Hands-on Intermediate Python

@__mharrison__
http://hairysun.com

About Me

- 12 years Python
- Worked in HA, Search, Open Source, BI and Storage
- Author of multiple Python Books

Get code

`inter_python.tar.gz`

- Thumbdrive has it

Unzip it somewhere (`tar -zxvf inter_python.tar.gz`)

Begin

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 until time is gone

Python 2 or 3?

Most of this is agnostic. I'll note the differences, but use 2.x throughout

Outline

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

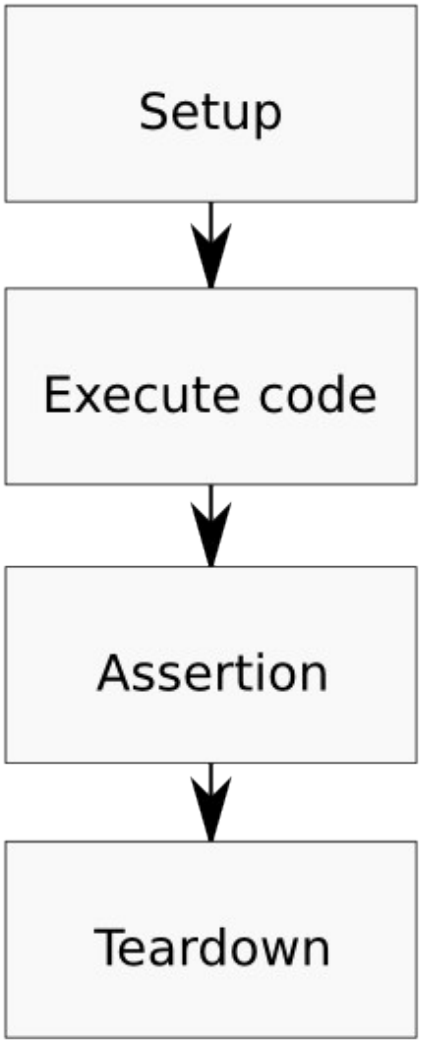
unittest

(Python 2.1)

`unittest`

Implements Kent Beck's *xUnit* paradigm

xUnit workflow



```
import unittest
import integr
class TestIntegr(unittest.TestCase):
    def setup(self):
        # setup is called *before* each test is run
        # if I need to adjust to a well known state before starting
        # I can do that here
        pass
    def teardown(self):
        # teardown is called *after* the each test is run
        pass
    def test_basic(self):
        # any method beginning with "test" is a test
        results = integr.parse('1,3,4')
        self.assertEquals(results, [1,3,4])
if __name__ == '__main__':
    unittest.main()
```

```
class TestIntegr(unittest.TestCase):
    ...
if __name__ == '__main__':
    unittest.main()
```

```
def setup(self):
    # setup is called *before*
    # each test is run.
    # Adjust to a well known
    # state before each test.
    pass
```

```
def test_basic(self):
    # any method beginning with
    # "test" is a test
    results = integr.parse('1,3,4')
    self.assertEquals(results, [1,3,4])
```

```
def teardown(self):
    # teardown is called *after*
    # the each test is run
    pass
```

Assertion Methods

Method signature	Explanation
`assert_(expression, [message])`	Complains if expression is False
`assertEqual(this, that, [message])`	Complains if this != that
`assertNotEqual(this, that, [message])`	Complains if this == that
`assertRaises(exception, callable, *args, **kwargs)`	Complains if `callable(*args, **kwargs)` does not raise `exception`

Critique of `unittest`

- **Cons**
  - Modeled after java, why classes? (inheritance/abstraction bad)
  - "heavyweight", Too much baggage, "frameworky"
  - Focus on testing, makes hard to update - why is a test failing?
- **Pros**
  - In the standard library
  - Straightforward

Other options

- Nose
- py.test
- Twisted Trial

None of these are included in the standard library.

Is there another way?

Other options???

doctest

(Python 2.1)

doctest

A post from Tim Peters in 1999...

doctest

- Examples are priceless.
- Examples that don't work are worse than worthless.
- Examples that work eventually turn into examples that don't.
- Docstrings too often don't get written.
- Docstrings that do get written rarely contain those priceless examples.

`doctest` (2)

- The rare written docstrings that do contain priceless examples eventually turn into rare docstrings with examples that don't work. I think this one may follow from the above ...
- Module unit tests too often don't get written.

