Hands-on Intermediate Python with Drones

@__mharrison__
http://metasnake.com

About Me

- Utah Python
- Books (Treading on Python)
- Training - Python and Data Science/ML

http://hairysun.com/ http://metasnake.com/

TREADING ON PYTHON SERIES

Intermediate Python Programming

Learn Decorators, Generators, Functional Programming & more

Matt Harrison

METASNAKE

@__mharrison__

Begin Intermediate

Impetus

You can get by in Python with basic constructs ...

Impetus (2)

But you might:

- get bored
- be confused by others' code
- be less efficient

Warning

- Starting from basic Python knowledge
- Hands on
  - (short) lecture
  - (short) code
  - repeat

Python 2 or 3?

Most of this is agnostic. I'll note the differences.
Labs work with either.

Outline

- Functional Programming
- Functions
- Decorators
- Class Decorators
- Iteration
- Generators
- Context Managers

Multi-paradigmatic

Imperative Programming

Using statements to affect a program's state

Imperative Programming

```
>>> total = 0
>>> for i in range(10):
...     total += i


>>> total
45
```

Object Oriented Programming

Using objects and methods to affect a program's state

Object Oriented Programming

```
>>> class Summer:
...     def __init__(self):
...         self.sum = 0
...     def add(self, num):
...         self.sum = self.sum + num

>>> s = Summer()
>>> for num in range(10):
...     s.add(num)

>>> s.sum
45
```

Declarative Programming

```sql
SELECT *
FROM sales
WHERE store_id = 5;
```

Functional Programming

Change state by applying functions, avoiding state, side effects and mutable data

Functional Programming

```
>>> import operator
>>> reduce(operator.add, range(10))
45
```

Functional Programming

```
>>> sum(range(10))
45
```

Functional Programming

(Python 1.4)

Functional Programming

Change state by applying functions, avoiding state, side effects and mutable data:

```
>>> sum(range(10))
45
```

Imperative Programming

Using statements to affect a program's state:

```
>>> total = 0
>>> for i in range(10):
...     total += i

>>> total
45
```

First-class functions

Functions are treated as data. They can be passed around, not just invoked.

Higher-order functions

Function that accept functions as parameters or returns a function.

Pure functions

- Always produces the same result (ie not accessing global state)
- No side effects (writing to disk, mutating global state, etc)

Pure functions (2)

Pure: `math.cos`

Impure: `print`, `random.random`

lambda

Create simple functions in a single line:

```
>>> def mul(a, b):
...     return a * b

>>> mul_2 = lambda a, b: a*b

>>> mul_2(4, 5) == mul(4,5)
True
```

`lambda` examples

Useful for `key` and `cmp` when sorting

lambda key example

```
>>> data = '3,1,100,99,0'.split(',')
>>> sorted(data)
['0', '1', '100', '3', '99']

>>> sorted(data, key=lambda x: int(x))
['0', '1', '3', '99', '100']
```

Hint: Use `key` not `cmp`

`lambda` *expressions*

Statements cause problems:

```
>>> is_pos = lambda x: if x >=0: 'pos'
Traceback (most recent call last):
  ...
    is_pos = lambda x: if x >=0: 'pos'
                       ^
SyntaxError: invalid syntax
```

`lambda` *expressions* (2)

(Conditional) expressions don't:

```
>>> is_pos = lambda x: 'pos' if x >= 0 else 'neg'
>>> is_pos(3)
'pos'
```

See PEP 308

`lambda` *expressions* (3)

Simple rule for *expressions*: Something that could be returned from a function:

```
def func(args):
    return expression
```

Std lib example

from `cookielib.py`

```
# add cookies in order of most specific
# (ie. longest) path first
cookies.sort(key=lambda arg: len(arg.path),
    reverse=True)
```

`lambda` *expressions* (5)

Good for one-liners

map

Higher-order function that applies a function to items of a sequence:

```
>>> map(str, [0, 1, 2])
['0', '1', '2']
```

map (2)

With a `lambda`:

```
>>> pos = lambda x: x >= 0
>>> map(pos, [-1, 0, 1, 2])
[False, True, True, True]
```

Std lib example

from `tarfile.py`:

```
def namelist(self):
  return map(lambda m: m.name,
self.infolist())
```

`map` (3)

In Python 3, `map` is not a function but a lazy class:

```
>>> map(str, range(10))
<map object at 0x7fa285727b90>
>>> next(_)
'0'
```

`map` (4)

Use `itertools.imap` in Python 2 to apply to an infinite sequence (generator)

reduce

Apply a function to pairs of the sequence:

```
>>> import operator
>>> reduce(operator.mul, [1,2,3,4])
24 # ((1 * 2) * 3) * 4
```

`reduce` (2)

Reduce moved to `functools` module in Python 3.
Unlike `map`, still a function and not lazy.

