Lecture 12: Generators
Check-in and Reminders

- Pick up **graded Homework 4** from box up front
- Reminder: **Midterm exam on March 12 (Thursday)**
  - Room: TPL 203, 5.45-7.45 pm and 8-10 pm
  - Closed book exam
  - Practice thinking about code on paper
- **Midterm review session:** **Monday, March 9**
  - TPL 203, 7-8.30 pm
  - Come with all your questions
- Next week’s lab (on plotting data) will also be partnered
- It will be short and due the day off

**Do You Have Any Questions?**
Review: Functions

• Functions taken in some input and return some output

• Parameters of a function are “holes” in the body of the function, that are filled in with the argument value for each invocation

• A particular name for a parameter is irrelevant, as long as we use it consistently within the body

```python
def square(x):
    return x*x
def square(num):
    return num*num
def square(apple):
    return apple*apple
```
Review: Function Call Model

- **Function frame.** Model to understanding how a function call works

Return value replaces the function call

```
square (2+3) → square (5) → 25
```

```
square frame

x 5

return x * x
```
```
square frame

x 5

return 5 * 5
```
```
square frame

x 5

return 25
```
Review: Return Statement

• When a function returns a value, where does it “end up”?

• Can a function have **multiple return** statements?
  • How many of them will ever be reached during a particular invocation of the function?

• What happens to the **“control flow” of a program when we hit a return statement** inside a function frame (invocation of a function)
  • Is any code after a return that is reached executed?

• What happens to the **the function frame** (the state of the local variables inside it) **after we hit return**?

• How can a function **return a sequence of multiple values**?

• Is any information that was computed within a function, that is not returned, remembered?
Recall: Variable Scope

- **Local variables.** An assignment to a variable within a function definition creates/changes a local variable.
- Local variables exist only within a functions body, and cannot be referred outside of it.
- **Parameters** are also local variables that are assigned a value when the function is invoked.

```python
def square(num):
    return num*num
```

In [1] square (5)
Out [1] 25
In [2] num
```
NameError: name ‘num’ is not defined
Recall: Variable Scope

def myfunc (val):
    val = val + 1
    print('val = ', val)
    return val

val = 3
newVal = myfunc(val)
Recall: Variable Scope

def myfunc(val):
    val = val + 1
    print('val =', val)
    return val

val = 3
newVal = myfunc(val)
Recall: Variable Scope

def myfunc (val):
    val = val + 1
    print(`val =`, val)
    return val

val = 3
newVal = myfunc(val)
Recall: Variable Scope

def myfunc (val):
    val = val + 1
    print(`val =`, val)
    return val

val = 3
newVal = myfunc(val)

Information flow out of a function is only through return statements!

Function frame destroyed (and all local variables lost) after return from call.
New Type of Functions with Yield

- A function that has a `yield` statement in it is called a **generator function**
- `yield` statement completely changes the behavior of the function

A generator function

```python
def genF(num):
    yield num
```

```python
>>> g = genF(10)
>>> g
<generator object genF at 0x10a55ac50>
```

A “normal” function

```python
def simpleF(num):
    return num
```

```python
>>> f = simpleF(10)
>>> f
10
```

Invoking a generator function creates a **generator object**

Invoking a regular (non-generator) function **returns the output**
Generator Functions

- **genF** does nothing other than *yield* the value that is passed as an argument. Invoking it like a “normal” function does not produce a returned value but results in a **generator object**
- If we call the **next()** method on the generator object **g**, it “yields” or “produces” a value. After it, the generator **g** is exhausted

```python
A generator function

```def genF(num):
    yield num
```

```python
A “normal” function