`doctest` (3)

- The best *Python* testing gets done in interactive mode, esp. trying endcases that almost never make it into a test suite because they're so tedious to code up.
- The endcases that were tested interactively (but never coded up) also fail to work after time.

OK, so what is `doctest`?

Turns *Python* docstrings (module, class, function, method) that contain interactive session snippets into tests. (Session must have output!)

An example

```
def add_10(x):
    """
    adds 10 to the input value
    >>> add_10(5)
    15
    >>> add_10(-2)
    8
    """
    return x + 10

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Broken documentation

```
def add_10(x):
    """
    adds 10 to the input value
    >>> add_10(5)
    15
    >>> add_10(-2)
    6
    """
    return x + 10

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Broken output

```
**********************************************************
File "add10.py", line 7, in __main__.add_10
Failed example:
    add_10(-2)
Expected:
    6
Got:
    8
**********************************************************
1 items had failures:
   1 of   2 in __main__.add_10
***Test Failed*** 1 failures.
```

doctest details

```
>>> # comments are ignored
>>> foo = "bar"
```

doctest details (2)

```
>>> foo  #implicit print
'bar'

>>> print foo # explicit print
bar
```

doctest details (3)

Caveat: can't print whitespace directly. Use <BLANKLINE> instead

```
>>> print "" # this fails!

>>> print """
<BLANKLINE>
```

doctest details (4)

Backslashes. Use raw docstrings (this is raw) or escape them (\\n)

```
>>> line = "foo\n"  # need to escape (\\n) if not raw
>>> print line
foo
<BLANKLINE>
```

doctest details (5)

Starting column is irrelevant

```
>>> 2 + 2
4
This is more indented but it
doesn't matter
    >>> 2 + 2
    4
```

doctest details (6)

Expected output for an exception must start with a traceback header.

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call
last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not
in list
```

`doctest` hint

Try to remove magic numbers (such as line numbers)

doctest details (7)

Traceback stack can also be replaced by ellipsis "..."

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  ...
ValueError: list.remove(x): x not in list
```

doctest details (8)

Various options and directives. Note the "#doctest:"

```
>>> print range(20) #doctest:
+NORMALIZE_WHITESPACE
[0,   1,  2,  3,  4,  5,  6,  7,
8,  9,  10,  11, 12, 13, 14, 15,
16, 17, 18, 19]
```

doctest details (9)

Combine directives

```
>>> print range(20) # doctest:
+ELLIPSIS, +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

doctest details (10)

Other directives:
DONT_ACCEPT_TRUE_FOR_1,
DONT_ACCEPT_BLANKLINE,
IGNORE_EXCEPTION_DETAIL, SKIP,
COMPARISON_FLAGS, REPORT_*DIFF

doctest details (11)

Dealing with dicts

```
>>> foo = dict(a=1, b="hello", \
...                c="bye")
>>> foo # this might fail
{'a': 1, 'c': 'bye', 'b':
'hello'}
```

doctest details (12)

dict workaround. Sort keys

```
>>> items = foo.items()
>>> items.sort()
>>> items
[('a', 1), ('b', 'hello'), ('c',
'bye')]
```

doctest details (13)

Dealing with addresses

```
>>> class C: pass
>>> c = C()
>>> c # will most likely fail
<__main__.C instance at
0xb7a210ec>
>>> c #doctest: +ELLIPSIS
<__main__.C instance at 0x...>
```

doctest details (14)

One fails... why?

```
>>> print "foo"
foo
>>> print "bar"
bar
```

doctest details (15)

One fails... output

```
**********************************************************************
File "slides.rst", line 536, in slides.rst
Failed example:
    print "bar"
Expected:
    bar
Got:
    bar
```

doctest details (16)

Trailing whitespace can bite you...

```
>>> print "bar"
bar  # 2 spaces after bar!
```

Running doctest on a module

```
import doctest
doctest.testmod()
```

Running doctest on a file

```
import doctest
doctest.testfile(filename)
```

`doctest` soapbox

3 clear uses:

- Check examples in docstrings
- Regression testing
- Exectuable documentation

Not necessarily the same thing (in appearance or utility)

Executable documentation

```
Schema Defined Buttons
----------------------

Let's now create a schema that describes ...