Std lib example

from `csv.py`. Guessing the quote character:

```
quotechar = reduce(lambda a, b, quotes=quotes:
                   (quotes[a] > quotes[b]) and
                    a or b, quotes.keys())
```

Lambda equivalent:

```
if quotes[a] > quotes[b]:
    return a
return b
```

Std lib example

from Python 3 `csv.py`. Guessing the quote character:

```
quotechar = max(quotes, key=quotes.get)
```

filter

Takes a function and a sequence. Return a sequence items for which `function(item)` is `True`:

```
>>> filter(lambda x:x >= 0, [0, -1, 3, 4, -2])
[0, 3, 4]
```

`filter` (2)

Lazy in Python 3. Use `itertools.ifilter` in Python 2 for infinite sequences.

Std lib example

from `tarfile.py`:

```
def infolist(self):
    return filter(
        lambda m: m.type in REGULAR_TYPES,
        self.tarfile.getmembers())
```

Notes about “functional” programming in *Python*

- `sum` or `for` loop can replace `reduce`
- List comprehensions replace `map` and `filter`
- No tail call optimization (means limit on recursion depth)

Example Assignment

Run hello world

Drone Hello World

```
from turtledrone import TRDrone
drone = TRDrone()
drone.takeoff()
time.sleep(1)
drone.land()
drone.halt()
```

Assignment Notes

- Use spaces not tabs (PEP 8)
- define functions as globals

Assignment

Use `map` to spin the drone around

More about functions

a function is an instance of a `function`

```
>>> def foo():
...     'docstring for foo'
...     print 'invoked foo'

>>> foo #doctest: +ELLIPSIS
<function foo at ...>
```

a function is `callable`

```
>>> callable(foo)
True
```

function invocation

Just add `()`:

```
>>> foo()
invoked foo
```

a function has attributes

```
>>> foo.__name__
'foo'

>>> foo.__doc__
'docstring for foo'
```

(PEP 234 Python 2.1)

function scope

A function knows about itself:

```
>>> def foo2():
...     print "NAME", foo2.__name__

>>> foo2()
NAME foo2
```

function attributes

Can attach data to function prior to invocation:

```
>>> def foo3():
...     print foo3.stuff

>>> foo3.stuff = "Data"

>>> foo3()
Data
```

function definition

```
def func_name(arg1, arg2=value,
              *args, **kwargs):
    """docstring"""
    # implementation
```

Parameter types

- No parameters
- standard parameters (many)
- default/keyword/named parameters (many)
- variable parameters (one), preceded by *
- variable keyword parameters (one), preceded by **

Standard/named parameters

```
>>> def param_func(a, b=2, c=5):
...     print [a, b, c]

>>> param_func(2)
[2, 2, 5]

>>> param_func(3, 4, 5)          # override defaults
[3, 4, 5]

>>> param_func(c=4, b=5, a=6)    # order changed if name
[6, 5, 4]                        # provided
```

`*args` and `**kwargs`

`*args` (variable parameters) is a *tuple* of parameters values.

`**kwargs` (keyword parameters) is a *dictionary* of name/value pairs.

Only one of each type. Naming above is standard convention

*args

```
>>> def demo_args(*args):
...     print type(args), args

>>> demo_args()
<type 'tuple'> ()

>>> demo_args(1)          # Note type
<type 'tuple'> (1,)

>>> demo_args(3, 'foo')
<type 'tuple'> (3, 'foo')
```

`*args` (2)

The `*` before a sequency *parameter* in an invocation “unpacks” (or splats) the sequence

*args (3)

```
>>> args = [1, 2, 3]

>>> demo_args(args[0], args[1], args[2])
<type 'tuple'> (1, 2, 3)

>>> demo_args(*args)          # same as above
<type 'tuple'> (1, 2, 3)

>>> demo_args(args)           # only 1 arg passed in
<type 'tuple'> ([1, 2, 3],)   # List(!) in a tuple
```

*args (4)

```
>>> def add3(a, b, c):     # No *args! yet...
...     return a + b + c

>>> add3(4, 5, 6)
15

>>> add3(*[4, 5, 6])   # unpack list
15
```

**kwargs

```
>>> def demo_kwargs(**kwargs):
...     print type(kwargs), kwargs

>>> demo_kwargs()
<type 'dict'> {}

>>> demo_kwargs(one=1)
<type 'dict'> {'one': 1}

>>> demo_kwargs(one=1, two=2)
<type 'dict'> {'two': 2, 'one': 1}
```

`**kwargs` (2)

The ** before a dict *parameter* in an invocation "unpacks" (or splats) the dict

**kwargs (3)

```
>>> from math import pow
>>> def distance(x1, y1, x2, y2):
...     return pow(pow(x1-x2, 2) +
...                pow(y1-y2, 2), .5)
```

**kwargs (3)

```
>>> points = {'x1':1, 'y1':1,
...           'x2':4, 'y2':5}

>>> # these calls are the same
>>> distance(**points)
5.0

>>> distance(x1=1, y1=1, x2=4, y2=5)
5.0
```

