Generators Inside Out

What is this talk about?

- Iterators
- Generators
- Coroutines
- Async
- Async IO

How does *iteration* work?

Iterating over a list

```
for x in [1, 2, 3, 4]:
    print(x)
```

```
1
2
3
4
```

Iterating over a string

```
for c in "hello":
    print(c)
```

```
h
e
l
l
o
```

Iterating over a dictionary

```
for k in {"x": 1, "y": 2, "z": 3}:
    print(k)
```

```
y
x
z
```

The Iteration Protocol

```
>>> x = iter(["a", "b", "c"])

>>> next(x)
'a'
>>> next(x)
'b'
>>> next(x)
'c'
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

win!

```
# Largest word in the dictionary
>>> max(open('/usr/share/dict/words'), key=len)
'formaldehydesulphoxylate\n'
```

Generators

What is a generator?

```
def squares(numbers):
    for n in numbers:
        yield n*n
```

```
>>> for x in squares([1, 2, 3]):
...     print(x)
1
4
9
```

Let me add some prints to understand it better.

```
def squares(numbers):
    print("BEGIN squares")
    for n in numbers:
        print("Computing square of", n)
        yield n*n
    print("END squares")
```

```
>>> sq = squares([1, 2, 3])
>>> sq
<generator object squares at 0xb6c73720>
```

```
>>> next(sq)
BEGIN squares
Computing square of 1
1
>>> next(sq)
Computing square of 2
4
>>> next(sq)
Computing square of 3
9
>>> next(sq)
END squares
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  StopIteration
```

```
>>> for x in squares([1, 2, 3]):
...     print(x)
BEGIN squares
Computing square of 1
1
Computing square of 2
4
Computing square of 3
9
END squares
```

Example: Fibbonacci Numbers

Write a program to find n^{th} fibbonacci number.

```
def fibn(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibn(n-1) + fibn(n-2)
```

```
>>> fibn(10)
55
```

Write a program to compute first n fibbonacci numbers.

```
def fibs(n):
    result = []
    a, b = 1, 1
    for i in range(n):
        result.append(a)
        a, b = b, a+b
    return result
```

```
>>> fibs(10)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

What is the largest fibbonacci number below one million?

```
def largest_fib(upperbound):
    a, b = 1, 1
    while b < upperbound:
        a, b = b, a+b
    return a
```

```
>>> largest_fib(1000000)
832040
```

Issue

Three different implementations to compute fibbonacci numbers!

The generator-based solution

```
def gen_fibs():
    """Generates sequence of fibbonacci numbers.
    """
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b
```

Lets write some generic generator utilities.

```
def first(seq):
    """Returns the first element of a sequence.
    """
    return next(iter(seq))

def last(seq):
    """Returns the last element of a sequence.
    """
    for x in seq:
        pass
    return x
```

```
def take(n, seq):
    """Takes first n elements of a sequence.
    """
    seq = iter(seq)
    return (next(seq) for i in range(n))

def nth(n, seq):
    """Returns n'th element of a sequence.
    """
    return last(take(n, seq))
```

```
def upto(upperbound, seq):
    """Returns elements in the sequence until
    they are less than upper bound.
    """
    for x in seq:
        if x > upperbound:
            break
        yield x
```

```
# what is 10th fibbonacci number?
>>> nth(10, gen_fibs())
55

# find first 10 fibbinacci numbers
>>> list(take(10, gen_fibs()))
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

# find all fibbonacci numbers below 100
>>> list(upto(100, gen_fibs())
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

# What is the largest fibbonacci number below one million?
>>> last(upto(1000000, gen_fibs()))
832040
```

Coroutines

Lets look at this strange example:

```
def display(values):
    for v in values:
        print(v)
        yield

def main():
    g1 = display("ABC")
    g2 = display("123")

    next(g1); next(g2)
    next(g1); next(g2)
    next(g1); next(g2)
```

```
>>> main()
A
1
B
2
C
3
```

Slightly generalized.

```
def run_all(generators):
    # Runs all the generators concurrently
    # stop when any one of them stops
    try:
        while True:
            for g in generators:
                next(g)
    except StopIteration:
        pass

def main2():
    g1 = display("ABC")
    g2 = display("123")
    run_all([g1, g2])
```

```
>>> main2()
A
1
B
2
C
3
```

