Software Development
Python (Part C)

Davide Balzarotti

Eurecom – Sophia Antipolis, France

Overview

- Modules and packages
- Python standard library
  - The sys module
  - The os (Operating System) module
  - Spawning and controlling other processes
  - Regular expressions
  - Network
  - Threads
  - Signals

Modules

- Modules provide a basic way of organizing code
- In Python, a module is simply a Python file
  - Any python script can be seen (and imported) as a module
  - Module X is just a file named X.py

- Modules are imported using the `import` keyword
- When you write “`import foo`” Python:
  - If the module foo has already been imported, it does nothing (it is not imported again!)
  - Otherwise, it searches for the file `foo.py` (or `foo.pyc`) in the search path
    - By default, the search path includes the current working directory and the standard library directories

Modules

- Each module has a symbol table (`__dict__`) that contains all the names defined by the module
- A module usually contains variables and function (or class) definitions. But it can also contain normal code
  - When a module is imported, all the instructions contained in the python file are executed
  - Within a module, the module's name is available as the value of the global variable `__name__`
  - If you want to execute some code only if the file is executed as a script (not when imported as a module) you can use:

```
if __name__ == "__main__":
```

- `reload(modulename)` allows to re-import a module that has already been imported

Modules

```
# foo.py

def f(a,b,c):
    return a+b+c

print 'Hi'
if __name__ == '__main__':
   print 'I was directly invoked'
else:
   print 'I was imported as a module'
```

```
>> import foo
Hi
I was imported as a module
>> foo.f(1,2,3)
6
```

Modules

```
# foo.py

def f(a,b,c):
    return a+b+c

print 'Hi'
if __name__ == '__main__':
    print 'I was directly invoked'
else:
    print 'I was imported as a module'
```

```
>> import foo
Hi
I was imported as a module
>> foo.f(1,2,3)
6
```

```
>> from foo import f
Hi
I was imported as a module
>> import foo
>> reload(foo)
Hi
I was imported as a module
<module 'foo' from 'foo.pyc'>
```

Modules

```
#  foo.py

def  f(a,b,c):
    return  a+b+c

print  'Hi'
if  __name__  ==  '__main__':
    print  'I  was  directly  invoked'
else:
    print  'I  was  imported  as  a  module'
```

```
>> import foo
Hi
I was imported as a module
>> foo.f(1,2,3)
6
```

```
>> from foo import f
Hi
I was imported as a module
>> import foo
>> reload(foo)
Hi
I was imported as a m
<module 'foo' from 'fo
```

```
balzarot> python foo.py
Hi
I was directly invoked

balzarot>
```

Packages

- Packages are a way to group together and organize a set of related modules
- Module names in a package are accessible using the *dot* notation
  - For example, the module name A.B designates a module named B in a package named A
- The structure of the package hierarchy is determined by the organization of the modules in the filesystem
  - Useful since putting many modules in the same directory can be cumbersome

Packages

- A package in Python is simply a directory that contains a file named `__init__.py`
  - The `__init__.py` file can execute some initialization code, or it can be an empty file if no initialization is required

```
sound/                  Top-level package
  __init__.py           Initialize the sound package
  formats/              File format subpackage
     __init__.py
     wavread.py
     wavwrite.py
     aiffread.py
     aiffwrite.py
     auwrite.py
     ...
  effects/               Subpackage for sound effects
     __init__.py
     echo.py
     surround.py
     reverse.py
     ...
```

```
>>> import sound.effects.echo
>>> sound.effects.echo.echofilter(input, output, delay=0.7)
```

Importing * From a Package

- In presence of complex packages, import * could take a long time and have unwanted side-effects
- By default, when import * is used on a package it does not import all submodules into the current namespace
- The only solution is for the package author to provide an explicit index of the package
  - If a package’s `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when the user use `import *`

