

Iteration, Generator and Coroutine in Python

Weng Kai

Based on David Beazley's slides, <http://www.dabeaz.com>

列表

列表

- 列表 (list)
 - 由一系列按照指定顺序排列的元素组成。列表中的元素可以是不同类型。
 - 列表的表示用方括号 ([]) 将元素括起来，元素之间用逗号 (,) 分隔
 - 列表是序列类型的一种，序列所有的特性和操作对于列表都是成立的，除此之外，列表还有自己的特殊操作。

列表

- 列表的创建

1. 直接使用列表的字面量。

`a = []` # 创建一个空列表

`a = [2,3,5,7,11,13]`

2. 使用`list()`将其他数据类型转换成一个列表。

`a = list('hello')`

a的内容是: `['h', 'e', 'l', 'l', 'o']`

`list(range(1,10,2))`

结果是: `[1,3,5,7,9]`

字面量

- 字面量是写在源代码里面的直接可见的值
 - 是计算机内部数据的人可见的形态
 - 是人类表达计算机内部数据的形式
 - 与实际的内部数据是“一体两面”
 - 如0x16, 'hello'是字面量，内部数据的形态与此不同
 - print()以人类可见的形态将计算机内部数据展现给我们看，其结果与字面量也不相同
- 常量是不会变化的量，它可能以字面量的形态存在于源代码中，也可能存在于变量中

列表字面量

- 列表字面量是表达式而非常量
 - `[a, b, c+2]`, 这里的变量和运算都会被计算

列表推导式

- 列表推导式是从一个或者多个列表快速简洁地创建列表的一种方法，又被称为列表解析。它可以将循环和条件判断结合，从而避免语法冗长的代码，同时提高程序性能。
- `[expression for item in iterable]`
- `nl = [2*number for number in [1,2,3,4,5]]`
- `nl`
`[2, 4, 6, 8, 10]`

带条件的列表解析

[expression for item in iterable if condition]

- `>>> nl=[number for number in range(1,8) if number % 2 == 1]`

- `>>> nl`

`[1, 3, 5, 7]`

求和

- 求 $1 + 1/2 + \dots + 1/20$ 之和
- `print(sum([1/i for i in range(1,21)]))`
- 求 $1 - 1/2 + 1/3 - 1/4 + \dots$ 之前 n 项和 ($n \geq 10$)

`n=int(input())`

`print(sum([1/i if i%2==1 else -1/i for i in range(1,n+1)]))`

列表推导式的if条件和条件表达式同时使用

求 $1 - 1/3 + 1/5 - 1/7 + \dots - 1/47 + 1/49$

```
[i if i%4==1 else -i for i in range(1,50) if i %2==1]
```

```
print(sum([1/i if i%4==1 else -1/i for i in range(1,50) if  
i%2==1]))
```

求 $6+66+666+\dots+666\dots666$

- 生成5个6
- `int('6'*5)`

66666

```
n=int(input())
```

```
print(sum([int('6'*i) for i in range(1,n+1)]))
```

多变量的列表推导式

- `[x*y for x in [1,2,3] for y in [10,20,30]]`
- 如何写出以下列表的推导式:
 - `[['A', 1], ['A', 2], ['A', 3], ['B', 1], ['B', 2], ['B', 3], ['C', 1], ['C', 2], ['C', 3]]`

循环

- 循环作为一类控制结构，承载了几种不同的语义
- 基本语义：不断地在不同的数据上做相同的计算
- 不同的语义：
 - 重复确定的次数做计算
 - 对单一（一组）数据做计算以构造或分解数据
 - 在一个序列上遍历，对其中的每个元素做相同的计算

遍历

- Python的for语句用于遍历一个容器

```
for x in [1,4,5,10]:
```

```
    print(x, end=' ')
```

```
1 4 5 10
```

- 不仅限于列表，所有的序列及非序列的容器均可被遍历

可以遍历

- 元组
- 字典
- 字符串
- 文件

枚举协议

```
>>> item = [1, 4, 5]
>>> it= iter(item)
>>> it.__next__()
1
>>> it.__next__()
4
>>> it.__next__()
5
>>> it.__next__()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    it.__next__()
StopIteration
```


for语句

```
for x in obj:
    #statement

_iter = iter(obj)
while 1:
    try:
        x = _iter.__next__()
    except StopIteration:
        break
    #statement
```