  >>> import zope.interface
  >>> class IButtons(zope.interface.Interface):
  ...     apply = button.Button(title=u'Apply')
  ...     cancel = button.Button(title=u'Cancel')
```

from **zope** (note has 100% coverage)

Critique of `doctest`

- **Cons**
  - Can get messy (spacing workarounds)
  - Poor integration with coverage tools
  - Setup/teardown code can get in the way of documentation
- **Pros**
  - Lightweight
  - Easy, no API
  - Included with Python
  - Focus on documentation

Fuzzy areas

How do I know the effectiveness of my tests?

Coverage

How do I know the effectiveness of my coverage?

Coverage(2)

Different types of coverage:

- Function - Every function executed
- Line (statement) - Every line executed
- Condition (Branch) - Every possible condition executed
- Path - Every possible route through code

Coverage(3)

Most coverage tools deal with line coverage.

`coverage.py` has branch coverage.

Full Path Coverage

```
>>> def foo(bar):
...     """
...     To achieve line/statement coverage this is
...     sufficient:
...     >>> foo(-1)
...     not a positive integer
...
...     For path coverage need this too:
...     >>> foo(1)
...     a positive integer
...     """
...     if bar <= 0:
...         print 'not',
...     print 'a positive integer'
```

How many tests for full path coverage?

PyMetrics is your friend. Indicates Cyclomatic (McCabe) Complexity. This will tell the lower bound for number of tests for structured coverage, and the upper bound for tests required for line coverage.

Coverage tools for Python

- figleaf
- coverage.py

Neither tool is included with Python.

Running *coverage.py*

```
coverage run program.py
# generate html reports
coverage html -d /tmp/html_results
firefox /tmp/html_results/index.html
```

Nose

Tool for:

- Using `assert` instead of `assertEqual`/etc
- Testfinder
- Helpers for reporting coverage, stop or error, failure

Testing links

- **Python Testing Tools Taxonomy** (www.pycheesecake.org)
- **TIP - Testing in Python mailing list**

Functional Programming

(Python 1.x)

lambda

Create simple functions in a 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 = [dict(number=x) for x in '019234']
>>> data.sort(key=lambda x: float(x['number']))
>>> data #doctest: +NORMALIZE_WHITESPACE
[{'number': '0'}, {'number': '1'}, {'number': '2'},
{'number': '3'}, {'number': '4'}, {'number': '9'}]
```

lambda cmp example

```
>>> data = [dict(number=x) for x in '019234']
>>> data.sort(cmp=lambda x,y: cmp(x['number'], y['number']))
>>> data #doctest: +NORMALIZE_WHITESPACE
[{'number': '0'}, {'number': '1'}, {'number': '2'},
{'number': '3'}, {'number': '4'}, {'number': '9'}]
```

cmp is from the sort method in Python 3.x

map

Apply a function to items of a sequence

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

reduce

Apply a function to pairs of the sequence

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

filter

Return a sequence items for which `function(item)` is `True`

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

Notes about "functional" programming in *Python*

- `sum` or `for` loop can replace `reduce`
- List comprehensions replace `map` and `filter`

Example Assignment

`sample.py`

Assignment

`functional.py`

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.func_name
'foo'
>>> foo.func_doc
'docstring for foo'
```

(PEP 234 Python 2.1)

function scope

A function knows about itself

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

>>> foo2()
NAME foo2
```

function attributes

Can attach data to function a priori

```
>>> 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)
- keyword/named/default parameters (many)
- variable parameters (one), preceded by *
- variable keyword parameters (one), preceded by **

a gotcha

When the function is created, the named/default parameters values are assigned to the function (`.func_defaults`)

named parameters

Don't default to mutable types.

```
>>> def named_param(a, foo=[]):
...     if not foo:
...         foo.append(a)

>>> named_param.func_defaults
([],)

>>> named_param(1)
>>> named_param.func_defaults
([1],)
```

mutable types

lists and *dicts* are mutable. When you modify them you don't create a new list (or dict). *Strings* and *ints* are immutable.

Parameters are evaluated when the `def` they belong to is evaluated. This usually happens at module import.

named parameters (2)

Don't default to mutable types.

