Python & Stuff

All the things I like about Python, plus a bit more.

Jacob Perkins

- Python Text Processing with NLTK 2.0 Cookbook
- Co-Founder & CTO @weotta
- Blog: http://streamhacker.com
- NLTK Demos: http://text-processing.com
- @japerk
- Python user for > 6 years

What I use Python for

- web development with Django
- web crawling with Scrapy
- NLP with NLTK
- argparse based scripts
- processing data in Redis & MongoDB

Topics

- functional programming
- I/O
- Object Oriented programming
- scripting
- testing
- remoting
- parsing
- package management
- data storage
- performance

Functional Programming

- list comprehensions
- slicing
- iterators
- generators
- higher order functions
- decorators
- default & optional arguments
- switch/case emulation

List Comprehensions

```
>>> [i for i in range(10) if i % 2]
[1, 3, 5, 7, 9]
>>> dict([(i, i*2) for i in range(5)])
{0: 0, 1: 2, 2: 4, 3: 6, 4: 8}
>>> s = set(range(5))
>>> [i for i in range(10) if i in s]
[0, 1, 2, 3, 4]
```

Slicing

```
>>> range(10)[:5]
[0, 1, 2, 3, 4]
>>> range(10)[3:5]
[3, 4]
>>> range(10)[1:5]
[1, 2, 3, 4]
>>> range(10)[::2]
[0, 2, 4, 6, 8]
>>> range(10)[-5:-1]
[5, 6, 7, 8]
```

Iterators

```
>>> i = iter([1, 2, 3])
>>> i.next()
1
>>> i.next()
2
>>> i.next()
3
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Generators

```
>>> def gen_ints(n):
...     for i in range(n):
...         yield i
...
>>> g = gen_ints(2)
>>> g.next()
0
>>> g.next()
1
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Higher Order Functions

```
>>> def hof(n):
...     def addn(i):
...         return i + n
...     return addn
...
>>> f = hof(5)
>>> f(3)
8
```

Decorators

```
>>> def print_args(f):
...     def g(*args, **kwargs):
...         print args, kwargs
...         return f(*args, **kwargs)
...     return g
...
>>> @print_args
... def add2(n):
...     return n+2
...
>>> add2(5)
(5,) {}
7
>>> add2(3)
(3,) {}
5
```

Default & Optional Args

```
>>> def special_arg(special=None, *args, **kwargs):
...     print 'special:', special
...     print args
...     print kwargs
...
>>> special_arg(special='hi')
special: hi
()
{}
>>>
>>> special_arg('hi')
special: hi
()
{}
```

switch/case emulation

```
OPTS = {
    “a”: all,
    “b”: any
}

def all_or_any(lst, opt):
    return OPTS[opt](lst)
```

Object Oriented

- classes
- multiple inheritance
- special methods
- collections
- defaultdict

Classes

```
>>> class A(object):
...     def __init__(self):
...             self.value = 'a'
...
>>> class B(A):
...     def __init__(self):
...             super(B, self).__init__()
...             self.value = 'b'
...
>>> a = A()
>>> a.value
'a'
>>> b = B()
>>> b.value
'b'
```

Multiple Inheritance

```
>>> class B(object):
...     def __init__(self):
...             self.value = 'b'
...
>>> class C(A, B): pass
...
>>> C().value
'a'
>>> class C(B, A): pass
...
>>> C().value
'b'
```

Special Methods

- __init__
- __len__
- __iter__
- __contains__
- __getitem__

collections

- high performance containers
- Abstract Base Classes
- Iterable, Sized, Sequence, Set, Mapping
- multi-inherit from ABC to mix & match
- implement only a few special methods, get rest for free

defaultdict

```
>>> d = {}
>>> d['a'] += 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a'
>>> import collections
>>> d = collections.defaultdict(int)
>>> d['a'] += 2
>>> d['a']
2
>>> l = collections.defaultdict(list)
>>> l['a'].append(1)
>>> l['a']
[1]
```

I/O

- context managers
- file iteration
- gevent / eventlet

Context Managers

```
>>> with open('myfile', 'w') as f:
...     f.write('hello\nworld')
...
```

File Iteration

```
>>> with open('myfile') as f:
...     for line in f:
...             print line.strip()
...
hello
world
```

gevent / eventlet

- coroutine networking libraries
- greenlets: “micro-threads”
- fast event loop
- monkey-patch standard library
- http://www.gevent.org/
- http://www.eventlet.net/