```def simpleF(num):
    return num
```

```python
>>> g = genF(10)
>>> g
<generator object genF at 0x10a55ac50>
>>> next(g)
10
>>> next(g)
File "<stdin>", line 1, in <module>
    StopIteration
```

Calling next on it again throws a **StopIteration** exception
Understanding Yield

• If a `yield exp` statement is reached, the function’s state is frozen, and the value of the expression `exp` is returned to the `.next()` call.

• That is, all local state of variables is retained, and then function execution is “resumed” when `.next()` is invoked again, and the control flow proceeds exactly where it left off.

• A function can contain multiple `yield` (along with `return`) statements.

Yield vs Return

• **Similarity.** Both `yield` and `return` will return some value from a function to the caller.

• **Difference:** while a return statement terminates the function entirely, the yield statement **pauses** the function (saving all its state) and later continues from there on successive calls.
Mechanics of Generator Functions

- Generator function contains one or more `yield` statement
- When called returns an object (iterator) but does not start execution immediately
- When a generator function yields a value, it is paused and the control is transferred to the caller
- Local variables and their states are remembered between successive calls
- Finally, when the function terminates (either by reaching a return statement or reaching the end of function body), a `StopIteration` is raised automatically on further `.next()` calls
- Such exceptions are handled automatically if iterating over the generator object in a for loop
Generator Functions: Examples

In [1]: def ourSecondGen():
    yield "a"
    yield "b"
    yield "c"

In [2]: genObj = ourSecondGen()

In [3]: next(genObj)
Out[3]: 'a'

In [4]: next(genObj)
Out[4]: 'b'

In [5]: next(ourSecondGen()) #predict the answer!
Out[5]: 'a'

In [6]: next(ourSecondGen())
Out[6]: 'a'

Creates and calls `next()` on new generator object!
CountTo(n) : Three Versions!

In [7]:
```python
def countToPart1(n):
    i = 0
    while i <= n:
        print(i)
        i += 1
```

In [8]:
```
countToPart1(6)
```

Out[8]:
```
0
1
2
3
4
5
6
```

In [9]:
```python
def countToPart2(n):
    i = 0
    while i <= n:
        return i
        i += 1
```

In [10]:
```
countToPart2(12)
```

Out[10]: 0

In [12]:
```python
def countToPart3(n):
    i = 1
    while i <= n:
        yield i
        i += 1
```

In [13]:
```
gObj = countToPart3(6)
```

In [14]:
```
gObj
```

Out[14]:
```
<generator object countToPart3 at 0x10858f250>
```

In [15]:
```
next(gObj)
```

Out[15]: 1

In [16]:
```
next(gObj)
```

Out[16]: 2

In [17]:
```
next(gObj)
```

Out[17]: 3

In [18]:
```
next(gObj)
```

Out[18]: 4
Generating Infinite Sequences

```python
In [7]: def count(start = 0, step = 1): # optional parameters
   i = start
   while True: # read: forever!
       yield i
       print("Now incrementing i=", i)
       i += step

In [8]: g = count()

In [9]: next(g)
Out[9]: 0

In [10]: next(g)
   
   Now incrementing i= 0
   
Out[10]: 1

In [11]: next(g)
   
   Now incrementing i= 1
   
Out[11]: 2

In [12]: next(g)
   
   Now incrementing i= 2
   
Out[12]: 3
```

Can keep going on forever!
Fibonacci Sequence

• Can use generators to generate “infinite series” in a lazy manner

• For example, the fibonacci sequence

  • The fibonacci numbers $F_n$ form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

  • $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-2} + F_{n-2}$ for all $n \geq 2$.

• Named after mathematician Pisa (later called Fibonacci), although it appears in early Indian mathematical texts

• These sequences occur in nature (such as the arrangement of leaves on a steam), the flowering of an artichoke, etc
Generator Function for Fibonacci

- Lets write a generator function that yields the next fibonacci number in the sequence when called.

```python
In [1]: def fibo(a = 0, b = 1):
    yield a
    yield b
    while True:
        a, b = b, a+b
        yield b
```

Optional parameters (by default first parameter `a` is 0, second `b` is 1)

Fibonacci sequence on demand:

```
In [2]: fibN = fibo()
Out[2]:

In [3]: next(fibN)
Out[3]: 0

In [4]: next(fibN)
Out[4]: 1

In [5]: next(fibN)
Out[5]: 1

In [6]: next(fibN)
Out[6]: 2

In [7]: next(fibN)
Out[7]: 3

In [8]: next(fibN)
Out[8]: 5
```
Acknowledgments

These slides have been adapted from:

• http://cs111.wellesley.edu/spring19 and

• https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/