Python Generators, Iterators and Comprehensions
Generators: Motivation

The goal is to conveniently provide a means of "lazy evaluation"
Lazy vs Eager Evaluation: Definitions

"Lazy Evaluation" means you write an expression or computation at one time, and it is evaluated at a later time – when it becomes necessary.

"Eager Evaluation" means things are evaluated when they're defined.
Iterators: Why do we need them?

Iterators are the underlying protocol used to implement generators.

Usually best avoided in favor of Generators.
Quick Aside: What is the Fibonacci Sequence?

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...
- The first two numbers are defined as 0, 1
- $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$
- Appears in nature: tree branches, leaves on a stem, pinecones, etc.
Generator Example: The Fibonacci Sequence

# works in Python 2 or Python 3

def fibonacci_sequence():
    a, b = 0, 1
    yield a
    while True:
        yield b
        a, b = b, a + b
Generator example: Using `fibonacci_sequence()`

# works in Python 2 or Python 3

def gen_for_1_second():
    t0 = time.time()

    for number in fibonacci_sequence():
        print(number)

    t1 = time.time()

    if t1 - t0 >= 1.0:
        break
Generator Example: Alternative Means of Use (Python 2)

# works with Python 2

def gen_1st_3():
    gen = fibonacci_sequence()
    print(gen.next())
    print(gen.next())
    print(gen.next())
    print(gen.next())
Generator Example: Alternative Means of Use (Python 3)

# works with Python 3

def gen_1st_3():
    gen = fibonacci_sequence()
    print(next(gen))
    print(next(gen))
    print(next(gen))
    print(next(gen))
class fibonacci_sequence:
    def __init__(self):
        self.a = 0
        self.b = 1
        self.first_time = True
    def __iter__(self):
        return self
        return self
def __next__(self):
    if self.first_time:
        self.first_time = False
        return self.a
    else:
        result = self.b
        self.a, self.b = self.b, self.a + self.b
        return result
Generators and Iterators Summary

- Lazy evaluation of a sequence
- Generators and Iterators look the same to a caller
- The Generator is simpler to write
- Iterators look different in Python 2 and Python 3
- Generators are often the same in Python 2 and 3
Comprehensions and Generator Expressions – first introduced

- List Comprehensions (2.0)
- Generator Expressions (2.4)
- Dictionary Comprehensions (2.7, 3.0)
- Set Comprehensions (2.7, 3.0)
Comprehensions and Generator Expressions: Purpose

- List Comprehensions – Lightweight syntax for creating lists
- Generator Expressions – Lightweight syntax for creating generators
- Dictionary Comprehensions – Lightweight syntax for creating dictionaries
- Set Comprehensions – Lightweight syntax for creating sets
List Comprehension – Simple Example

>>> [2**i for i in range(4)]
[1, 2, 4, 8]
Generator Expression – Simple Example

```python
>>> (2**i for i in range(4))
<generator object <genexpr> at 0xb7483a2c>
>>> gen = (2**i for i in range(4))
>>> print(list(gen))
[1, 2, 4, 8]
```
List Comprehensions Compared to Generator Expressions - Helpers

#!/usr/local/cpython-3.1/bin/python3

def fn(description, x):
    print('%s: %d' % (description, x))
    return x + 1

def main():
    list_comprehension_examples()
    generator_expression_examples()

# main()
Sequential List Comprehension Example

def list_comprehension_examples():
    list1 = [fn('comprehension 1', x) for x in range(3)]
    list2 = [fn('comprehension 2', x) for x in list1]
    for element in list2:
        print(element)
Cascading Generator Expression Example

def generator_expression_examples():
    genexp1 = (fn('genexp 1', x) for x in range(3))
    genexp2 = (fn('genexp 2', x) for x in genexp1)
    for element in genexp2:
        print(element)
Sequential/Cascading Example

Outputs

comprehension 1: 0  genexp 1: 0
comprehension 1: 1  genexp 2: 1
comprehension 1: 2  2
comprehension 2: 1  genexp 1: 1
comprehension 2: 2  genexp 2: 2
comprehension 2: 3  3
2  genexp 1: 2
3  genexp 2: 3
4  4
Dictionary Comprehensions

```python
>>> print({i : chr(65+i) for i in range(4)})
```
Set Comprehensions

```python
>>> print({chr(i+65) for i in range(4)})
{'A', 'C', 'B', 'D'}
```
A Brief Return to Generators: Bidirectional Generators

- These are basically a generator that's able to accept input from the outside
- Naturally used for "coroutines" or "cooperative multitasking"
- Another kind of use follows...
def main():
    gen = generator()
    first_time = True

    while True:
        if random.random() < 0.9 or first_time:
            first_time = False
            value = gen.send(None)
            show_reverse = ''
        else:
            show_reverse = ' manual reverse'
            value = gen.send('reverse')
        time.sleep(0.1)
        print '%2d>%s*%s< %s' % (value, (value - 1) * ' ', (10 - value) * ' ', show_reverse)
def generator():
    position = 1
    direction = 1
    while True:
        received_value = yield position
        position += direction
        assert received_value in [None, 'reverse']

        if received_value == 'reverse' or direction == 1 and position == 10 or direction == -1 and position == 1:
            direction *= -1
Bidi Generator: Daffy Duck:
Example Output

1 >* <
2 > * <
3 > * < manual reverse
2 > * <
1 >* <
1 >* <
2 >* <
3 > * <
4 > * <
5 > * <
For Further Study

- On the difference between Iterables and Iterators:
  http://treyhunner.com/2018/02/python-range-is-not-an-iterator
Fini

We've discussed:

- Generators
- Iterators
- List Comprehensions
- Generator Expressions
- Dictionary Comprehensions
- Set Comprehensions
- Bidirectional Generators
Questions or Comments?