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

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**

Subroutine is subordinate to a main function, while coroutine operates in collaboration with other coroutines that cooperate.

---

**Coroutines in Python, or, the many faces of “yield”**

Previously: generator functions

- Produce data with yield
  ```python
def letters_generator():
    current = 'a'
    while current <= 'd':
        yield current
        current = chr(ord(current)+1)
  ```

Now: coroutines

- Consume data with yield
  ```python
  value = (yield)
  ```

```python
send(data)
```

(yield) returns the sent data. Execution resumes.

---

**Coroutines in Python**

Consuming data with yield:

```python
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**

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

Step 1: Initialize

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

Step 2: Start with `__next__()`

```python
>>> m.__next__()
Looking for Jabberwock
```

Step 3: Send data

```python
>>> m.send("the Jabberwock with eyes of flame")
Match found: the Jabberwock with eyes of flame
```

Step 4: close the coroutine

```python
>>> 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**

We can chain coroutines together to achieve complex behaviors.

Create a pipeline

Coroutines send data to others downstream.
A simple pipeline: reading words

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

A simple pipeline

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

A simple pipeline

```python
read
for word in text.split():
    next_coroutine.send(word)
(yield) -- wait for next send
```

A simple pipeline

```python
for loop
for word in text.split():
    next_coroutine.send(word)
read
```

A simple pipeline

```python
for loop
for word in text.split():
    next_coroutine.send(word)
read
send -- activate (yield)
```

A simple pipeline

```python
while True:
    s = (yield)
    if pattern in s:
        print(s)
    except GeneratorExit:
        print("*** Done ***")
```

A simple pipeline

```python
while loop
while loop
```

A simple pipeline

```python
while True:
    line = (yield)
    if pattern in line:
        print(line)
```

A simple pipeline

```python
while loop
while loop
```

A simple pipeline

```python
for loop
for word in text.split():
    next_coroutine.send(word)
read
```

A simple pipeline

```python
for loop
for word in text.split():
    next_coroutine.send(word)
read
send -- activate (yield)
```

A simple pipeline

```python
line = (yield)
if pattern in line:
    print(line)
```

A simple pipeline

```python
next_coroutine.close()
```

A simple pipeline

```python
next_coroutine
```

A simple pipeline

```python
next_coroutine
```

A simple pipeline: reading words

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

A simple pipeline

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

Produce, Filter, Consume

Coroutines can have different roles in a pipeline

Based on how they use send() and yield

```
producer
send
filter
send
call
filter
send
consumer
```

Produce, Filter, Consume

```
producer (yield)
filter (yield)
filter (yield)
consumer (yield)
```

Produce, Filter, Consume

```
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, pattern):
    for word in text.split():
        if pattern in word:
            print(word)
    next_coroutine.send(word)
    next_coroutine.close()
```

Example: simple pipeline

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

Example: simple pipeline

```
def read(text, pattern):
    for word in text.split():
        if pattern in word:
            print(word)
    next_coroutine.send(word)
    next_coroutine.close()
```
Breaking down match

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

```python
>>> printer = print_consumer()
>>> printer.__next__()
'Preparing to print'
```

```python
>>> m = match_filter('pend', printer)
>>> m.__next__()
'Looking for pend'
```

```python
>>> text = 'Commending spending is offending'
>>> read(text, m)
'spending'
```

```python
>>> text = 'Commending spending is offending'
>>> read_to_many(text, [m, p])
'Commending'
'spending'
'people'
'pending'
```

MULTITASKING

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

Read-to-many

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

```python
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

```python
>>> printer = print_consumer()
>>> printer.__next__()
'Preparing to print'
```

```python
>>> m = match_filter('mend', printer)
>>> m.__next__()
'Looking for mend'
```

```python
>>> p = match_filter('pe', printer)
>>> p.__next__()
'Looking for pe'
```

```python
>>> read_to_many(text, [m, p])
'Commending'
'spending'
'people'
'pending'
```

Any questions?

Next time

MAP REDUCE

http://www.infobarrel.com/Top_10_Tips_For_Snowboard_Beginners