Python
Executable Pseudocode

David Hilley

davidhi@cc.gatech.edu

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Roadmap

- Overview
  - General Intro
  - Marketing
  - Basics
- Language
  - Basic Procedural Features (plus some 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”
  - Web Applications
  - Introductory Programming

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

– Old Klingon Proverb

Marketing Slide

- Who uses Python?
  - Debian, Gentoo and Fedora standard systems
    - Portage
    - anaconda
    - yum
  - Red Hat, Google, NASA
  - `youtube.com`, `lwn.net`
  - BitTorrent (original)
  - GNU Mailman
  - Blender (for scripting)

Language Basics

- Python interactive interpreter: `python`
- Whitespace is significant: specifies block structure
- Comments are #
- Declaration by assignment
- Lists are everywhere: not like a LISP list, though!
- Large standard library

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

Basics

```
>>> x = "a"
>>> x = 3
>>> x
3
>>> "a" + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

- Dynamic typing; declaration by assignment
- Strict type checking
- Booleans, Numbers, Strings, Iterators, Modules, etc.
- Collections: Lists, Tuples, Dictionaries, Sets, Buffers, XRange

Basic datatypes

- Numbers/Booleans
  - `int`, `float`, `long`, `complex`
  - Boolean conditions accept anything; bitwise ops: int/long/bool
  - `None`, `False`, `0` and empty collections/sequences are false.
  - `or` and `and` short-circuit and return their operands
- Strings
  - Single or double quotes
  - Triple quoted-string literals
  - Strings are immutable
  - No character type; single character strings

Collections

- Sequences
  - List, Tuple, String – can be sliced
  - Lists: Mutable; Heterogeneous collection (more like Java vectors)
  - Tuples: Immutable; Heterogeneous, frequently used for multiple return values, or multiple assignment
  - Buffer, XRange – uncommon
- Unordered Collections
  - Dictionaries: `{'key' : value, }` literal syntax
  - Can return lists and lazy iterators over dictionary items
  - Sets: all common set operations
  - `frozensets` are immutable and therefore hashable

Slices

```
>>> x = [1, 2, "a", "b"]
>>> x[:2]
[1, 2]
>>> x[2:]
['a', 'b']
>>> x[-1]
'b'
>>> del x[0:2]; x
['a', 'b']
```

```
>>> x = [1, 2, "a", "b"]
>>> x[:-1]
[1, 2, 'a']
>>> x[-3:-1]
[2, 'a']
>>> x[0:100]
[1, 2, 'a', 'b']
>>> x[2:3] = [3, 4, 5]; x
[1, 2, 3, 4, 5, 'b']
```

- Works on sequences: lists, strings, tuples, etc.
- Negative indices specify from the end of the sequence
- Assignment, `del` and other operations work on slices

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) == type(1)]
[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)

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

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

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`; becomes tuple
- Pass sequences exploded to individual arguments with `*`
- Pass dictionaries exploded to keyword arguments with `**`

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` (or `pass`)
- Accepts an `else` clause, which is called when loop stops naturally (not `break`)
- Also a `while` loop
- Use `enumerate`, `sorted`, `zip`, `reversed`

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 has extended generators – PEP 342
- Generator Expressions: `sum(i*i for i in range(10))`

Python exceptions

```
try:
  f = file("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

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 file('foo', 'r') as f:
    for line in f:
        print line
```

```
lock = threading.Lock()
with nested (db_transaction(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

A simple example

```
import sys, re

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

f = file(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)

Python is OO

- Powerful object model, closer to Modula-3
- Python 2.2 introduced “new style” unified object model
- Supports multiple inheritance (but advises caution)
- Supports mixins, metaclasses and decorators
- Supports runtime introspection
- Operator definitions (infix, too) and interaction like built-ins:
  - Function call: `__call__` to support `obj(...)`
  - Containers: `__getitem__` to support `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]
```

- <item>.<method>(...) is the object method call
- Common things `len`, `del`, etc. are in top-level namespace.
- One can implement collections that behave like built-ins.
- Use `dir` function to see methods.

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

More about OO Python

- Fields/methods can be "private" with a `__` prefix
- Two types of non-instance methods:
  - `staticmethod` – like Java `static` (no `self` argument)
  - `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

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 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()
        for url in parser.urls:
            print url
```

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`, `httplib`, `smtpd`, `cookielib`, `mimelib`
- Formats: `zlib`, `gzip`, `zipfile`, `bz2`, `tarfile`, `uu`, `binhex`
- Crypto: `hashlib`, `hmac`, `md5`, `sha`
- XML: `expat`, `xml.dom`, `xml.sax`, `xml.etree`
- Persistence: `pickle`, `dbm`, `gdbm`, `sqlite3`, `bsddb`, `dumbdbm`
- Internals: `parser`, `symbol`, `tokenize`, `compileall`, `dis`

`http://docs.python.org/lib/lib.html`

Tools & Libraries

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

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