```
>>> def named_param(a, foo=None):
...     foo = foo or []
...     if not foo:
...         foo.append(a)
```

*args and **kwargs

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

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

*args and **kwargs (2)

```
>>> def param_func(a, b=2, c=5):
...     print [x for x in [a, b, c]]
>>> param_func(2)
[2, 2, 5]
>>> param_func(3, 4, 5)
[3, 4, 5]
>>> param_func(c=4, b=5, a=6)
[6, 5, 4]
```

`*args` and `**kwargs` (3)

The `*` before a sequency parameter in an invocation "flattens" (or splats) the sequence

*args and **kwargs (4)

```
>>> def args_func(a, *args):
...     print [x for x in [a, args]]
>>> args_func(2)
[2, ()]
>>> args_func(4, *(5, 6)) # splat
[4, (5, 6)]
>>> args_func(4, 5, 6)    # same as above
[4, (5, 6)]
>>> args_func(5, (6, 7)) # tricksey!
[5, ((6, 7),)]
```

*args and **kwargs (5)

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

*args and **kwargs (6)

```
>>> def kwargs_func(a, **kwargs):
...     print [x for x in [a, kwargs]]
>>> kwargs_func(2)
[2, {}]
>>> kwargs_func({'a' : 3})
[{'a': 3}, {}]
>>> kwargs_func({'b' : 4})
[{'b': 4}, {}]
>>> kwargs_func(**{'a' : 3}) # splat
[3, {}]
>>> kwargs_func(a = 3)  # same
[3, {}]
>>> kwargs_func(**{'b' : 4})
Traceback (most recent call last):
  ...
TypeError: kwargs_func() takes exactly 1 non-keyword
argument (0 given)
```

*args and **kwargs (7)

```
>>> def param_func(a, b='b', *args, **kwargs):
...     print [x for x in [a, b, args, kwargs]]
>>> param_func(2, 'c', 'd', 'e,')
[2, 'c', ('d', 'e,'), {}]
>>> params = ('f', 'g', 'h')
>>> param_func(3, params)
[3, ('f', 'g', 'h'), (), {}]

>>> param_func(4, *params) # tricksey!
[4, 'f', ('g', 'h'), {}]
>>> param_func(4, 'f', 'g', 'h')  # same
[4, 'f', ('g', 'h'), {}]

>>> param_func(6, **{'foo':'bar'})
[6, 'b', (), {'foo': 'bar'}]
>>> param_func(6, foo='bar')
[6, 'b', (), {'foo': 'bar'}]
```

*args and **kwargs (8)

Equivalent:

```
>>> params = ('f', 'g', 'h')
>>> param_func(*params) # tricksey!
['f', 'g', ('h',), {}]

>>> param_func(params[0], params[1], params[2])
['f', 'g', ('h',), {}]
```

*args and **kwargs (9)

See
http://docs.python.org/reference/expressions.html#
for gory details

Assignment

`funcargs.py`

Closures

(PEP 227 Python 2.1)

Closures

- **Wikipedia:** First-class function with free variables that are bound by the lexical environment
- **Python:** 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*.

Closures (2)

Useful as function generators

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

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

Closures (3)

Notice the function attributes

```
>>> add_5.func_name
'adder'
```

Closures (4)

Nested functions only have write access to global and local scope (Python 2.x)

```
>>> X = 3
>>> def outer():
...     X = 4 # now local
...     y = 2
...     def inner():
...         global X
...         X = 5
...     print X
...     inner()
...     print X

>>> outer()
4
4
>>> X
5
```

Closures (5)

Python 3.x has `nonlocal` scope to access variables in outer functions

Assignment

`closures.py`

Decorators

(PEP 318, 3129, Python 2.4)

Decorators

Since functions are `function` instances you can wrap them

Decorators (2)

Allow you to

- modify arguments
- modify function
- modify results

Decorators (3)

Count how many times a function is called

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

Decorators (4)

Attach it to a function

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

>>> 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'
```

equals

```
>>> 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)
```

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():
...     pass

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

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

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

Decorator Template

```
>>> def decorator(func_to_decorate):
...     def wrapper(*args, **kwargs):
...         # do something before invocation
...         result = func_to_decorate(*args,
**kwargs)
...         # do something after
...         return result
...     # update wrapper.__doc__ and .func_name
...     # or functools.wraps
...     return wrapper
```

Classes as Decorators

```
>>> class decorator_class(object):
...     def __init__(self, function):
...         self.function = function
...     def __call__(self, *args, **kwargs):
...         # do something before invocation
...         result = self.function(*args,
**kwargs)
...         # do something after
...         return result
```

Classes as Decorators (2)

```
>>> @decorator_class
... def function():
...     # implementation
```

