Python Full Meal Deal

matthewharrison@gmail.com
http://panela.blog-city.com/

Warning

- Starting from zero
- Hands on
  - (short) lecture
  - (short) code
  - repeat until time is gone

Get code

Thumbdrive has `fullmeal.tar.gz`. Unzip it somewhere (`tar -zxvf fullmeal.tar.gz`)

Why Python?

- Used (almost) everywhere
- Fun
- Concise

Hello World

hello world

```
print "hello world"
```

from interpreter

```
$ python
>>> print "hello world"
hello world
```

From script

Make file `hello.py` with

```
print "hello world"
```

Run with:

```
python hello.py
```

interpreter vs programs

Objects

Objects

Everything in *Python* is an object that has:

- an *identity* (`id`)
- a *type* (`type`). Determines what operations object can perform.
- a *value* (mutable or immutable)

id

```
>>> a = 4
>>> id(a)
6406896
```

type

```
>>> a = 4
>>> type(a)
<type 'int'>
```

Value

Mutable: When you alter the item, the `id` is still the same. Dictionary, List
Immutable: String, Integer, Tuple

Mutable

```
>>> b = []
>>> id(b)
140675605442000
>>> b.append(3)
>>> b
[3]
>>> id(b)
140675605442000 # SAME!
```

Immutable

```
>>> a = 4
>>> id(a)
6406896
>>> a = 5
>>> id(a)
6406872 # DIFFERENT!
```

Variables

```
a = 4 # Integer
b = 5.6 # Float
c = "hello" # String
a = "4" # rebound to String
```

naming

- lowercase
- underscore_between_words
- don't start with numbers

See PEP8

Assignment

`variables.py`

Math

+, **−**, ***, /, ****, **%** (modulus)

Careful with integer division

```
>>> 3/4
0
>>> 3/4.
0.75
```

What happens when you raise 10 to the 100th?

Long

```
>>> 10**100
100000000000000000000000000000000
000000000000000000000000000000000
000000000000000000000000000000000
00L
```

Strings

```
name = 'matt'
with_quote = "I ain't gonna"
longer = """This string has
multiple lines
in it"""
```

dir

```
>>> dir("a string")
['__add__', '__class__', ...
'startswith', 'strip',
'swapcase', 'title', 'translate',
'upper', 'zfill']
```

Whats with all the `'__blah__'`?

dunder methods

dunder (double under) methods determine what will happen when `+` (`__add__`) or `/` (`__div__`) is called.

help

```
>>> help("a string".startswith)
```

```
Help on built-in function startswith:

startswith(...)
  S.startswith(prefix[, start[, end]]) -> bool

  Return True if S starts with the specified prefix, False
otherwise.
  With optional start, test S beginning at that position.
  With optional end, stop comparing S at that position.
  prefix can also be a tuple of strings to try.
```

Assignment

`strings.py`

Comments

comments

Comments follow a **#**

More Types

booleans

```
a = True
b = False
```

sequences

- *lists*
- *tuples*
- *sets*

lists

```
>>> a = []
>>> a.append(4)
>>> a.append('hello')
>>> a.append(1)
>>> a.sort()
>>> print a
[1, 4, 'hello']
```

tuples

Immutable

```
>>> b = (2,3)
>>> b.append(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute
'append'
```

tuple vs *list*

- Tuple
  - Hetergenous state (name, age, address)
- List
  - Homogenous, mutable (list of names)

Assignment

`lists.py`

Dictionaries

dictionaries

Also called *hashmap* or *associative array* elsewhere

```
>>> age = {}
>>> age['george'] = 10
>>> age['fred'] = 12
>>> age['henry'] = 10
>>> print age['george']
10
```

dictionaries (2)

Find out if `'matt'` in `age`

```
>>> 'matt' in age
False
```

dictionaries (3)

```
>>> print age['charles']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'charles'
>>> print age.get('charles', 'Not found')
Not found
```

dictionaries (4)

```
>>> if 'charles' not in age:
...     age['charles'] = 10
```

shortcut

```
>>> age.setdefault('charles', 10)
10
```

dictionaries (5)

Even more useful if we map to a list of items

```
>>> room2members = {}
>>> member = 'frank'
>>> room = 'room5'
>>> room2members.setdefault(room,
[]).append(member)
```

dictionaries (6)

Even more useful if we map to a list of items

```
>>> room2members = {}
>>> member = 'frank'
>>> room = 'room5'
>>> if room in room2members:
...     members = room2members[room]
...     members.append(member)
... else:
...     members = [member]
...     room2members[room] = members
```

dictionaries (7)

Removing `'charles'` from `age`

```
>>> del age['charles']
```

Assignment

`dictionaries.py`

Functions

functions

```
def add_2(num):
    """ return 2
    more than num
    """
    return num + 2

five = add_2(3)
```

whitespace

Instead of { use a : and indent consistently (4 spaces)

default (named) parameters

```
def add_n(num, n=3):
    """default to
    adding 3"""
    return num + n

five = add_n(2)
ten = add_n(15, -5)
```

`__doc__`

Functions have *docstrings*. Accessible via `.__doc__` or `help`

__doc__

```
>>> def echo(txt):
...     "echo back txt"
...     return txt
>>> help(echo)
Help on function echo in module __main__:
<BLANKLINE>
echo(txt)
    echo back txt
<BLANKLINE>
```

naming

- lowercase
- underscore_between_words
- don't start with numbers
- verb

Assignment

`functions.py`

Conditionals

conditionals

```
if grade > 90:
    print "A"
elif grade > 80:
    print "B"
elif grade > 70:
    print "C"
else:
    print "D"
```

Remember the colon/whitespace!

