Generators in Python

Square Processing

Let’s say you need to do something with square numbers.

```python
def fetch_squares(max_root):
    squares = []
    for x in range(max_root):
        squares.append(x**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

This works. But…

Maximum MAX

What if MAX is not 5, but 10,000,000? Or 10,000,000,000? Or more?

What if you aren’t doing arithmetic to get each element, but making a truly expensive calculation? Or making an API call? Or reading from a database?

Now your program has to wait… to create and populate a huge list… before the second for-loop can even START.

Lazily Looping

The solution is to create an iterator to start with, which lazily computes each value just as it's needed. Then each cycle through the loop happens just in time.

The Iterator Protocol

Here’s how you do it in Python:

```
class Squares:
    def __init__(self, max_root):
        self.max_root = max_root
        self.root = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.root == self.max_root:
            raise StopIteration
        value = self.root ** 2
        self.root += 1
        return value

for square in Squares(5):
    print(square)
```

There's got to be a better way

Good news. There's a better way.

It's called the **generator**. You're going to love it!

- Sidesteps potential memory bottlenecks, to greatly improve scalability and performance
- Improves real-time responsiveness of the application
- Can be chained together in clear, composable code patterns for better readability and easier code reuse
- Provides unique, valuable mechanisms of encapsulation. Concisely expressive and powerfully effective coding
- A key building block of the async services in Python 3

Yield for Awesomeness

A generator looks just like a regular function, except it uses the `yield` keyword instead of `return`.

```
>>> def gen_squares(max_root):
...     root = 0
...     while root < max_root:
...         yield root**2
...         root += 1
...
>>> for square in gen_squares(5):
...     print(square)
...
0
1
4
9
16
```

Generator Functions & Objects

The function with `yield` is called a **generator function**.

The object it returns is called a **generator object**.

```
>>> def gen_squares(max_root):
...     root = 0
...     while root < max_root:
...         yield root**2
...         root += 1
...
>>> squares = gen_squares(5)
>>> type(squares)
<class 'generator'>
>>> list(squares)
[0, 1, 4, 9, 16]
```

Pop quiz

Create a new file called `gensquares.py`. Type this in and run it:

```
def gen_squares(max_root):
    root = 0
    while root < max_root:
        yield root**2
        root += 1
squares = gen_squares(5)
for square in squares: print(square)
```

It should print:

```
0
1
4
9
16
```

When done: Thumbs up, comment out the `for` loop, and replace it with `print(next(squares))` repeated several times. What does that do?

The next() thing

```
>>> squares = gen_squares(5)
>>> next(squares)
0
>>> next(squares)
1
>>> next(squares)
4
>>> next(squares)
9
>>> next(squares)
16
>>> next(squares)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Future-proofing "next"

```python
def gen_up_to(limit):
    n = 0
    while n <= limit:
        yield n
        n += 1

it = gen_up_to(10)

# Works in Python 3 only
it.__next__()
# Works in Python 2 only
it.next()
# Works in Python 2, 3, 4, ...
next(it)
# next() also lets you supply a default value
next(it, None)
```

Multiple Yields

You can have more than one yield statement.

```
>>> def myitems(top):
...     while top > 0:
...         yield top**2
...         top -= 1
...     yield "All done"
...
>>> for item in myitems(3):
...     print(item)
...
9
4
1
All done
```

Tokenizing

```python
# Produce the tokens/words in a
# string, one at a time.

def tokens(text):
    start = 0
    end = text.find(' ', start)
    while end > 0:
        token = text[start:end]
        yield token
        start = end+1
        end = text.find(' ', end+1)
    yield text[start:]
```

```
>>> body = "int main() { return
0; }"

>>> for token in tokens(body):
...     print(token)
int
main()
{
return
0;
}
```

Imagine we want `tokens()` to immediately stop producing tokens if it encounters the word "EOF". What’s the best way to do that?

Returning

```python
def tokens(text):
    start = 0
    end = text.find(' ', start)
    while end > 0:
        token = text[start:end]
        # Insert the next two lines:
        if token == 'EOF':
            return
        yield token
        start = end+1
        end = text.find(' ', end+1)
    yield text[start:]
```

```
>>> body = "int main() { return
0; } EOF Write comments here!"

>>> for token in tokens(body):
...     print(token)
...
int
main()
{
return
0;
}
```

In a generator function, "return" with no args exits, raising `StopIteration`.

(Older Python versions let you write `raise StopIteration` instead, but that's deprecated, and removed in Python 3.7.)

Lab: Generators

Lab file: `generators/generators.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up…
- … and then do `generators/generators_extra.py`

Instructions: `LABS.txt` in courseware.

NOTE: If the test fails saying it sees `<class 'generator'>`, but expected `<type 'generator'>` - or the other way around - check your Python version.

Scalable Generators

Here’s another way to implement `myitems`:

```
>>> def myitems(top):
...     for x in range(top, 0, -1):
...         yield x**2
...     yield "All done"
...
>>> for item in myitems(3):
...     print(item)
...
9
4
1
All done
```

Same output. But … is there a problem hiding here?

Iterator Protocol

Any object in Python can be an iterator. It just needs to define proper `__iter__` and `__next__` methods.

```
class Squares:
    def __init__(self, max_root):
        self.max_root = max_root
        self.root = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.root == self.max_root:
            raise StopIteration
        value = self.root ** 2
        self.root += 1
        return value

for square in Squares(5):
    print(square)
```

We call this the *iterator protocol.*