- 支持iter()的对象是“可枚举的”

自定义可枚举类型

```
for x in countdown(10):  
    print(x, end=' ')
```

- countdown()是构造函数
- 对象支持
 - __iter__()
 - __next__()

countdown类

```
class countdown(object):  
    def __init__(self, start):  
        self.start = start  
  
    def __iter__(self):  
        return countdown_iter(self.start)
```

```
class countdown_iter(object):  
    def __init__(self, count):  
        self.count = count  
  
    def __next__(self):  
        if self.count <= 0:  
            raise StopIteration  
        r = self.count  
        self.count -= 1  
        return r
```

生成器generator

```
def countdown(n):  
    while n > 0:  
        yield n  
        n -= 1
```

- 生成器是一个函数
- 它不返回值，而是生成一个可枚举的序列

生成器

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

```
>>> x = countdown(10)  
>>> x  
<generator object countdown at 0x7f8908faed60>  
>>>
```

没有输出

- 生成器函数返回的是一个generator对象
- 函数的代码并没有执行

生成器对象操作

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

```
>>> x.__next__()  
begin  
10  
>>> x.__next__()  
9  
>>>
```



开始执行

- 第一次调用__next__()的时候执行了函数
- Yield生成了一个值，然后使得函数挂起
- 调用__next__()恢复了函数的继续运行

结束生成

3

```
>>> x.__next__()
```

2

```
>>> x.__next__()
```

1

```
>>> x.__next__()
```

Traceback (most recent call last):

File "<pyshell#23>", line 1, in <module>

x.__next__()

StopIteration

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

- $n==0$ 时循环结束，函数结束，抛出StopIteration异常

生成器函数

- 用生成器函数是产生可枚举对象的非常方便的方法
- 不再需要实现__next__()和__iter__()等函数

生成器vs可枚举数据

- 生成器函数是一次性的操作，调用一次可以枚举一次，再要枚举就需要再次调用（生成）
- 可枚举数据，如列表，则始终存在，可多次枚举

生成器表达式

```
>>> a = [1, 2, 3, 4]
>>> b = [2*x for x in a]
>>> c = (2*x for x in a)
>>> b
[2, 4, 6, 8]
>>> c
<generator object <genexpr> at 0x7f8908fd3f90>
>>> for i in c:
    print(i, end=' ')
```

- b是列表推导式
- c是生成器表达式，循环的每一轮，c的下一个元素才生成
- c只能用于遍历，一旦被遍历完，就不能再遍历了

生成器表达式

- (expression for i in s if condition)
- 意思是

```
for i in s:  
    if condition:  
        yield expression
```

作为参数值

- 把生成器表达式用作函数的参数值时，外面的圆括号可以省略

```
sum(x*x for x in range(10))
```

Web服务器日志

你的程序会读入一个日志文件，日志文件的每一行可能是以下的形式：

```
10.180.17.246 - - [24/Dec/2017:09:31:35 +0800] "GET /apple-touch-icon.png HTTP/1.1" 404 509 "-" "Safari/13604.4.7.1.3 CFNetwork/893.13.1 Darwin/17.3.0 (x86_64)"
10.180.17.246 - - [24/Dec/2017:09:31:44 +0800] "POST /login.php HTTP/1.1" 302 452 "http://fm.zju.edu.cn/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_2) AppleWebKit/604.4.7 (KHTML, like Gecko) Version/11.0.2 Safari/604.4.7"
10.180.17.246 - - [24/Dec/2017:09:32:17 +0800] "GET /showCourse.php?id=57 HTTP/1.1" 200 1894 "http://fm.zju.edu.cn/index.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_2) AppleWebKit/604.4.7 (KHTML, like Gecko) Version/11.0.2 Safari/604.4.7"
```

程序要从中找出浏览器所访问的网址，统计所有的网址中，访问量最大的网址，输出日志中的访问量和网址。当网址带有GET的请求时要去掉请求数据来统计。当存在多个网址的访问量相同时，按照字母顺序输出。