Today

- Modules and packages
- Python standard library
  - The sys module
  - The os (Operating System) module
  - Spawning and controlling other processes
  - Regular expressions
  - Network
  - Threads
  - Signals

sys Module

- `path` – list of the paths where Python is looking for when it imports modules
- `version, subversion, version_info` – info about the version of Python running
- `stdin, stdout, stderr` – no need to explain
- `argv` – list of the command-line arguments given to the program
  - Each argument is always a string
  - The first argument is the name of the program
- `exit(value)` – terminates the program

Simple Parameters Handling

```
import sys

if len(sys.argv) < 2:
   sys.stderr.write("Use: %s value\n”%sys.argv[0])
   sys.stderr.write("  value - just a number\n")
   sys.exit(1)

value = int(sys.argv[1])
...
```

Getopt (the C way)

Provide an easier way to parse the command line arguments

```
import getopt
# example of parameter list (you should use sys.argv)
arglist = ['-ab', '-cfoo', '-d', 'bar', 'a1', 'a2']
# parsing the options
opts, args = getopt.getopt(arglist, 'abc:d:')

# opts=[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
# args=['a1', 'a2']
```

Short options
the ':' characters means that the option
requires an argument

Getopt (the C way)

Provide an easier way to parse the command line arguments

```
import getopt
# parameters (containing long options too)
arglist = ['--cond=foo', '--testing', '--ofile', 'abc.def',
           '-x', 'a1', 'a2']
# parsing the parameters
optlist, args = getopt.getopt(args, 'x',
                              ['cond=', 'ofile=', 'testing'])

# optlist = [('--cond', 'foo'), ('--testing', ''),
             ('--ofile', 'abc.def'), ('-x', '')]
# args=['a1', 'a2']
```

Long options
the '=' characters means that the option requires an argument

ArgParse (the Python 2.7 way)

- Flexible, and powerful library for parsing command-line options
- Replace OptParse introduced (and already depracated) in 2.6

```
from argparse import ArgumentParser
# create a new option parser
parser = ArgumentParser(description="process some integers")

# add the allowed arguments
parser.add_argument("integers", metavar="N", type=int,
                  nargs='+', help="value for the accumulator")

parser.add_argument("--sum", dest="operator", default="max",
                    action="store_const", const="sum",
                    help="sum the values (instead of finding
                        the max")

# parse the command line arguments
args = parser.parse_args()
print "Integers:",args.integers
print "Operator:",args.operator
```

ArgParse (the Python 2.7 way)

- Flexible, and powerful library for parsing command-line options
- Replace OptParse introduced (and already depracated) in 2.6

```
from argparse import ArgumentParser
# create a new option parser
parser = ArgumentParser(description="process some integers")

#
pa

ad
pa




#
ar
print "integers:",args.integers
print "Operator:",args.operator
```

```
balzarot> script.py -h
usage: script.py [-h] [--sum] N [N ...]

process some integers

positional arguments:
  N             value for the accumulator

optional arguments:
  -h, --help    show this help message and exit
  --sum         sum the values (instead of finding the max
```

OS

- The `os` module provides a portable way of using operating system dependent functionalities
- Access to the process environment
  - `environ` – dictionary containing the environment variables
- File/directory management
  - Shell-like commands `(chdir, chmod, chown, mkdir, rmdir, link..)`
  - File descriptors operators: `dup, open, close, fstat..`
  - `listdir(path)` – return a list containing all the files and directories in *path*
- Process management:
  - `execl, execle, execv, popen, popen2, popen3..`

Paths Manipulation

- The `os.path` module implements some useful functions on pathnames
  - Available in different flavors to fit different operating systems (`posixpath, ntpath, macpath`..)
- Info about a file:
  `getsize(), getatime(), getmtime(), getctime()`
- Absolute path and current directory:
  `os.path.abspath(os.path.curdir)`
- Check for file/directory existence:
  `exists(path), isdir(path), isfile(path)`
- Join multiple path together: `join(path1, path2,` …`)`
- Split a path in its directory and file parts: `split(path)`