Scripting

- argparse
- __main__
- atexit

argparse

```
import argparse

parser = argparse.ArgumentParser(description='Train a
NLTK Classifier')

parser.add_argument('corpus', help='corpus name/path')
parser.add_argument('--no-pickle', action='store_true',
  default=False, help="don't pickle")
parser.add_argument('--trace', default=1, type=int,
  help='How much trace output you want')

args = parser.parse_args()

if args.trace:
    print ‘have args’
```

__main__

```
if __name__ == '__main__':
do_main_function()
```

atexit

```
def goodbye(name, adjective):
    print 'Goodbye, %s, it was %s to meet you.' % (name,
adjective)

import atexit
atexit.register(goodbye, 'Donny', 'nice')
```

Testing

- doctest
- unittest
- nose
- fudge
- py.test

doctest

```
def fib(n):
    '''Return the nth fibonacci number.
    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(2)
    1
    >>> fib(3)
    2
    >>> fib(4)
    3
    '''
    if n == 0: return 0
    elif n == 1: return 1
    else: return fib(n - 1) + fib(n - 2)
```

doctesting modules

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

unittest

- anything more complicated than function I/O
- clean state for each test
- test interactions between components
- can use mock objects

nose

- http://readthedocs.org/docs/nose/en/latest/
- test runner
- auto-discovery of tests
- easy plugin system
- plugins can generate XML for CI (Jenkins)

fudge

- http://farmdev.com/projects/fudge/
- make fake objects
- mock thru monkey-patching

py.test

- http://pytest.org/latest/
- similar to nose
- distributed multi-platform testing

Remoting Libraries

- Fabric
- execnet

Fabric

- http://fabfile.org
- run commands over ssh
- great for "push" deployment
- not parallel yet

fabfile.py

```
from fabric.api import run

def host_type():
    run('uname -s')
```

fab command

```
$ fab -H localhost,linuxbox host_type
[localhost] run: uname -s
[localhost] out: Darwin
[linuxbox] run: uname -s
[linuxbox] out: Linux
```

execnet

- http://codespeak.net/execnet/
- open python interpreters over ssh
- spawn local python interpreters
- shared-nothing model
- send code & data over channels
- interact with CPython, Jython, PyPy
- py.test distributed testing

execnet example

```
>>> import execnet, os
>>> gw = execnet.makegateway("ssh=codespeak.net")
>>> channel = gw.remote_exec("""
...     import sys, os
...     channel.send((sys.platform, sys.version_info,
os.getpid()))
... """)
>>> platform, version_info, remote_pid = channel.receive()
>>> platform
'linux2'
>>> version_info
(2, 4, 2, 'final', 0)
```

Parsing

- regular expressions
- NLTK
- SimpleParse

NLTK Tokenization

```
>>> from nltk import tokenize
>>> tokenize.word_tokenize("Jacob's presentation")
['Jacob', "'s", 'presentation']
>>> tokenize.wordpunct_tokenize("Jacob's presentation")
['Jacob', "'", 's', 'presentation']
```

nltk.grammar

- CFGs
- Chapter 9 of NLTK Book: http://nltk.googlecode.com/svn/trunk/doc/book/ch09.html

more NLTK

- stemming
- part-of-speech tagging
- chunking
- classification

SimpleParse

- http://simpleparse.sourceforge.net/
- Parser generator
- EBNF grammars
- Based on mxTextTools: http://www.egenix.com/products/python/mxBase/mxTextTools/ (C extensions)

Package Management

- import
- pip
- virtualenv
- mercurial

import

```
import module
from module import function, ClassName
from module import function as f
```

- always make sure package directories have __init__.py

pip

- http://www.pip-installer.org/en/latest/
- easy_install replacement
- install from requirements files

```
$ pip install simplejson
[... progress report ...]
Successfully installed simplejson
```

virtualenv

- http://www.virtualenv.org/en/latest/
- create self-contained python installations
- dependency silos
- works great with pip (same author)

mercurial

- http://mercurial.selenic.com/
- Python based DVCS
- simple & fast
- easy cloning
- works with Bitbucket, Github, Googlecode

Flexible Data Storage

- Redis
- MongoDB

Redis

- in-memory key-value storage server
- most operations O(1)
- lists
- sets
- sorted sets
- hash objects

MongoDB

- memory mapped document storage
- arbitrary document fields
- nested documents
- index on multiple fields
- easier (for programmers) than SQL
- capped collections (good for logging)

Python Performance

- CPU
- RAM

CPU

- probably fast enough if I/O or DB bound
- try PyPy: http://pypy.org/
- use CPython optimized libraries like numpy
- write a CPython extension

RAM

- don't keep references longer than needed
- iterate over data
- aggregate to an optimized DB

import this

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```