Lecture 4: Functional Programming

Chirag Sakhuja

Overview

Recap

List comprehensions

Lambdas

Map/filter/reduce

Generators

Coroutines

Decorators

Conclusion

"The Zen of Python" – Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.

LBE: Max of a list (without the library)

```
def max(nums):
    if nums:
        res_idx = 0
        for idx, val in enumerate(nums):
            if val > nums[res_idx]:
                res_idx = idx
        return res_idx, nums[res_idx]

idx, val = max([3, 2, 1])    # => (0, 3)
idx, val = max([])           # => ???
```

Default arguments

```
def func(x, y=0):
    return x + y

func(1)         # => 1
func(1, 2)      # => 3
```

Keyword arguments

```
def func(x, y=0):
    return x + y

func(1)         # => 1
func(1, y=2)    # => 3
func(y=2, 1)    # => SyntaxError!
```

Positional arguments must come before keyword arguments

Variadic positional arguments

```
def func(*args):
    print(*args, sep=', ')
```

Unpack the tuple as individual arguments to print

```
print(0, sep=', ')             # => 0
print(1, 2, 3, sep=', ')       # => 1, 2, 3
func(0)                        # => 0
func(1, 2, 3)                  # => 1, 2, 3
```

Variadic keyword arguments

Excess keyword arguments are packed into dict

```
def cite(quote, **info):
    print('>', quote)
    print('-' * (len(quote) + 2))
    for k, v in info.items():
        print(k, v, sep=': ')

cite('Readability counts.',        # => > Readability counts.
     Title='The Zen of Python',          ---------------------
     Author='Tim Peters')                Title: The Zen of Python
                                         Author: Tim Peters
```

Variables bind to nearest scope

x in local and global scope

```
x = 1
def foo(x):
    z = 3
    print(locals())
    print(globals())

foo(2)     # => {'z': 3, 'x': 2}
           # => { ..., 'x': 1, ...}
```

It picks the nearest scope for locals

But globals are unchanged

Functions are objects

```
def add(x, y):
    return x + y

def operate(fun, x, y):
    return fun(x, y)

operate(add, 1, 2)    # => 3
```

Function factory

```python
def factory(x):
    def helper(y):
        return x + y
    return helper

add1 = factory(1)
add2 = factory(2)
add1(10)        # => 11
add2(10)        # => 12
factory(3)(10)  # => 13
```

Overview

- Recap
- List comprehensions
- Lambdas
- Map/filter/reduce
- Generators
- Coroutines
- Decorators
- Conclusion

List comprehensions

- Syntactic sugar to transform some for loops into a single line
- Useful to easily generate or filter lists (or any iterable object)
- List comprehensions create new lists

Basic syntax

Loop iterator variable

```
[f(x) for x in iterable]
```

Function to call per iteration

Object being iterated over

List comprehension functionality

```
[1, 2, 3, 4, 5]

[f(x) for x in iterable]

[f(1), f(2), f(3), f(4), f(5)]
```

List comprehensions are concise!

```
nums1 = list()
for i in range(5):
    nums1.append(i ** 2)
    # => [0, 1, 4, 9, 16]

nums2 = [i ** 2 for i in range(5)]
# => [0, 1, 4, 9, 16]

nums1 == nums2    # => True
```

Ugly list copy

```python
nums = [1, 2, 3]
copy = [x for x in nums]

nums == copy    # => True
```

Iterate over any iterable thing

```python
letters = '01234'
nums = [int(x) + 1 for x in letters]
# => [1, 2, 3, 4, 5]

nums = [x ** 2 for x in nums]
# => [1, 4, 9, 16, 25]
```

Predicates

```
[f(x) for x in iterable if cond(x)]
```

Optional predicate

Predicate functionality

```
[1, 2, 3, 4, 5]

[f(x) for x in iterable if cond(x)]

[f(1), f(3), f(4)]
```

Only values that pass the predicate get processed

Filtering

```python
nums = [1, 2, 3, 4, 5]

filtered1 = list()
for x in nums:
    if x % 2 == 0:
        filtered1.append(x)
        # => [2, 4]

filtered2 = [x for x in nums if x % 2 == 0]
# => [2, 4]
```

LBE: Length of the longest lower-case string

```python
s = 'Hi my name is Chirag'
max_len = max(
    [len(x) for x in s.split(' ') if x.islower()]
)
# => 4
```

Going further…sort of

- Nested list comprehensions
  - Simple is better than complex.
  - Readability counts.
- Dictionary comprehensions
- Set comprehensions
- Generator comprehensions

Overview

- Recap
- List comprehensions
- Lambdas
- Map/filter/reduce
- Generators
- Coroutines
- Decorators
- Conclusion

Operator overloading in C++

```
struct Foo {
    int operator+(int x) { return x + 1; }
};

int main(void) {
    Foo x;
    std::cout << x + 1;    // => 2
}
```

Function objects in C++

Parenthesis are an operator in C++

```
struct Foo {
    int operator()(int x) { return x + 1; }
};

int main(void) {
    Foo x;
    std::cout << x(1);      // => 2
}
```

"Indistinguishable" from a function call!