*args and **kwargs

```
>>> def demo_params(normal, kw="Test", *args, **kwargs):
...     print normal, kw, args, kwargs
>>> args = (0, 1, 2)
>>> kw = {'foo': 3, 'bar': 4}

>>> demo_params(*args, **kw)   # args can migrate
0 1 (2,) {'foo': 3, 'bar': 4}

>>> demo_params(args[0], args[1], args[2],
...                foo=3, bar=4)
0 1 (2,) {'foo': 3, 'bar': 4}
```

`*args` and `**kwargs` (2)

Python website [1] has gory details

[1] See http://docs.python.org/reference/expressions.html#calls

Closures

(PEP 227 Python 2.1)

Closure Definitions

First-class function with free variables that are bound by the lexical environment

Wikipedia

In Python functions can return new functions. The inner function is a *closure* and any variable it accesses that are defined outside of that function are *free variables*. (Inner function "closes-over" the variables)

me

Closures (2)

Useful as function generators:

```
>>> def add_x(x):
...     def adder(num):
...         # we have read access to x
...         return num + x
...     return adder

>>> add_5 = add_x(5)
>>> add_5 #doctest: +ELLIPSIS
<function adder at ...>
>>> add_5(10)
15

>>> add_bang = add_x("!")
>>> add_bang("hello world")
'hello world!'
```

Closures (3)

Notice the function attributes:

```
>>> add_5.__name__
'adder'
```

Assignment

Write a function that takes a drone and a number. Return a new function that will move forward that number of times when run.

Decorators

(PEP 318, 3129, Python 2.4)

Decorators

Functions are first class objects! You can wrap them to alter behavior

Decorators (2)

Allow you to

- modify arguments
- modify function
- modify results

Uses for decorators

- caching
- monkey patching stdio
- jsonify
- logging time in function call

Decorator Definition

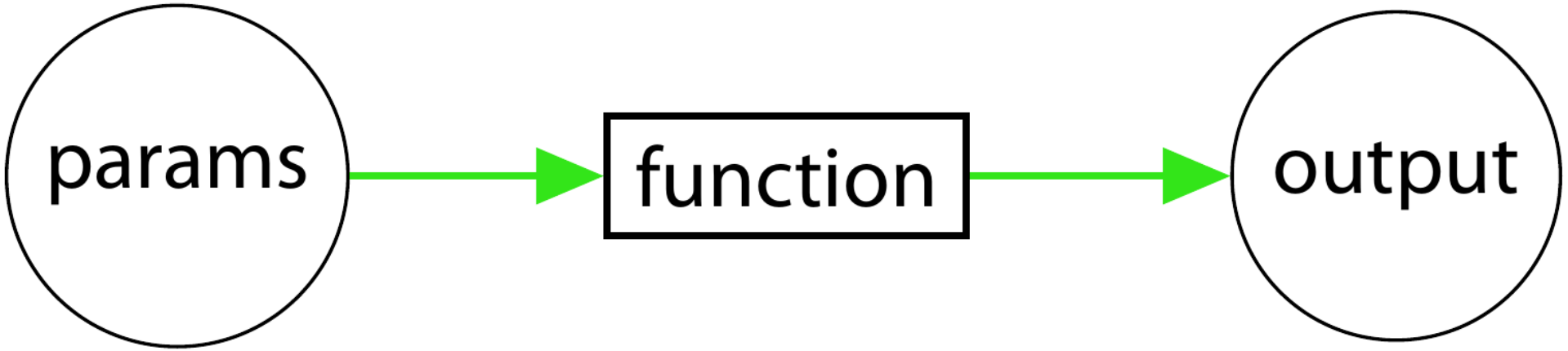
[A]llows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.

