CS3101

Lecture 6

This week

- Threads & processes
- Performance & profiling
- Extending and embedding Python (Boost, SWIG)
- More libraries
- Web frameworks & CGI (Django, Pylons)

Final Projects & Demos

- Due via courseworks March 2nd
- Demos later that week
- Submit a single compressed archive containing well documented source code and the following:
  - project.txt – a 1 page write-up describing your project, results, lessons learned.
  - readme.txt – a short document describing any libraries your code depends on, where to download them (and which versions), as well as instructions for running your project
- Demos
  - Schedule a short demo via Doodle: see email
  - 10-15 minutes, demonstrate usage, features

Threads & Processes

http://docs.python.org/library/multiprocessing.html

Threads: Characteristics

- A thread is an independent flow of control that shares global state with other threads
- All threads appear to execute simultaneously
- Experience with threading in other languages?
  - E.g. PTHREAD in C, threads in Java
- Can be complex, not easy to master
- Can also be powerful
  - Allows you to solve some problems with better architecture, performance
- Consider threading non-compute bounded tasks via thread pools

Processes: Characteristics

- A process is an instance of a running program
- OS protects processes from one another
  - Inter-process communication (IPC) must be used for processes to communicate between one another
  - Communication may also be done via files and databases
- Processes can run on different nodes of a network or on different cores of a local machine
- Each Python process contains its own instance of the interpreter

The Global Interpreter Lock

- Python's core implementation uses a GIL which protects internal data structures
- Why? Easy integration of C libraries that are usually not thread-safe.
  - 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 (parameterized)
- Released and reacquired around I/O operations
- Difficult to speed up compute bounded tasks in Python using multithreading alone
  - One GIL per interpreter
  - use the mulitprocessing module which spawns processes to achieve full concurrency, as each interpreter has its own GIL.

Thread pools: consider for non-compute bounded tasks

```
import subprocess
sites= ['www.google.com',
             'www.yahoo.com’,
             'www.columbia.edu’]

for ip in sites:
  subprocess.call("ping -c2 %s" % ip, shell=True)
```

Thread pools (cont'd)

```
import subprocess
from threading import Thread

Sites = ['www.google.com',
         'www.yahoo.com',
         'www.columbia.edu']

class ping_worker(Thread):
   def __init__ (self,ip):
      Thread.__init__(self) # must explicitly call
      self.ip = ip
   def run(self):
       subprocess.call("ping -c2 %s" % ip, shell=True)
```

Thread pools (cont'd)

```
pool = []

for ip in sites:
    worker = ping_worker(ip)
    pool.append(worker)
    worker.start()

for worker in pool:
    worker.join()
```

Multiprocessing: map and apply

```
# example from the python doc

from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':

   # start 4 worker processes
   pool = Pool(processes=4)

   # evaluate "f(10)" asynchronously
   result = pool.apply_async(f, [10])

   print (result.get(timeout=1))
   # prints "100" unless your computer is *very* slow

   print (pool.map(f, range(10))) # distribute and compute in parallel
   # prints "[0, 1, 4,..., 81]"
```

Pools

- Multiprocessing.Pool, Multiprocessing.
- A process pool controls pool of workers
  - Accepts submitted jobs
- `class multiprocessing.Pool([processes[,initializer [, initargs]]])`
  - If initializer is not None: each process will call `initializer(*initargs)` when it starts
  - Defaults to one worker per available cpu core
- apply – blocks till results is ready
- apply_async – non blocking
- map / map_asyncy – distributes jobs among processes
- close() – prevents more jobs from being submitted
- join() - waits for the worker processes to terminate
- terminate() – immediately brings down the worker processes
- Async methods return a multiprocessing.pool.AsyncResult objected, like a queue
  - get(timeout), wait(timeout), ready(), sucessful()

Multiprocessing Queues

- Import from multiprocessing
- Meant to be shared among threads and processes
- Supplies FIFO queues that provide multithread and *multiprocess* access
- Comprised of one main class and two exception classes
  - Queue: main class
  - Empty: exception arising when trying to get an item from an empty queue
  - Full: arises when queue is full
- Supports blocking and non-blocking put and get
  - Threads can specify timeouts
- A container for arbitrary objects

Using Queues for Inter-process Communication

```
# queues are process and thread safe
# ex. from the python doc

from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print (q.get()) # blocking call
    p.join()
```