Classes as Decorators (3)

```
>>> class Decorator(object):
...     # in __init__ set up state
...     def __call__(self, function):
...         def wrapper(*args, **kwargs):
...             # do something before
invocation
...             result = function(*args,
**kwargs)
...             # do something after
...             return result
...         return wrapper
```

This lets you have access to the instance of a decorator later

Classes as Decorators (4)

```
>>> deco = Decorator()
>>> @deco
... def function():
...     # implementation

>>> # perhaps modify deco later
```

Classes as Decorators (5)

Not the same as "Class Decorators". See PEP 3129

Parameterized decorators (need 2 closures)

```
>>> def limit(length):
...     def decorator(function):
...         def wrapper(*args, **kwargs):
...             result = function(*args, **kwargs)
...             result = result[:length]
...             return result
...         return wrapper
...     return decorator

>>> @limit(5) # notice parens
... def echo(foo): return foo

>>> echo('123456')
'12345'
```

Parameterized decorators

```
>>> @limit(5)
... def echo(foo): return foo
```

syntactic sugar for

```
>>> echo = limit(5)(echo)
```

decorator tidying

function attributes get mangled

```
>>> def echo2(input):
...     """return input"""
...     return input
>>> echo2.__doc__
'return input'
>>> echo2.func_name
'echo2'

>>> echo3 = limit(3)(echo2)
>>> echo3.__doc__    # empty!!!
>>> echo3.func_name
'wrapper'
```

decorator tidying (2)

```
>>> def limit(length):
...     def decorator(function):
...         def wrapper(*args, **kwargs):
...             result = function(*args, **kwargs)
...             result = result[:length]
...             return result
...         wrapper.__doc__ = function.__doc__
...         wrapper.func_name = function.func_name
...         return wrapper
...     return decorator

>>> echo4 = limit(3)(echo2)
>>> echo4.__doc__
'return input'
>>> echo4.func_name
'echo2'
```

decorator tidying (3)

```
>>> import functools
>>> def limit(length):
...     def decorator(function):
...         @functools.wraps(function)
...         def wrapper(*args, **kwargs):
...             result = function(*args, **kwargs)
...             result = result[:length]
...             return result
...         return wrapper
...     return decorator

>>> echo5 = limit(3)(echo2)
>>> echo5.__doc__
'return input'
>>> echo5.func_name
'echo2'
```

Multple decorators

```
>>> @count
... @limit(4)
... def long_word():
...     return "supercalafrag"
>>> long_word()
'supe'
```

equivalent to:

```
>>> long_word = count(limit(4)(long_word))
```

Gotcha

```
>>> class verbose(object):
...     def __init__(self, func):
...         self.func = func
...
...     def __call__(self, *args, **kwargs):
...         print "BEFORE"
...         result = self.func(*args, **kwargs)
...         print "AFTER"
...         return result
```

Gotcha (2)

```
>>> class Adder(object):
...     def __init__(self): pass
...
...     @verbose
...     def add(self, x, y):
...         return x + y

>>> a = Adder()
>>> a.add(2, 4)
BEFORE
Traceback (most recent call last):
  ...
TypeError: add() takes exactly 3 arguments (2 given)
```

Gotcha (3)

`verbose` decorates an *unbound* method (has no class instance during decoration time). But invoked on an instance.

Gotcha (4)

```
>>> class Foo(object):
...     def bar(self):
...         pass
```

Bound

```
>>> f = Foo()
>>> f.bar
<bound method Foo.bar of <__main__.Foo object at 0x910b90>>
```

Unbound

```
>>> Foo.bar
<unbound method Foo.bar>
```

Gotcha (5)

Solution - `verbose` needs to implement *descriptor* protocol to bind method to the instance

Gotcha (6)

```
>>> class verbose(object):
...     def __init__(self, func):
...         self.func = func
...
...     def __call__(self, *args, **kwargs):
...         print "BEFORE"
...         result = self.func(*args, **kwargs)
...         print "AFTER"
...         return result
...
...     def __get__(self, obj, type=None):
...         print "OBJ", obj
...         print "TYPE", type
...         if obj is None:
...             return self
...         # Create new instance with bound method
...         new_func = self.func.__get__(obj, type)
...         return self.__class__(new_func)
```

Gotcha (7)

```
>>> class Adder(object):
...     def __init__(self): pass
...
...     @verbose
...     def add(self, x, y):
...         return x + y

>>> a = Adder()
>>> a.add(2, 4)
OBJ <__main__.Adder object at 0x9190d0>
TYPE <class '__main__.Adder'>
BEFORE
AFTER
6
```