Wikipedia

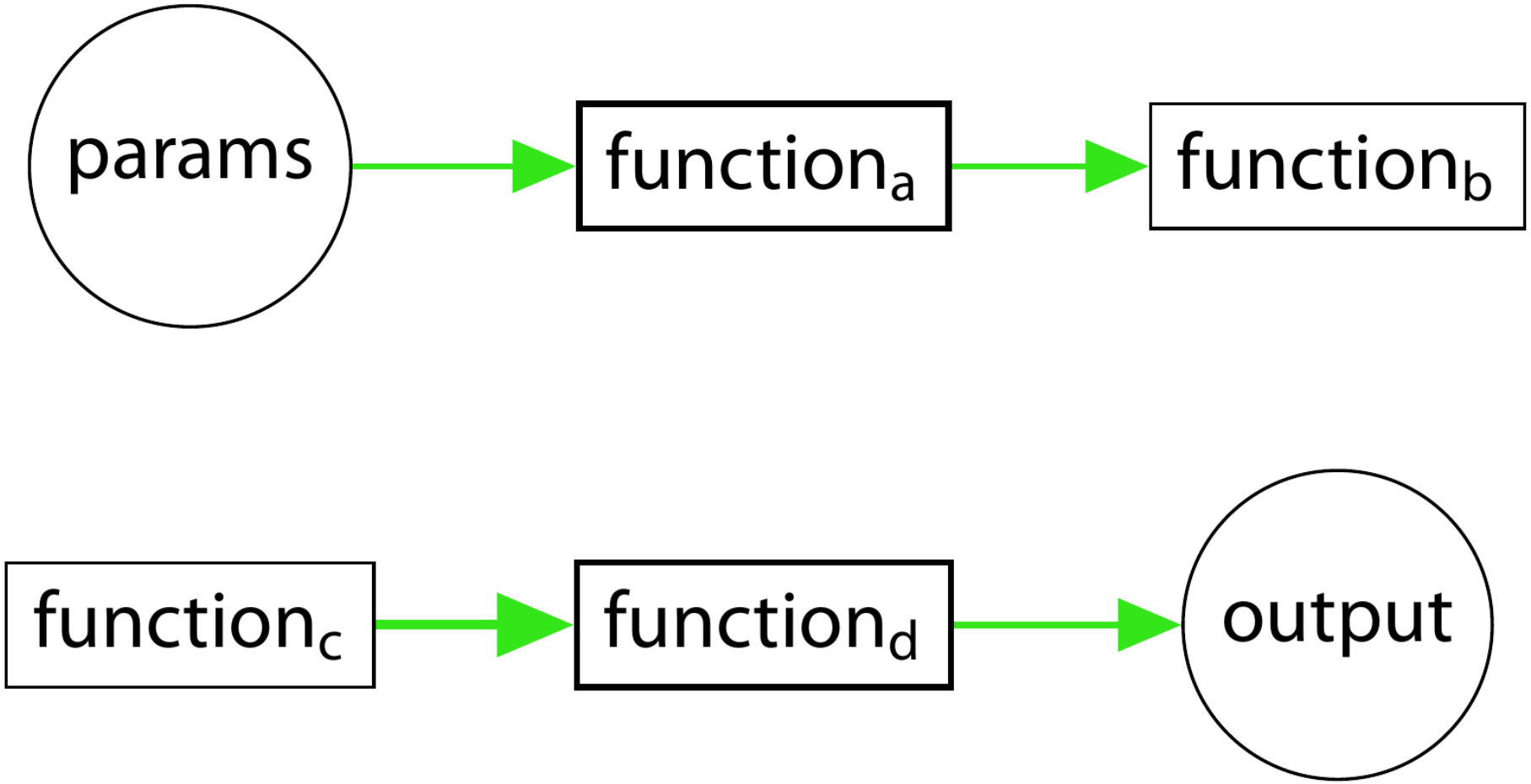
A *callable* that accepts a *callable* and returns a *callable*

