

61A Lecture 35

Monday, 28th November, 2011

Last time: sequential data and iterators

Sequences

- The sequence abstraction so far
 - Length
 - Element selection
- Lists and tuples
 - Store all elements up-front
 - can't deal with huge data
 - can't deal with infinite sequences

Iterators

- Store how to compute elements
- Compute one element at a time
- Delay evaluation

Last time: sequential data and iterators

Streams -- a unit of delayed evaluation.

- 2 elements, first and rest.
 - "first" is stored
 - "compute_rest" is stored
 - calculate "rest" on demand

Native python iterator interface

- `__iter__()`
- `__next__()`
- for-loops rely on these methods

Generator functions

- Functions that use `yield` to output values
- Creates a generator object
- `__iter__()` and `__next__()` automatically defined

Today: modularity, processing pipelines, and coroutines

Modularity in programs so far

- Helper functions a.k.a "subroutines"

Coroutines: what are they?

Coroutines in python

Types of coroutines

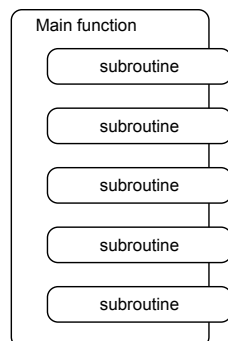
Multitasking

Modularity so far: helper functions

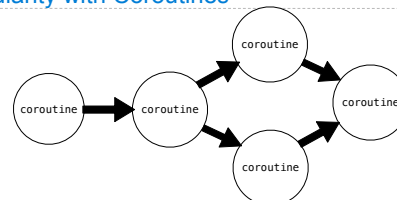
Modularity in programming?

- Helper functions!
 - a.k.a. "subroutines"
- A sub-program responsible for a small piece of computation

A main function is responsible for calling all the subroutines



Modularity with Coroutines

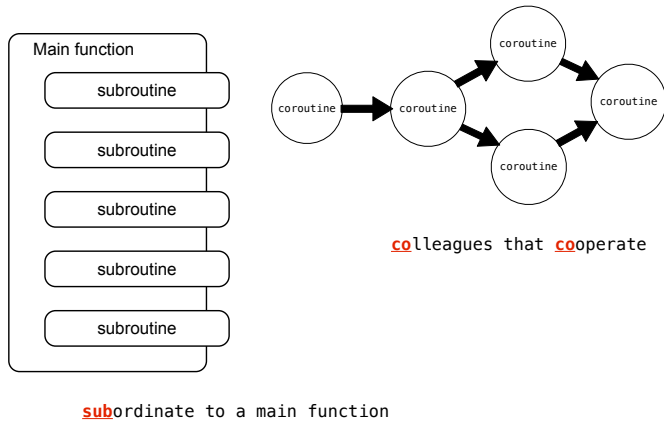


Coroutines are *also* sub-computations

The difference: no main function

Separate coroutines link together to form a complete pipeline

Coroutines vs. subroutines: a conceptual difference



Coroutines in python, or, the many faces of "yield"

Previously: generator functions

• Produce data with yield

```
def letters_generator():
    current = 'a'
    while current <= 'd':
        yield current
        current = chr(ord(current)+1)
```

pauses execution
local variables preserved
resumes when `._next_` is called
returns the yielded value

Now: coroutines

• Consume data with yield

```
value = (yield)
```

pauses execution
local variables preserved
resumes when `.send(data)` is called
assigns value to yielded data

```
send(data)
```

value = (yield)
(yield) returns the sent data.
Execution resumes

Coroutines in Python

Consuming data with yield:

- value = (yield)
- Execution pauses waiting for data to be sent

Send a coroutine data using send(...)

Start a coroutine using `__next__()`

Signal the end of a computation using `close()`

- Raises `GeneratorExit` exception inside coroutine

Example: print out strings that match a pattern

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

execution starts
resumes here, waiting for the data
match found
catch exception

Step 1: Initialize

```
>>> m = match("Jabberwock")
```

does nothing
creates a new object

Step 2: Start with `__next__()`

```
>>> m.__next__()
```

```
'Looking for Jabberwock'
```

Step 3: Send data

```
>>> m.send("the Jabberwock with eyes of flame")
```

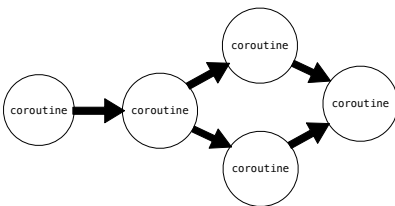
```
'the Jabberwock with eyes of flame'
```

Step 4: close the coroutine

```
>>> m.close()
```

```
'=== Done ==='
```

Pipelines: the power of coroutines

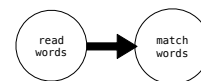


We can chain coroutines together to achieve complex behaviors

Create a pipeline

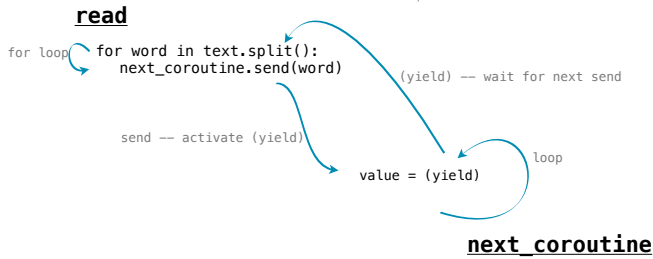
Coroutines send data to others downstream

