

Python

Executable Pseudocode

David Hilley

davidhi@cc.gatech.edu

College of Computing
Georgia Institute of Technology

Roadmap

- Overview
 - General Intro
 - Context / Marketing
- Language
 - Basic Procedural Features (+ unique things)
 - OO Stuff (including advanced features)
 - Standard Library
- Nifty Tools

There are live examples throughout, so follow along on your laptop.

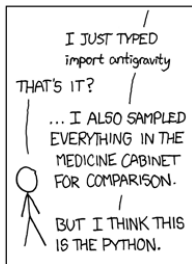
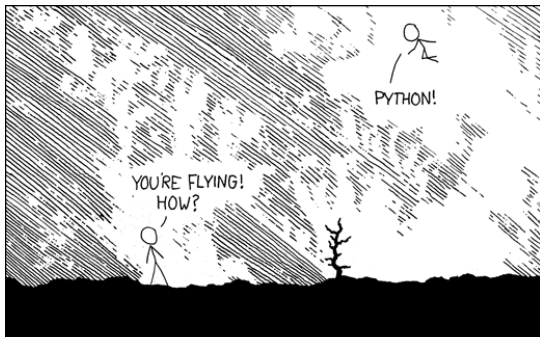
General Intro

- Python: dynamically typed, procedural, object-oriented
 - Large standard library (not external)
 - Linus Torvalds : Linux :: Guido Van Rossum :
-
- Application Domains
 - Shell Scripting / Perl Replacement
 - Rapid Prototyping
 - Application “Glue” (or in-app scripting)
 - Web Applications
 - Introductory Programming

“Python is executable pseudocode. Perl is executable line noise.”

– Old Klingon Proverb

xkcd on Python



Language Basics

- Whitespace is significant – block structure
- Comments are #
- Lists are everywhere: not like LISP lists!
- Note: Python 3000 is somewhat different

```
print "Batteries Included"  
for i in [1, 2, 3]:  
    print i
```

```
_print_ "Hello"
```

```
File "<stdin>", line 1  
    print "Hello"  
    ^
```

IndentationError: unexpected indent

Rookie Mistake

Getting Started

```
print "Hello World!"  
x = 1  
print x, x+1  
x = "Hello World!"  
print x[6:], x[:5]  
print x.lower()  
print x.split()  
y = ["a", "b", 1, 2]  
print y[2], "c" in y, 1 in y  
print ["Z"] + y
```

- Python interactive interpreter: `python`
- Dynamic typing; declaration by assignment
- `dir(obj)` – very useful

Common datatypes

- Numbers/Booleans
- Strings
 - Single or double quotes
 - Triple quoted-string literals
 - Docstrings
 - Strings are immutable
 - No character type
- Collections:
 - Ordered – Lists & Tuples:
[1, 2, 3] or (1, 2, 3)
 - Unordered – Dictionaries & Sets:
{'a':1, 'b':2} or set([1, 2])

Collections

- Sequences
 - List, Tuple, String – can be sliced
 - Buffer, xrange – uncommon
- Unordered Collections
 - Dictionaries: `{'key' : value, }`
literal syntax
 - Can return lists and lazy iterators over dictionary
 - Sets: all common set operations
 - `frozensets` are immutable and therefore hashable

Basic Control Flow

```
def foo(x):  
    if x == "Hello":  
        print "is Hello"  
    elif x == "Bye":  
        print "is Bye"  
        return 2  
    else:  
        print "N/A!"
```

```
>>> foo("Hello")  
is Hello  
>>> x = foo(3)  
N/A!  
>>> print x  
None  
>>> print foo("Bye")  
is Bye  
2
```

- If `if / elif / else`
- `==` comparisons are "intuitive"
- Define functions with `def`
- Use `return` to return values
- `None` is the special "nothing" return value

Looping

```
x = ['a', 'b', 'c']
```

```
for i in x:  
    print i
```

```
for idx, item in enumerate(x):  
    print "%s at %d" % (item, idx)
```

- Iterate over sequences and collections
- Can use `break` and `continue`
- Also a `while` loop
- Use `enumerate`, `sorted`, `zip`, `reversed`
- Accepts an `else` clause, which is called when loop stops naturally (not `break`)

A simple example

```
import sys, re

if len(sys.argv) <= 2:
    print "%s <pattern> <file >" % sys.argv[0]
    sys.exit(2)

f = open(sys.argv[2])
r = re.compile(sys.argv[1])
for line in f:
    if r.search(line):
        print line,
f.close()
```

- A very basic grep(1)

An example with dictionaries

```
# imports & test args...

f = open(sys.argv[1])
d = {} # empty dictionary
for line in f:
    for word in line.split():
        d[word] = d.get(word, 0) + 1
f.close()

it = sorted(d.items(), cmp=lambda x, y: y[1]-x[1])

for (word, freq) in it:
    print freq, word
```

- Simple word histograms

How do I count?

```
for i in range(0, 3):  
    print i
```

```
for i in xrange(0, 3):  
    print i
```

