

Python for EECS Majors

Scott Wolchok
`scott@wolchok.org`

Welcome!

- Me: annoyed with “all code in C++”
- You: taken 280, 281/381 strongly advised
- Goal: your next program is in Python
- Why: C++ is not efficient for programmers; students famously hate reading docs

Disclaimers

- Not University-run: this isn't my job
 - Not a class: you get what you put in
 - Not done: you're guinea pigs!
 - Not enough: you need to read/write code
-
- I did give this talk in EECS 398

Following Along

- Install Python 2.6 from <http://python.org/>
- ssh `username@login.engin.umich.edu` and run `python2.5`
- Don't use Python 3.x; it's not ready (poor library support)

Survey

- Class year?
- Programming languages other than C++?
- Special interests? (web development?)

Why Python?

- Dynamic typing: **no type declarations**

C++: **Person** *p = **new** Person();

Python: p = Person()

- Strong typing: **no implicit conversions**

'3' + 5 is an error, not '8'

Why Python (cont.)

- Very high-level: **shorter code**

```
[x*x for x in range(10)]
```

```
knights = {'Gallahad': 'pure',  
           'Robin': 'brave'}
```

```
for k in knights:
```

```
    print 'Sir %s the %s' % (k, knights[k])
```

Why Python? (cont. 2)

- Interactive interpreter
- General-purpose: **standard** libraries for networking, Web, email, testing, profiling, and more (see docs.python.org)
- No memory leaks (mostly)
- Readability (contrast with Perl)
- Runs on .NET and Java Virtual Machine
- Google and MIT use it :)

Why not Python?

- It's slow, but:
 - Speed usually doesn't matter
 - You can make it fast (more later)
 - Unladen Swallow coming Real Soon Now
- No customizable syntax (unlike Perl, Ruby, Lisp, etc.)
- Style/convention disagreements
- It's not JavaScript

The Basics

- Create variables by assigning to them:

```
x = 3
```

- Arithmetic operators are the same as C++:

```
y = 2
```

```
z = x + y      # 5
```

```
s = 'Hello ' + 'world'      # 'Hello world'
```

Strings and Lists

- Strings can't be modified (immutable)

```
s[3] = 'q'      # error!
```

- Lists – mutable sequence of any type

```
lst = [1,2,3]
```

```
lst[-1] = 'spam'      # fine, l is [1,2,'spam']
```

```
print lst[1:] + lst[:1]      # [2,'spam'] + [1]
```

```
                             # = [2, 'spam', 1]
```

```
print len(lst)           # 3
```

- Lists resemble `std::vector<Object *>`

Sequence operations

- Indexing: $s[0]$ – “the 0th element of s ”
- Slicing: $s[1:3]$ – “start at 1, include up to 3”
- Membership testing – x **in** s , x **not in** s
- Concatenation – $s + t$
- Multiplication – $s * n$ (concat n copies of s)
- Length – **len**(s)
- **min**(s), **max**(s)

Mapping Types -- dicts

- `d = {'one': 1, 'two': 2}`
- Map keys to values, like C++'s `std::map`
- **NOT** sorted; uses hashing internally
- Keys can be any hashable (immutable) type, values can be any type

Mapping type operations

- Indexing – `d[key]`
- Removal – `del d[key]`
- Membership testing – `key in d`, `k not in d`
- `d.clear()`
- Index w/default – `d.get(key, default=None)`
- Lists of elts – `d.keys()`, `d.values()`, `d.items()`
- Iteration: `for key in d: ...`
- Size: `len(d)`

Control Flow Summary

- **if, elif, else:** as in C++
- **for x in s:** iterate over a sequence
- **while:** as in C++
- No **do..while**
- No **switch**
- No assignment in loop conditions, as in
while line = fyl.readline(): *# Error!*

Control flow

- **if...elif...else**

```
if square_type == 'RAILROAD':
```

```
    ...
```

```
elif square_type == 'UTILITY':
```

```
    ...
```

```
else:
```

```
    print 'invalid square type'
```

- **elif** avoids excessive indentation

More control flow

- **while** should be familiar
- The **for** loop iterates over a sequence.
for num **in** range(99):
 print (99-num), 'bottles of beer on the wall'
- **range** returns a list from zero to its argument, exclusive (range(3) = [0,1,2])

Functions

- The **def** keyword denotes functions:

```
sofar = [1, 1]
```

```
def fib(n):
```

```
    """Return the nth Fibonacci number."""
```

```
    assert n >= 0, "no negative fib"
```

```
    while len(sofar) < n+1:
```

```
        sofar.append(sofar[-2] + sofar[-1])
```

```
    return sofar[n]
```

C++ version

```
int fib(int n) {  
    static vector<int> sofar; // assume initied.  
    while (sofar.size() < n+1) {  
        int last = *sofar.rbegin();  
        int prev = *(sofar.rbegin()+1);  
        sofar.push_back(prev + last);  
    }  
    return sofar[n];  
}
```

- What a pain! (and what about fib(1000)?)

Default arguments

- Very similar to C++

```
def search_web(terms, site='google.com'):
```

```
...
```

- Default args go after non-default args

Keyword Arguments

- **def** f(a, b=2,c=7):
...
f(a=10, c=5)
- Override default args selectively
- Make the call site more clear
- **def** g(x, y, z, **kwargs):
 print kwargs['w']
g(1,2,3, w=7)

File I/O

- **open(filename, mode)** is like C's **fopen()**
- **f = open('/tmp/myfile', 'r')** # Open for read
- **f.read()** – return entire file as string
- **f.readline()** – return line, leaving the '\n'
- **f.readlines()** – return list of lines in file
- **for line in f** – loop over lines in file
- **f.close()** – close file
- **f.write(s)** – write string s to file

Let's DO something!