Example

```
import os
import os.path

def whereis(program):
  for d in os.environ.get('PATH', '').split(':'):
    full_path = os.path.join(d, program)
    if os.path.exists(full_path) and not \
       os.path.isdir(full_path):
         return full_path
  return None
```

subprocess

- Provide functionalities to spawn new processes, connect to their input/output/error pipes, and obtain their return codes
    - To manage subprocesses, Python had many functions spread in different modules - the subprocess module intends to replace them all
- The process creation and communication is done through the `subprocess.Popen` class and its methods

```
p = subprocess.Popen(
   args,
   shell=False,
   stdin=None, stdout=None, stderr=None,
   cwd=None,
   env=None)
```

```
p = subprocess.Popen(
    args,
    shell=False,
    stdin=None, stdout=None, stderr=None,
    cwd=None,
    env=None)
```

If `shell=False`, the Popen class uses os.execvp() to create the new process. In this case, args must be a list containing the program name and its arguments

If `shell=True`, args is a string that is executed through the shell

```
p = subprocess.Popen(
    args,
    shell=False,
    stdin=None, stdout=None, stderr=None,
    cwd=None,
    env=None)
```

If `shell=False`, the Popen class uses os.execvp() to create the new process. In this case, args must be a list containing the program name and its arguments

If `shell=True`, args is a string that is executed through the shell

Specify the file handles for standard input, output, and error
Valid values are PIPE, an existing file object or descriptor, or None.

```
p = subprocess.Popen(
    args,
    shell=False,
    stdin=None, stdout=None, stderr=None,
    cwd=None,
    env=None)
```

If `shell=False`, the Popen class uses os.execvp() to create the new process. In this case, args must be a list containing the program name and its arguments

If `shell=True`, args is a string that is executed through the shell

Specify the file handles for standard input, output, and error

Valid values are PIPE, an existing file object or descriptor, or None.

If cwd is not `None`, the child's current directory will be changed to cwd before it is executed

```
p = subprocess.Popen(
  args,
  shell=False,
  stdin=None, stdout=None, stderr=None,
  cwd=None,
  env=None)
```

If `shell=False`, the Popen class uses os.execvp() to create the new process. In this case, args must be a list containing the program name and its arguments
If `shell=True`, args is a string that is executed through the shell

Specify the file handles for standard input, output, and error
Valid values are PIPE, an existing file object or descriptor, or None.

If cwd is not `None`, the child's current directory will be changed to cwd before it is executed

Defines the environment variables for the new process. If None, the new process will inherit the current process' environment

The Popen Object

- Methods:
  - `poll()` - check if the process is still running
  - `wait()` - wait until the process terminates
    (it may deadlock if PIPEs are full)
  - `send_signal(signal)` - send a signal to the process
  - `kill()` - kill the process
- Fields:
  - `stdin, stdout, stderr` – corresponding file objects (only if set to PIPE when the process was created)
  - `pid` – the process ID
  - `returncode` – contains the process return code (if it is already terminated, or None otherwise)

Example

```
import subprocess

cat = subprocess.Popen(['cat', 'example.py'],
                        stdout=subprocess.PIPE,
                        )

grep = subprocess.Popen(['grep', 'subprocess'],
                        stdin=cat.stdout,
                        stdout=subprocess.PIPE,
                        )

cut = subprocess.Popen(['cut', '-f1', '-d='],
                        stdin=grep.stdout,
                        stdout=subprocess.PIPE,
                        )

end_of_pipe = cut.stdout

for line in end_of_pipe:
    print line.strip()
```