- 10.180.17.246 - - [24/Dec/2017:09:31:35 +0800] "GET /apple-touch-icon.png HTTP/1.1" 404 509 "-" "Safari/13604.4.7.1.3 CFNetwork/893.13.1 Darwin/17.3.0 (x86_64)"
- 10.180.17.246 - - [24/Dec/2017:09:31:44 +0800] "POST /login.php HTTP/1.1" 302 452 "http://fm.zju.edu.cn/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_2) AppleWebKit/604.4.7 (KHTML, like Gecko) Version/11.0.2 Safari/604.4.7"
- 10.180.17.246 - - [24/Dec/2017:09:32:17 +0800] "GET /showCourse.php?id=57 HTTP/1.1" 200 1894 "http://fm.zju.edu.cn/index.php" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_2) AppleWebKit/604.4.7 (KHTML, like Gecko) Version/11.0.2 Safari/604.4.7"

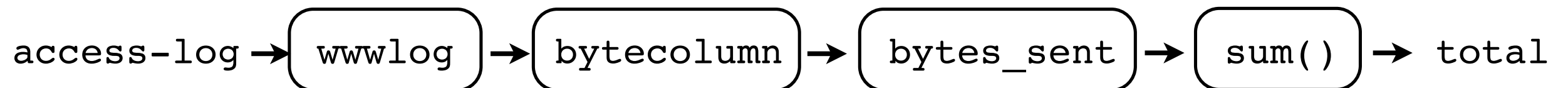
两个版本

```
with open("access-log") as wwwlog:
    total = 0
    for line in wwwlog:
        bytes_sent = line.split()[9]
        if bytes_sent != '-':
            total += int(bytes_sent)
    print(total)
```

```
with open("access-log") as wwwlog:
    bytcolumn = (line.split()[9] for line in wwwlog)
    byte_sent = (int(x) for x in bytcolumn if x != '-')
    print("Total", sum(byte_sent))
```

生成器的流水线

```
with open("access-log") as wwwlog:  
    bytcolumn = (line.split()[9] for line in wwwlog)  
    byte_sent = (int(x) for x in bytcolumn if x != '-')  
    print("Total", sum(byte_sent))
```



- 三个生成器连接
- 每个生成器表达枚举一个元素时要做的操作
- sum()实现了reduce，拉着所需的数据从头走过整条线

表达关系而非动作

- 代码不是表达要一步一步执行的步骤
- 而是表达容器中每个数据的计算关系
 - 这是函数式编程的范儿
- 仍然会想要理解如何执行
 - 命令式语言学久了
 - 计算机的执行是一步步的
 - 描述过程是人类首先掌握的“讲故事”的手段

性能

- 不再需要大规模的临时数据存储
- 有可能利用并行

协程

Generators as Pipelines

- One of the most powerful applications of generators is setting up processing pipelines
- Similar to shell pipes in Unix



- Idea: You can stack a series of generator functions together into a pipe and pull items through it with a for-loop

A Pipeline Example

- Print all server log entries containing 'python'

```
def grep(pattern, lines):  
    for line in lines:  
        if pattern in line:  
            yield line  
  
# Set up a processing pipe : tail -f | grep python  
logfile = open("access-log")  
loglines = follow(logfile)  
pylines = grep("python", loglines)  
  
# Pull results out of the processing pipeline  
for line in pylines:  
    print(line)
```

Yield as an Expression

- You could now use yield 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?

Consumer Func.

```
>>>g = grep('yes')
>>>g.send(None)
Looking for yes
>>>g.send('Oh no')
>>>g.send('Oh yes')
Oh yes
```

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

- Values sent are returned by yield

Coroutine defacto

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

```
>>>g = grep('yes')
```

Nothing happens

```
>>>g.send(None)
```

```
Looking for yes
```

Coroutine starts

```
>>>g.send('0h no')
```

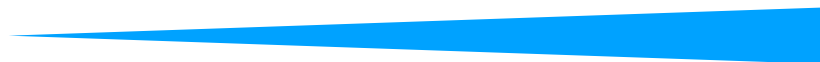
```
>>>g.send('0h yes')
```

```
0h yes
```

