CS 3101.3 Python: Lecture 6

February 26, 2009

Project Proposals

- We have a range of great ideas
  - Craigslist mining and statistical analysis
  - Genetic algorithms
  - Financial engineering
  - Hardware interface to lab equipment
  - Music crawler
  - Network subterfuge
  - Prime number sieves
  - ...

Final Projects

- Due Thursday March 5th 6pm by email to joshua@cs.columbia.edu
- Submit a single zipped archive containing source code and the following:
    - README.txt - a short plaintext file detailing any libraries your code depends on, where to download them, and instructions for running your code
    - PROJECT.pdf - a pdf writeup describing your project, results, lessoned learned, etc.
- Well documented internally - especially if you anticipate trouble areas
- Neatly packaged, should work out of the box - all paths must be relative

Demos

- A couple of projects depend on external resources - databases, web servers, lab equipment
- Submit your code as normal
- Send me an email to schedule a demo

Office Hours

I'll be camping in the TA room:

Monday March 2nd: 6pm-8pm

Tuesday March 3rd: 6pm-8pm

and

by email appointment
(joshua@cs.columbia.edu)

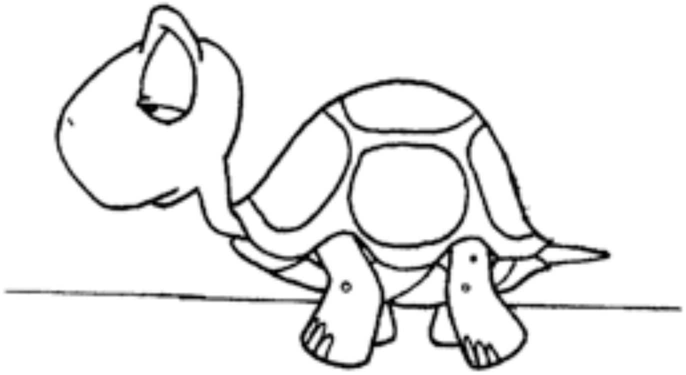
CVN

- First lecture is Wednesday March 4th, 4pm
- Hopefully no one needs to sit that one :)
- Regular expressions, however, are coming up

Agenda

- Performance:
  - Optimization: when and why
  - Profiling, identifying hotspots, timing execution
  - Memoizing function return values
- Integrating Python with C / C++
  - Distutils and SWIG, Boost
- Common questions
  - Sorting a dictionary
  - Documentation
  - The None keyword
- Scheduling
- Sending Email
- Threads 101

Performance Basics



- Data Structures and Algorithms
- Focus on the big picture
- Disk, network I/O
- Almost always when I see scripts performing poorly it’s one of these
- Batch operations - cache data in memory, read and write en masse
- Code generally spends 90% of it’s time in 10% of its context

Optimizing performance

- In scripting we are much more interested in
  - correctness
  - readability
  - efficiency w.r.t. to development time
- When you need to be fast with Python, you have options
  - Identifying hotspots with the profile module
  - Small scale optimization with timeit
  - Rewriting libraries in C++

Starting small with timeit

- A good introduction to benchmarking
- Useful for small scale optimizations, i.e., measuring the performance of a single routine
- Covers many common gotchas - i.e., setup code, multiple runs
- Quick question: Say you benchmark a function with identical inputs several times. The running times are 100ms, 90ms, and 110ms respectively.
  - Which time would you report as the most accurate estimate of performance?

From the command line

- ./python -mtimeit -s ‘setup statements(s)’ ‘bechmark statements’
- josh$ python -mtimeit -s 'x=[5,4,3]*100' 'x.sort()'
  - **100000** loops, best of 3: 13 **usec** per loop
- josh$ python -mtimeit -s 'x=[5,4,3]*100' 'sorted(x)'
  - **10000** loops, best of 3: 88 **usec** per loop
- Notice timeit automatically adjusts the number of loops run. Cool right?
- Difference between sorted and sort?