Regular Expression

- The `re` module provides regular expression functionalities
- Regex use '\' to escape special characters: `.^$*+?{}[]\|()`
  - But python use '\' to escape special string sequences: `\n,\t,\r..`
  - Therefore, to match a '\' in a regex you have to write '\\\\'
    - Or (better) use raw strings: `r'regex'`
- Usual regex syntax:
  - `'^[0-9]+'` : one or more digits at the beginning of the line
- Grouping support:
  - Simple positional group: `(regex)`
  - Named group: `(?P<name>regex)`
  - Match whatever text was matched by the group name: `(?P=name)`

Regex

- If you plan to match the same expression more than once, it's better to compile it to improve performance
- The result of a regex matching is a `MatchObject` instance

```
>>> import re
>>> pattern = re.compile(r'spam:([a-z]+)')
>>> m = pattern.search('this is spam:egg')
>>> m.group(0)   # zero corresponds to the entire regex
spam:egg
>>> m.group(1)   # non-zero is the n-th group in the regex
egg
>>> m.span(1)    # start and end position of a group
(13, 16)
```

Regex

- If you plan to match the same expression more than once, it's better to compile it to improve performance
- The result of a regex matching is a `MatchObject` instance

```
>>> import re
>>> pattern = re.compile(r'spam:(?P<spam>[a-z]+)')
>>> m = pattern.search('this is spam:egg')
>>> m.group(0)    # zero corresponds to the entire regex
spam:egg
>>> m.group("spam")
egg
>>> m.span("spam")     # start and end position of a group
(13, 16)
```

Searching, Matching, and more..

- `match()` determines if the RE matches at the beginning of the string
- `search()` scans through a string, looking for any location where the RE matches
- `finditer()` returns an iterator through all the matches
- `split()` splits the string into a list, splitting it wherever the RE matches
- `findall()` returns all the non-overlapping matches of the pattern as a list of strings
- `sub()` finds all substrings where the RE matches, and replace them with a different string

Regex substitution

- It's possible to substitute a regex with a string

```
>>> pattern = re.compile(r'number:(?P<number>[0-9]+)')
>>> pattern.sub('number:XXX', "name:Jack number:06213123")
name:Jack number:XXX
```

Regex substitution

- It's possible to substitute a regex with a string

```
>>> pattern = re.compile(r'number:(?P<number>[0-9]+)')
>>> pattern.sub('number:XXX', “name:Jack number:06213123”)
name:Jack number:XXX
```

- But it's also possible to substitute with the output of a function

```
def scale(match):
    value = int(match.group('coord'))
    return 'X:%d'%(value*3)

pattern = re.compile(r'X:(?P<coord>[0-9]+)')
pattern.sub(scale, “ X:22  Y:55”)
 X:66   Y:55
```

More on Regex: Greedy Matching

```
>>> text="<a href='l1'>some text</a>
          <a href="l2">more text</a>"
>>> print re.search("l1'>(.*)</a>", text).group(1)
```

More on Regex: Greedy Matching

```
>>> text="<a href='l1'>some text</a>
          <a href="l2">more text</a>"
>>> print re.search("l1'>(.*)</a>", text).group(1)
some text</a> ... <a href='l2'>more text
```

- By default, the + and * operators are greedy (i.e., they try to match as many characters as possible)
- Solutions
  - If possible, refine the regular expression (e.g., use [^<] instead of .)
  - Add a ? after the operator to make it match as few characters as possible

Lazy Matching

```
>>> text="<a href='l1'>some text</a>
          <a href="l2">more text</a>"
>>> print re.search("l1'>(.*)</a>", text).group(1)
some text</a> ... <a href='l2'>more text
```

```
>>> text="<a href='l1'>some text</a>
          <a href="l2">more text</a>"
>>> print re.search("l1'>(.*?)</a>", text).group(1)
some text
```

Network

- Python standard library includes several modules covering multiple Internet protocols
  - TCP/IP sockets
  - HTTP
  - FTP
  - SSL
  - Telnet
  - XML RPC
  - Mail (IMAP, POP, SMTP)
- External libraries cover the rest:
  - Scapy: powerful low level packet manipulation library
  - Twisted: an event-driven networking engine that supports a large number of protocols (including HTTP, NNTP, DNS, IMAP, SSH, SFTP, IRC, FTP, instant messaging and many others)