A simple pipeline

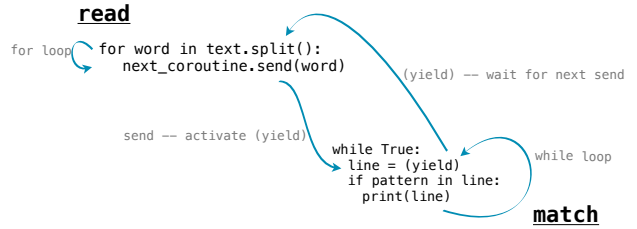


A simple pipeline: reading words

```
def read(text, next_coroutine): needs to know where to send()
    for word in text.split():
        next_coroutine.send(word)
    next_coroutine.close()
```

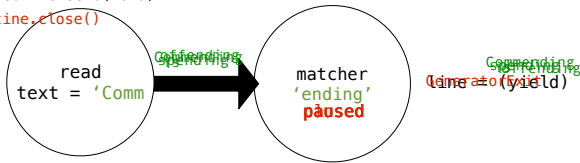


A simple pipeline



A simple pipeline

```
for word in text.split():
    next_coroutine.send(word)
next_coroutine.close()
```

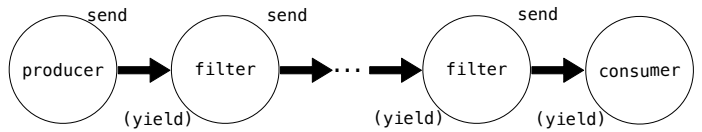


```
>>> matcher = match('ending')
>>> matcher.__next__()
'Looking for ending'
>>> text = 'Commending spending is offending to people pending lending!'
>>> read(text, matcher)
'Commending'
'spending'
'offending'
'pending'
'lending!'
'=== Done ==='
```

last word!

Produce, Filter, Consume

Coroutines can have different roles in a pipeline Based on how they use send() and yield



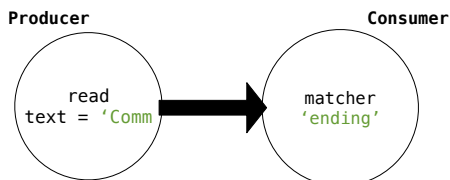
The **producer** only sends data

The **filter** consumes with (yield) and sends results downstream

The **consumer** only consumes data

There can be many layers of filters

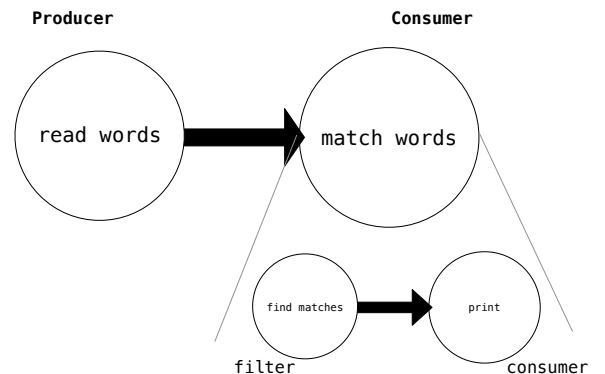
Example: simple pipeline



```
def read(text, next_coroutine):
    for word in text.split():
        next_coroutine.send(word)
    next_coroutine.close()

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

Breaking down match

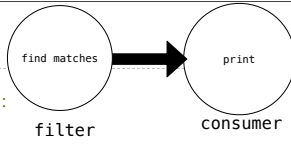


Breaking down match

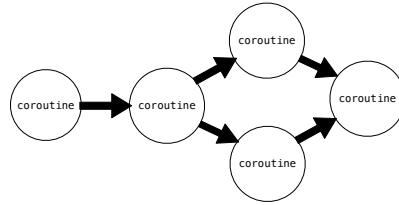
```
def match_filter(pattern, next_coroutine):
    print('Looking for ' + pattern)
    try:
        while True:
            s = (yield)
            if pattern in s:
                next_coroutine.send(s)
    except GeneratorExit:
        next_coroutine.close()

def print_consumer():
    print('Preparing to print')
    try:
        while True:
            line = (yield)
            print(line)
    except GeneratorExit:
        print("=== Done ===")

>>> printer = print_consumer()
>>> printer.__next__()
'Preparing to print'
>>> matcher = match_filter('pend', printer)
>>> matcher.__next__()
'Looking for pend'
>>> text = 'Commending spending is offending'
>>> read(text, matcher)
'spending'
'=== Done ==='
```



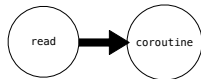
Multitasking



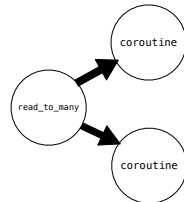
We do not need to be restricted to just one next step

Read-to-many

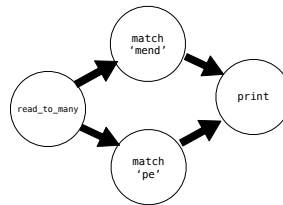
```
def read(text, next_coroutine):
    for word in text.split():
        next_coroutine.send(word)
    next_coroutine.close()
```



```
def read_to_many(text, coroutines):
    for word in text.split():
        for coroutine in coroutines:
            coroutine.send(word)
    for coroutine in coroutines:
        coroutine.close()
```



Matching multiple patterns



Any questions?

```
>>> printer = print_consumer()
>>> printer.__next__()
'Preparing to print'
>>> m = match_filter('mend', printer)
>>> m.__next__()
'Looking for mend'
>>> p = match_filter("pe", printer)
>>> p.__next__()
'Looking for pe'
>>> read_to_many(text, [m, p])
'Commending'
'spending'
'people'
'pending'
'=== Done ==='
```