me

Function Takes parameters as input returns output

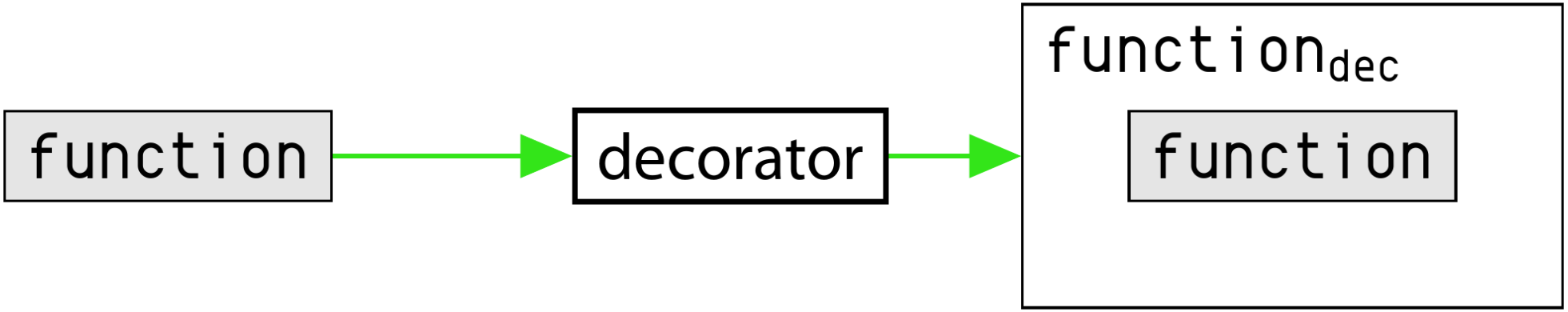


Higher-Order Function **Accepts or returns functions**

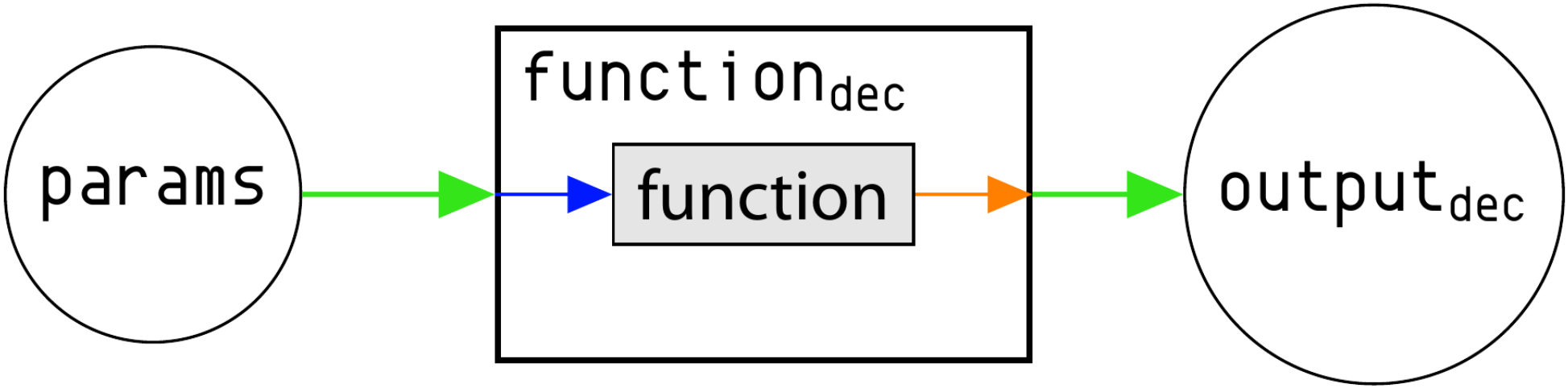


Decorator Takes function as input, returns a function

Decorator Takes function as input, returns a function



Decorated Function can do something before or after



Identity Decorator

```
>>> def iden(func):
...     return func

>>> def add(x, y):
...     return x + y

>>> add = iden(add)

>>> add(3, 4)
7
```

Identity Decorator (2)

```
>>> def iden(func):
...     def wrapper(*args, **kwargs):
...         # before
...         res = func(*args, **kwargs)
...         # after
...         return res
...     return wrapper

>>> def add(x, y):
...     return x + y

>>> add = iden(add)

>>> add(3, 4)
7
```

Decorators

Count how many times a function is called.
Create a decorator—`count`:

```
>>> call_count = 0
>>> def count(func):
...     def wrapper(*args, **kwargs):
...         global call_count
...         call_count += 1
...         return func(*args, **kwargs)
...     return wrapper
```

Decorators (4)

Create a function:

```
>>> def hello():
...     print 'invoked hello'
```

"Decorate" the `hello` function:

```
>>> hello = count(hello)
```

Decorators (5)

Test it:

```
>>> hello()
invoked hello
>>> call_count
1

>>> hello()
invoked hello
>>> call_count
2
```

Syntactic Sugar

```
>>> @count
... def hello():
...     print 'hello'
```

equivalent to:

```
>>> hello = count(hello)
```

Syntactic Sugar(2)

Don't add parens to decorator:

```
>>> @count()   # notice parens
... def hello():
...     print 'hello'
Traceback (most recent call last):
  ...
TypeError: count() takes exactly 1 argument (0
given)
```

Same as `hello = count()(hello)`

There was a problem with `count`

Better decorator

Attach data to `wrapper`:

```
>>> def count2(func):
...     def wrapper(*args, **kwargs):
...         wrapper.call_count += 1
...         return func(*args, **kwargs)
...     wrapper.call_count = 0
...     return wrapper
```

Better decorator(2)

```
>>> @count2
... def bar():
...     "my docstring"
...     pass

>>> bar(); bar()
>>> print bar.call_count
2

>>> @count2
... def snoz():
...     pass

>>> snoz()
>>> print snoz.call_count
1
```

Another problem

```
>>> bar.__name__
'wrapper'
>>> bar.__doc__
```

Better decorator (2)

Update `__name__` and `__doc__` (or use `@functools.wraps`):

```
>>> def count3(func):
...     def wrapper(*args, **kwargs):
...         wrapper.call_count += 1
...         return func(*args, **kwargs)
...     wrapper.call_count = 0
...     wrapper.__name__ = func.__name__
...     wrapper.__doc__ = func.__doc__
...     return wrapper
```

Decorator Template

```
>>> import functools
>>> def decorator(func_to_decorate):
...     @functools.wraps(func_to_decorate)
...     def wrapper(*args, **kwargs):
...         # do something before invocation
...         result = func_to_decorate(*args,
**kwargs)
...         # do something after
...         return result
...     return wrapper
```

Std lib Example

From `contextlib.py`:

```
def contextmanager(func):
    @wraps(func)
    def helper(*args, **kwds):
        return GeneratorContextManager(
            func(*args, **kwds))
    return helper
```

Django Example

From `django/views/decorators/http.py`:

```
def require_http_methods(request_method_list):
    """
Decorator to make a view only accept particular request methods. Usage::

@require_http_methods(["GET", "POST"])
def my_view(request):
# I can assume now that only GET or POST requests make it this far
# ...

Note that request methods should be in uppercase.
"""
    def decorator(func):
        @wraps(func, assigned=available_attrs(func))
        def inner(request, *args, **kwargs):
            if request.method not in request_method_list:
                logger.warning('Method Not Allowed (%s): %s',
                    request.method, request.path,
                    extra={
                        'status_code': 405,
                        'request': request
                    }
                )
                return HttpResponseNotAllowed(request_method_list)
            return func(request, *args, **kwargs)
        return inner
    return decorator
```

