Learning the **Pythonic** Way

Athula Balachandran
Wolf Richter
The Man, The Legend: Zed Shaw

So, he made a web server too. It's called mongrel2.

Oh, and Learning Python the Hard Way [LPTHW].

and... Programming, Motherfuckers...

Need I say more?
Why Python?

• My job is to convince you that:
  • Python is incredibly easy to program in
  • Python “comes with batteries”
  • Python enables rapid prototyping
  • All your pseudo-code are belong to Python

• Practicality?
  • Systems scripting language of choice
  • Alongside Perl and Ruby; OK, fine
Let's do this.
One at a time.

All your pseudo-code are belong to Python
Wikipedia: What is Quicksort?

Simple version

In simple pseudocode, the algorithm might be expressed as this:

```plaintext
function quicksort('array')
    create empty lists 'less' and 'greater'
    if length('array') ≤ 1
        return 'array'  // an array of zero or one elements is already sorted
    select and remove a pivot value 'pivot' from 'array'
    for each 'x' in 'array'
        if 'x' ≤ 'pivot' then append 'x' to 'less'
        else append 'x' to 'greater'
    return concatenate(quicksort('less'), 'pivot', quicksort('greater'))
```
def quicksort(array):
    less = []; greater = []
    if len(array) <= 1:
        return array
    pivot = array.pop()
    for x in array:
        if x <= pivot: less.append(x)
        else: greater.append(x)
    return quicksort(less)+[pivot]+quicksort(greater)
Really? Yes!

```python
>>> quicksort([9,8,4,5,32,64,2,1,0,10,19,27])
[0, 1, 2, 4, 5, 8, 9, 10, 19, 27, 32, 64]
```
Python “comes with batteries”
There's an a library for that

- `import httpplib`
  - HTTP protocol client
  - Test your web servers!
  - Also: ftplib, poplib, imaplib, nntplib, smtplib...

- `import os and import sys`
  - misc. OS interfaces, and system-specific parameters and functions

- `import random`
  - Generate pseudo-random numbers
  - Sampling, shuffling, etc. – good for testing

- `import socket`
  - First test script used this for PJ1CP1
  - also SimpleHTTPServer, BaseHTTPServer...

- `import fileinput → fileinput.input() → iterable`
  - Reads lines from stdin, files listed on command line, etc.
Python enables rapid prototyping
Give me a dynamic web app NOW

```python
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```
Python is incredibly easy to program in
Use the Interpreter

- Code and experiment interactively
- Use help()
- Explore functionality and ideas
- Then code in your main editor
Just one thing

- **Whitespace matters**
- Defines blocks → C-world thinks `{ }`
- Use spaces
- 4 spaces per indentation level
- spaces > tab → just be consistent
- Really though, generally aids readability
- Set your editor preferences ahead of time
The Colon

• Required for if/for/while/with/def/class statements
• Enhances readability
• Has English meaning
• Helps auto-indenting editors
• From the Python Design FAQ
Starting a Script and Comments

• Start with:
  
  `#!/usr/bin/env python`

• Then you can `chmod +x script.py`

• The `#!` is a special character combination

• Tells the OS how to execute a file

• Comments start with a `#`

• They go to the end of the line
Math – Business as Usual

- `import math` → extra math functions
- Convert between: `int()` and `float()`
- Convert to string: `str()`

```python
>>> 2 * 8
16
>>> 4 / 3
1
>>> 4 / 3.
1.3333333333333333
>>> 2 ** 4
16
>>> 18 % 3
0
>>> 18 % 4
2
>>> float(4) / 3
1.3333333333333333
>>> float(4 / 3)
1.0
>>> int(4 / 3.)
1
>>> str(2**4)
'16'
```
Danger: Division from the Future

- Python 3 is coming...and __future__
- Yes, basic math changes...

```python
>>> from __future__ import division
>>> 6 / 7 # defaults to float
0.8571428571428571
>>> 6 // 7 # “floor” division
0
>>> 6 // 7.
0.0
>>> 6.5 // 7
0.0
>>> 7. // 7
1.0
>>> 7 // 7
1
```
Danger: Division from the Future

- Always read the top of a script first

SyntaxError: from __future__ imports must occur at the beginning of the file
Booleans

- **True/False** – actual values

- **Logical Operators**
  - `and` – not `&&` (although `&` is set and bit and)
  - `or` – not `||` (although `|` is set and bit or)
  - `not` – not `~` (although `~` is bit not)

- As expected...