A classic (newly defunct?) example of a common pitfall: building a string

```
def slow():          O(n^2)
    big = ''
    small = 'foo'
    for i in range(10000):
        big+= small
    return big

def fast():           O(n)
    big = []
    small = 'foo'
    for i in range(10000):
        big.append(aDonut)
    return ''.join(big)

if __name__ == '__main__':
    from timeit import Timer
    t1 = Timer('fast()', 'from __main__ import fast')
    t2 = Timer('slow()', 'from __main__ import slow')
    print t1.timeit(number=100) / t2.timeit(number=100)
```

1.54   Notice the unexpected results (Using Python 2.5)?

Profiling

- Typically code spends 90% of its time in 10% of it's context
- Don't guess where - it's often not obvious
- Pattern: use the profile module with standardized inputs to analyze code, then analyze the data with pstats
- Profiling is not just for algorithms intensive work
  - worth considering when working with large data sets
  - a must if you're sending code out into the world

Profiling

- Calibration: if you're doing serious work you'll want to calibrate profile to your machine - takes care of the overhead
  - See profile.calibrate, the python doc, or Python in a nutshell p.480
- You can run profile directly and see statistical output, or write to disk with an optional filename=... named parameter.
  - Consolidate several runs and analyze with pstats

Profiling

```python
def recFib(n):
    if n == 0 or n == 1:  # base case
        return n
    else:
        return recFib(n-1 ) + recFib(n-2)

def iterfib(n):
  sum,a,b = 0,1,1
  if n<=2: return 1
  for i in range(3,n+1):
    sum=a+b
    a=b
    b=sum
  return sum

def go():
    print recFib(20)
    print iterfib(20)

if __name__ == '__main__':
    import profile
    profile.run('go()')
```

$$F_n = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n \sqrt{5}}.$$

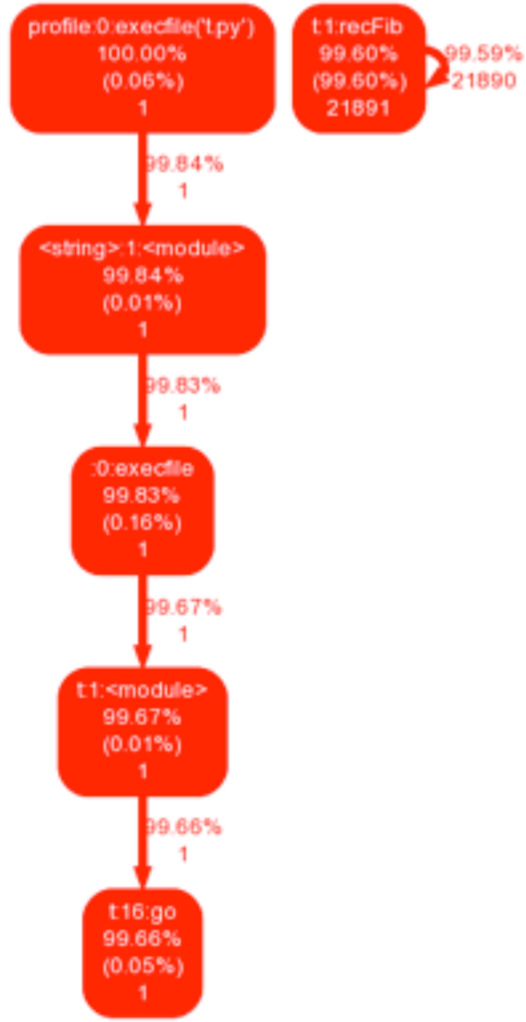
21897 function calls (7 primitive calls) in 0.312 CPU seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	:0(range)
1	0.001	0.001	0.001	0.001	:0(setprofile)
1	0.000	0.000	0.311	0.311	<string>:1(<module>)
1	0.000	0.000	0.312	0.312	profile:0(go())
0	0.000		0.000		profile:0(profiler)
21891/1	0.311	0.000	0.311	0.311	t.py:1(recFib)
1	0.000	0.000	0.311	0.311	t.py:16(go)
1	0.000	0.000	0.000	0.000	t.py:7(iterfib)

Visualizing results

- Generating call graphs
- References:
- http://www.graphviz.org/
- http://code.google.com/p/jrfonseca/wiki/Gprof2Dot
- http://docs.python.org/library/profile.html



python -m profile -o output.pstats
python gprof2dot.py -f pstats output.pstats | dot -Tpng -o output.png

Memonization using function decorators

- Idea: cache a functions return results in a dictionary, keyed by the arguments that produced that value
    - One student used this technique on homework 2
- Worth understanding - useful for optimizing recursive functions, server side code

Memoizing a recursive function

```
#cache the return values of a fn
def memoize(cache=None):
    if cache is None: cache = {}
    def decorator(function):
        def decorated(*args):
            if args not in cache:
                cache[args] = function(*args)
            return cache[args]
        return decorated
    return decorator
```

```
@memoize({})
def memFib(n):
    if n < 2: return 1
    return memFib(n-1) +
memFib(n-2)

def fib(n):
    if n < 2: return 1
    return fib(n-1) + fib(n-2)

if __name__ == '__main__':
    import profile
    profile.run('memFib(20)')
    profile.run('fib(20)')
```

