Discrete Structures: CMPSC 102

Oliver BONHAM-CARTER

Fall 2022

Week 10

Let's Discuss

Key Questions

How do I use **dynamically generated streams** of data to implement **memory efficient** and **predictable** Python programs?

Learning Objectives

To **remember** and **understand** some the concept of a **monoid**, seeing how it connects to **practical applications** with strings and sequences

Static versus Dynamic

In Chapter 7.1, Stavely Says ...

An input stream, for example, appears to a program to be a sequence of values - lines, characters, numbers from sensors, whatever they may be - that are not present all at once, but appear dynamically over time. Some input streams don't even have an end: the data keeps coming indefinitely.

Simple Definitions

- Static sequences exist as a **complete structure**
- Dynamic sequences are **generated incrementally**
- Dynamically generated sequences are **streams**

Streams: Static variables

What is "Static"?

- **Static**: Does not change. **Dynamic** is able to change.
- A *static* data structure is an organized container or collection of data in memory of a fixed size
- A "static" sequence may be mutable like a `list` but at any one time, it exists as a complete data structure.
- **Static** lists and **Active** lists

Static `list`

```
stringList = ['count_'+str(i+1) for i in range(4)]
```

Active `list`

```
a = 2; b = 3
myList_list = [a+b, b+a, len(["a","b"])]
```

Some Static Lists

Static sequence exists as a complete structure

```
listOfWords = ["this","is","a","list","of","words"]
items = [ word[0] for word in listOfWords ]
print(items) # first chars of each word
# ['t', 'i', 'a', 'l', 'o', 'w']
```

```
print([x+y for x in 'tea' for y in 'pot'])
# ['tp', 'tt', 'ep', ..., 'at']

print([x+y for x in 'tea' for y in 'pot' if x != 't' and y != 'o' ])
# ['ep', 'et', 'ap', 'at']

print ([x+y for x in 'tea' for y in 'pot' if x != 't' or y != 'o' ])
# ['tp', 'tt', 'ep', ..., 'at']
```

```
zeros_list=[0]*5
print(zeros_list)
```

```
item_list=['item']*3
print(item_list) #['item', 'item', 'item']
```

File Input Involves the Use of Streams

```
file = open("emails")
for line in file:
  name, email = line.split(",")
  if name == "John Davis":
    print(email)
```

- The **file** is a **sequence** of characters
- A **character** is a sequence of **numbers**
- Does the entire **file exist** in the computer's memory?
- What would happen if the file was **many gigabytes** in size?
- Are there **alternatives** for static sequences? What are they?

Using Comprehensions and Generators

Dynamic sequence is generated incrementally

Generator Function

```
even_squares =  (x * x for x in range(10)
  if x % 2 == 0)

print(even_squares)
#<generator object <genexpr> at 0x370d878>

for value in even_squares:
    print(value)
```

Output

```
0 4 16 36 64
```

Using Comprehensions and Generators

```
# Note: square backets
list_comprehension = ['Hello' for i in range(3)]
print(list_comprehension)

# Note: curly brackets
generator_expression = ('Hello' for i in range(3))
print(generator_expression)
```

