Generators in Python

Square Processing

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

```
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]
```

Syntax Exercise

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, 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
```

next() with default

You can pass a second argument to `next()`. If you do, then instead of raising `StopIteration` at the end, it will return that value.

```
>>> # Returns 42:
... nums = gen_up_to(1)
>>> next(nums, 42)
0
>>> next(nums, 42)
1
>>> next(nums, 42)
42
```

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

```
>>> 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

```
>>> 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`.

Raising StopIteration?

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

```
def cause_trouble(x):
    if x > 10:
        raise StopIteration
    yield x
    x += 1
```

```
>>> vals = cause_trouble(11)
>>> next(vals)
Traceback (most recent call last):
  File "<stdin>", line 3, in cause_trouble
StopIteration

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: generator raised StopIteration
```

Scalable Generators

Imagine for a moment you're using Python 2.7.

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?

xrange and range

In Python 2:

```
>>> type(range(5))
<type 'list'>
>>> type(xrange(5))
<type 'xrange'>
```

In Python 3:

```
>>> type(range(5))
<class 'range'>
>>> type(xrange(5))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'xrange' is not defined
```

The larger point:

Your generator functions are only as scalable as their components.

Only As Scalable As...

```
from textlib import words_in_text

def complex_words(text):
    'Text is a string. Generate the words, dropping "a", "an" and "the".'
    words_to_omit = {'a', 'an', 'the'}
    for word in words_in_text(text):
        word = word.lower()
        if word not in words_to_omit:
            yield word
```

Does `words_in_text()` return an iterator? A generator object? Or a list? Or a set?

If the first two, then `complex_words()` is scalable. If the last two, then it's not.

Lab: Generators

Lab file: `generators/generators.py`

- In `labs` folder
- When you are done, study the solution - compare to what you wrote.
- ... and then optionally do `generators/generators_extra.py`

Instructions: `LABS.txt` in courseware.

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