4 Generators

Key terms: `next yield`

Reading: Dan Bader's Generator Expressions in Python: An Introduction[12]

Reading: Jeff Knupp's Improve Your Python: yield and Generators Explained[13]

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. This behavior is essential for problems with "big data," as otherwise processing can take so long that no output is ever available. Because the contents are not calculated in advance, a generator object has no `len()` nor slice notation; they only operation they support is access to the next result. 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 explicitly consume the results. 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.

```
# Naive and lazy ways to read the beginning of a log file:

# Try to calculate all in advance, in about 20.00 seconds
log = '/etc/httpd/logs/access_log'
ip_addresses = [line.split(" ")[0] for line in open(log)]
print (ip_addresses[:10])

# Yields ten results at a time, in about 00.02 seconds
ip_addresses = (line.split(" ")[0] for line in open(log))
print([next(ip_addresses) for _ in range(10)])
```

Generators are also written as functions; to iterate over their results, instantiate a generator object by calling a generator function. The iteration happens when the `next()` function is called repeatedly with reference to the object: `next(gen)`, not `gen.next()`.

Define a generator function to define the specific steps the generator is to take. Unlike traditional functions that conclude and return, generators suspend and expect to resume execution later, thus serving as a concurrency alternative to threads. A generator function uses `yield` instead of `return`, and raises a `StopIteration` event at the end of the data. For development you may find it useful to print out data in your generator function and once it looks right, swap `yield` for `print`.

```
# Demonstrate generation of the Fibonacci sequence:

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(10)])
```

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

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