How about writing a function to print two sets of values?

```
def display2(values1, values2):
    # WARNING: this doesn't work
    display(values1)
    display(values2)
```

```
def display2(values1, values2):
    yield from display(values1)
    yield from display(values2)

def main3():
    g1 = display2("ABC", "XYZ")
    g2 = display2("...", "...")
    run_all([g1, g2])
```

```
>>> main3()
A
.
B
.
C
.
X
.
Y
.
Z
.
```

Lets try to build a simple concurrency library based on coroutines.

```
from collections import deque
_tasks = deque()

def run(task):
    _tasks.append(task)
    run_all()

def spawn(task):
    _tasks.appendleft(task)
    yield
```

```
def run_all():
    while _tasks:
        task = _tasks.popleft()
        try:
            next(task)
        except StopIteration:
            pass
        else:
            _tasks.append(task)
```

Returning a value from a generator

```
def square(x):
    """Computes square of a number using square microservice.
    """
    response = send_request("http://square.io/", x=x)

    # Let something else run while square is being computed.
    yield

    return response.json()['result']
```

```
def sum_of_squares(x, y):
    x2 = yield from square(x)
    y2 = yield from square(y)
    return x2 + y2
```

Generators are overloaded

- Used to build and process data streams
- Also used as coroutines

Confusing!

Native Coroutines

```
async def square(x):
    return x*x

async def sum_of_squares(x, y):
    x2 = await square(x)
    y2 = await square(y)
    return x2+y2
```

Coroutine Protocol

```
>>> square(4)
<coroutine object square at 0xb57a6510>

>>> x = square(4)
>>> x.send(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  StopIteration: 16
```

Running Coroutines

```
def run(coroutine):
    try:
        while True:
            coroutine.send(None)
    except StopIteration as e:
        return e.value
```

```
>>> run(square(4))
16
```

Generator-based coroutines

```
import types

@types.coroutine
def aprint(x):
    print(x)
    yield
```

```
async def display(values):
    for v in values:
        await aprint(v)
```

```
>>> run(display("ABC"))
A
B
C
```

Coroutine Library

```
"""coro.py - a simple coroutine conncurrency library.
"""
import types
from collections import deque

_tasks = deque()

def run(task):
    _tasks.append(task)
    run_all()

@types.coroutine
def spawn(task):
    _tasks.appendleft(task)
    yield
```

```
def run_all():
    while _tasks:
        task = _tasks.popleft()
        try:
            task.send(None)
        except StopIteration:
            pass
        else:
            _tasks.append(task)
```

Async Example

```
from coro import spawn, run
import types

@types.coroutine
def aprint(x):
    print(x)
    yield
```

```
async def display(values):
    for v in values:
        await aprint(v)

async def main():
    await spawn(display("ABC"))
    await spawn(display("123"))

if __name__ == "__main__":
    run(main())
```

Output:

A
1
B
2
C
3

Async IO?

```
"""asocket - simple async socket implementation.
"""
from socket import *
import types
import select

# Rename the original socket as _socket as
# we are going to write a new socket class
_socket = socket
```

```
class socket:
    """Simple async socket.
    """
    def __init__(self, *args):
        self._sock = _socket(*args)
        self._sock.setblocking(0)

    def __getattr__(self, name):
        return getattr(self._sock, name)
```

```
def connect(self, addr):
    try:
        self._sock.connect(addr)
    except BlockingIOError: pass

async def send(self, data):
    await wait_for_write(self._sock)
    return self._sock.send(data)

async def recv(self, size):
    await wait_for_read(self._sock)
    return self._sock.recv(size)
```

```
@types.coroutine
def wait_for_read(sock):
    while True:
        r, w, e = select.select([sock], [], [], 0)
        if r: break
        yield

@types.coroutine
def wait_for_write(sock):
    while True:
        r, w, e = select.select([], [sock], [], 0)
        if w: break
        yield
```

Async IO Example

```
from asocket import *
from coro import spawn, run

async def echo_client(host, port, label):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.connect((host, port))

    for i in range(3):
        await sock.send(str(i).encode('ascii'))
        data = await sock.recv(1024)
        print(label, data.decode('ascii'))
```

```
async def main():
    host, port = 'localhost', 1234
    await spawn(echo_client(host, port, 'A'))
    await spawn(echo_client(host, port, 'B'))
    await spawn(echo_client(host, port, 'C'))
    await spawn(echo_client(host, port, 'D'))

if __name__ == "__main__":
    run(main())
```

```
$ python echo_client.py
B 0
A 0
D 0
C 0
B 1
B 2
A 1
D 1
C 1
A 2
D 2
C 2
```

Questions?

Anand Chitipothu
@anandology