Decorator rehash

Allows you to:

- Before function invocation
  - modify arguments
  - modify function
- After function invocation
  - modify results

Assignment

Create a decorator `use_battery` that takes a method and calls `draw_battery` (on `args[0]`). Wrap the `takeoff` method

Class Decorators

(PEP 3129, Python 2.6)

Class Decorators

A callable that takes a class and returns a class

Class Decorators (2)

```
>>> def shoutclass(cls):
...     def shout(self):
...         print self.__class__.__name__.upper()
...     cls.shout = shout
...     return cls

>>> @shoutclass
... class Loud: pass

>>> loud = Loud()
>>> loud.shout()
LOUD
```

Class Decorators (3)

Occurs during class definition time (not instance creation):

```
>>> def time_cls_dec(cls):
...     print "BEFORE"
...     def new_method(self):
...         print "NEW METHOD"
...     cls.new_method = new_method
...     return cls

>>> @time_cls_dec   # definition time
... class Timing(object): pass
BEFORE

>>> t = Timing()    # instance creation time
```

Class Decorators (4)

Works with subclasses:

```
>>> class SubTiming(Timing): pass

>>> s = SubTiming()
>>> s.new_method()
NEW METHOD
```

Std lib example

`functools.total_ordering` in Python3.2 adds `__le__`, `__gt__`, and `__ge__` if `__lt__` and `__eq__` are defined.

Assignment

Create a class decorator `battery_class` that takes a class and wraps every method that starts with 'move' with `use_battery`

List comprehensions

(PEP 202, Python 2.0)

Looping

Common to loop over and accumulate:

```
>>> seq = range(-10, 10)
>>> results = []
>>> for x in seq:
...     if x >= 0:
...         results.append(2*x)
```

List comprehensions

```
>>> results = [ 2*x for x in seq if x >= 0 ]
```

Shorthand for accumulation:

```
>>> results = []
>>> for x in seq:
...     if x >= 0:
...         results.append(2*x)
```

Construction

- Assign result:

```
results = []
```

- Insert for loop:

```
results = [for x in seq]
```

- Add filter (if any):

```
results = [for x in seq if x >= 0]
```

- Put accumulated object in front:

```
results = [2**x for x in seq if x >= 0]
```

List comprehensions (2)

`if` statement optional:

```
>>> results = [ 2*x for x in xrange(9)]
>>> results
[0, 2, 4, 6, 8, 10, 12, 14, 16]
```

List comprehensions (3)

Can be nested:

```
>>> nested = [ (x, y) for x in xrange(3) \
...             for y in xrange(4) ]
>>> nested
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2,
0), (2, 1), (2, 2), (2, 3)]
```

Same as:

```
>>> nested = []
>>> for x in xrange(3):
...     for y in xrange(4):
...         nested.append((x,y))
```

List comprehensions (4)

Acting like `map` (apply `str` to a sequence):

```
>>> [str(x) for x in range(5)]
['0', '1', '2', '3', '4']
```

List comprehensions (5)

Acting like `filter` (get positive numbers):

```
>>> [x for x in range(-5, 5) if x >= 0]
[0, 1, 2, 3, 4]
```

Std lib example

From `csv.py`:

```
ascii = [chr(c) for c in range(127)]  # 7-bit ASCII
```

Assignment

Use a list comp. to spin the drone around

Iterators

(PEP 234)

Iterators

Sequences in *Python* follow the iterator pattern (PEP 234):

```
>>> sequence = [ 'foo', 'bar', 'baz']
>>> for x in sequence:
...     # body of loop
```

equivalent to:

```
>>> iterator = iter(sequence)
>>> while True:
...     try:
...         x = iterator.next()  # py3 .__next__()
...     except StopIteration, e:
...         break
...     # body of loop
```

Iterators (2)

```
>>> sequence = [ 'foo', 'bar']
>>> seq_iter = iter(sequence)
>>> seq_iter.next()
'foo'
>>> seq_iter.next()
'bar'
>>> seq_iter.next()
Traceback (most recent call last):
  ...
StopIteration
```

Making objects iterable

```
>>> class Foo(object):
...     def __iter__(self):
...         return self
...     def next(self):    # py3 __next__
...         # logic
...         return next_item
```

Object example

```
>>> class RangeObject(object):
...     def __init__(self, end):
...         self.end = end
...         self.start = 0
...     def __iter__(self): return self
...     def next(self):
...         if self.start < self.end:
...             value = self.start
...             self.start += 1
...             return value
...         raise StopIteration

>>> [x for x in RangeObject(4)]
[0, 1, 2, 3]
```

