Lecture 13:
Generators and Iterators
Check-in and Reminders

- Submit **Homework 5** according to anonymous ID in 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:** today
  - TPL 203, 7-8.30 pm
  - Come with all your questions
- This week’s lab is on plotting data
- Partner lab, due 11 pm Mon/ Tues

Do You Have Any Questions?
Review: 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
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,

    \[
    F_0 = 0, \quad F_1 = 1, \quad \text{and} \quad F_n = F_{n-2} + F_{n-2} \quad \text{for all} \quad n \geq 2.
    \]

• 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
```
Iterators

- All sequences in Python are **iterable**, they can be iterated over
  - Examples, strings, lists, ranges, tuples, even files
- A Python object is **iterable** if it supports the `iter` function—that is, it has the special method `__iter__` defined—and returns an iterator
- An iterator is something that
  - supports the `next` function (that is, the special method `__next__` is defined and can be called to access the next item)
  - throws a `StopIteration` when the iterator “has run dry”
  - returns itself under an `iter` call
- Iterations may be defined using class (with special methods `__iter__` and `__next__` implemented)
- **Generators provide an easy way to define iterators in Python!**
Generator Iterators

• One can iterate across an object \texttt{obj} using an iterator
• You can ask an object \texttt{obj} for its iterator with \texttt{it = iter(obj)}
• An iterator generates values in the sequence by its \texttt{next()} method

Generator as Iterators

• A generators can be used to implement our own iterator objects
• If \texttt{genObj} is a generator, then \texttt{iter(genObj)} is \texttt{genObj}
• The yield statement produce the next values of the iterator

• \textbf{Question}. How can we iterate implicitly (without calls to next)?
For loop: Behind the Scenes

- A for loop iterates across some object \texttt{obj}. For example:

```python
# a simple for loop to iterate over a list
for item in numList:
    print(item)
```

- The \texttt{for} loop is simply a \texttt{while} loop in disguise, driving an iteration within a \texttt{try-except} statement. The above loop is really:

```python
try:
    it = iter(numList)
    while(True):
        item = next(it)
        print(item)
except StopIteration:
    pass
```

Call the \texttt{iter} method on object to get an iterator. Recall that if \texttt{g} is generator then \texttt{iter(g)} is just \texttt{g}

access the next item if it exists, then print it
What *try-except* does

• The *try/except* statement has the following form:

```python
try:
    <possibly faulty suite>
except <error>:
    <cleanup suite>
```

• The *<possibly faulty suite>* is a collection of statements that has the potential to fail, with error. If occurs, the of statements is executed.

• You can have more than one *except*, handling different types of errors.
Generator Expressions

• Similar to list comprehensions, we can write generator functions with concise expressions

• For example, below is a generator expression to generate all squares from 1 to 10. **Look how concise it is!**

```python
>>> genExp = (i*i for i in range (1, 10))
>>> next(genExp)
1
>>> next(genExp)
4
>>> next(genExp)
9
```
Recall: Random Module

- We used the random module in Python to generate random integers in Lecture 9
- Here we review how to generate random integers in Python

```python
>>> import random
>>> lo = 0, hi = 31
>>> randomIndex = random.randint(lo, hi)  # generates a random integer between lo and hi (both inclusive)
>>> randomIndex # try the above a bunch of times!
5
```
Acknowledgments

These slides have been adapted from:

- [http://cs111.wellesley.edu/spring19](http://cs111.wellesley.edu/spring19) and