Looking back at Python

```
def factory(x):
    def helper(y):
        return x + y
    return helper
```

helper is a function object

```
add1 = factory(1)
add1(10)          # => 11
factory(2)(10)    # => 12
type(add1)        # => <class 'function'>
```

Lambda functions

- Concise syntax for function objects we've seen before
- Useful when it's easier to create a function inline
- Creates its own scope, similar to inner functions
- Should be unnamed

Lambda syntax

Arguments

```
lambda x, y: x * y
```

Operation

Contrived example

```python
def add(x, y):
    return x + y
add(2, 3)                     # => 5

(lambda x, y: x + y)(2, 3)    # => 5
```

We'll see something meaningful shortly

More contrived examples

```python
lambda x: x ** 2
```

```python
lambda tup: tup[0] + tup[1]
```

Function factory with lambdas (bad!)

```
def factory(x):
    return lambda y: x + y

add1 = factory(1)
add1(10)            # => 11
factory(2)(10)      # => 12
type(add1)          # => <class 'function'>
```

Bad use of lambdas! If it's going to be given a name, don’t use a lambda!

Bad use of lambdas! Look at the wacky scoping!

Lambdas should be used inline

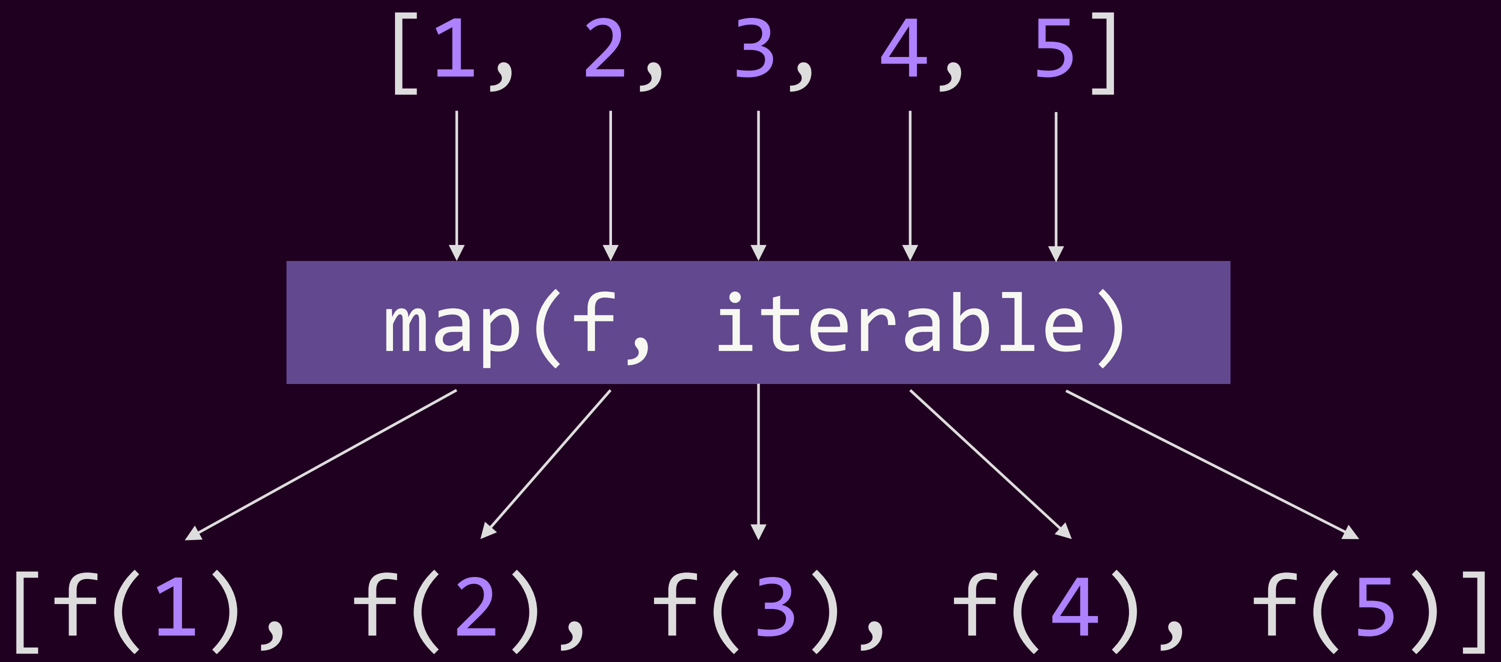
Overview

- Recap
- List comprehensions
- Lambdas
- Map/filter/reduce
- Generators
- Coroutines
- Decorators
- Conclusion

Map function, what's the point?