Std lib example

From `csv.py`:

```
class DictReader:

    def __iter__(self):
        return self

    def next(self):
        if self.line_num == 0:
            # Used only for its side effect.
            self.fieldnames  # property: calls .next()
        row = self.reader.next()
        self.line_num = self.reader.line_num
```

Std lib example (2)

```
# unlike the basic reader, we prefer not to return blanks,
# because we will typically wind up with a dict full of None
# values
while row == []:
    row = self.reader.next()
d = dict(zip(self.fieldnames, row))
lf = len(self.fieldnames)
lr = len(row)
if lf < lr:
    d[self.restkey] = row[lf:]
elif lf > lr:
    for key in self.fieldnames[lr:]:
        d[key] = self.restval
return d
```

Generators

(PEP 255, 342, Python 2.3)

generators

Functions with the `yield` keyword remember their state and return to it when iterating over them

generators (2)

Can be used to easily “generate” sequences

generators (3)

Can be useful for lowering memory usage (ie `range(1000000)` vs `xrange(1000000)`)

Note `xrange` is *not* a generator

generators (4)

```
>>> def gen_range(end):
...     cur = 0
...     while cur < end:
...         yield cur
...         # returns here next
...         cur += 1
```

generators (5)

Generators return a generator instance. Iterate over them for values:

```
>>> gen = gen_range(4)
>>> gen #doctest: +ELLIPSIS
<generator object gen_range at ...>
```

generators (6)

Follow the iteration protocol. A generator is iterable!

```
>>> nums = gen_range(2)
>>> nums.next()
0
>>> nums.next()
1
>>> nums.next()
Traceback (most recent call last):
  ...
StopIteration
```

Generators (7)

Generator in for loop or list comprehension:

```
>>> for num in gen_range(2):
...     print num
0
1

>>> print [x for x in gen_range(2)]
[0, 1]
```

Generators (8)

Re-using generators may be confusing:

```
>>> gen = gen_range(2)
>>> [x for x in gen]
[0, 1]

>>> # gen in now exhausted!
>>> [x for x in gen]
[]
```

generators (9)

Can be chained:

```
>>> def positive(seq):
...     for x in seq:
...         if x >= 0:
...             yield x
>>> def every_other(seq):
...     for i, x in enumerate(seq):
...         if i % 2 == 0:
...             yield x
>>> nums = xrange(-5, 5)
>>> pos = positive(nums)
>>> skip = every_other(pos)
>>> [x for x in skip]
[0, 2, 4]
```

generators (10)

Generators can be tricky to debug. Can't step into them when invoked, only when *iterated over*.

Objects as generators

```
>>> class Generate(object):
...     def __iter__(self):
...         # just use a
...         # generator here
...         yield result
```

list or generator?

List:

- Need to use data repeatedly
- Enough memory to hold data
- Negative slicing

Generator Hints

- Make it “peekable”
- Generators always return `True`, [] (empty list) is `False`
- Might be useful to cache results
- If recursive, make sure to iterate over results

Generator Hints (2)

- Rather than making a complicated generator, consider making simple ones that chain together (Unix philosophy)
- Sometimes one at a time is slow (db) - wrap with “fetchmany” generator
- `itertools` is helpful (`islice`)

`xrange`

`xrange` doesn't really behave as an generator.

- you can index it directly (but not slice)
- it has no `.next()` method
- it doesn't exhaust

Std lib example

From `collections.py`

```
class OrderedDict(dict):
    ...

    def iteritems(self):
        'od.iteritems -> an iterator over the (key, value)
pairs in od'
        for k in self:
            yield (k, self[k])
```

Assignment

Make two infinite generators, `forward` and `right`, that take a drone and when iterated over move it forward or right. Run with:

```
f = forward(drone)
r = right(drone)
for i in range(72):
    next(f)
    next(r)
```

Generator Expressions

(PEP 289 Python 2.4)

Generator expressions

Like list comprehensions. Except results are generated on the fly. Use (and) instead of [and] (or omit if expecting a sequence)

Generator expressions (2)

```
>>> [x*x for x in xrange(5)]
[0, 1, 4, 9, 16]

>>> (x*x for x in xrange(5)) # doctest: +ELLIPSIS,
<generator object <genexpr> at ...>
>>> list(x*x for x in xrange(5))
[0, 1, 4, 9, 16]
```

Generator expressions (3)

```
>>> nums = xrange(-5, 5)
>>> pos = (x for x in nums if x >= 0)
>>> skip = (x for i, x in enumerate(pos) if i % 2 == 0)
>>> list(skip)   # materialize
[0, 2, 4]
```

Generator expressions (4)

If Generators are confusing, but List Comprehensions make sense, you can simulate some of the behavior of generators as follows....

