: [1, <generator object bug_range at 0x0000028A18AC7DB0>]
```

Powerset generator

A generator for all subsets of elements in L; assumes elements in L are unique.

In [7]:
```
# yield all subsets of list L; all elements of L assumed to be unique
def all_subsets(L):
    if len(L) == 0:
        yield set()
    else:
        first = {L[0]}
        for s in all_subsets(L[1:]):
            yield s
            yield first | s
```

In [8]:
```
print(list(all_subsets([1,2])))
```

```
[set(), {1}, {2}, {1, 2}]
```

In [9]:
```
print(list(all_subsets([1,2,3])))
```

```
[set(), {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}]
```

Powerset generator (with print statements to understand operation)

In [10]:
```
# yield all subset of list L; all elements of L assume to be unique
def all_subsets(L):
    print("call all_subsets with L =", L)
    if len(L) == 0:
        print("yield empty set")
        yield set()
    else:
        first = {L[0]}
        print("first =", first)
        for s in all_subsets(L[1:]):
            print("yield s =", s)
            yield s
            print("yield", first, "|", s, "=", first | s)
            yield first | s
    print("return -- nothing more to yield")
```

In [11]:

```
list(all_subsets([1,2]))
```

```
call all_subsets with L = [1, 2]
first = {1}
call all_subsets with L = [2]
first = {2}
call all_subsets with L = []
yield empty set
yield s = set()
yield s = set()
yield {1} | set() = {1}
yield {2} | set() = {2}
yield s = {2}
yield {1} | {2} = {1, 2}
return -- nothing more to yield
return -- nothing more to yield
return -- nothing more to yield
```

Out[11]:

```
[set(), {1}, {2}, {1, 2}]
```

Generator for all subsets of given size

Recall our all_subsets generator

In [12]:

```
# yield all subsets of list L; all elements of L assumed to be unique
def all_subsets(L):
    if len(L) == 0:
        yield set()
    else:
        first = {L[0]}
        for s in all_subsets(L[1:]):
            yield s
            yield first | s
```

In [13]:

```
list(all_subsets([1,2,3]))
```

Out[13]:

```
[set(), {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}]
```

brain-dead version: all_subsets_of_size

A very inefficient all_subsets_of_size:

In [14]:

```
# yield all subsets of L equal in size to size
# NOTE: horrible version not likely to ever earn full credit in 6.009!
def all_subsets_of_size(L, size):
    for s in all_subsets(L):
        if len(s) == size:
            yield s
```

In [15]:

```
print(list(all_subsets_of_size([1,2,3], 2)))
```

```
[{1, 2}, {1, 3}, {2, 3}]
```

In [16]:

```
print(list(all_subsets_of_size(list(range(1,10)), 2))) # try with larger values than 10, e.g.,
25
```

```
[{1, 2}, {1, 3}, {2, 3}, {1, 4}, {2, 4}, {3, 4}, {1, 5}, {2, 5}, {3, 5}, {4, 5}, {1, 6}, {2, 6},
{3, 6}, {4, 6}, {5, 6}, {1, 7}, {2, 7}, {3, 7}, {4, 7}, {5, 7}, {6, 7}, {8, 1}, {8, 2}, {8, 3},
{8, 4}, {8, 5}, {8, 6}, {8, 7}, {1, 9}, {9, 2}, {9, 3}, {9, 4}, {9, 5}, {9, 6}, {9, 7}, {8, 9}]
```

It "works" for small enough L and size, but it's **horrible**. (Why?)

So let's write a direct generator...

direct generator version: all_subsets_of_size

In [17]:

```
# yield all subsets of L equal in size to size
def all_subsets_of_size(L, size):
    return # Exercise left to student
```

In [ ]:

```
print(list(all_subsets_of_size([1,2,3], 2)))
```