Coroutine Priming

- All coroutines must be "primed" by first calling `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)
```



Send(None)
advances the
coroutine to the first
yield

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

Closing a Coroutine

- A coroutine might run indefinitely • Use `.close()` to shut it down

```
>>> g = grep("python")
>>> g.next() # Prime it
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!
>>> g.close()
```

- Note: Garbage collection also calls `close()`

Catching close()

- close() can be caught (GeneratorExit)

```
@coroutine
def grep(pattern):
    print("Looking for %s" % pattern)
    try:
        while True:
            line = (yield)
            if pattern in line:
                (print line)
    except GeneratorExit:
        print("Going away. Goodbye")
```

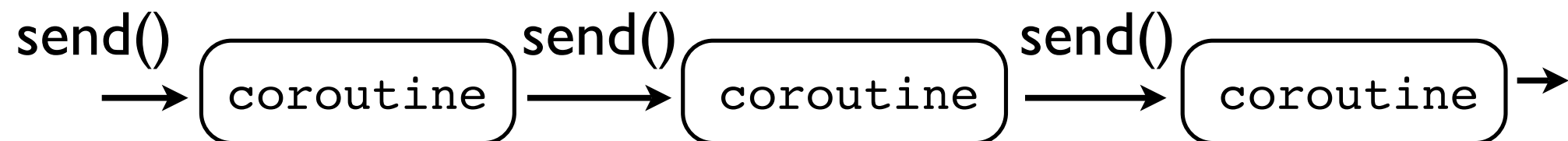
- You cannot ignore this exception
- Only legal action is to clean up and return

Interlude

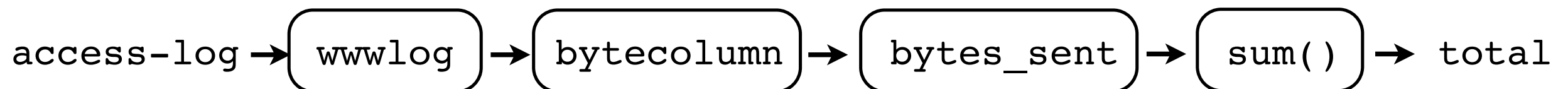
- Generators produce data for iteration
- Coroutines are consumers of data
- Coroutines are not related to iteration

Processing Pipelines

- Coroutines can be used to set up pipes



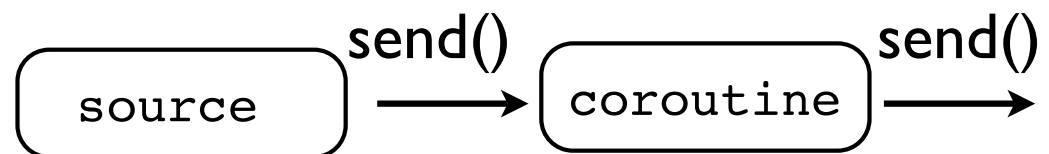
- You just chain coroutines together and **push** data through the pipe with `send()` operations
- Compare with generator



- Key difference. Generators **pull** data through the pipe with iteration. Coroutines push data into the pipeline with `send()`

Pipeline Sources

- The pipeline needs an initial source (a producer)



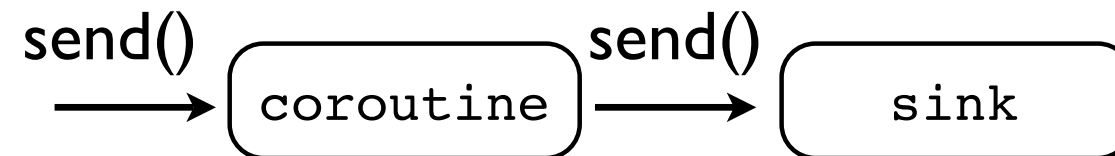
- The source drives the entire pipeline