Sockets

Server

```
import socket

# create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# associate the socket with a port
s.bind(('', 1234))

# (optional) reuse a socket to prevent waiting..
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# accept connections
s.listen(5)

client_s, addr = s.accept()
print 'Connection from ', addr

client_s.send('Welcome to the server\n')
print client_s.recv(100)
client_s.close()
s.close()
```

Sockets

```
import socket

# create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# associa
s.bind(('

# (option
s.setsock

# accept
s.listen(

client_s,
print 'Co

client_s.send('Welcome to the server\n')
print client_s.recv(100)
client_s.close()
s.close()
```

Client

```
import socket

# create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# open a connection to a certain port
s.connect(('localhost', 1234))

print s.recv(100)

s.send('Bye bye\n')
s.close()
```

Sockets

```
import socket

# create a socket
s = socke

# associa
s.bind(('

# (option
s.setsock

# accept
s.listen(

client_s,
print 'Co

client_s.
print client_s.recv(100)
client_s.close()
s.close()
```

Client

```
import socket

# create a socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# open a connection to a certain port
s.connect(('localhost', 1234))

print s.recv(100)

s.send('Bye bye\n')
s.close()
```

Receives up to 100 bytes !!

Sends some characters !!
Use `sendall()` to be sure that the whole string is sent

Line-based Sockets

- A socket receives and sends sequences of bytes
  - No nice functions like `readline()`
  - Usually implemented manually looping on `recv(1)`

- A way around this problem is to convert the socket to a file-like object

```
s.connect(('localhost', 1234))

# Return a file object associated with the socket
# (for read-write, unbuffered)
fs = s.makefile('rw',0)

print fs.readline()
fs.write('Bye bye\n')
```

The WEB (client-side)

```
import urllib
import HTMLParser

class GetLinks(HTMLParser.HTMLParser):
    def handle_starttag(self,tag,attrs):
        if tag == 'a':
            for name,value in attrs:
                if name == 'href':
                    print value

gl = GetLinks()

url =  'http://www.iseclab.org/softdev/material.html'
urlconn = urllib.urlopen(url)
urlcontents = urlconn.read()

gl.feed(urlcontents)
```

The WEB (client-side)

```
import urllib
import HTMLParser

class GetLinks(HTMLParser.HTMLParser):
    def handle_starttag(self,tag,attrs):
        if tag == 'a':
            for name,value in attrs:
                if name == 'href':
                    print value

gl = GetLinks()

url =  'http://www.iseclab.org/softdev/material.html'
urlconn = urllib.urlopen(url)
urlcontents = urlconn.read()

gl.feed(urlcontents)
```

The WEB (client-side)

```
import urllib
import HTMLParser

class GetLinks(HTMLParser.HTMLParser):
    def handle_starttag(self,tag,attrs):
        if tag == 'a':
            for name,value in attrs:
                if name == 'href':
                    print value

gl = GetLinks()

url =  'http://www.iseclab.org/softdev/material.html'
urlconn = urllib.urlopen(url)
urlcontents = urlconn.read()

gl.feed(urlcontents)
```

Need to get some data out of some badly-formatted HTML page?
Check out the Beautiful Soup module

```
opener = urllib2.build_opener()
opener.addheaders = [('User-agent','Mozilla/5.0')]
opener.open('http://www.example.com/')
```

Headers

```
import os, cookielib, urllib2
cj = cookielib.MozillaCookieJar()
cj.load(os.path.join(os.environ["HOME"],".netscape/cookies.tx
t"))
opener =
urllib2.build_opener(urllib2.HTTPCookieProcessor(cj
r = opener.open("http://example.com/")
```