Generator expressions (5)

```
>>> def pos_generator(seq):
...     for x in seq:
...         if x >= 0:
...             yield x

>>> def pos_gen_exp(seq):
...     return (x for x in seq if x >= 0)

>>> list(pos_generator(range(-5, 5))) == \
...   list(pos_gen_exp(range(-5, 5)))
True
```

Std lib example

from `string.py`

```
def capwords(s, sep=None):
    """capwords(s [,sep]) -> string

    Split the argument into words using split, capitalize
    each word using capitalize, and join the capitalized
    words using join. If the optional second argument sep
    is absent or None, runs of whitespace characters are
    replaced by a single space and leading and trailing
    whitespace are removed, otherwise sep is used to split
    and join the words.

    """
    return (sep or ' ').join(x.capitalize() for x in s.split(sep))
```

Context Managers

(PEP 343 Python 2.5)

Context Mgr

Shortcut for “try/finally” statements

Context Mgr (2)

Makes it easy to write

```
# setup
try:
    variable = value
    # body
finally:
    # cleanup
```

as

```
with some_generator() as variable:
    # body
```

Try/Finally Diversion

What does this do?

```
>>> def foo():
...     try:
...         return 1
...     finally:
...         return 2
```

Try/Finally Diversion

What does this do?

```
>>> foo()
2
```

Try/Finally Diversion

What does this do?

```
>>> def foo2():
...     try:
...         1/0
...     finally:
...         return 2
```

Try/Finally Diversion

What does this do?

```
>>> foo2()
2
```

Try/Finally Diversion

What does this do?

```
>>> def foo3():
...     try:
...         raise KeyError("Bad!")
...     finally:
...         return 2
```

Try/Finally Diversion

What does this do?

```
>>> foo3()
2
```

Try/Finally Diversion

Takeaway - `finally` always runs (unless infinite loop before)

Context Mgr (3)

Seen in files:

```
fin = open('/tmp/foo')
# do something with  fin
fin.close()
```

Context Mgr (4)

Seen in files:

```
with open('/tmp/foo') as fin:
    # do something with fin
# fin is automatically closed here
```

Context Mgr (5)

Two ways to create:

- class
- decorated generator

Context Mgr (5)

Context managers can optionally return an item with `as`

Lock example (PEP 343)

```
lock.acquire()
# run some code while locked
lock.release()
```

Lock example (PEP 343)

Buggy, should be:

```
lock.acquire()
try:
    # run some code while locked
finally:
    lock.release()
```

Lock example (PEP 343)

```
with locked(myLock):
    # Code here executes with
    # myLock held. The lock is
    # guaranteed to be released
    # when the block is left
    # (even if via return or
    # by an uncaught exception).
```

Lock example (PEP 343) (2)

Class style:

```
class locked:
    def __init__(self, lock):
        self.lock = lock
    def __enter__(self):
        self.lock.acquire()
    def __exit__(self, type, value, tb):
        # if error in block, t, v, & tb
        # have non None values
        # return True to hide exception
        self.lock.release()
```

Lock example (PEP 343) (3)

Generator style:

```
from contextlib import contextmanager

@contextmanager
def locked(lock):
    lock.acquire()
    try:
        yield
    finally:
        lock.release()
```

Context Manager with `as`

Seen in files:

```
with open('/tmp/foo') as fin:
    # do something with fin
# fin is automatically closed here
```

Context Manager with `as` (2)

Class style:

```
class a_cm:
    def __init__(self):
        # init
    def __enter__(self):
        # enter logic
        return self
    def __exit__(self, type, value, tb):
        # exit logic
```

Context Manager with `as` (3)

Generator style yield object:

```
from contextlib import contextmanager

@contextmanager
def a_cm():
    # enter logic
    try:
        yield object
    finally:
        # exit logic
```

Error Handling

- In generators can use bare `raise` from `finally`
- In class, return `True` to swallow error. Can inspect error if needed. Arguments to `__exit__` correspond to results of `sys.exc_info()` (class, instance, traceback)

Error Handling

In class:

```
def __exit__(self, type, value, tb):
    # type is class of exception
    # value is instance
    # tb is traceback
    # return True to swallow exception
```

Rollback Code

From PEP

```
@contextmanager
def transaction(db):
    db.begin()
    try:
        yield None
    except:
        db.rollback()
        raise
    else:
        db.commit()
    # don't have to have finally
```

Rollback Code

Not in PEP

```
class transaction(object):
    def __init__(self, db):
        self.db = db

    def __enter__(self):
        self.db.begin()

    def __exit__(self, type, value, tb):
        if type:
            self.db.rollback()
            return False
        self.db.commit()
        return True
```

Uses for Context Managers

- Managing external resources (socket, file, connection)
- Transactions
- Acquiring locks
- closing/cleaning up
- nesting for generating html/xml

Std lib example

from `tempfile.py`

```
class SpooledTemporaryFile:
    """Temporary file wrapper, specialized to switch from
    StringIO to a real file when it exceeds a certain size or
    when a fileno is needed.
    """

    # Context management protocol
    def __enter__(self):
        if self._file.closed:
            raise ValueError("Cannot enter context with closed file")
        return self

    def __exit__(self, exc, value, tb):
        self._file.close()
```

Assignment

Write a context manager that launches the drone and always lands it

That's all

matthewharrison@gmail.com
@__mharrison__
http://hairysun.com