Booleans

```
a = True
b = False
```

Boolean tests

Supports (>, >=, <, <=, ==, !=)

```
>>> 5 > 9
False
>>> 'matt' != 'fred'
True
>>> isinstance('matt', basestring
)
True
```

Iteration

iteration

```
for number in [1,2,3,4,5,6]:
    print number

for number in range(1, 7):
    print number
```

iteration (2)

If you need indices, use `enumerate`

```
animals = ["cat", "dog", "bird"]
for index, value in enumerate(animals):
    print index, value
```

iteration (3)

Can `break` out of loop

```
for item in sequence:
    # process until first negative
    if item < 0:
        break
    # process item
```

iteration (4)

Can `continue` to skip over items

```
for item in sequence:
    if item < 0:
        continue
    # process all positive items
```

iteration (5)

Can loop over lists, strings, iterators, dictionaries... sequence like things:

```
my_dict = { "name": "matt", "cash": 5.45}
for key in my_dict.keys():
  # process key

for value in my_dict.values():
  # process value

for key, value in my_dict.items():
  # process items
```

pass

Take no action

```
for i in range(10):
    # do nothing 10 times
    pass
```

Assignment

`loops.py`

Slicing

Slicing

Sequences (lists, tuples, strings, etc) can be *sliced*

```
my_pets = ["dog", "cat", "bird"]
favorite = my_pets[0]
bird = my_pets[-1]
```

Slicing (2)

Slices can take an end index

```
my_pets = ["dog", "cat", "bird"]
# a list
cat_and_dog = my_pets[0:2]
cat_and_bird = my_pets[1:3]
```

Slicing (3)

Slices can take a stride

```
my_pets = ["dog", "cat", "bird"]
# a list
dog_and_bird = [0:3:2]
zero_three_etc = range(0,10)[::3]
```

Slicing (4)

Just to beat it in

```
veg = "tomatoe"
correct = veg[:-1]
tmte = veg[::2]
oetamot = veg[::-1]
```

File IO

File Input

Open a file to read from it:

```
fin = open("foo.txt")
for line in fin:
    # manipulate line

fin.close()
```

File Output

Open a file using `'w'` to `write` to a file:

```
fout = open("bar.txt", "w")
fout.write("hello world")
fout.close()
```

Always remember to close your files!

closing with `with`

implicit `close`

```
with open('bar.txt') as fin:
    for line in fin:
        # process line
```

Hint

Much code implements "file-like" behavior (`read`, `write`). Try to use interfaces that take files instead of filenames where possible.

Hint (2)

```
def process(filename):
    fin = open(filename)
    process_file(fin)
    fin.close()

def process_file(fin):
    # go crazy
```

Classes

Classes

```
class Animal(object):
    def __init__(self, name):
        self.name = name

    def talk(self):
        print "Generic Animal Sound"

animal = Animal("thing")
animal.talk()
```

Classes (2)

notes:

- `object` (base class) (fixed in 3.X)
- *dunder* init (constructor)
- all methods take `self` as first parameter

Classes(2)

Subclassing

```
class Cat(Animal):
    def talk(self):
        print '%s says, "Meow!"' % (self.name)

cat = Cat("Groucho")
cat.talk() # invoke method
```

Classes(3)

```
class Cheetah(Cat):
    """classes can have
    docstrings"""

    def talk(self):
        print "Growl"
```

naming

- CamelCase
- don’t start with numbers
- Nouns

Assignment

`classes.py`

Packages, Modules and Importing

importing

Python can `import` *packages* and *modules* via:

```
import package
import module
from math import sin
import longname as ln
```

Packages

File layout (excluding README, etc):

```
packagename/
   __init__.py
   code1.py
   code2.py
   subpackage/
      __init__.py
```

Modules

Just a `.py` file

naming

- lowercase
- nounderscorebetweenwords
- don’t start with numbers

importing code

First Install

- via operating system
- `easy_install`
- updating `PYTHONPATH` (env variable)
- changing `sys.path` in code

Import code (2)

- `import packagename`
- `import module`

Exceptions

Exceptions

Can catch exceptions

```
try:
      f = open("file.txt")
except IOError, e:
      # handle e
```

Exceptions (2)

