A Gentle Introduction to

# Generators

and

# Coroutines

# Hello!

Hacker/Programmer @ WalletKit, Inc.

Open Source Enthusiast

Python Fanboy

@kirang89

http://kirang.in

# Generators

# Coroutines

# Let's back up a bit

# Functions

Takes some input

Returns some output (or flow of control)

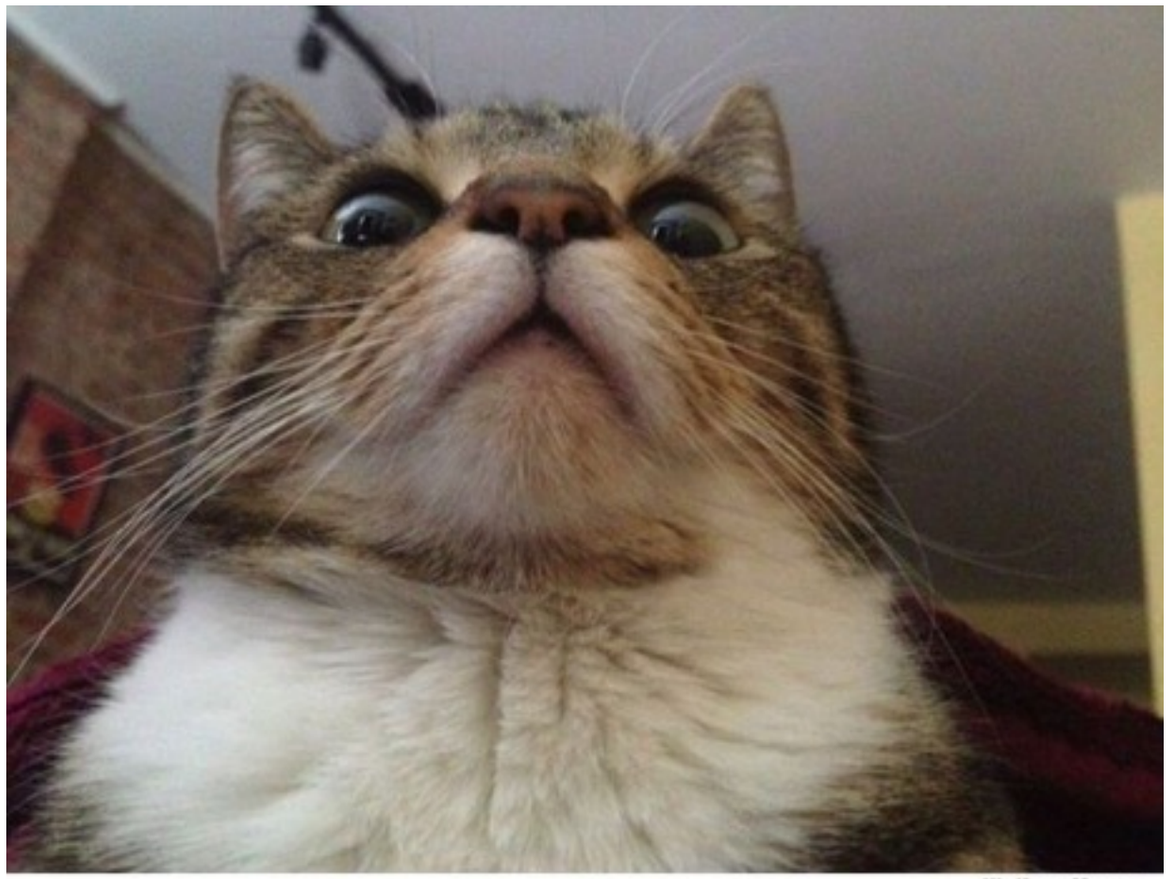
Stateless

Now consider a new type of function

Pretty much the same (Well, almost)

Instead of returning, it **yields** some output

Remembers state after returning flow of control




# yield = return + sorcery*

* It's not all that magical from a computer's perspective

Functions that 'yield' are called

# Generator Functions

a.k.a

# Generators

# Why are Generators awesome?

Coz they're **Lazy**

```
def infinite_seq():
    i = 0
    while True:
        yield i
    i = i +1
```

OR

https://projecteuler.net/problem=10

# Example

```
def fibonacci_gen():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a + b
```

```
>>> f = fibonacci_gen()
>>> type(f)
<type 'generator'>
>>> f
<generator object fibonacci at 0x107ef3870>
```

# Print first 5 terms of the fibonacci series

```
f = fibonacci_gen()
print(next(f))  # Prints 1
print(next(f))  # Prints 2
print(next(f))  # Prints 3
print(next(f))  # Prints 5
```

# Using Generators

- Generator Functions
- Generator Expressions

```python
def even_range(n):
    for i in range(n):
        if i % 2 == 0:
            yield i
```

VS

```python
even_range_10 = (a for a in range(10) if a % 2 ==0)
```

Performance ?

Memory Usage ?

```
kiran:~/ $ python -m timeit "s=[a for a in range(1000000)]”
10 loops, best of 3: 61.3 msec per loop
```

70 MB

```
kiran:~/ $ python -m timeit "s=(a for a in range(1000000))"
10 loops, best of 3: 16.4 msec per loop
```

~66 MB

```
kiran:~/ $ python -m timeit "s=(a for a in xrange(1000000))"
1000000 loops, best of 3: 0.802 usec per loop
```

~3.4 MB

# Generators

# Coroutines

# Co-operative Multitasking/Communication

via

## Enhanced Generators

# Enhanced ?

Allow **values/exceptions** to be passed as **arguments** when a **generator** resumes

yield as a keyword

vs

yield as an expression

| | |
|---|---|
| Receive a value | `value = (yield)` |
| Send a value | `coroutine.send(data)` |
| Close | `coroutine.close()` |

# Example

```
def match(pattern):
    print('Looking for ' + pattern)
    try:
        while True:
            s = (yield)
            if pattern in s:
                print(s)
except GeneratorExit:
    print("Done")
```

```
>>> matcher = match('python')
>>> matcher.send(None)
Looking for python
>>> matcher.send('Hello World')
>>> matcher.send('python is awesome!')
python is awesome!
>>> matcher.close()
Done
```

Let's look at a bit more

# awesome

example

```
def word_count():
    wc = 0
    try:
        while True:
            _ = (yield)
            wc = wc + 1
    except GeneratorExit:
        print "Word Count: ", wc
        pass


def match(pattern, counter):
    print('Looking for ' + pattern)
    try:
        while True:
            s = (yield)
            if pattern in s:
                counter.send(None)
    except GeneratorExit:
        counter.close()
        print("Done")


def parse(file, matcher):
    f = open('sampleset.txt', 'r')
    for line in f.readlines():
        for word in line.split(' '):
            matcher.send(word)
    matcher.close()
```

```
counter = word_count()
counter.send(None)
matcher = match('python', counter)
matcher.send(None)
parse('sampleset.txt', matcher)
```

We've not even scratched the surface!

# Resources

- PEP 255 (Generators)
- PEP 289 (Generator Expressions)
- PEP 342 (Coroutines)
- http://www.dabeaz.com/coroutines/

# Thank you !