Cookies

```
import urllib
import urllib2

url = 'http://www.someserver.com/cgi-bin/register.cgi'
values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }

data = urllib.urlencode(values)
req = urllib2.Request(url, data)
response = urllib2.urlopen(req)
the_page = response.read()
```

Post Data

The Web (server-side)

```
import BaseHTTPServer
from SimpleHTTPServer import
SimpleHTTPRequestHandler

HandlerClass.protocol_version = 'HTTP/1.0'
httpd = BaseHTTPServer.HTTPServer(
  ('127.0.0.1', 8080), SimpleHTTPRequestHandler )
httpd.serve_forever()
```

Or simply...

```
> python -m SimpleHTTPServer
```

Python for Webpages (CGI)

```
#!/usr/local/bin/python
import cgi
import cgitb; cgitb.enable()
Import Cookie

print "Content-type: text/html\n"

form = cgi.FieldStorage()
first_name = form.getvalue('first_name')
last_name  = form.getvalue('last_name')


c = Cookie.SimpleCookie(os.environ['HTTP_COOKIE'])
session_id = c['session_id'].value

print '''<html><body>
<h1>Title</h1>
…'''
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur

Provide access to all the submitted form data (GET and POST)

Cookie Management

Python for Webpages

- Web Frameworks
  - Python data model with object-relational mapper
  - Request routing
  - Automatic administration interface
  - Template system
  - Caching
  - Internationalization
  - http://wiki.python.org/moin/WebFrameworks lists 49 web frameworks written in python (Django, Zope, …)
- Tens of CMS written in python

Threads

- Threads allow a process to do multiple things at once
- A process can have multiple threads
  - Each thread has its own local state
  - All threads share the same global state
- How threads are scheduled to run is dependent on how they are implemented
- Threads in Python are pre-emptive
  (i.e, they can be interrupted at any time)
- Two modules:
  - `thread` – primitive thread functionality
  - `threading` - higher-level threading interfaces built on top of the thread module

Thread

```
thread.start_new_thread(tfunction ,(tparameters))
lockname = thread.allocate_lock()
```

- Basic threading and locking functionalities
- `start_new_thread()` starts a new python thread
  - The new thread will execute the function `tfunction` invoked with the parameter `tparameters`
  - When the main thread exits, it is system dependent whether the other threads will survive or get killed
- `allocate_lock()` creates a mutex to allows different threads to synchronize

Locking

- `lockname = thread.allocate_lock()`
- `lockname.aquire()` - takes the lock
  - Trying to acquire an already taken lock will block the current thread until the lock is released
  - Trying to acquire a lock that was already acquired by the same thread leads to a deadlock situation (!)
- `lockname.release()` - release the lock
  - Releasing a lock that was not previously acquired generate a `thread.error`

Example

```
import thread
import socket

main_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
main_s.bind(('', 1234))
main_s.listen(5)

total = 0
total_lock = thread.allocate_lock()
for x in range(2):
   s,client = main_s.accept()
   thread.start_new_thread(client_manager,(s,))
main_s.close()

print total
```

Example

```
import thread
import socket

main_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
main_s.bind(('', 1234))
main_s.listen(5)

total = 0
total_lock = thread.allocate_lock()
for x in range(2):
   s,client = main_s.accept()
   thread.start_new_thread(client_manager,(s,))
main_s.close()

print total
```

```
def client_manager(s):
   global total, total_lock
   v = int(s.recv(10))
   total_lock.acquire()
   total += v
   total_lock.release()
   s.close()
```

Example

```
import thread
import socket

main_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
main_s.bind(('', 1234))
main_s.listen(5)

total = 0
total_lock = thread.allocate_lock()
for x in range(2):
   s,client = main_s.accept()
   thread.start_new_thread(client_manager,(s,))
main_s.close()

print total
```

When does the total get printed?