63 function calls (5 primitive calls) in 0.010 CPU seconds
21/1 0.000 0.000 0.001 0.001 t.py:11(memFib)
39/1 0.000 0.000 0.001 0.001 t.py:4(decorated)

21894 function calls (4 primitive calls) in 0.302 CPU seconds
21891/1 0.301 0.000 0.301 0.301 t.py:16(fib)

Data structure internals

- Python optimizes the common case heavily - make use of that
- In general you can expect excellent performance when using the built in types appropriately
- Caution when not - you need a basic underlying knowledge to sense where performance might degrade

Lists

- List operations:
  - internally implemented as vectors
  - Chaining two lists together O(len(a) + len(b)
  - Accessing or rebinding any item: O(1)
  - Len: O(1)
  - Slicing, O(M)
  - Rebinding with segments of different length
    - cheap when appending to the tail of the list, if you need FIFO operations on large lists - see collections.deque

Deques

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print elem.upper()
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'
```

source: http://docs.python.org/library/collections.html

Strings

- String operations
  - Most methods are O(N) where N is the string length, len is O(1)
  - Fastest way to produce a copy of the string with transliterations / removal is the string's translate method

```
>>> 'read this short text'.translate(None, 'aeiou')
'rd ths shrt txt'
```

source: http://docs.python.org/library/stdtypes.html

Dictionaries and sets

- Internally hash tables
  - one of the more highly optimized implementations around
- Accessing, rebinding, adding, removing: generally 0(1)
- Iterkeys vs. keys
  - Methods: keys, values, items are O(n)
  - Methods: iterkeys, iteritems, itervalues are O(1)
  - Iterkeys() return an element at a time, keys() returns a list
  - consider the different memory charactistics
- Gotchas: testing if a value is in a dictionary
  - Never use if x in d.keys() - that's O(n): instead use: if x in d
- Sets are similar

Sort

- Using operator 'in' is the natural tool for checking membership
- O(1) for dictionaries and sets
- O(n) for sequences (lists, strings, or tuples)
- If you find yourself performing many lookups on a sequence, consider restructuring with a dictionary
- Alternatively, it may be worth your time to maintain a sorted copy
- Internally, (as of 2.4) merge-sort: stable (equivalent items retain their relative position)
  - Close to 1200 lines of C code, handles many common cases (already sorted lists, reverse sorted, mostly sorted aside from a few random elements, the input is the concatenation of two already sorted sequences, and onward)
  - Performance drops off fast when using custom comparators - best bet to use the built in types

Common question
Sorting a dictionary by keys and values

```
d = {'homer' :350, 'marge': 140, 'bart': 80, 'lisa': 70, 'maggie': 6}

keys = d.keys()
keys.sort()
print [(key, d[key]) for key in keys]
[('bart', 80), ('homer', 350), ('lisa', 70), ('maggie', 6), ('marge', 140)]

# couple ways
from operator import itemgetter
print sorted(d.items(), key=itemgetter(1))
# or
items = d.items()
items.sort(key = itemgetter(1))
print items
[('maggie', 6), ('lisa', 70), ('bart', 80), ('marge', 140), ('homer', 350)]
```

- If you find yourself frequently needing to sort a dictionary consider a support data-structure
- By keys: simplest approach: sort the keys then extract the corresponding elements
- Many ways: see: http://writeonly.wordpress.com/2008/08/30/sorting-dictionaries-by-value-in-python-improved/

C / C++ Integration

- There are instances when scripting languages won’t cut it from a performance perspective
  - Often as your intuition develops you can get a sense for this in advance
- Additionally, life is heterogeneous - many instances in which you’ll need to connect to a driver or library written in C
  - You can make your life easier by scripting the bulk of code, and interfacing the special cases

Extending and Embedding

- Recall that Python itself runs in a C-coded VM
  - built in types (including numbers, sequences, dictionaries, sets) are coded in highly optimized C
  - as well as many standard library modules
- Extending
  - building C / C++ modules that Python code can access using the import statement (as well as other languages)
- Embedding
  - executing Python code from an external C application

Common cases

- Performance:
  - you re-implement functionality originally coded in Python
  - rapid prototyping
- Leveraging existing functionality in a C library
  - avoid reinventing the wheel
  - many high quality highly optimized libraries written
- Exposing Python functionality to a host language in the process of embedding Python

The common case: exposing an existing library

- Recall Python's use as Glue
- Exposing the functionality of an existing C library is a common task
    - getting at hardware drivers, math libraries, vision packages, ontologies, etc
- **Many** existing tools to help you

Fundamentals

- A C-coded extension is guaranteed to run only with the version of Python it is compiled for
- You generally need an identical compiler to that used to build your version of Python
  - on *nix systems - it's gcc
  - microsoft is usually MSVC
- A Python extension module named 'foo' generally lives in a dynamic library with the same filename (foo.pyd on Win32, foo.so on *nix)
- That library is customarily placed in the site-packages sub directory of the Python library

Manual decoration

