4 Generators

Key terms: lazy `next` `yield`

Reading: Dan Bader's Generator Expressions in Python: An Introduction[11]

Reading: Jeff Knupp's Improve Your Python: yield and Generators Explained[12]

Exercise: Write a program that demonstrates a generator yielding one timestamp at a time from `/etc/httpd/logs/access_log`.

Generators, first defined in PEPs 202 and 255, perform lazy evaluation, in which content is calculated only as needed. The reason for generating results on the fly is the advantageous performance of "lazy" evaluation. This approach is related to "pay as you go" accounting and "just in time" manufacturing. It's essential for problems with "big data," when all contents cannot be calculated in advance. Generators can play either or both roles in the producer-consumer model. They can be written in two general ways: as expressions, and as functions.

Creating a Python generator expression is as simple as changing a comprehension's square or curly brackets to parentheses. Rather than slicing the result, we need to consume the results sequentially. This can be done either with a loop or with a comprehension — a mechanism that definitely consumes a certain number of results, without reference to how many of them have been calculated yet.

```
# Demonstrate naive and lazy ways to read the top of a log file;
# the lazy way is many times faster and uses much less memory.

log = '/etc/httpd/logs/access_log'

ip_addresses = [line.split(" ")[0] for line in open(log)]
print (ip_addresses[:10])

ip_addresses = (line.split(" ")[0] for line in open(log))
print([next(ip_addresses) for _ in range(10)])
```

Note that it's easy to destroy laziness by accidentally consuming a whole dataset, especially with `read()` and `readlines()` over file contents. Any line of a program that requires the whole file in hand before proceeding creates a bottleneck that prevents lazy evaluation. Compare to the effect of a bucket brigade worker who insists on receiving *all* buckets before passing any on!

To define a stateful generator function, spell out the specific steps the generator is to take. A generator function uses `yield` instead of `return`, and raises a `StopIteration` event at the end of the data. To iterate over generated results, instantiate a generator object by calling a generator function, then call `next()` with it repeatedly: `next(gen)`, not `gen.next()`.

```
# Demonstrate generation of the Fibonacci sequence;
# Notice the multiple yields.

def fibonacci():
 a, b = 1, 1
 yield a
 yield b
 while True:
  c = a + b
  yield c
  a, b = b, c

f=fibonacci()
print([next(f) for _ in range(20)])
```

Next week, we will example our chronological tools.

[11] `https://dbader.org/blog/python-generator-expressions`

[12] `https://web.archive.org/web/20200307071345///jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/`