- `range` creates a list
- `xrange` simply keeps track of your place
- Iterators:
 - define `__iter__()` – returns the iterator
 - define `next()` on the iterator
- Generators – even fancier, automatically creates an iterator

Generators

```
def foo(x):  
    while True:  
        x += 1  
        yield x
```

```
>>> foo(1)
```

```
<generator object at 0x2b72dc725cf8>
```

- Like simple co-routines
- Use `yield` to “return” values
- Python 2.5: extended generators (PEP 342)
- Generator Expressions:

```
sum(i*i for i in range(10))  
any(x > 42 for x in range(41, 45))  
all("a" in s for s in ['abba', 'lad', 'amp'])
```

List Comprehensions

```
>>> x = [1, 2, "a", "b"]
>>> [i*2 for i in x]
[2, 4, 'aa', 'bb']
>>> [i for i in x if type(i) == int]
[1, 2]
>>> x = (1, 2, 3, 4)
>>> [(i, j) for i in x if i%3 for j in x if not j%3]
[(1, 3), (2, 3), (4, 3)]
```

- Haskell anyone?
- Handy for implicit iteration over collections
- Can map / filter implicitly; can iterate over multiple collections
- (Python also has map/filter/reduce/lambda for FP party people – see `functools`)

Sort with List Comprehensions

```
def qsort(lst):  
    if len(lst) <= 1:  
        return lst  
    pivot = lst.pop(0)  
    ltList=[y for y in lst if y < pivot]  
    gtList=[y for y in lst if y >= pivot]  
    return qsort(ltList) + [pivot] + qsort(gtList)
```

```
>>> qsort([4, 2, 3, 1, 5])
```

```
[1, 2, 3, 4, 5]
```

- Haskell-like example
- For novelty purposes only: use the list sort method

A more practical example

```
import sys

if len(sys.argv) <= 2:
    print "%s <file> [stopwords]" % sys.argv[0]
    sys.exit(2)

f = open(sys.argv[1])
stopwords = frozenset(sys.argv[2:])
for line in f:
    lst = [w for w in line.split()
           if w not in stopwords]
    print ' '.join(lst)
f.close()
```

- Filter out specific words from a text corpus

Fancy Function Stuff

```
def bar(x, y=2):  
    if x == y:  
        print "same"  
    else:  
        print "not same"
```

```
>>> bar(2)  
same
```

```
>>> bar(y=3, x=2)  
not same  
>>> bar(y=3, 2)  
SyntaxError: non-keyword arg  
>>> t = [2, 2]  
>>> bar(t)  
not same  
>>> bar(*t)  
same
```

- Default arguments; keyword arguments
- Also varargs: last arg is `*names` (tuple)
- Pass sequences exploded to individual arguments with `*`
- Pass dictionaries exploded to keyword arguments with `**`

Python exceptions

```
try:  
    f = open("abc", "r")  
    for l in f:  
        print l  
except IOError:  
    print "Error"  
else:  
    print "Close"  
    f.close()  
finally:  
    print "Finally!"
```

- Use `raise` to raise exceptions
- `except` can handle more than one exception
- Exceptions can also have arguments
- 2.6 – `except <ex> as <varname>`

Modules

```
>>> sys.argv
```

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'sys' is not defined
```

```
>>> import sys
```

```
>>> sys.argv
```

```
['']
```

- `import` imports a module – not like Java `import`, though
- `from <module> import ...` more like Java `import`
- Use `dir` to introspect into a module or type
- Python also supports packages

Future Module

```
from __future__ import with_statement
with open('foo', 'r') as f:
    for line in f:
        print line
```

```
lock = threading.Lock()
with nested (db_trans(db), lock) as (cursor, locked):
    #... do something in transaction with lock ...
```

- Like modules, you import things from `__future__`
- Add new language features, and possibly new syntax
- Must come before other code and imports

Python is OO

- Powerful object model, closer to Modula-3
- Python 2.2 – “new style” unified object model
- Supports:
 - Multiple inheritance (but advises caution)
 - Mixins, metaclasses and decorators
 - Runtime introspection
- Operator definitions (infix, too) and built-ins:
 - Function call: `__call__` to support `obj(...)`
 - Containers: `__getitem__ == obj[key]`
 - Infix operators: `__add__` to support `+`
 - Comparison: `__cmp__`, `__lt__`, `__le__`, etc.
 - Iterators: `__iter__` and `next`
 - Customize attribute access: `__getattr__`

A word on methods

```
>>> x = [1, 2, "a", "b"]
>>> len(x)
4
>>> x.__len__()
4
>>> type(x)
<type 'list'>
>>> type(x).__len__(x)
4
```

```
>>> x = [1, 2, "a", "b"]
>>> del x[3]; x
[1, 2, 'a']
>>> x = x + ['b']; x
[1, 2, "a", "b"]
>>> x.__delitem__(3); x
[1, 2, 'a']
>>> x.append(5); x
[1, 2, 'a', 5]
```

Common things `len`, `del`, etc. are in top-level namespace.

- Objects that behave like built-ins
- Use `dir` function to see methods
- “Open Kimono” language

Basic Class definition