```c
//gcd.c
int gcd(int a,int b)
{
   int c;
   while (a!=0) {
        c = a;
       a = b%a;
       b = c;
   }
   return b;
}
```

```c
// gcd_wrapper.c
#include <Python.h>

extern int gcd(int, int);

PyObject *wrap_gcd(PyObject *self, PyObject *args){
   int x,y;
   if (!PyArg_ParseTuple(args, "ii", &x, &y)) return NULL;
   int g = gcd(x, y);
   return Py_BuildValue("i", g);
}

/* List of all functions to be exposed */
static PyMethodDef gcdmethods[] = {
 { "gcd", wrap_gcd, METH_VARARGS}, {NULL, NULL}
};

void initgcd(void){
   /* Called upon import */
   Py_InitModule("gcd", gcdmethods);
}
```

Building and installing with distutils

- Distribution utilities automates the building and installation of C-coded modules
  - cross platform: definitely the way to go rather than a manual approach
- Assuming you have a properly decorated C module ready to go, say foo.c, create a new file: setup.py in the same directory, execute the below
- then run from the shell $python setup.py install
- you're now free to import your module
  - import gcd
  - gcd(40, 4)

```
from distutils.core import setup, Extension
setup(name='gcd',ext_modules=[Extension('gcd',sources=['gcd.c'])])
```

SWIG

- Manual decoration is cumbersome
  - Appropriate when you're coding a new built-in data type, or core Python extension, otherwise: use a tool
- Simplified Wrapper and Interface Generator: http://www.swig.org
- SWIG decorates C source with the necessary Python markup
- Markup generation is guided by the library's header file (occasionally with some help)
- Not Python specific, support for:
  - Scripting: Perl, PHP, Python, Tcl, Ruby.
  - Non-scripting languages: C#, Common Lisp, Java, Lua, Modula-3, OCAML, Octave and R

SWIG (much easier)

```c
//example.c
int gcd (int a, int b)
{
  int c;
  while (a!=0) {
      c = a;
      a = b%a;
      b = c;
  }
  return b;
}
```

```c
//example.h
int gcd(int,int);
```

Importing a function

```
//example.i - swig directions
%module example
/* Parse the header file to generate wrappers */
%include "example.h"
```

```
#leverage distutils!
#setup.py
from distutils.core import setup, Extension
setup(name='example', ext_modules=[Extension('example', sources=['example.c'])])
```

```
#install using shell commands
$swig -python example.i
$python setup.py install
```