```python
>>> True and True
True
>>> True or False
True
>>> not True
False
>>> not False
True
```
Lists

- Think arrays of arbitrary objects—can mix and match type!

- Sorting

  \[
  \text{sorted}(x) \quad \text{– returns a new list}
  \]

  \[
  \text{x.sort()} \quad \text{– sorts in place}
  \]

```python
>>> x = [3, 5, 7, 2, 8, 1, 4, 9, 6]
>>> sorted(x)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
[3, 5, 7, 2, 8, 1, 4, 9, 6]
>>> x.sort()
>>> x
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```
Lists

- **Comprehensions** – construct lists dynamically; they nest too!
- **Functional Programmers**: think `map()`

```python
global evens

evens = [x*2 for x in xrange(10)]

evens

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

evens = [x for x in xrange(10) if x % 2 == 0]

evens

[0, 2, 4, 6, 8]
```
Lists

- **Slicing** – cutting up lists and other iterables (strings etc.)

```python
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[:] # copy x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[-1] # last position in list
9
>>> x[0:3] # 0\textsuperscript{th} through 2\textsuperscript{nd} positions
[0, 1, 2]
>>> x[1:] # copy starting at 1\textsuperscript{st} position
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[:4] # copy up to 3\textsuperscript{rd} position
[0, 1, 2, 3]
```
Lists

- **Stacks and Queues** – LIFO and FIFO – lists are just so versatile

```
>>> x = []
>>> x.append(0)
>>> x.append(1)
>>> x.pop()
1
>>> x.append(2)
>>> x.pop()
2
>>> x = []
>>> x.append(0)
>>> x.append(1)
>>> x.pop()
0
>>> x.append(2)
>>> x.pop()
1
```
Dictionaries

- **Key-Value Storage** – arbitrary keys, arbitrary values
- **del** – remove object from dictionary or list

```python
>>> d = {'a': 0, 'b': 1, 2: 0}
>>> d[2]
0
>>> d['a']
0
>>> d['b']
1
>>> del d['b']
>>> d
dict({'a': 0, 2: 0})
```
Dictionaries

- `len()` – get length of dictionary or list
- `keys()`, `values()` – get lists of these
- `key in d` – membership in dictionary or list

```python
>>> d = {'a': 0, 'b': 1, 2: 0}
>>> len(d)
3
>>> d.keys() # note, no ordering
['a', 2, 'b']
>>> d.values() # ordering...
[0, 0, 1]
>>> 'a' in d
True
>>> 'x' in d
False
```
Tuples and Strings = Sequences

- Tuples are just values separated by ' , '
- They are both (strings too) immutable
- Otherwise, they behave like lists

>>> t = ('x', 'y')
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>" , line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t[0]
'x'
>>> t[1]
'y'
Tuples: Packing and Unpacking

• Quick and easy way to name values

```python
>>> position = 249,576
>>> x, y = position
>>> x
249
>>> y
576
```
Sets: Creating

s1 = set([1, 1, 1, 2, 3, 4, 5])  No duplicates
>>> s1
set([1, 2, 3, 4, 5])
>>> s1.add(4)
>>> s1.add(7)
>>> s1
set([1, 2, 3, 4, 5, 7])
>>> sorted(s1)
[1, 2, 3, 4, 5, 7]
>>> 6 in s1
False
>>> 6 not in s1
True

Adding elements

You can sort sets!?  Returns a list

Test element membership too...
Sets: Manipulating

```python
>>> s1 = set([1, 1, 1, 2, 3, 4, 5])
>>> s2 = set([7, 4, 64, 62, 5, 1])
>>> s1 & s2
set([1, 4, 5])
>>> s1 | s2
set([64, 1, 2, 3, 4, 5, 7, 62])
>>> s1 ^ s2
set([64, 2, 3, 7, 62])
>>> s1 - s2
set([2, 3])
>>> s2 - s1
set([64, 62, 7])
```

Regular set operations just work
Strings

- **Strip** – remove surrounding white space
  ```python
  >>> ' this is a test '.strip()
  'this is a test'
  ```

- **Length** – same as lists: `len()`

- **Slicing** – same as lists/other sequences

- **Formatted** – C printf-style inline
  ```python
  >>> '%d\t%d\t%s\n' % (6, 7, 'hello')
  '6\t7\thello\n'
  ```
Strings: Me, Myself, and Irene

- So there are several types of strings...
- Single- or double-quotes accepted
- Triple and you got something special
  - Keeps newlines and whitespace generally

```python
>>> 'string'
'string'
>>> "string"
'string'
>>> '''test
... ... ... yeah
... ... ...
'test\n\n\tyeah
'```

Raw Strings

• Maintain escapes inside them
• That is, the ' \ ' stays put

```python
>>> r'This string\t has escapes\n\n.'
'This string\t has escapes\n\n.'

>>> 'This string\t won\'t have escapes\n\n.'
'This string\t won\'t have escapes\n\n.'
```
Looping: In Theory

- **for** – always a foreach
  - Use `enumerate` to get more C-stylish with an i
- **while** – similar to C while
- **range, xrange** – create ranges to iterate on
  - `range` – actually creates a list in memory
  - `xrange` – does not create a list in memory
  - **Just use** `xrange`
- **break, continue** – similar to C
Looping: Applied

Tricky: Modifying lists etc. while looping. Generally work on copies.

