Computer Science 134C
Introduction to Computer Science, in Python
Lecture #12 (Generators)
October 3, 2018

Keywords
fibonacci, exception, for loop, generator, next, primes, try-except

We learn how to generate values on demand.

1. Homework out: this homework is due on Wednesday.
2. Questions?
3. A *generator* is an object that constructs a (possibly infinite) stream of values *on demand*.

   (a) Whenever we write a function that mentions the `yield` keyword, the result of the function, when called, is a *generator*.

   ```
   def countTo(n):
       i = 1
       while i <= n:
           yield i
           i += 1
   ```

   (b) The generator object, g, can be asked to compute and return the next value in the sequence by calling `next(g)`. This causes the generator to execute the function until a value is returned with `yield`:

   ```
   >>> g = countTo(3)
   >>> print(next(g))
   1
   >>> print(next(g))
   2
   >>> print(next(g))
   3
   >>> print(next(g))
   Traceback (most recent call last):
     File ''<stdin>'', line 1, in <module>
   StopIteration
   ```

   If you call `next` to get a value from a generator that has run dry, it raises a `StopIteration` exception.

   (c) This exception could be *caught* with a `try-except` statement, but a more efficient mechanism is to use a `for` loop:

   ```
   >>> for v in countTo(10):
   >>>     print(v)
   1
   2
   3
   ```

(d) Generators have the potential to generate an infinite number of values:

```
def count(start = 0, step = 1):
    i = start
    while True:
        yield i
        i += step
```

(e) How would you generate all the Fibonacci numbers? Assume the first two are 1.

(f) How would you generate all prime numbers?

(g) How would compute the orbit of a function `f` on a value `n`? As an example, can you compute `f` as the product of digits of a value.

⋆