```
def client_manager(s):
   global total, total_lock
   v = int(s.recv(10))
   total_lock.acquire()
   total += v
   total_lock.release()
   s.close()
```

Threading

- Using the threading module, each thread is represented by an instance of the class `Thread`
  - Normally, an application defines a subclass of Thread and redefines the method `run()` to implement the thread behavior
- Thread objects
  - `start()` - start the thread
  - `run()` - implement the thread logic (automatically invoked by start())
  - `join([timeout])` – wait till the thread ends (or the optional timeout expires)
  - `is_alive()` - check if the thread is still alive
  - `daemon` – if set to True before calling start(), set the thread as a daemon thread
- Other useful functions in the threading module:
  - `current_thread()` - returns the current Thread object
  - `active_count()` - returns the number of Thread object alive
  - `enumerate()` - returns a list of all Thread objects currently alive

```
def client_manager(s):
   global total, total_l
   v = int(s.recv(10))
   total_lock.acquire()
   total += v
   total_lock.release()
   s.close()
```

```
class client_manager(threading.Thread):
  total = 0
  total_lock = threading.Lock()

  def __init__(self, socket):
      threading.Thread.__init__(self)
      self.s = socket

  def run(self):
      v = int(self.s.recv(10))
      client_manager.total_lock.acquire()
      client_manager.total += v
      client_manager.total_lock.release()
      self.s.close()
```

```
import thread
import socket

main_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
main_s.bind((’’, 1234))
main_s.listen(5)

total = 0
total_lock = thread.allocate_lock()
for x in range(2):
   s,client = main_s.accept()
   thread.start_new_thread(client_manager,(s,))
main_s.close()

print total
```

```
import threading
import socket

main_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
main_s.bind((’’, 1234))
main_s.listen(5)

for x in range(2):
   s,client = main_s.accept()
   new_client = client_manager(s)
   new_client.start()
main_s.close()
for t in threading.enumerate():
   if t is not threading.current_thread():
      t.join()
print client_manager.total
```

The global interpreter lock

- To facilitate garbage collection, the CPython implementation has a global interpreter lock (GIL) that is used to ensure that only one thread runs at a certain time
- The GIL prevents multiple thread to run in parallel on multi-processor machines
  - It also degrades the performance (http://www.dabeaz.com/python/GIL.pdf)
- So, what if your really need that kind of efficiency?

The global interpreter lock

- To facilitate garbage collection, the CPython implementation has a global interpreter lock (GIL) that is used to ensure that only one thread runs at a certain time
- The GIL prevents multiple thread to run in parallel on multi-processor machines
  - It also degrades the performance (http://www.dabeaz.com/python/GIL.pdf)
- So, what if your really need that kind of efficiency?
  - Maybe you shouldn't use Python in the first place :)
  - You can still use processes instead of threads
  - You can switch to a different Python implementation (e.g., IronPython)

`signal`, dealing with asynchronous events

- signal handlers can only occur between “atomic” instructions of the Python interpreter
  - signals arriving during long calculations implemented purely in C (such as regular expression matches) may be delayed for an arbitrary amount of time
- Python installs a small number of signal handlers by default:
  - SIGPIPE is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions)
  - SIGINT is translated into a KeyboardInterrupt exception
- Signal *cannot* be used to communicate between threads
  - Only the main thread can set a new signal handler, and the main thread will be the only one to receive signals

Signals

- `signal.alarm(time)` – a SIGALRM signal will be sent to the process in *time* seconds
- `signal.signal(signalnum, handler)` – set the handler for signal *signalnum* to the function *handler*

```
import signal

# handler for the SIGINT signal (ignore the control-c)
def quit_handler(signum, frame):
   print 'no no no..'

# handler for the alarm
def alarm_handler(signum, frame):
    print 'Wake up'

signal.signal(signal.SIGINT, quit_handler)
signal.signal(signal.SIGALRM, alarm_handler)
signal.alarm(5)

for x in range(10):
   time.sleep(10)
```