```
#import as normal
#test.py
from example import gcd
print gcd(7890, 12)
```

Boost

- Uniformly high quality C++ libraries
  - Development partially funded by LLNL and LBNL
  - Mathematics intensive
- References:
  - www.boost.org/libs/python/doc

Detailed references

- http://www.python.org/doc/ext/ext.html
- http://www.python.org/doc/api/api.html
- http://www.swig.org/tutorial.html
- www.boost.org/libs/python/doc
- Python in a Nutshell, 2nd Edition: Chapter 25

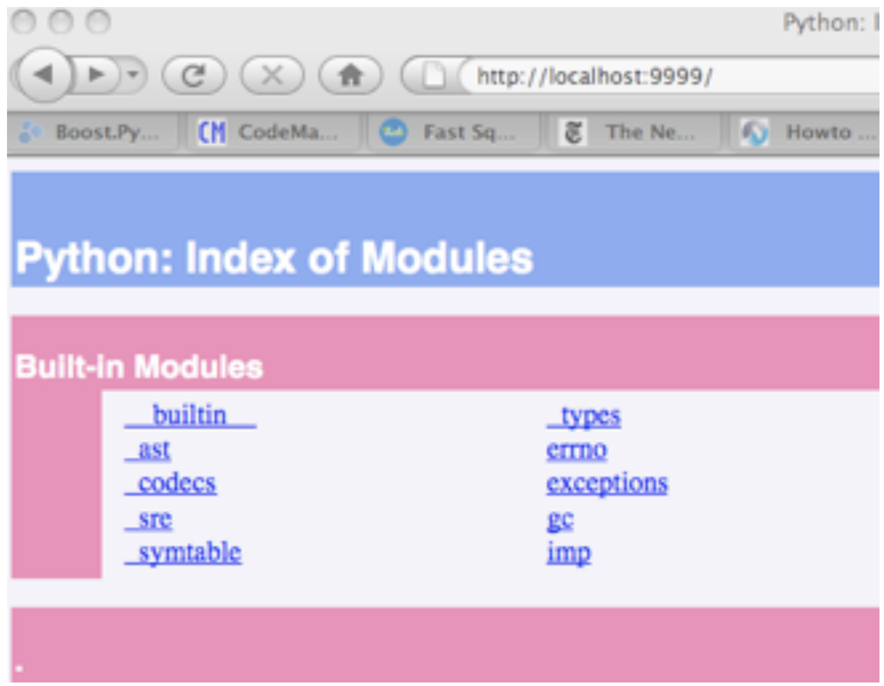
Common question
Pydoc & Docstrings

- Docstrings are used by source parsing tools
- Viewing doc from the terminal
  - pydoc sys
- Producing html
  - pydoc -w hello > hello.html
- Starting a local webserver
  - pydoc -p 9999

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    """
    if imag == 0.0 and real == 0.0: return complex_zero
    ...
```



Common question None vs 'None'

```
def kungfu(punches=5, kicks=None):
    if kicks == None:
        print 'boring'
    else:
        print punches, kicks

kungfu(5, 5) 5 5
kungfu(5) boring
```

- Take a look at Pythons internal types:
- http://docs.python.org/library/types.html
- "The sole value of types.NoneType. None is frequently used to represent the absence of a value, as when default arguments are not passed to a function."

Common question Scheduling Events

- Often have the need to run scripts incrementally
- Useful for maintenance, updates
- Many operating systems have this capability build in - cron, windows scheduler
- Nice to have a little more control

```
# simplistic
import time, os, sys

def main(cmd, inc=60):
    while True:
        os.system(command)
        time.sleep(inc)

if __name__ == '__main__':
    cmd = sys.argv[1]
    if numargs <3:
        main(cmd)
    else:
        inc = sys.argv[2]
        main(cmd, inc)
```

Using sched

```
import sched
schedule = sched.scheduler(time.time, time.sleep)
```

- Why the input of a delay function? When would you not want to use real-time?
- Adding an event returns a unique token which may be used to check status, cancel, etc
- enter - schedules an event at a relative time
- enterabs - schedules a future event at a specific time
- support for priorities
- won't overlap or cancel tasks by default
- useful for guaranteeing a scheduled task completes at the given rate on average
- see: http://docs.python.org/library/sched.html