```
def source(target):  
    while not done:  
        item = produce_an_item()  
        ...  
        target.send(item)  
        ...  
    target.close()
```

- It is typically not a coroutine

Pipeline Sinks

- The pipeline must have an end-point (sink)



- Collects all data sent to it and processes it

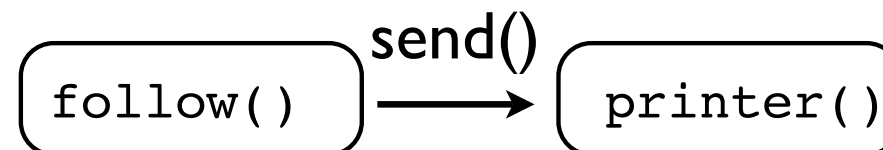
```
@coroutine
def sink():
    try:
        while True:
            item = (yield)
            ...
    except GeneratorExit:
        # Done
        ...
```

An Example of 'tail -f'

```
import time
def follow(thefile, target):
    thefile.seek(0,2)          # Go to the end of the file
    while True:
        line = thefile.readline()
        if not line:
            time.sleep(0.1)     # Sleep briefly
            continue
        target.send(line)
```

A sink. A coroutine that receives data

```
@coroutine
def printer():
    while True:
        line = (yield)
        print(line)
```



```
f = open("access-log")
follow(f, printer())
```

- follow() is driving the entire computation by reading lines and pushing them into the printer() coroutine

Pipeline Filters

- Intermediate stages both receive and send



- Typically perform some kind of data transformation, filtering, routing, etc.

```
@coroutine
def filter(target):
    while True:
        item = (yield) # Receive an item
        # Transform/filter item
        ...
        # Send it along to the next stage
        target.send(item)
```


A Filter Example

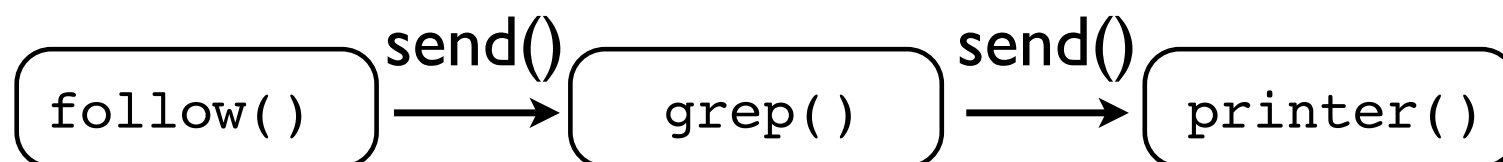
```
@coroutine
def grep(pattern, target):
    while True:
        line = (yield)
        if pattern in line:
            target.send(line)
```

Receive a line

Send to next stage

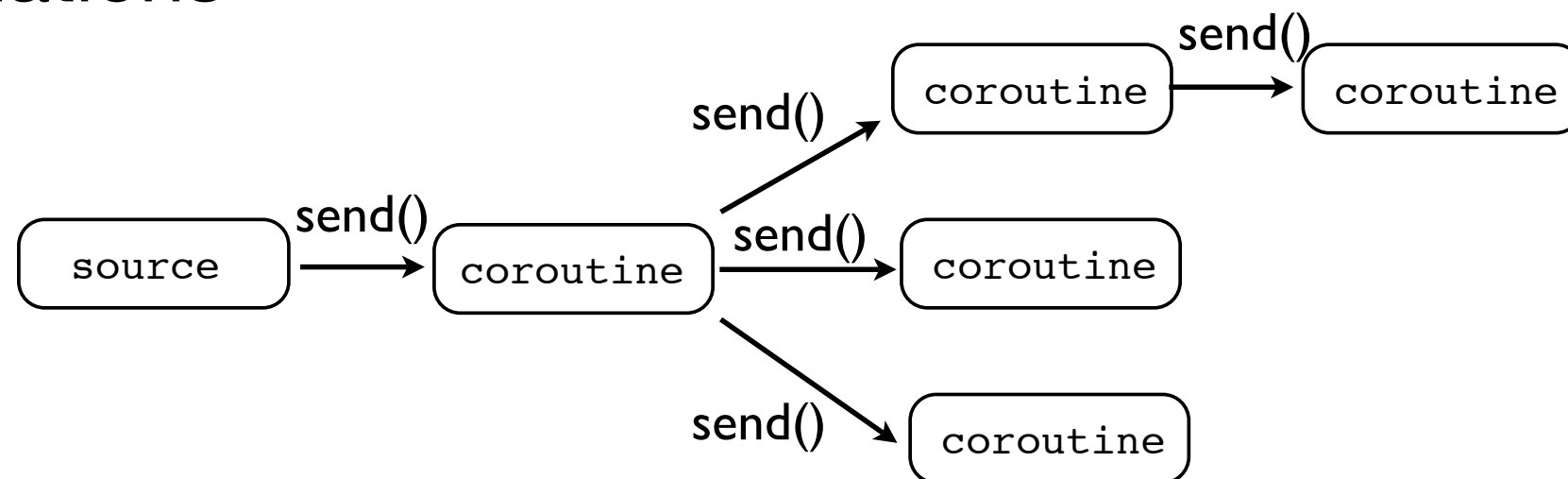
```
f = open("access-log")
follow(f,
      grep('python',
           printer()))
```