- Applies a function for each element in an iterable
  - A less powerful list comprehension
- Python community has deemed comprehensions more clear
- Lazy evaluation
  - Generator comprehensions can also do this
- Comprehensions have supplanted map/filter functions in Python, but they are fundamental to other functional programming languages
  - I know…this is supposed to be a Python class…

Map functionality



Output isn't really a list...

Length of strings

```
parts = 'Hi my name is Chirag'.split(' ')

l1 = list()
for x in parts:
    l1.append(len(x))
    # => [2, 2, 4, 2, 6]

l2 = [len(x) for x in parts] # => [2, 2, 4, 2, 6]

l3 = map(len, parts) # => <map object at 0x7f5b485ccf60>
```

What?! Hold that thought

Maps and lambdas

```
nums1 = list()
for i in range(5):
    nums1.append(i ** 2)
    # => [0, 1, 4, 9, 16]

nums2 = map(lambda x: x ** 2, range(5))
# => <map object at 0x7f5b485ccf98>
```

Lazy evaluation

RIP computer

```
[i ** 2 for i in range(10 ** 100)]
```

```
map(lambda x: x ** 2, range(10 ** 100))
```

This actually doesn't do anything because of lazy evaluation, brought to you by generators

Small aside: generator comprehensions

```
map(lambda x: x ** 2, range(10 ** 100))
```

```
(i ** 2 for i in range(10 ** 100))
```

Notice parentheses instead of brackets

Both of these use lazy evaluation

Explicit conversion to list

```
nums1 = list()
for i in range(5):
    nums1.append(i ** 2)
    # => [0, 1, 4, 9, 16]

nums2 = list(map(lambda x: x ** 2, range(5)))
# => [0, 1, 4, 9, 16]
```

Filter function, what's the point?

- Selects elements from an iterable
  - A less powerful list comprehension
- Python community has deemed comprehensions more clear
- Lazy evaluation
  - Generator comprehensions can also do this
- Comprehensions have supplanted map/filter functions in Python, but they are fundamental to other functional programming languages
  - I know…this is supposed to be a Python class…

Filter functionality

```
[1, 2, 3, 4, 5]

filter(cond, nums)

[1, 3, 4]
```

Only values that pass the predicate get selected

Filtering

```python
nums = [1, 2, 3, 4, 5]

filtered1 = list()
for x in nums:
    if x % 2 == 0:
        filtered1.append(x)
        # => [2, 4]

filtered2 = filter(lambda x: x % 2 == 0, nums)
# => <filter object at 0x7f273d420160>
list(filtered2)    # => [2, 4]
```

Reduce function, what's the point?

- Performs an operation on adjacent elements in an iterable
  - Result is a single value
- Python community has deemed reductions ugly so it's hidden away in `functools` library

Reduce functionality

```
[1, 2, 3, 4, 5]

reduce(f, nums)

f(f(f(f(1, 2), 3), 4, 5)
```

Sum of numbers

```python
from functools import reduce
nums = [1, 2, 3, 4, 5]

sum = 0
for x in nums:
    sum = sum + x
    # => 15

sum = reduce(lambda x, y: x + y, nums)    # => 15
```

Necessary to use reduce

LBE: Sum of first N even squares

```
from functools import reduce

sq = map(lambda x: x ** 2, range(N + 1))
even_sq = filter(lambda x: x % 2 == 0, sq)
sum = reduce(lambda x, y: x + y, even_sq)
# => 20 if N = 5
```

Notice we didn't convert to lists in between operations
Map and Filter objects are iterable

Overview

Recap

List comprehensions

Lambdas

Map/filter/reduce

Generators

Coroutines

Decorators

Conclusion

Brace yourselves

Subroutines vs. coroutines

- Subroutines executed their code and then eventually return
- Subroutine creates new scope that only exists while the subroutine is executing
- Same thing as functions

- Coroutines execute part of their code and suspend until it is resumed later
- Local variables in a coroutine persist between suspensions
- Think of them as "resumable functions"

Generators

- Generators are a subset of coroutines
- We've used lots of generators already!
  - Every iterable thing in Python is backed by a generator
- Lazy evaluation is also backed by a generator