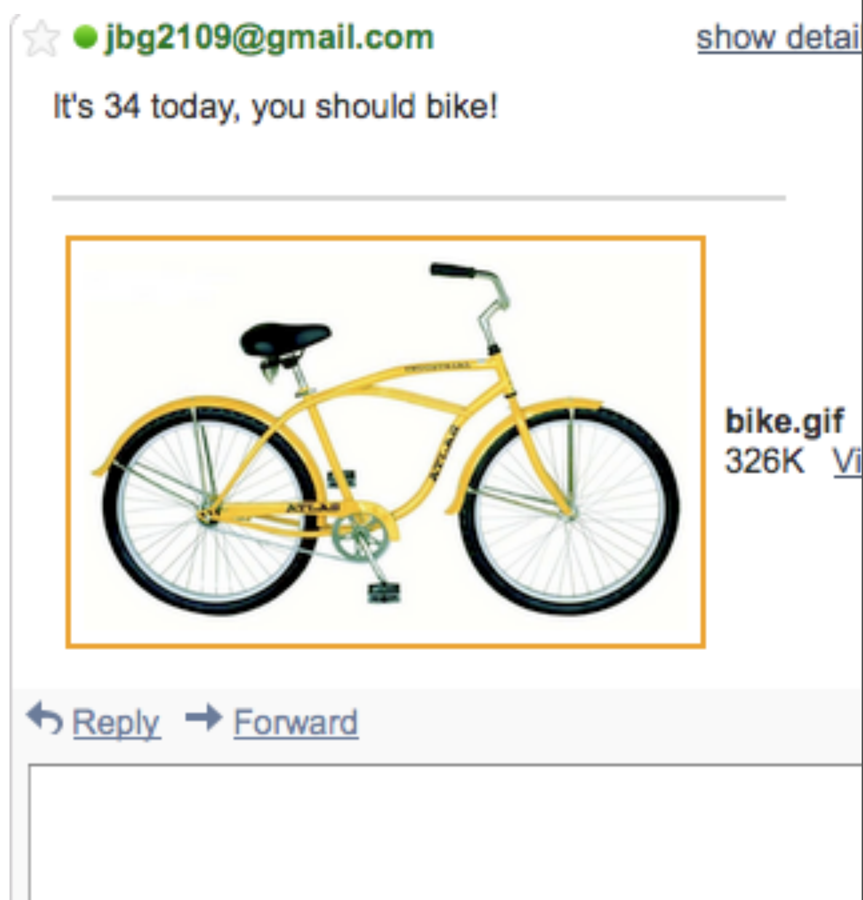
More libraries: email

```
import smtplib,os
#import classes
from email.MIMEMultipart import MIMEMultipart
from email.MIMEBase import MIMEBase
from email.MIMEText import MIMEText
from email import Encoders

def mail(to, subject, text, attach):
   msg = MIMEMultipart()
   msg['From'], msg['To'], msg['Subject'] = user, to,subject
   msg.attach(MIMEText(text))
   part = MIMEBase('application', 'octet-stream')
   part.set_payload(open(attach, 'rb').read())
   Encoders.encode_base64(part)
   part.add_header('Content-Disposition',
           'attachment; filename="%s"' %
         os.path.basename(attach))
   msg.attach(part)
   mailServer = smtplib.SMTP("smtp.gmail.com", 587)
   mailServer.ehlo()
   mailServer.starttls()
   mailServer.ehlo()
   mailServer.login(user, pwd)
   mailServer.sendmail(user, to, msg.as_string())
   mailServer.close()
```

see: http://docs.python.org/library/smtplib.html

```
message = os.system('sports')
image = os.system('sports_image')
subject = 'Sports!'
mail("jbg2109@gmail.com", "Sports!", \
message, image, 'user', 'pass')
```



Common questions
Reversing a string by characters or words

- Lists have the .reverse() method
- Strings do not, but it's still straightforward

```
s = 'ford'
print s[::-1]
s = 'homer drives a ford'
rev = s.split()
rev.reverse()
print ' '.join(rev)

drof
ford a drives homer
```

Process, Threads

- Process
  - run in separate, protected logical address space
  - interprocess communication occurs through special channels - network, a file, a protocol, etc
- Threads
  - Execute simultaneously in a given program w/o protection - sharing data is easy - lower overhead, faster communication
- Examples of multithreaded programming?
- Concurrency issues, atomic and non-atomic operations

The Global Interpreter Lock

- Python's core implementation using a GIL which protects internal data structures
- The key to this lock must be held by a thread before it can safely access objects
- The interpreter releases and reacquires the lock every 100 byte code instructions (settable)
- Released and reacquired around I/O operations
- Effective performance enhancement from multithreading is difficult in Python in compute bounded scenarios
  - Anyone know how to do it?
- When do Python threads still make sense?

Consider threading non-cpu bounded tasks

```
import os, sys

for host in range(60,70):
  ip = "128.59.245."+str(host)
  proc = os.popen("ping -q -c2 "+ip,"r")
  print "Testing ",ip,
  sys.stdout.flush()
  while 1:
    line = proc.readline()
    if not line: break
    print line
```

Compare the execution time of each

Notice join?

Constructors do not execute threads

```
import os, time, sys
from threading import Thread

class pingWorker(Thread):
  def __init__ (self,ip):
    Thread.__init__(self)
    self.ip = ip
    self.status = -1
  def run(self):
    print "Testing ", ip
    proc = os.popen("ping -q -c2 "+self.ip,"r")
    while True:
      line = proc.readline()
      if not line: break
      print line
      self.status+=1

pool = []

for host in range(60,70):
  ip = "128.59.245."+str(host)
  worker = pingWorker(ip)
  pool.append(worker)
  worker.start()

for worker in pool:
  worker.join()
  print "From ",worker.ip," received:  ", worker.lines
```

Take away
Concatenative Programming

- The heart of scripting is concatenation
- Juxtaposing existing libraries is the way to achieve results
- You no longer have to be an expert to accomplish technically complex tasks
  - Doesn’t mean you should ignore the fundamentals
- Basic programming knowledge is becoming ubiquitous