ECE326

PROGRAMMING LANGUAGES

Lecture 36 : Lambdas, Generators and Coroutines

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Closure

- Function that can *capture* enclosing environment
- Capture
  - The use of outer variables in a closure
- Heavy use of type inference to make code succinct
- Rust

```
let closure_annotated = |i: i32| -> i32 { i + 1 };
let closure_inferred = |i | i + 1 ;
```

- C++11

```
auto func = [] { cout << "Hello world"; };
auto printi = [](int val) { cout << val; };
```

No parameter closures can omit parantheses ()

Capture

- In C++, captures can be specified in detail

`[]`	Capture nothing
`[&]`	Capture any referenced variable by reference
`[=]`	Capture any referenced variable by copy
`[=, &foo]`	Capture any ref'ed variable by copy, but capture *foo* by reference
`[bar]`	Capture *bar* by making a copy. don't copy anything else
`[this]`	Capture the this pointer of the enclosing class

- Closures are implemented as *functors*
  - Recall that functor is a class that overloads the call operator
  - All captures are stored as member variables

Anonymous Function

- In Rust and C++, closures are also anonymous
  - Name of the closure is unspecified
- In Python, anonymous functions are called lambdas
  - Closure can be named or nameless in Python

```
>> func = lambda x: x + 1
>> func(2)
3
>> full_name = lambda first, last: "%s %s"%(first, last)
>> full_name("Hello", "World")
'Hello World'
>> list(map(lambda x: x*x, range(1, 5)))
[1, 4, 9, 16]
```

Anonymous Function

- Other names
  - Lambda expression
  - Function literals
- Main uses
  - As closure
  - Pass to higher order functions
  - Allows function code to be physically closer to usage

```
def square(x):
        return x*x
… many lines later …
map(square, range(1, 5))
```

```
map(lambda x: x*x, range(1, 5))
```

Iterator Revisited

- To support iteration, implement __iter__ and __next__

```
class CountdownIterator(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

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

```
for n in countdown(5):
        print(n, end= " ")

5 4 3 2 1
```

Returns an iterator object!

Generator

- A function that can pause and resume where it left off
  - Requires its own stack and context to preserve state
    - Similar to a thread
- Context
  - The data that must be saved to resume a thread after a switch
    - Most important of which are processor registers
- Generator produces a sequence of results
  - Instead of a single value, *without creating temporary lists!*
  - Similar to an iterator, but *much easier to write*

Generator

- Uses *yield* instead of *return* to return value
- Has similar interface to an iterator
  - Not callable, but next() will resume the function
  - Raises StopIteration when the generator finishes

```
def generator(n):
      yield n+1
      yield n-1
      yield n

>> g = generator(3)
>> print(type(g))
<class 'generator'>
```

```
>> next(g)
4
>> next(g)
2
>> next(g)
3
>> next(g) # exception occurs here
StopIteration
```

Generator

- Calling a generator creates a generator object
  - The function starts running on the first next() invocation
- A Generator object cannot be reused
  - must create new one

```
Output:

About to consume g
About to launch
5 4 3 2 1
```

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

g = countdown(5)
print("About to consume g")
for x in g:
        print(x, end=" ")
```

Yield From

- Delegates iteration to another iterator
- Allow you to chain multiple iterators together

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

def countup(stop):
      n = 1
      while n < stop:
            yield n
            n += 1
```

```
def up_and_down(n):
      yield from countup(n)
      yield from countdown(n)

>> for x in up_and_down(3):
..      print(x)
1
2
3
2
1
```

Generator Expression

- Similar to list expression, but creates generator not list
  - `( P(x) for x in iterable if Q(x) )`
- Reduces memory footprint if you don't need the list
  - Python has been moving towards using lazy iterators
    - E.g. range() used to create a list instead of a range object

```
>> a = [1, 2, 3, 4]
>> b = (2*x for x in a)
>> b
<generator object at 0x58760>
>> for i in b:
..     print(b, end=' ')
2 4 6 8
```

Example

- Apache web server log file
  - Process the log file to find total bytes send to clients
  - Log file can be huge (in gigabytes)
    - Interpreter may run out of memory with list comprehension
  - Ends with bytes sent or “-” in case of error

```
81.107.39.38 - ... "GET /ece326.html HTTP/1.1" 200 2359
66.249.72.134 - ... "GET /index.html HTTP/1.1" 200 4447
81.107.39.38 - ... "GET /DoesNotExist/ HTTP/1.1" 404 -
```

Use generator here instead of list!

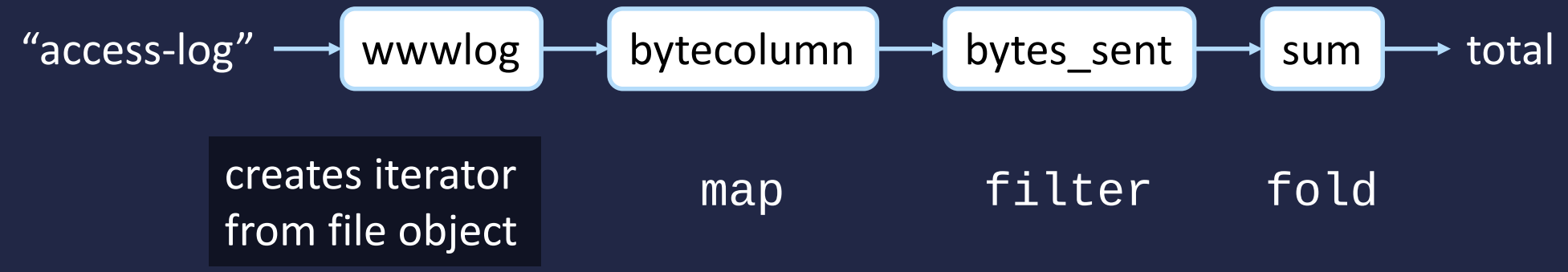
```
with open("access-log") as wwwlog:
    bytecolumn = (line.rsplit(maxsplit=1)[1] for line in wwwlog)
    bytes_sent = (int(x) for x in bytecolumn if x != '-')
    print("Total", sum(bytes_sent))
```

Pipelining

- Pulling input through set of data processing elements