Gotcha (8)

Functions invoke methods with correct instance

```
>>> def verbose(func):
...     def wrapper(*args, **kwargs):
...         print "BEFORE", args
...         result = func(*args, **kwargs)
...         print "AFTER"
...         return result
...     return wrapper

>>> class Adder(object):
...     @verbose
...     def add(self, x, y):
...         return x + y

>>> a = Adder()
>>> a.add(2, 4)
BEFORE (<__main__.Adder object at 0x91b350>, 2, 4)
AFTER
6
```

Uses for decorators

- caching
- monkey patching stdio
- jsonify
- logging time in function call
- change cwd
- timeout a function call

Decorator rehash

Allows you to

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

What if I want to tweak decoration parameters at runtime?

(ie `@limit(4)` instead of `@limit(3)`)

Tweak Parameters

- Use class instance decorator
- Tweak `wrapper` attributes
- Use context manager
- or

Don't decorate

Since a decorator is just a closure, you can invoke at run time. Like this:

```
result = limit(4)(echo2)('input')
```

Another option

Context managers let you dictate before and after conditions of execution. It is possible to create decorators that serve as context managers.

Assignment

`decorators.py`

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

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

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

>>> t = Timing()
```

Class Decorators (4)

Works with subclasses

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

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

Class Decorators (5)

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

Properties

(Python 2.2, 2.6 added `getter`, `setter`, `deleter`)

Properties

Utilize *descriptors* to allow attribute to invoke methods. If you have an attribute you can later add an underlying method to do setting, getting, deleting.

Properties (2)

```
>>> class Person(object):
...     def __init__(self):
...         self.name = None
```

to

```
>>> class Person(object):
...     def __init__(self):
...         self._name = None
...     def get_name(self):
...         return self._name
...     def set_name(self, name):
...         self._name = name.replace(';', '')
...     name = property(get_name, set_name)
```

Properties (3)

```
>>> p = Person()
>>> p.name = 'Fred; Drop TABLE people;'
>>> print p.name
Fred Drop TABLE people
```

Properties (4)

2.6 style

```
>>> class Person2(object):
...     def __init__(self):
...         self._name = None
...
...     @property
...     def name(self):
...         return self._name
...
...     @name.setter
...     def name(self, value):
...         self._name = value.replace(';', '')
...
...     @name.deleter
...     def name(self):
...         del self._name
```

Properties (5)

```
>>> p = Person2()
>>> p.name = 'Fred; Drop TABLE people;'
>>> print p.name
Fred Drop TABLE people
```

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(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)
```

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]
```

Assignment

`listcomprehensions.py`

Iterators

(PEP 234)

Iterators

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

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

equals

```
>>> iterable = iter(sequence)
>>> while True:
...     try:
...         # py3 .__next__()
...         x = iterable.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]
```

Assignment

`iterators.py`

Generators

(PEP 255, 342, Python 2.3)

generators

Functions with `yield` remember 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)`)

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.

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

Generator example

```
def fetch_many_wrapper(result, count=20000):
    """
    In an effort to speed up queries, this wrapper
    fetches count objects at a time.  Otherwise our
    implementation has sqlalchemy fetching 1 row
    at a time (~30% slower).
    """
    done = False
    while not done:
        items = result.fetchmany(count)
        done = len(items) == 0
        if not done:
            for item in items:
                yield item
```

Recursive generator example

```
def find_files(base_dir, recurse=True):
    """
    yield files found in base_dir
    """
    for name in os.listdir(base_dir):
        filepath = os.path.join(base_dir, name)
        if os.path.isdir(filepath) and recurse:
            # make sure to iterate when recursing!
            for child in find_files(filepath, recurse):
                yield child
        else:
            yield filepath
```

Assignment

`generators.py`

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)
[0, 2, 4]
```

Generator expressions (4)

If Generators are confusing, but List Comprehensions make sense, you 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
```

Assignment

`genexp.py`

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
```

Context Mgr (3)

Seen in files

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

Context Mgr (3)

Two ways to create:

- class
- decorated generator

Context Mgr (4)

Context managers can optionally return an item with `as`

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
```

Uses for Context Managers

- Transactions
- Acquiring locks
- closing/cleaning up
- nesting for generating html/xml

Assignment

`ctxmgr.py`

Not covered

- class/static methods
- metaclasses

That's all

Questions? Tweet or email me

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