Range generator

```
for i in range(3):
    print(i)    # => 0
                     1
                     2
```

Range generator

```
def range_gen(n):
    for i in range(n):
        yield i

for i in range_gen(3):
    print(i)    # => 0
                     1
                     2
```

Defined like a normal function, but `yield` keyword indicates it's a generator

`yield` keyword suspends generator with the given return value

Next time generator resumes, it picks up from exactly where the last `yield` was

Decomposing range generator

```
def range_gen(n):
    for i in range(n):
        yield i

gen = range_gen(3)
next(gen)     # => 0
next(gen)     # => 1
next(gen)     # => 2
next(gen)     # => StopIteration!
```

The in operator is just continuously calling the next operator until it excepts!

LBE: Fibonacci number generator

```python
def fib_gen():
    a, b = 0, 1
    while True:
        a, b = b, a + b
        yield a

for fib in fib_gen():
    if fib > N: break
    print(fib, end=',')  # => 1,1,2,3,5, if N = 5
```

Infinitely generate Fibonacci numbers

Lazy evaluation!

Overview

Recap

List comprehensions

Lambdas

Map/filter/reduce

Generators

Coroutines

Decorators

Conclusion

Coroutines are a superset of generators

- Generators can yield results to the caller
- Coroutines can also receive results from the caller
- Overload the `yield` keyword

Simple coroutine

```
def print_co():
    while True:
        val = yield
        print('Received: {}'.format(val))

co = print_co()
next(co)
co.send(1)    # => Received: 1
co.send(2)    # => Received: 2
```

Coroutine suspends here for data from caller

Need to get to the first yield before coroutine is useful

Simple coroutine (fixed)

```
def print_co():
    while True:
        val = yield
        print('Received: {}'.format(val))

co = print_co()
next(co)
co.send(1)    # => Received: 1
co.send(2)    # => Received: 2
co.close()
```

Explicitly stop coroutine since it's an infinite loop

I will skip this step in my slides for brevity since it will close on terminate automatically!

String filter

```python
def filter_co(pattern):
    print('Searching for {}'.format(pattern))
    while True:
        line = yield
        if pattern in line:
            print(line)

co = filter_co('help')
next(co)                           # => Searching for help
co.send('Please send help')        # => Please send help
co.send('This is easy')            # =>
co.send("You don't need help!")    # => You don't need help!
```

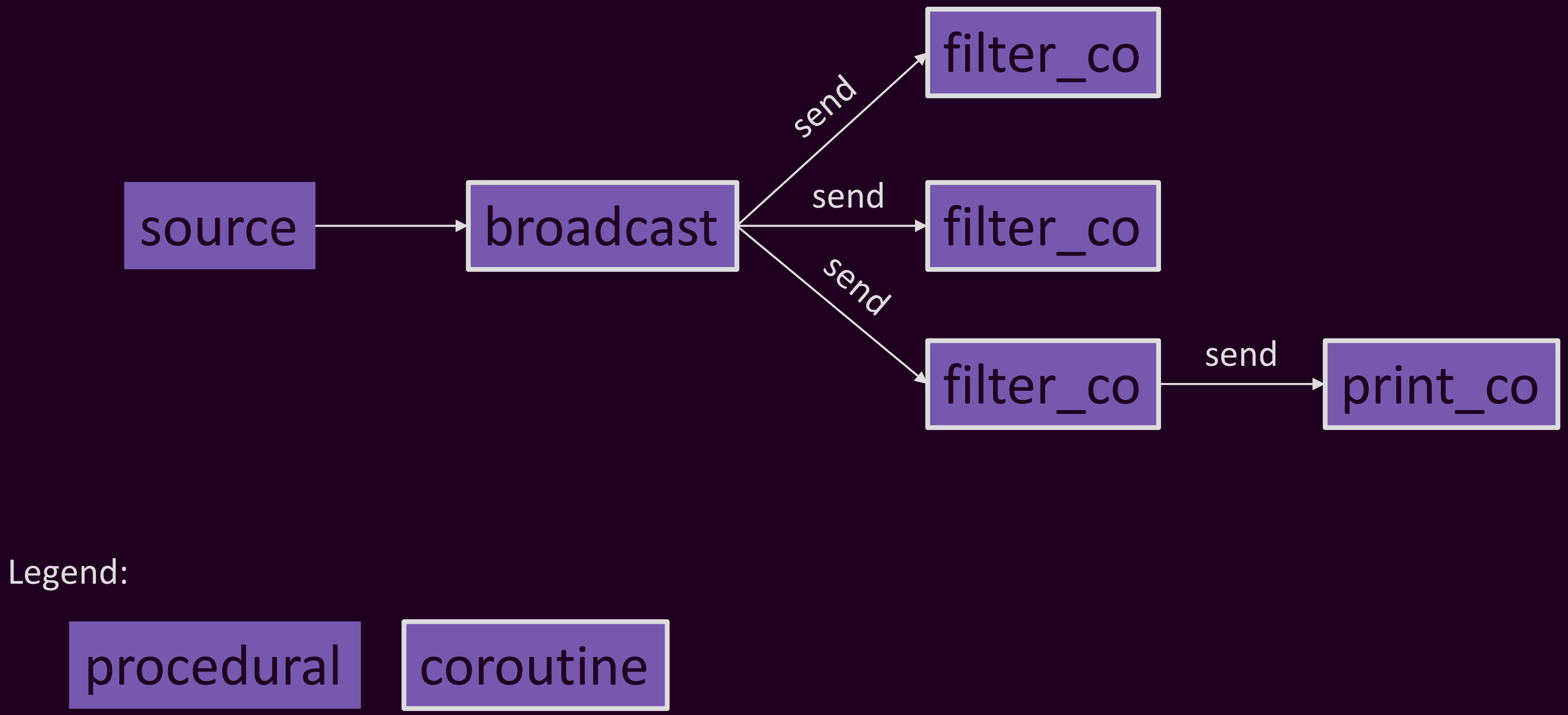
LBE: Squaring numbers like a mad man

```python
def range_gen(n):
    for i in range(n):
        yield i
def square_co():
    while True:
        num = yield
        print(num ** 2, end=',')
co = square_co()
next(co)
for i in range_gen(3):
    co.send(i)    # => 0,1,4,
```