- Will cover classes, modules, standard library later
- Next up: case studies!

Poor man's wget

```
#!/usr/bin/env python
```

```
import sys
```

```
import urllib2
```

```
if __name__ == '__main__':
```

```
    print urllib2.urlopen(sys.argv[1]).read()
```

Unpack Flash image

- Working on embedded system, have images of the Flash memory
- Stored as blocks with 2 KB of “main” area followed by 64 bytes of “spare” area
- Want to separate main and spare
- **struct** FlashChunk {
 char main[2048];
 char spare[64];
};

```
#!/usr/bin/env python
```

```
import struct
```

```
import sys
```

```
# Input is 2048-byte "string" followed by
```

```
# 64-byte "string"
```

```
st = struct.Struct('=2048s64s')
```

```
if __name__ == '__main__':
```

```
    if len(sys.argv) != 2:
```

```
        print 'Usage: %s img' % sys.argv[0]
```

```
        sys.exit(1)
```

```
img = open(sys.argv[1], 'rb')
main = open(sys.argv[1] + '.main', 'wb')
spare = open(sys.argv[1] + '.spare', 'wb')
```

```
chunk = img.read(st.size)
while chunk:
    ms, ss = st.unpack(chunk)
    main.write(ms)
    spare.write(ss)
    chunk = img.read(st.size)
```

“Txt spk”

- Make text fit in an SMS message

REPLACEMENTS = {

 'back': 'bak',

 'please': 'plz',

 ...

}

```
def replace_words(words):  
    ret = []  
    for word in words:  
        lword = word.lower()  
        if lword in REPLACEMENTS:  
            word = REPLACEMENTS[lword]  
        ret.append(word)  
    return ret
```

```
def replace_words(words):
```

```
    ret = []
```

```
    for word in words:
```

```
        lword = word.lower()
```

```
        if lword in REPLACEMENTS:
```

```
            word = REPLACEMENTS[lword]
```

```
        ret.append(word)
```

```
    return ret
```

```
def replace_words(words):
```

```
    ret = []
```

```
    for word in words:
```

```
        lword = word.lower()
```

```
        word = REPLACEMENTS.get(lword, word)
```

```
        ret.append(word)
```

```
    return ret
```

```
def replace_words(words):  
    ret = []  
    for word in words:  
        lword = word.lower()  
        word = REPLACEMENTS.get(lword, word)  
        ret.append(word)  
    return ret
```

```
def shorter_replace_words(words):  
    return [REPLACEMENTS.get(word.lower(),  
                               word)  
            for word in words]
```

```
def aggressive(words):  
    ret = []  
    for word in words:  
        if word != 'a' and word != 'i':  
            word = word.translate(None,  
                                   'aeiouAEIOU')  
        ret.append(word)  
return ret
```

```
def make_it_fit(s, chars):  
    words = s.split()  
    if len(s) > chars:  
        words = replace_words(words)  
    return ' '.join(words)
```

This function has a bug. What is it?

Pythonic Code

- Careful! Don't just code C++ in Python
- Use the standard types and helpful syntax whenever possible and straightforward
- This means you'll be referring to <http://docs.python.org> a lot (hint, hint)
- Try `import this` in the interpreter

Getting Help

- The **help** function displays docstrings
- The **dir** function lists attributes & methods
- <http://docs.python.org/lib> is best
- Google is your friend – <http://imgtfy.com/>

Further Reading

- Python Tutorial
<http://docs.python.org/tutorial>
- PEP 8: Style Guide for Python Code
<http://www.python.org/dev/peps/pep-0008/>
- PEP 20: The Zen of Python (short!)
<http://www.python.org/dev/peps/pep-0020/>
- Dive Into Python
<http://diveintopython.org/>

Next Talk: TBD

- How to write “real” Python programs
- Will discuss modules, classes, useful standard library modules
- Please send feedback!
- Website: <http://scott.wolchok.org/pytalks/>

Other Talks/Topics?

- Simple web development with CherryPy
- Python: the bad news (improving perf)
- Using C/C++ libraries from Python
- Easy XML parsing with ElementTree
- What's new in Python 3.0
- Packaging Python programs with distutils

Pitfalls for C++ Programmers

- Python **always** passes by reference
- $n += 1$ means $n = n + 1$
- “Take the object referenced by n , make a **new** int object, and make n reference that new object.”

More Pitfalls

- **def** append_1_to_list(L):
 L.append(1)
lst = [1,2,3]
append_1_to_list(lst)
print lst

- **def clear_list(L):**
 L = []
lst = [1, 2, 3]
clear_list(lst)
print lst

- **def clear_list(L):**

```
    L = []
```

```
lst = [1, 2, 3]
```

```
clear_list(lst)
```

```
print lst
```

- L refers to the same list that lst does, but clear_list makes L refer to something else!
- **No change** to the list lst refers to!

The right way

- **def clear_list(L):**
 L.clear() # or L[:] = []

Tidbits

- Unpacking: one, two, three = (1, 2, 3)
- Swap idiom: a, b = b, a
- **for** (idx, item) in **enumerate**(s):
- Chained comparisons: **if** 1 < x < 8:

Extra Slides

Nested Lists

- Lists can contain anything!

```
k = [1, 2]
```

```
lst = [3, k, 9] # prints as [3, [1, 2], 9]
```

```
lst[1].append(3)
```

```
print lst # [3, [1, 2, 3], 9]
```

```
print k # [1, 2, 3] !
```

- `lst[1]` and `k` are the same object (variables have pointer semantics)