Assigning worker functions to processes

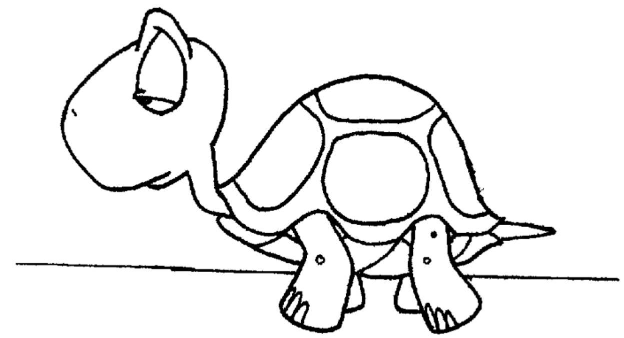
```
#ex from the pydoc

from multiprocessing import Process
import os

def worker(name):
    print ('parent process:', os.getppid())
    print ('process id:', os.getpid())
    print ('hello', name)

if __name__ == '__main__':
   pool = []
   for arg in ['homer', 'lisa']:
       p = Process(target=worker, args=('bob',)) # note that trailing
   # comma
       pool.append(p)
       p.start()

   for p in pool:
       p.join() # block until this process completes
```

Performance & Profiling

http://docs.python.org/library/profile.html

Performance Basics

- Python is substantially slower than C (about one to two orders of magnitude)
- The tradeoff is often well worth it:
  - more of your time to focus on the problem
- General methodology:
  - Focus on algorithms, use Python to prototype
  - If it's fast enough, move on
  - If it's not: profile, rewrite in Python
  - Finally, rewrite modules in C and import
- 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
  - Finding a high performance library (e.g., numpy, Boost)
  - Rewriting your own modules in C and importing

Playing 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 on the same machine. 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?

A classic pitfall: string concatenation in object languages (but newly defunct in Python 2.5+ due to architecture changes)

```
def slow(): # create lots of objects unnecessarily
    big = ''
    small = 'foo'
    for i in range(10000):
        big+= small
    return big

def fast(): # perform a single concatentation
    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: why never to teach recursion via Fib()

```
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()')
```

```
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)
```

Memonization using function decorators

- Idea: cache a functions return results in a dictionary, keyed by the arguments that produced that value
- Worth understanding - useful for optimizing recursive functions, server side code

Memoizing a recursive function

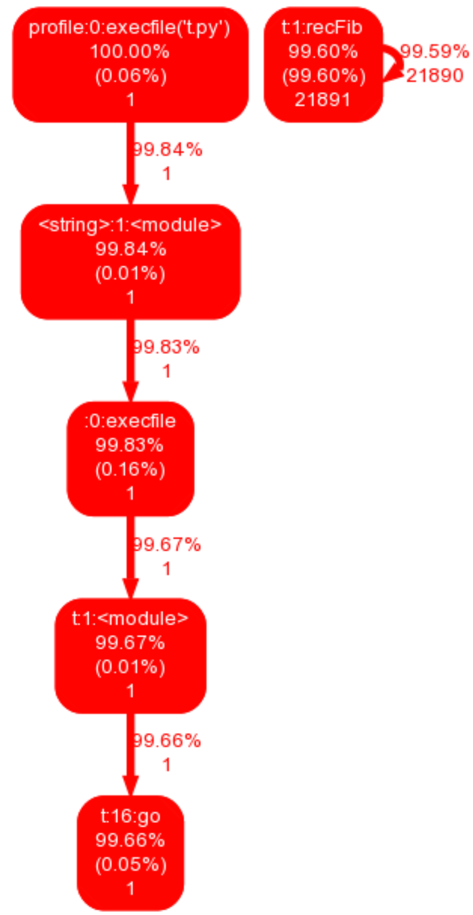
```
#cache the return values of a fn
def memoize(cache=None):
    if cache is None: cache = {}
    def decorator(function):
        def decorated(*args):
            if args not in cache:
                cache[args] = function(*args)
            return cache[args]
        return decorated
    return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

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
```

Extending and Embedding

http://www.swig.org/Doc1.3/Python.html

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
- Software 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
- Which means Python is highly extensible!
  - 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
  - Exposing Python libraries to a host language in the process of embedding Python

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

Manually (avoid when possible – there are tools to assist)

```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 in native Python
  - `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)