Output: square brackets

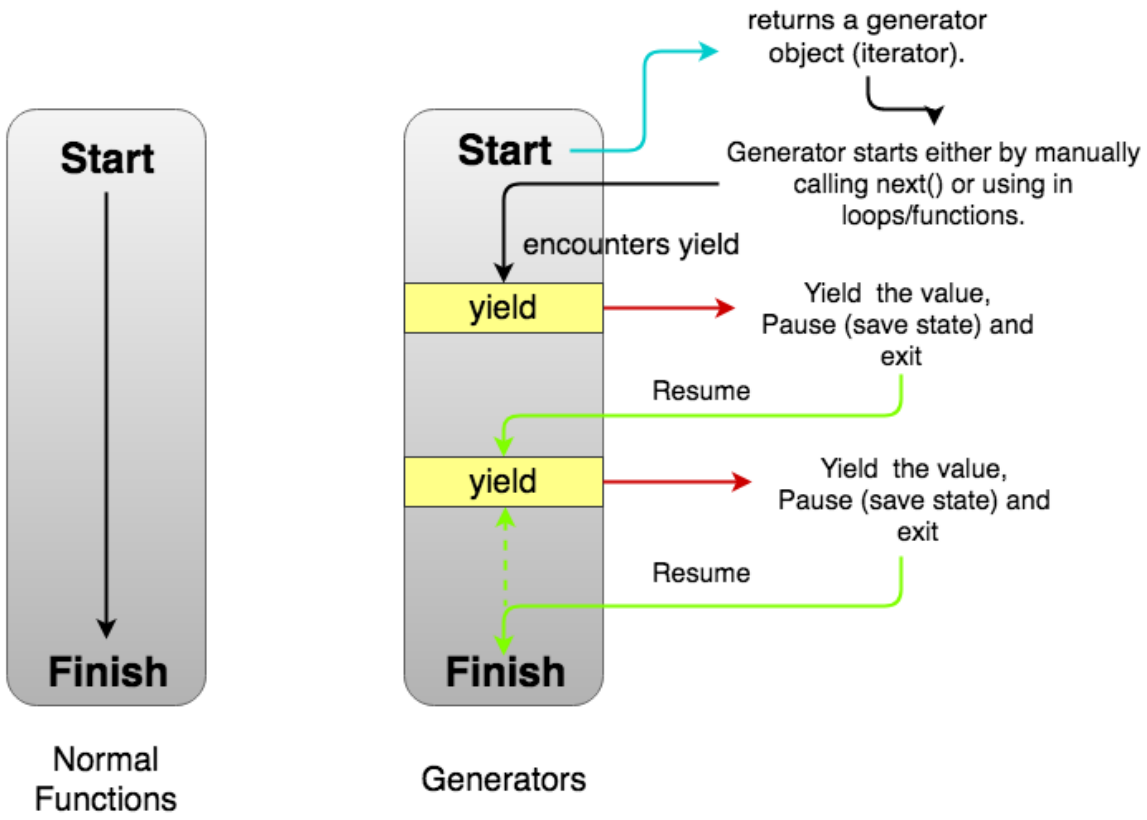
```
['Hello', 'Hello', 'Hello']
```

Output: curly brackets

```
<generator object <genexpr> at 0x2c34cc8>
```

Generator Functions

"Data when you need it!"



- Generators are convenient way of creating iterators.
- They are functions returning objects (iterators) for iteration of one value at a time.

Process List Items When Needed

Another Stream Invoking an iterator

```
l_list = ["Apples", "Oranges", "Apricots",
"Avocado", "Ananas (pineapple)", "Asparagus"]

names = (line[:] for line in l_list) # create a generator
print(names) # generator function, no data added just yet
type(names) # <class 'generator'>

for i in names:print("\t First round :",i)
print("\t Let's try that again! ")

for i in names:print("\t Second round :",i) #... ?
```

- The generator expression is evaluated, creating an iterator, and the *name* variable is bound to that iterator.
- The `for`-statement invokes names for values one after the next
- The `name` generator is then destroyed

