Functional Python and Comprehensions

@__mharrison__
http://hairysun.com

Utah Python Jan 2013

About Me

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

Sample code

Tell me if you want to do the samples

Python 2 or 3?

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

Outline

- Functional Programming
- List Comprehensions
- Generator Expressions
- Dict/Set Comprehensions

Functional Programming

(Python 1.4)

Functional Programming

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

```
>>> sum(range(10))  # cheating
>>> reduce(operator.add, range(10))
```

Imperative Programming

Using statements to affect a program's state

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

First-class functions

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

Higher-order functions

Functions that accept functions as parameters.

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`

Tail call optimization

Optimization for recursion to not create a new stack. Python does not have it (Guido says no).

Tail call optimization (2)

Decorator recipe that *slowly* hacks the stack

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'}]
```

Use key not cmp

`lambda` parameters

Supports

- `normal`
- `named`
- `*args`
- `**kwargs`

`lambda` *expressions*

Statements cause problems

```
>>> is_pos = lambda x: if x >=0: 'pos'
  File "<stdin>", line 1
    is_pos = lambda x: if x >=0: 'pos'
                         ^
SyntaxError: invalid syntax
```

lambda *expressions* (2)

Expressions don't

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

`lambda` *expressions* (3)

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

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

lambda *expressions* (4)

Example from stdlib (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-line statements

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]
```

map (3)

Example from stdlib (`tarfile.py`)

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

`map` (4)

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

`map` (5)

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.

reduce (3)

Example from stdlib (`csv.py`). Guessing the quote character

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

`reduce` (4)

Note the `lambda` uses a trick. Named parameter to pass in `quotes`.

filter

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.

filter (3)

Example from stdlib (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

`sample.py`

Assignment

`functional.py`

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`

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`

Dict Comprehensions

(PEP 274 Python 2.7)

Dict comprehensions

Similar to list comprehensions. (Yes, "dict" not "dictionary")

Dict comprehensions (2)

This

```
>>> result = {x:x*x for x in range(5)}
```

Instead of

```
>>> result = dict((x,x*x) for x in range(5))
```

Dict comprehensions (3)

- More legible
- No list created first (when `dict` combined with LC)

Set Comprehensions

(PEP 274 Python 2.7)

Set comprehensions

Similar to list comprehensions. But with { and }.

Set comprehensions (2)

This

```
>>> result = {x for x in range(5)}
>>> result
set([0, 1, 2, 3, 4])
```

Instead of

```
>>> result = set(x for x in range(5))
```

(`range(5)` is lousy here)

Set comprehensions (3)

- More legible
- No list created first (when `set` combined with LC)

That's all

Questions? Tweet or email me

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