```
with open("access-log") as wwwlog:
    bytecolumn = (line.rsplit(maxsplit=1)[1] for line in wwwlog)
    bytes_sent = (int(x) for x in bytecolumn if x != '-')
    print("Total", sum(bytes_sent))
```

- Uses familiar higher order functions
  - No need to create temporary list, enables high performance



Functional Programming

- Writing declarative expressions
  - No statements, everything is an expression
- Prefers immutability
  - Enables *pipelining* of data from one function to another
- Uses higher order functions instead of control flow
- Verdict:
  - Good for processing large amount of data
  - Poor for simulating real life interactions
    - FP requires *no side effect*, but real life objects are frequently stateful

Similar Constructs

- Mutual recursion
  - Two functions calling each others

```
def foo(n):
    return bar(n-1) if n > 0 else 0
```

```
def bar(n):
    return print(foo(n-1))
    return n
```

  - Each function always starts from beginning
    - Shares a stack
- Cooperative thread
  - Thread voluntarily yields, triggers scheduler immediately
    - E.g. by calling thread_yield(next_id)
  - Cannot pass data between threads through interface

Coroutines

- Cooperative threads with two-way communication
  - Generators are only one-way
- In Python, coroutines are implemented as generators
  - The only difference is that it receives input via send()

```
def recv_count():
    try:
        while True:
            n = yield
            print(n, end=“ ”)
    # triggers upon close()
    except GeneratorExit:
        print("Kaboom!")
```

```
r = recv_count()
# required to “prime” the function
r.send(None)
for i in range(5, 0, -1):
    r.send(i)
# closes the generator object
r.close()
```

5 4 3 2 1 Kaboom!

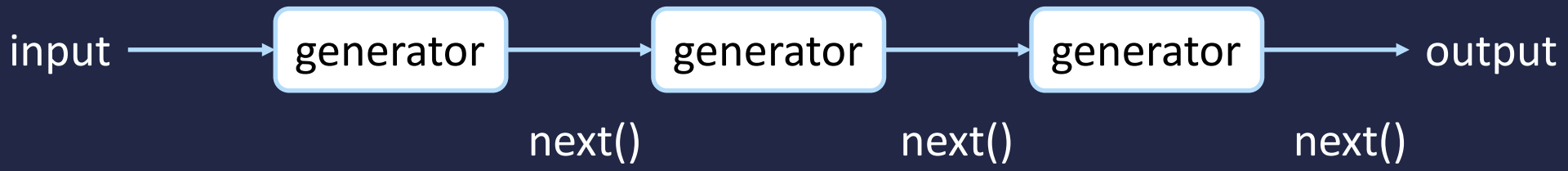
Priming Coroutine

- Use a decorator

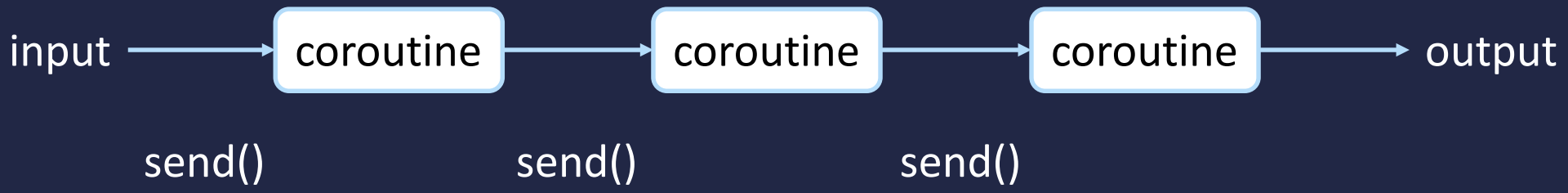
```
def coroutine(func):
        def start(*args,**kwargs):
                cr = func(*args,**kwargs)
                cr.send(None)
                return cr
        return start

# using decorator the manual way
@coroutine
def recv_count():
        …

# don't have to prime it anymore!
r = recv_count()
for i in range(5, 0, -1):
        r.send(i)
r.close()             # still have to close it manually
```

Pipelining

- Coroutines are frequently used only for sending
- Generators actively *pulls* data from its iterable



- Coroutines passively waits for data to be *pushed*



Coroutines

- When next() is called, function stops at yield
- If send() is called, its return value is the next yield
- If send() not called before next(), yield returns None

```
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided,
        # change counter
        if val is not None:
            i = val
        else:
            i += 1
```

```
it = counter(10)
print(next(it), next(it))
print(it.send(8))
print(next(it))
print(next(it))
```

```
0 1
8
9
StopIteration
```