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

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

SWIG (cont'd)

```
//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

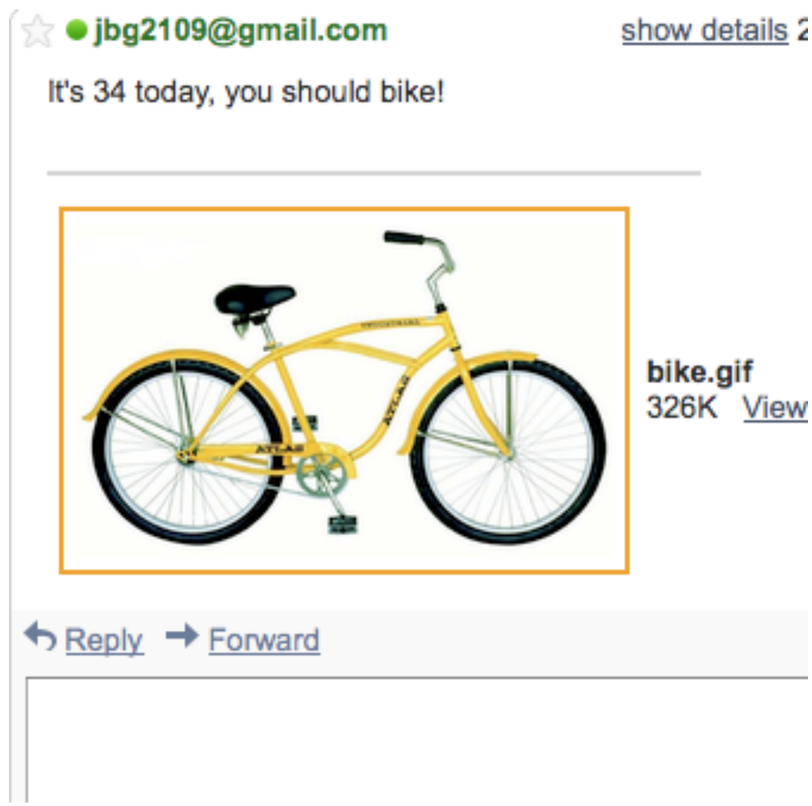
Libraries III

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')
```



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

Simple Scheduling

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

```
# basic, python 2.6
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 for simualtions

```
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

Python and CGI

http://docs.python.org/library/cgi.html

http://httpd.apache.org/

CGI (Slides courtesy of John Zhang)

- When a web browser requests a page from a web server, the server may return either static or dynamic content
  - Serving dynamic content requires the server-side programs to generate and deliver the content
- The Common Gateway Interface (CGI) is a long-standing web- wide standard for server-side programming
- What happens:
  - First, browser sends request to server
  - Server executes another program, passing content of request
  - Server captures standard output of other program
  - Server returns output to the browser as response to request
- A CGI program / script is the "other program" in this case

CGI (cont'd)

- CGI is a standard, so you can code scripts in any language
- Scripts often handle submitted HTML forms
  - ACTION attribute of the FORM tag specifies the URL for a CGI script to handle the form
  - Method is GET or POST
  - GET encodes form contents as query string and sends as part of URL, POST transmits form's contents as encoded stream of data
  - GET is faster, you can use a fixed GET-form URL, but can't send large amounts of data, and URL length is limited
  - POST has no size limits
- With CGI, GET data is sent as query string, POST data is sent through standard input

The cgi Module

- Recovers data only from the query string if it is present, otherwise, recovers data from standard input
- Module supplies one function and one class that will be used often
  - Function escape(...)
    - Escapes a string, i.e. replaces some characters with appropriateHTML entites such as <, >, & with <, > and &
  - Class FieldStorage
    - Used for parsing input

The FieldStorage Class

- When FieldStorage is instantiated, the query string and/or standard input is parsed
  – Distinction between POST and GET is hidden
- Must only be instantiated once, since it consumes stdin
- FieldStorage instances are mappings
  – Keys are the NAME attributes of the form's controls
  – Contains a subset of dict's functionality
    - Iteration, checking if a key is present, indexing are possible