```python
>>> for x in xrange(5):
...     print x
...
0 1 2 3 4

>>> while (x > 0):
...     print x
...     x -= 1
...
4 3 2 1

>>> for i,x in enumerate(['test', '15-441', 'test']):
...     print i,x
...
0 test
1 15-441
2 test
```
Branching

- **if** → as expected
- **elif** → else if construct
- **else** → as expected

```python
>>> if '' or None or 0 or [] or set([]) or ():
    ...     pass
    ... else:
    ...     print 'huh, they all appear as false.'
    ...
    huh, they all appear as false.
>>> if False:
    ...     pass
    ...     pass
    ... elif True:
    ...     print 'else if!'
    ...
else if!
```
Defining Functions

- The magic keyword: `def`
- **Formal parameters** – as normal C args
- *arguments* – contains non-formal args
- **kwargs** – contains a dictionary with non-formal keyword parameters
- Be thinking: `varargs from C`
- Parameters can have defaults
- Parameters can be named
One Function to Rule Them All

```python
>>> def example(x, y=None, *args, **keywords):
...     print x, '\t', y, '\t',
...     print args, '\t', keywords
...
>>> example(1, 2, 3, 4, test='test', test2='test2')
1 2 (3, 4) {'test': 'test', 'test2': 'test2'}
```
The Power of Passing

- Rapidly create a skeleton/think abstractly
- `pass` – a **noop statement** – it does nothing

```python
def log_error(message):
    pass

def open_binary(path):
    pass

def close_binary(path):
    pass

def new_client(client_socket):
    pass

while True:
    pass

for x in xrange(10):
    pass
```
None

- None is kind of like NULL
- That's pretty much it.
- You can use it as a guard value
Classes: The What

- The magic keyword: `class`
- Another magic keyword: `self`
- `self` refers to the current object
- `self` stores instance variables etc.
- `self` is always an argument to an instance method
Classes: The How

```python
>>> class myclass(object):
...     def __init__(self):
...         self.x = 0
...     def increment(self):
...         self.x += 1
...     def get(self):
...         return self.x

>>> instance = myclass()
>>> instance.get()
0
>>> instance.increment()
>>> instance.get()
1
```
Iterators and Generators

- The power to create your own `xrange`
- **Classes** with `next()` and `__iter__()` methods
  - Then their instantiated objects may be used as iterator objects
- **Functions** can use the `yield` keyword
  - State is retained for successive yields
Iterator Example

```python
>>> class myiter:
...     def __iter__(self):
...         return self
...     def next(self):
...         raise StopIteration
...     for x in myiter():
...         print x
```
Yield Example

```python
>>> def myiter():
...     for x in [1, 2, 3, 4, 5]:
...         yield x
...

>>> for x in myiter():
...     print x
...
1
2
3
4
5
```
Exceptions: Except and Finally

- **try...except**
  - Often enough for most tasks
  - Multiple exceptions in one except
  - Or one except per exception type

- **try...except...finally**
  - finally executed on the way out, cleanup handler
  - Also on return, break, continue
Exceptions: In Practice

```python
>>> try:
...    open('test.txt', 'r')
... except IOError:
...    print 'error'
... finally:
...    print 'code that is guaranteed to run'

error

code that is guaranteed to run
```
Exceptions: Making Them...

```python
>>> raise ValueError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError

raise special classes you have created with meaningful names.
```
with: Better IO in Practice

- with keyword uses `__enter__` and `__exit__`
- `__exit__` executes no matter what
- Only lives for a block
- Better semantics
  - Definitely closing file descriptors etc.
- Replaces standard `try/finally` blocks
- Uhmmm (Python < 2.5):
  - `from __future__ import with_statement`
with: Better IO in Practice

```python
>>> with open('test.txt', 'r') as f:
...     f.read()
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'test.txt'
```
Writing Tests

- `import doctest`
  - Dynamically finds tests in your documentation!
  - Check examples in `docstrings`
- `import unittest`
  - Test whole programs or APIs or other programs
Writing Documentation

- PEP 257 -- Docstring Conventions
  http://www.python.org/dev/peps/pep-0257/
- Sphinx – Python Documentation Generator
  http://sphinx.pocoo.org/
Pythonic Style

PEP 8 -- Style Guide for Python Code
http://www.python.org/dev/peps/pep-0008/
@zedshaw LPTHW Reference

• We did lessons
  • 1-5, 10, 13, 18, 23, 29, 32, 40, 47
• Bonus: Up to 10 points towards HW's/Projects
• What should you do:
  • Finish the rest of LPTHW/fill in the gaps
  • Type in all the Python code yourself
  • Follow instructions!
• Place all code into your Project 1 repo
  – /scripts/ subfolder
  – Email us your feelings/experience
How do I get LPTHW?

• Free online
  
  http://learnpythonthehardway.org/book/

• Zed Shaw provided PDF
  
  CMU IP-only site
  

• How long does it take?
  
  • ~1-2 days for an experienced programmer
More Python References

Python Tutorial
http://docs.python.org/tutorial/

Super Useful Python Documentation
http://docs.python.org/library/

Python Interpreter

```
python
>>> help(x)
```
Shameless plug.

www.phototags.org

Help Wolf's research.
GitHub:

Git it, got it, good.

```
git clone git://github.com/theonewolf/15-441-Recitation-Sessions.git
```