Intermediate Python

Functional Programming

Python has support for basic functional constructs

lambda

`lambda` allows for the creation of basic one line functions:

```
>>> def mul(a, b):
...     return a * b
>>> mul_2 = lambda a, b: a*b
>>> mul_2(4, 5) == mul(4,5)
True
```

map

`map` applies a function to items of a sequence:

```
>>> map(str, range(3))
['0', '1', '2']
```

reduce

`reduce` applies a function to pairs of the sequence:

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

filter

`filter` returns 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`

Functions

Functions can take `*args` (variable parameters) and `**kwargs` (variable keyword parameters). During function invocation the `*` and `**` operator flattens or *splats* the parameters.

**args and **kwargs*

```
>>> 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,'), {}]
>>> args = ('f', 'g')
>>> param_func(3, args)
[3, ('f', 'g'), (), {}]
>>> param_func(4, *args) # tricksey!
[4, 'f', ('g',), {}]
>>> param_func(5, 'x', *args)
[5, 'x', ('f', 'g'), {}]
>>> param_func(*args) # tricksey!
['f', 'g', (), {}]
```

same as:

```
>>> param_func('f', 'g')
['f', 'g', (), {}]
```

Also:

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

Same as:

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

Closures PEP 227

Closures are inner functions that have (readonly in Python 2.x) access to the state in which they were defined. (Use `nonlocal` keyword in Python 3.x for write access). One common use is for generating functions:

```
>>> def add_x(x):
...     def add(num):
...         # note x is a "free" variable
...         return x + num
...     return add
...
>>> add_2 = add_x(2)
>>> add_2(5)
7
```

Decorators PEP 318, 3129

Decorator Template

Note that `__doc__` and `__name__` should be updated to ensure non-breakage (`pickle`, `help`, etc). Wrapper the wrapper with `@functools.wraps(function)` will also suffice:

```
>>> 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
```

Syntactic Sugar

The following are the same:

```
>>> @decorator
... def foo():
...     print "hello"
```

and:

```
>>> def foo():
...     print "hello"
>>> foo = decorator(foo)
```

Invoking a decorated function:

```
>>> foo()
before invocation
hello
after invocation
```

Parameterized decorators (need 2 closures)

```
>>> 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
```

The following are the same:

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

and:

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

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

Note that `@limit(5)` is syntactic sugar for `echo = limit(5)(echo)`

Class Decorator with `__get__` workaround

Binds methods to correct instances:

```
>>> 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):
...         if obj is None:
...             return self
...         # Create new instance with bound method
...         new_func = self.func.__get__(obj, type)
...         return self.__class__(new_func)
```

Class Decorators PEP 3129

A callable that takes a class and returns a class:

```
>>> 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
```

List Comprehension PEP 202

List comprehensions allow for easy creation of lists, as well as `map` and `filter` operations:

```
>>> 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)
```

Nested List Comprehensions

List comprehensions can also 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))
```

Iteration Protocol PEP 234

- get an iterator (`__iter__`)
- call `next` on it
- `StopIteration` error means iteration is done

```
>>> 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 instances iterable

```
>>> class Iter(object):
...     def __iter__(self):
...         return self
...     def next(self):
...         # return next item
```

Generators PEP 255, 342

Generators create sequences one item at a time (`yield`). They remember state and return to the line following `yield` during iteration:

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

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

Making instances generate

```
>>> class Generate(object):
...     def __iter__(self):
...         # generate here
...         # logic
...         yield result
```

Generator expressions PEP 289

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

```
>>> [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]
```

Dict/Set Comprehensions PEP 274

Like List Comps, but use `{` instead of `(`:

```
>>> {x:x for x in xrange(5)}  # dict comp
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4}

>>> {x for x in xrange(5)}  # set comp
set([0, 1, 2, 3, 4])
```

Context Managers PEP 343

Context Managers provide entry/exit level changes at a block level (ie `try/except/finally`):

```
>>> with a_cm() as foo:
...     # block logic
```

When `a_cm()` is invoked, the entry logic is performed. When the block is done, the exit logic is performed. Note it is not necessary to return an object from `__enter__` (ie `as` is not required).

Class Style Context Manager

```
>>> class a_cm:
...     def __init__(self):
...         # init
...     def __enter__(self):
...         # enter logice
...         return object
...     def __exit__(self, type, value, tb):
...         # exit logic
...         # can handle block exceptions with params
...         # return a boolean to suppress exceptions
```

Decorator Style Context Manager

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def a_cm():
...     # enter logic
...     try:
...         yield object
...     finally:
...         # exit logic
```

Thanks

@__mharrison__
http://hairysun.com/