Special Topic

# Coroutines

# Yield as an Expression

- In generators, yield can be used as an *expression*
- For example, on the right side of an assignment

```
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = yield
        if pattern in line:
            print line,
```

- Question : What is its value?

# Coroutines

- If you use yield like this, you get a "coroutine"
- It defines a function to which you send values

```
>>> g = grep("python")
>>> g.next()                # Prime it (explained shortly)
Looking for python
>>> g.send("Yeah, but no, but yeah, but no")
>>> g.send("A series of tubes")
>>> g.send("python generators rock!")
python generators rock!
>>>
```

- Sent values are returned by (yield)

# Coroutine Execution

- Execution is the same as for a generator
- When you call a coroutine, nothing happens
- They only run in response to next() and send() methods

```
>>> g = grep("python")
>>> g.next()
Looking for python
>>>
```

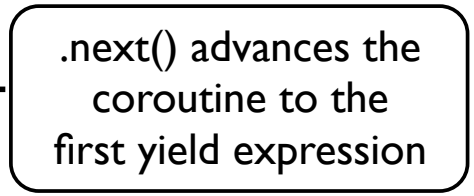
Notice that no output was produced

On first operation, coroutine starts running

# Coroutine Priming

- All coroutines must be "primed" by first calling .next()  (or send(None))
- This advances execution to the location of the first yield expression.

```
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = yield
        if pattern in line:
            print line,
```




- At this point, it's ready to receive a value

# Processing Pipelines

- Coroutines can be used to set up pipelines


- You just chain coroutines together and push data through the pipe with send() operations

# Other Generator Uses

- Generators are also used in other contexts
  - Concurrency (tasklets, greenlets, etc.)
  - Inline control flow (deferred ops, etc.)
- This is a big topic
- Will give a few simple examples

# Example : Concurrency

- Define some "task" functions

```
def countdown(n):
    while n > 0:
        print "T-minus", n
        yield
        n -= 1

def countup(n):
    x = 0
    while x < n:
        print "Up we go", x
        yield
        x += 1
```

- Carefully observe: just a bare "yield"

# Example : Concurrency

- Instantiate some tasks in a queue

```
tasks = deque([
    countdown(10),
    countdown(5),
    countup(20)
])
```

- Run a little scheduler

```
while tasks:
    t = tasks.pop()          # Get a task
    try:
        next(t)              # Run it until it "yields"
        tasks.appendleft(t)  # Reschedule
    except StopIteration:
        pass
```

# Example : Concurrency

- Output

```
T-minus 10
T-minus 5
Up we go 0
T-minus 9
T-minus 4
Up we go 1
T-minus 8
T-minus 4
Up we go 2
...
```

- We see tasks cycling, but there are no threads

# Concurrency Note

- Coroutine based concurrency frameworks are strongly tied to I/O processing
- yield used to wait for I/O events
- A big topic: Beyond the scope of what can be effectively covered here

# Callback Programming

- Coroutines are increasingly being used in Python to simplify callback based programming
- Example: asynchronous I/O

# Example : Async Callback

- Callback function prototype

```
def when_done(result):
    # finish
    ...

def foo():
    ...
    do_something(args,callback=when_done)
```

- Initiates some work, but invokes a callback function upon completion (notification)
- Often leads to a head-exploding mess

# Inlined Callback

- Generators can be used to unify multiple processing stages into a single function

```
def foo():
    ...
    result = yield do_something, args
    # finish
    ...
```

- You use straightforward looking control-flow
- Behind the scenes there is magic
- Example: Twisted inlined deferreds

# More Information

- "A Curious Course on Coroutines and Concurrency" tutorial from PyCon'09

  http://www.dabeaz.com/coroutines

# Sample Code

PythonClass/Solutions/coroutines