Can raise exceptions

```
raise RuntimeError("Program
failed")
```

Chaining Exceptions (3)

```
try:
    some_function()
except ZeroDivisionError, e:
    # handle specific
except Exception, e:
    # handle others
```

finally

`finally` always executes

```
try:
    some_function()
except Exception, e:
    # handle others
finally:
    # cleanup
```

else

runs if no exceptions

```
try:
    print "hi"
except Exception, e:
    # won't happen
else:
    # first here
finally:
    # finally here
```

re-raise

Usually a good idea to re-raise if you don't handle it. (just `raise`)

```
try:
    # erorry code
except Exception, e:
    # handle higher up
    raise
```

File Organization

program layout

```
PackageName/
  README
  setup.py
  bin/
    script
  docs/
  test/ # some include in package_name
  packagename/
    __init__.py
    code1.py
    subpackage/
      __init__.py
```

scripts vs libraries

- Executable?
- Importable?

script layout

- #!/usr/bin/env python
- docstrings
- importing
- meta data
- logging
- implementation
- testing?
- `if __name__ == '__main__':`
- optparse (now argparse)

ifmain

```
if __name__ == '__main__':
    sys.exit(main(sys.argv))
```

main

```
def main(arguments):
    # process args
    # run
    # return exit code
```

What if I want to reuse logic from my script?

Script wrapper

Put logic in library, have script be a simple wrapper

```
#!/usr/bin/env python
import sys
import scriptlib
sys.exit(scriptlib.main(sys.argv)
)
```

Functional Programming

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'}]
```

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`

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 properties

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

function definition

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

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.

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 **kw

*args is a tuple of parameters values.

**kw is a dictionary of name/value pairs.

*args and **kw (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 **kw (3)

```
>>> def args_func(a, *args):
...     print [x for x in [a, args]]
>>> args_func(2)
[2, ()]
>>> args_func(3, 4, 5)
[3, (4, 5)]
>>> args_func(4, *(5, 6))
[4, (5, 6)]
>>> args_func(5, (6, 7)) # tricksey!
[5, ((6, 7),)]
```

*args and **kw (4)

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

*args and **kw (5)

```
>>> def param_func(a, b='b', *args, **kw):
...     print [x for x in [a, b, args, kw]]
>>> 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(*params) # tricksey!
['f', 'g', ('h',), {}]
>>> param_func(5, 'x', *params)
[5, 'x', ('f', 'g', 'h'), {}]
>>> param_func(6, **{'foo':'bar'})
[6, 'b', (), {'foo': 'bar'}]
```

*args and **kw (6)

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

Closures

Closures

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

In Python: Wrap functions with functions. Outer functions have free variables that are bound to inner functions. (ie attach data to inner functions)

Closures (2)

Useful as function generators

```
>>> def add_x(x):
...     def adder(num):
...         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'
```

Assignment

`closures.py`

Decorators

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, **kw):
...         global call_count
...         call_count += 1
...         return func(*args, **kw)
...     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)
```

Decorator Template

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

Decorators can also be classes

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

Decorators can also be classes (2)

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

This lets you have an instance of a decorator that stores state (rather than using global state)

Decorators can also be classes (3)

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

Parameterized decorators (need 2 closures)

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

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

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

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, **kw):
...             result = function(*args, **kw)
...             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, **kw):
...             result = function(*args, **kw)
...             result = result[:length]
...             return result
...         return wrapper
...     return decorator

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

Uses for decorators

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

Decorator rehash

Allows you to

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

Assignment

`closures.py`

List comprehensions

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]
```

Iterators

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:
...         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):
...         # 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]
```

Generators

generators

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

generators (2)

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

generators (3)

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

generators (4)

Generators return a generator instance.
Iterate over them for values

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

generators (5)

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 (6)

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 (7)

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 (8)

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 (9)

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`)

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
```

Generator Expressions

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 can create (some) 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
```

Not covered

- context managers
- class/static methods
- properties
- metaclasses

Debugging

Poor mans

`print` works a lot of the time

Remember

Clean up `print` statements. If you really need them, use `logging` or write to `sys.stdout`

pdb

```
import pdb; pdb.set_trace()
```

pdb commands

- `h` - help
- `s` - step into
- `n` - next
- `c` - continue
- `w` - where am I (in stack)?
- `l` - list code around me

Testing

testing

see `unittest` and `doctest`

nose

nose is useful to run tests with coverage, profiling, break on error, etc

3rd party

Packaging

packaging

Somewhat of a mess and in flux. Find something else that does what you want and steal.... er ....copy it.

packaging (2)

I use `virtualenv` and `easy_install`

3rd party

packaging (3)

`pypi` hosts packages

Other Tools

Editors

Most editors have some notion of Python support

Linting

- `pyflakes`- least verbose (dead/redundant code)
- `pychecker` - more verbose, imports code, slower
- `pylint` - most verbose, configurable, "rates" code

3rd party

Refactoring

- `rope` - not perfect, somewhat slow

3rd party