More Generators

Pat Riehecky

Programming Video Journal Club - Session 9

17 March 2021

Why talk about this?

Generators are a fundamental (foundational?) part of the python language. When used cleanly “Generators Will Free Your Mind” (PyData 2014)

Generators help you think about your DATA, not your code base!

Why this talk?

More About Generators - James Powell PyData 2018

https://www.youtube.com/watch?v=m6asOJmfGpY

James Powell has a lot of good talks about generators. This one teases at some of the most interesting bits without skipping over the fundamentals.

He asks a central question, “Why don’t generators show up more in data science?”

Why this talk?

More About Generators - James Powell PyData 2018

https://www.youtube.com/watch?v=m6asOJmfGpY

James Powell has a lot of good talks about generators. This one teases at some of the most interesting bits without skipping over the fundamentals.

He asks a central question, “Why don’t generators show up more in data science?”

Talk Outline

1) Review
2) What Generators Are
3) Why they are not what you think they are
4) “fun”

Review - Computing Principles

From a user perspective, what is the difference between an Anonymous Function and a Named Function? There really isn't one.

What about between an Object and a Named Function? While an Object makes some things easier, there still isn't a clear difference to the end user.

In the end Objects and Closures are identical.

For folks like me who forgot, a Closure basically a function that contains "variables" that are used by a function within the function. James' code is probably clearer than this sentence

Review - The dumb compute function

If a trivial compute function can become an object, it can become a generator:

```
def compute():
    sleep(0.1)
    return randrange(10)

def f():
    rv = []
    for _ in range(10)
        rv.append(compute())
    return rv
```

```
class F:
    def __iter__(self):
        self.size = 10
        return self
    def __next__(self):
        if not self.size:
            raise StopIteration
        self.size -= 1
        return compute()
f = F()
```

```
def f():
  for _ in range(10):
      yield compute()
```

The generator object uses no storage, and can be modified with something like `__call__(self, size)` to permit user flexibility. Lazy fetch saves time/memory.

What Are Generators

Generators are Lazy - Only get a value when you ask for a value.

Itertools - Functions creating iterators for efficient looping - even over unknown (infinite) sizes

A Generator is a sequencing mechanism. A Generator can quietly model an application API within itself.

James quickly touches on what it takes to make Generators use Python's `async` framework. If you don't write the `async` bits, Generators are not `async`.

What Generators Are Not

Python numbers, lists, and generators are not a good tool for computational analysis.

Numpy, super fast. Python no so much...

Generators are not “multi processing” (unless you write the multi processing in there)

Generators don’t have attributes (functions do)

Each element needs to be generated to be returned *in sequence*.

Fun With Generators

Generators can help you construct your program around your data.

James had some issues here, so the code is “a bit sparse”… The example he mentioned is in “Generators Will Free Your Mind” (PyData 2014)

The goal of the example code is clear - a chain of connected values can store their state (from an expensive calculation) and provide it to others or if it needs to be recalculated it can call “expensive operation” only as needed.

The most obvious non-generator version of this code would be written mostly around the “message passing” rather than the data blobs. Generators let you skip over writing the synchronization code by becoming a synchronization interface.