```
def follow(thefile, target):
    thefile.seek(0,2)      # Go to the end of the file
    while True:
        line = thefile.readline()
        if not line:
            time.sleep(0.1)  # Sleep briefly
            continue
        target.send(line)
```



Being Branchy

- With coroutines, you can send data to multiple destinations



- The source simply "sends" data. Further routing of that data can be arbitrarily complex

Example : Broadcasting

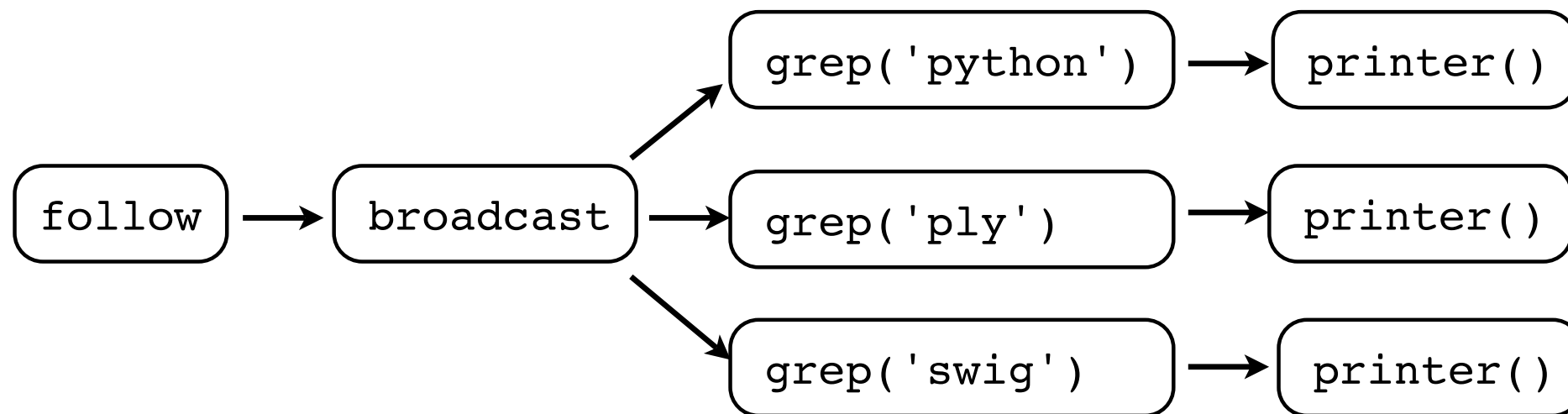
- Broadcast to multiple targets