The Yield function

File: createGen.py

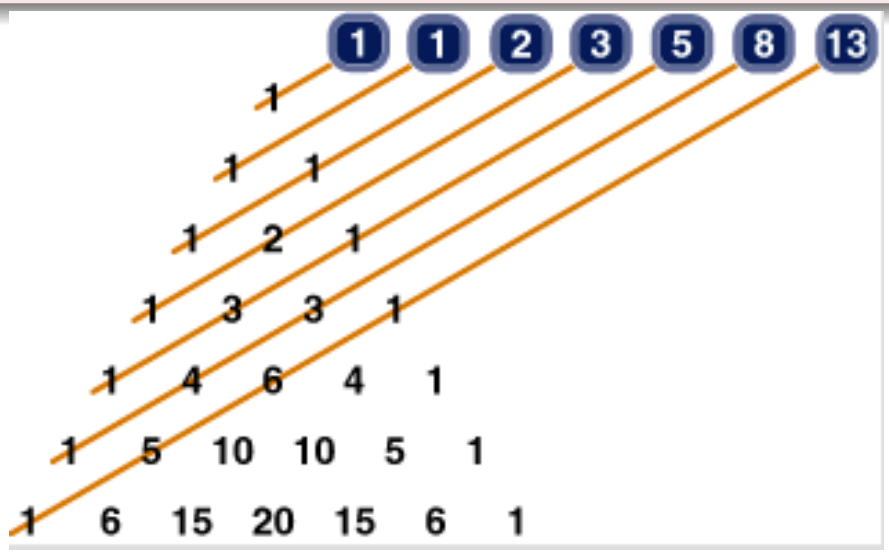
Create another generator

```
def createGenerator():
  mylist = range(3)
  for i in mylist:
  # find the square of the value as needed
    yield i*i
# end of createGenerator()

# Initiation: create a generator
myGenerator = createGenerator()
# Where is this generator in memory?
print(myGenerator)
for i in myGenerator:
  print("\t A: myGenerator: ",i)

for i in myGenerator:
  print("\t B: myGenerator: ",i)
```

Sequences of Fibonacci

Sequence

- $F_n = F_{n-1} + F_{n-2}$
- For $n = 1, 2, \cdots, 8$
- The sequence follows as: 1, 1, 2, 3, 5, 8, 13, 21



Pascal's Triangle to find the sequence

Interesting reference:

`http://mathworld.wolfram.com/FibonacciNumber.html`

Calculate by a Generator Functions

Generator function: Return elements as necessary with `yield`

Make a `tuple` containing the results

```
def fibonacci_generator(n):
    a = 1; b = 1
    for i in range(n):
        yield a
        a, b = b, a + b # increment a and b

print(fibonacci_generator)

for value in fibonacci_generator(10):
    print(value, end=" ")
```

- Generator function `yields` results when they are called.

Binet's Formula: Static Function

Similar to a Generator: Return elements as needed

Binet's Formula

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

- Static equation using Binet's formula

A static function for the Fibonacci sequence

```
import math
def fibsBinet(n):
    a = (1/math.sqrt(5))
    b = ((1 + math.sqrt(5))/2)**n
    c = ((1 - math.sqrt(5))/2)**n
    return a * (b - c)
#end of fibsBinet()
for i in range(8): # calculate each value as needed
    print(int(fibsBinet(i)))
```

Calculate by Tuple-Maker Function

Generator: Return elements of the sequence all at once in a structure

Make a tuple containing the results

```
def fibsTuple(n):
    result = ( )
    a=1
    b=1
    for i in range(n):
        result += (a,)
        a, b = b, a + b
    return result
print("  My type is: ",type(fibsTuple))
print(fibsTuple(5)) #(1, 1, 2, 3, 5)
```

- Every time around the loop, the function creates a new tuple, a copy of result with another value concatenated onto the end. Each tuple but the last is never used again.
- Result is returned in one structure

List-Maker Functions For Fibonacci Sequences

Not a Generator: Return elements of the sequence all at once in a structure

A list maker

```
def fibsList(n):
    # produce a list
    result = [ ]
    a=1; b=1
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result

print("  My type is: ",type(fibsList))
print(fibsList(4)) #[1, 1, 2, 3]
```

- More efficient function than `fibsTuple()`: as a result is modified in place rather than creating a whole new data structure during each iteration
- When n is large the difference may be significant
- Result is returned in one data structure

Generator Functions For Fibonacci Sequences

Creating sequences dynamically with *yield*

- Functions having *yield*-statement are generator
- This function works as a generator or otherwise

A generator function for the Fibonacci sequence

```
def fibs(n):
    a=1
    b=1
    for i in range(n):
        yield a
        a, b = b, a + b
print([x for x in fibs(6)])
print("  My type is:",type(fibs))
f = fibs(6)
for i in f: print(i)
print("  My type is: ",type(fibs(6)))
```