Discrete Structures: CMPSC 102

Oliver BONHAM-CARTER

Fall 2019
Week 7

Streams: Static variables

What is "Static"?

- 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 Actively created lists

Create a static `list`

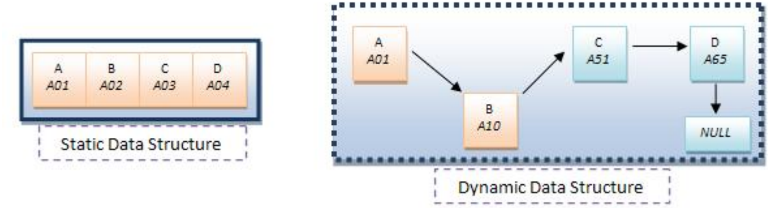
```
stringList = ['count_'+str(i+1) for i in range(4)]
```

Create an active `list`

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

The `lists` are still of a set size.

Dynamic vs. Static Data Structures



The difference between the dynamic and static data structures

- Static data structures are ideal for storing a fixed number of data items, lack the dynamic data structures flexibility to consume additional memory if needed or to free up memory when possible for improved efficiency.

Other Ways to Make Static Lists

https://en.wikibooks.org/wiki/Python_Programming/Lists

Streams
Static
Static vs. Dynamically-Created Sequences
Streams and Iterators
Yield function
Generator Application
Functions to Generate


```
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']
```

Dynamically-Generated Sequences

- The size of the list was settled at the time of the creation of the list
- The list could be printed to the screen item-by-item or all-at-once
- Enter **dynamically generated sequences**: Items are created, printed, consumed as needed.

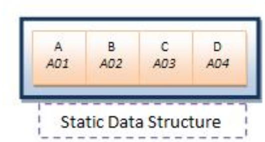
In Chapter 7.1, Stavely Says ...

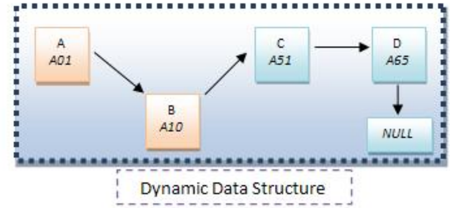
An input stream, for example, appears to a program to be a sequence of values - lines, characters, numbers from sensors, whatever they may be - that are not present all at once, but appear dynamically over time. Some input streams don't even have an end: the data keeps coming indefinitely.

$$\infty$$

Dynamic vs. Static Data Structures

Let's see that graphic again!





The difference between the dynamic and static data structures

Streams and Iterators

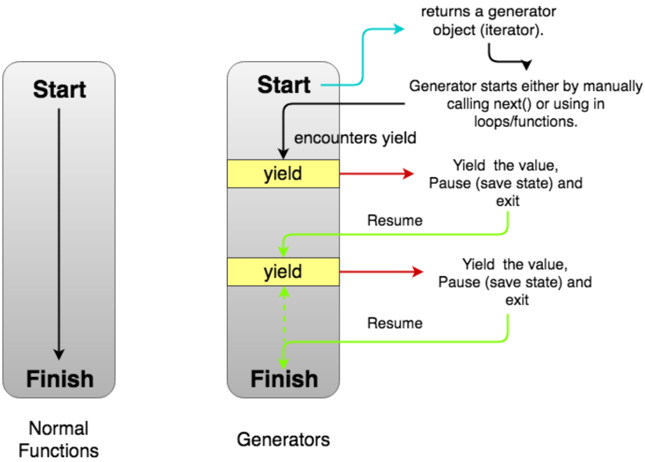
- The term *stream* denotes any dynamically-generated sequence of values
- Two kinds of sequences:
  - Static sequences (similar to any other list that we have already seen)
  - Streams: *generated* data structures using iterators and range objects

Streams by Invoking an iterator with a `for`-statement

```
#for i in iterator:
#   statements
l_list = ["Apples", "Oranges", "Apricots",
"Avocado", "Ananas (pineapple)", "Asparagus"]
print(" Starting with 'A' ")
for line in l_list:
   if line.startswith("A"): print(line)
```

Generators

The overview



- The loop continues only when the `yield` code is run

Using Iterators as Defined By Others

Generator data type: *names*

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

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

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)
```

YAY: *Yet Another Yield*

File: yay.py

{ Let's Code! }

Summations of Large Lists

File: seq_nonGen.py

- Suppose we want to find the sum of all numbers between 1 and n
- We could build a list of these numbers in memory and then add each
- Note: the whole list must be supported by the memory of the machine

```
# Build and return a list
# ref: https://wiki.python.org/moin/Generators
def listBuilder(n):
    num, nums = 0, []
    while num < n:
        nums.append(num)
        num += 1
    return nums
#end of listBuilder()
sum_of_first_n = sum(listBuilder(1000000))
print("\t The sum of first n :",sum_of_first_n)
```

Summations of Large Lists

File: seq_gen.py

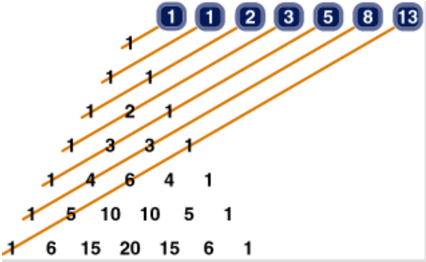
- Suppose we *still* want to find the sum of all numbers between 1 and n but we do not want to use all our memory.
- Generator functions to build the list and get each value as requested

```
# Using the generator pattern (an iterable)
# ref: https://wiki.python.org/moin/Generators
class listBuilder(object):
    def __init__(self, n):
        self.n = n
        self.num, self.nums = 0, []

    def __iter__(self):
        return self

    # Python 3 compatibility
    def __next__(self):
        return self.next()

    def next(self):
        if self.num < self.n:
            cur, self.num = self.num, self.num+1
            return cur
        else:
            raise StopIteration()

sum_of_first_n = sum(listBuilder(1000000))
print("\t The sum of first n:", sum_of_first_n)
```

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

Static Function

The n_{th} term of the Fibonacci sequence

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):
   print(fibsBinet(i)) # calculate each value as needed
```

Tuple-Maker Functions For Fibonacci Sequences

Not a 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):
    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

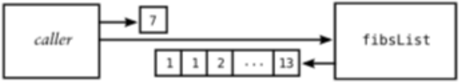
Creating sequences dynamically with *yield*

- 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)))
```

Call versus List Maker



- Non-Generator function: WIth `fibsTuple()` or `fibsList()`, the code that calls the function "pushes" a value of n to the function and the function "pushes" a sequence object back (Click to see Tuples)
- Generator function: With `fibs()`, the caller pushes a value of n to the function and then "pulls" values from the function (or, more precisely, from the iterator returned by the function) as it needs them. (Click to see fibs)

Combinations *(to make another generator function)*

- How many ways are there to choose k things from a set of n?
- Said: *n choose k*
- $Choose(n, k) = \frac{n!}{k!(n-k!)}$

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

Finding Combinations using a Generator

File: combinations.py

python™

{ Let's Code! }