```
@coroutine
def broadcast(targets):
    while True:
        item = (yield)
        for target in targets:
            target.send(item)
```

- This takes a sequence of coroutines (targets) and sends received items to all of them

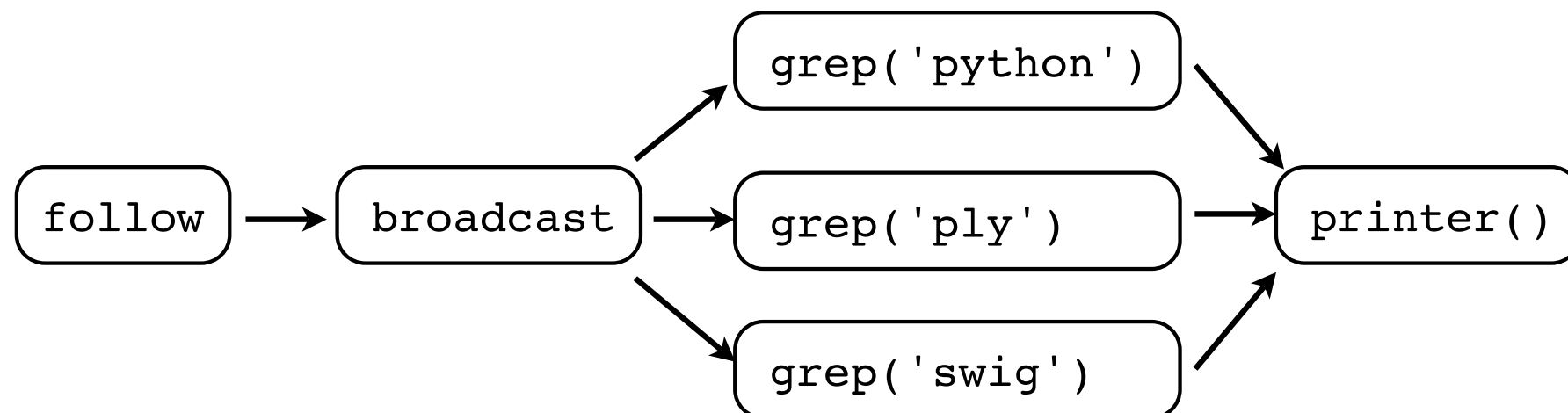
Example : Broadcasting

```
f = open("access-log")
follow(f,
    broadcast([grep('python',printer()),
               grep('ply',printer()),
               grep('swig',printer())])
)
```



Example : Broadcasting

```
f = open("access-log")
p = printer()
follow(f,
    broadcast([grep('python',p),
               grep('ply',p),
               grep('swig',p)]))
```



Interlude

- Coroutines provide more powerful data routing possibilities than simple iterators
- If you built a collection of simple data processing components, you can glue them together into complex arrangements of pipes, branches, merging, etc.
- Although there are some limitations

coroutine vs task

- Coroutines have their own control flow
- Coroutines have their internal own state
 - The locals live as long as the coroutine is active
- Coroutines can communicate
 - The `.send()` method sends data to a coroutine
 - `yield` expressions receive input
- Coroutines can be suspended and resumed
 - `yield` suspends execution
 - `send()` resumes execution
 - `close()` terminates execution

Further Reading


- <http://www.dabeaz.com>

[dabeaz](#) [Courses](#) [Writing](#) [Speaking](#) [Software](#) [About](#) [Newsletter](#) [Chat](#) [Contact](#)

Welcome!

Dabeaz is David Beazley, an independent computer scientist, educator, and researcher with more than 35 years of experience. Dave has been most active in the Python community where he has created various [software packages](#), given conference [talks](#) and [tutorials](#), and is known as the author of the [Python Essential Reference \(Addison-Wesley\)](#) and the [Python Cookbook \(O'Reilly Media\)](#). He supports this work by offering a variety of computer science and programming [courses](#).


Upcoming Events



Holiday of Compilers

December 21-January 1, 2021.
ONLINE with David Beazley


Spend the holidays in front of a computer! Write a compiler for a new programming language and learn about a number of interesting technologies including LLVM and WebAssembly. [More....](#)



Structure and Interpretation of Computer Programs

January 4-8, 2021.
In Chicago with David Beazley

Shatter your brain into bits as you tackle one of the classic texts of computer science. You'll write mostly Scheme with a bit of Python, implement two Lisp interpreters, and explore the foundations of functional programming. You'll likely never look at programming languages the same way again. [More...](#)



Rafting Trip

January 25-29, 2021.
ONLINE with David Beazley

Learn about network programming, concurrency, distributed systems, and more as you tackle the challenge of implementing the Raft distributed consensus algorithm--and likely failing. You'll learn a lot. [More....](#)