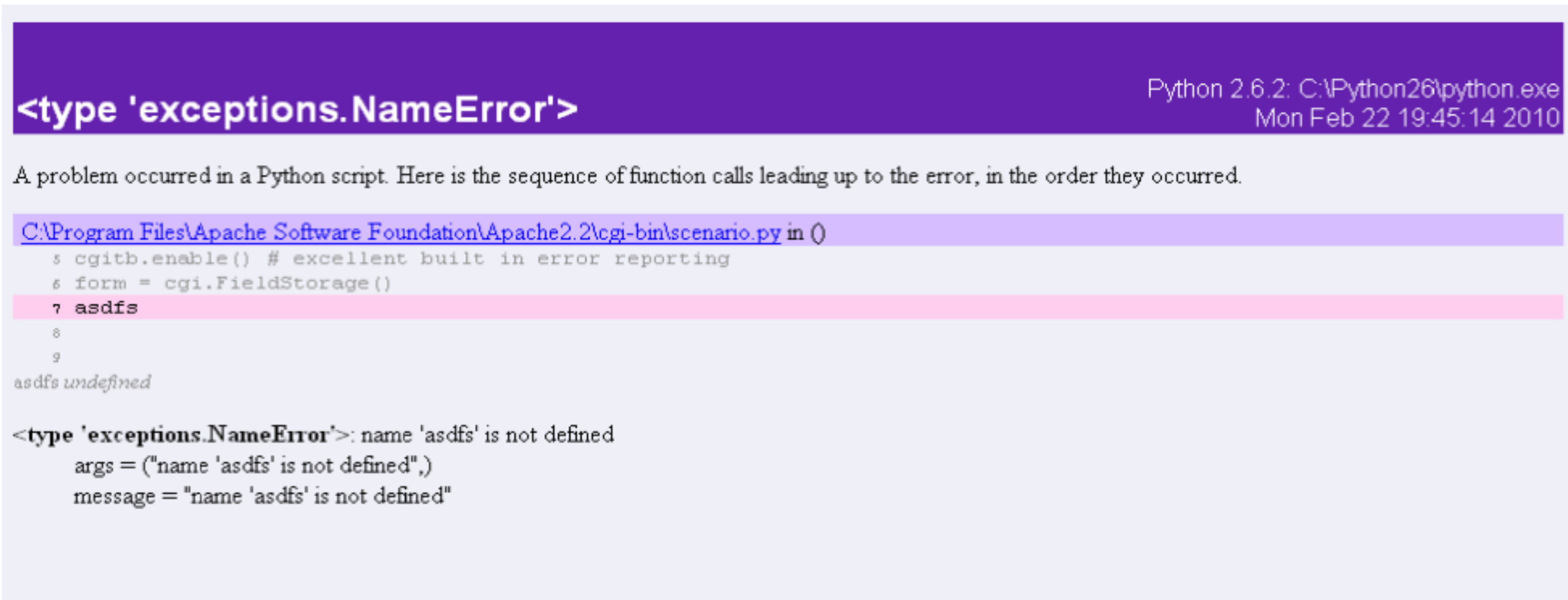
Output

- The response to a HTTP request is the standard output of the CGI script
- Script must output:
  - Content-type header (often just text/html)
  - Followed by a blank line
  - Followed by response body
- Script may also output be any MIME type followed by a response body that conforms to the type
  - Response is often in HTML or XML

Example (python 2.6)

```
import cgi, cgitb
cgitb.enable() # built in error handling

print "Content-Type: text/html" # HTML is following
print # blank line, end of headers
print "<TITLE>CGI script output</TITLE>"
print "<H1>This is my first CGI script</H1>"
print "Hello, world!"

form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print "<H1>Error</H1>"
    print "Please fill in the name and addr fields."
    return
print "<p>name:", form["name"].value
print "<p>addr:", form["addr"].value
```

cgitb



<type 'exceptions.NameError'>
Python 2.6.2: C:\Python26\python.exe
Mon Feb 22 19:45:14 2010
A problem occurred in a Python script. Here is the sequence of function calls leading up to the error, in the order they occurred.
C:\Program Files\Apache Software Foundation\Apache2.2\cgi-bin\scenario.py in ()
5 cgitb.enable() # excellent built in error reporting
6 form = cgi.FieldStorage()
7 asdfs
8
9
asdfs *undefined*
<**type 'exceptions.NameError'**>: name 'asdfs' is not defined
args = ("name 'asdfs' is not defined",)
message = "name 'asdfs' is not defined"

Installing Scripts on Apache

- Depends on web browser and host platform
  - Here, we assume you are using Apache
- In configuration file httpd.conf
  - ScriptAlias /cgi-bin/ /usr/local/apache/cgi-bin/
    - Enables any executable script in aliased directory to run as CGI script
  - Or, you may enable CGI execution in a specific directory using Apache directive
    - Options +ExecCGI
    - You'd also need to add a global AddHandler directive
    - AddHandler cgi-script py
    - Enables scripts with extension .py to run as CGI
- Also see mod python: www.modpython.org

Web Frameworks

- Offer different functionality and philosophies
- Some integrate database access, others focus on web part, etc.
- Some examples
  - Django www.djangoproject.com
  - Pylons www.pytlonshq.com
- Usually offer built in support for sessions
- Worth it for larger projects

Final Projects & Demos

- Due via courseworks March 2nd
- Demos later that week
- Submit a single compressed archive containing well documented source code and the following:
  - project.txt – a 1 page write-up describing your project, results, lessons learned.
  - readme.txt – a short document describing any libraries your code depends on, where to download them (and which versions), as well as instructions for running your project
- Demos
  - Schedule a short demo via Doodle: see email
  - 10-15 minutes, demonstrate usage, features