LBE: Combining string filters

```
def filter_co(pattern, target=None):
    while True:
        line = yield
        if pattern in line:
            if target is None: print(line)
            else:              target.send(line)
f2 = filter_co('help')
f1 = filter_co('send', f2)
next(f2), next(f1)
f1.send('Please send help')       # => Please send help
f1.send('This is easy')           # =>
f1.send("You don't need help!")   # =>
```

Broadcasting



Where coroutines strive

- Producer-consumer relationships
  - Each coroutine we've seen is a consumer
- Modeling state machines
  - Each state could be an individual coroutine that changes state based on inputs
- Event-driven simulations
- Processing inputs and forwarding result to one or more targets
- Anything "reactive" in nature
- Food for thought: coroutines for modeling hardware?

Overview

Recap

List comprehensions

Lambdas

Map/filter/reduce

Generators

Coroutines

Decorators

Conclusion

Decorators are fancy wrappers

- Decorators wrap functions
- Defined as a function factory
- Invoked using the @ operator above the function you want to decorate

Basic function

```
def foo(x, y):
    print(x + y)

foo(1, 2)     # => 3
```

Debug decorator

```
def debug(func):
    def wrapper(*args, **kwargs):
        print('Arguments: ', args, kwargs, end=' -> ')
        return func(*args, *kwargs)
    return wrapper

def foo(x, y):
    print(x + y)

foo(1, 2)             # => 3
foo_debug = debug(foo)
foo_debug(1, 2)       # => Arguments: (1, 2) {} -> 3
```

Do some stuff, then call the function and return the result (forwarding in the arguments)

This looks ugly...

LBE: Debug decorator

```
def debug(func):
    def wrapper(*args, **kwargs):
        print('Arguments: ', args, kwargs, end=' -> ')
        return func(*args, *kwargs)
    return wrapper

@debug
def foo(x, y):
    print(x + y)

foo(1, 2)    # => Arguments: (1, 2) {} -> 3
```

LBE: Coroutine priming decorator

```python
def coroutine(func):
    def wrapper(*args, **kwargs):
        co = func(*args, **args)
        next(co)
        return co
    return wrapper
@coroutine
def print_co():
    while True:
        val = yield
        print('Received: {}'.format(val))
co = print_co()
co.send(1)    # => Received: 1
```

Initialize the coroutine, prime it with the next operator, then return the primed coroutine

No need to call next when using the coroutine now!

Overview

Recap

List comprehensions

Lambdas

Map/filter/reduce

Generators

Coroutines

Decorators

Conclusion

Key takeaways

- Once again, there are a lot of ways to use the language
- You don't need to master, or even fully understand, everything we've talked about to be an effective Python programmer
- Python has strong support for functional programming
- Python provides many features to remove boilerplate from programming

References

[1] http://www.dabeaz.com/coroutines/Coroutines.pdf

[2] http://stanfordpython.com/