```
class Rectangle:
    sides = 4
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def isSquare(self):
        return self.length == self.width
```

```
>>> r = Rectangle(4, 5)
```

- Constructor is `__init__`
- Explicit `self` for all instance methods
- Attributes are dynamic and public (can use `property`)

Basic Classes Continued

```
class Stack(list, object):  
    def push(self, item):  
        self.append(item)  
  
    @staticmethod  
    def foo():  
        print "Hello world."  
  
    def bar():  
        print "Another static method."  
bar = staticmethod(bar)
```

- Super-classes in parentheses (can extend primitives)
- Static methods using decorators (@) or old style syntax

Decorators

```
@synchronized
@logging
def myfunc(arg1, arg2, ...):
    # ...do something
# decorators are equivalent to ending with:
# myfunc = synchronized(logging(myfunc))
# Nested in that declaration order
```

- Powerful metaprogramming technique
- Write your own: functions that return a new function
- Python Cookbook has a tail call optimization decorator

Longer Class Example

```
from sgmllib import SGMLParser
class URLLister(SGMLParser):
    def reset(self):
        SGMLParser.reset(self)
        self.urls = []

    def start_a(self, attrs):
        self.urls.extend([v for k, v in attrs if k=='href'])

    @staticmethod
    def grab_site(url):
        import urllib
        fd = urllib.urlopen(url)
        parser = URLLister()
        parser.feed(fd.read())
        parser.close()
        fd.close()
        return parser.urls
```

More about OO Python

- “Private” fields/methods with a `__` prefix
- Two types of non-instance methods:
 - `staticmethod` – like Java `static`
 - `classmethod` – like Smalltalk (with `class` argument)
- Metaprogramming:
 - Override `__new__`
 - Set `__metaclass__` attribute
 - Decorators `@` can provide generic method modification
 - Override `__getattr__` and `__setattr__`
 - Use `super` with multiple inheritance
 - Override descriptors: `__get__`, `__set__` and `__delete__`
 - Use the “magic” in the `new` module

Library Tour

- **OS:** `os`, `stat`, `glob`, `shutil`, `popen2`, `posix`, `subprocess`
- **String IO:** `string`, `re`, `difflib`, `pprint`, `getopt`, `optparse`
- **Daemons:** `select`, `socket`, `threading`, `asyncore`
- **Tools:** `unittest`, `test`, `pydoc`, `profile`, `trace`
- **Net:** `urllib2`, `httpplib`, `smtpd`, `cookielib`
- **Formats:** `zlib`, `gzip`, `zipfile`, `bz2`, `tarfile`
- **Crypto:** `hashlib`, `hmac`, `md5`, `sha`
- **XML:** `expat`, `xml`.`{dom, sax, etree}`
- **Persistence:** `pickle`, `dbm`, `gdbm`, `sqlite3`
- **Internals:** `parser`, `symbol`, `tokenize`, `compileall`, `dis`

<http://docs.python.org/lib/lib.html>

Tools & Libraries

- Python Debugger: `pydb`
- Python Documentation Tool: `pydoc`
- `distutils` & `setuptools` (Eggs/EasyInstall)
- Object Relational Mapping: SQLAlchemy, Elixir
- Networking Framework: Twisted Python
- Web Frameworks: Django, Zope, Pylons, TurboGears
- JIT Compiler: PyPy, Psyco
- Numerical Computing: NumPy & SciPy
- Image Manipulation: Python Imaging Library (PIL)
- Graphing/Graphics: Matplotlib & VPython
- Libraries: Boost.Python

Python Evolution

The Future¹ – Python 3000

- Various syntactic changes – `print`
- Strings unicode by default
- `range` & `xrange` merged
- Some libraries merged
- `2to3` tool available

¹Actually available now

Resources/Bibliography

- Python Programming Language Official Website
<http://www.python.org>
- Python Tutorial
<http://docs.python.org/tut/tut.html>
- Python Library Reference
<http://docs.python.org/lib/lib.html>
- Python Reference Manual
<http://docs.python.org/ref/ref.html>
- Python Enhancement Proposals (PEPs)
<http://www.python.org/dev/peps/>
- Python Wiki
<http://wiki.python.org/moin/>

Resources/Bibliography cont.

- **C2 – Python Language**
<http://c2.com/cgi/wiki?PythonLanguage>
- **Python Cookbook**
<http://aspn.activestate.com/ASPN/Cookbook/Python>
- **Dive Into Python**
<http://diveintopython.org/>
- **Thinking in Python**
<http://www.mindview.net/Books/TIPython>
- **Charming Python: Decorators make magic easy**
<http://www-128.ibm.com/developerworks/linux/library/l-cpdecor.html>

Example Sources

- Sort with List Comprehensions:
C2 - Python Samples
- Future Module:
PEP 343: The 'with' statement
- Basic Classes Continued:
C2 - Python Samples
- Decorators:
Charming Python: Decorators make magic easy (Listing 4)
- Longer Example:
Dive Into Python: 8.3. Extracting data from HTML documents