Scalable Code With Generators

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.

Writing Generator Functions

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):
...     for root in range(max_root):
...         yield root**2
...
>>> for square in gen_squares(5):
...     print(square)
```

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):
...     for root in range(max_root):
...         yield root**2
...
>>> 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):
    for root in range(max_root):
        yield root**2

squares = gen_squares(5)
for square in squares:
    print(square)
```

It should print:

```
0
1
4
9
16
```

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;
}
```

Lab: Generators

Lab file: `generators.py`

- In `labs` folder
- When you are done, study the solution - compare to what you wrote.

Instructions: `LABS.txt` in courseware.