



Overview:

Understand TCP socket programming by developing a (multi-threaded) Web server. You may use Python, Java, or C++ as the programming language. The project manuals for Python and Java are attached to this document, which have provided partial code.

Submission Guide:



Name: xxx

ID: xxx

Operating system: Windows/Linux

Programming language: Python/Java/C++

Compiling instructions: xxx

Running instructions: xxx

The code must be well-documented. The grader will run your code on one host and test with a standard web browser from another host in the same network. The grader will use Windows and Linux-based platforms for testing. In case of problems or if you use an uncommon platform, you may need to bring your own computer and demonstrate the code.

Grading Criteria:

Successful compilation	10%
Correct logic	10%
Helpful comments	10%
Code clarity	10%
Handling HTML file	15%
Handling .jpeg image	15%
Handling .pdf File	15%
Handle file not found error	15%
Extra credit: multi-threaded web server	15%

Getting Started with Python

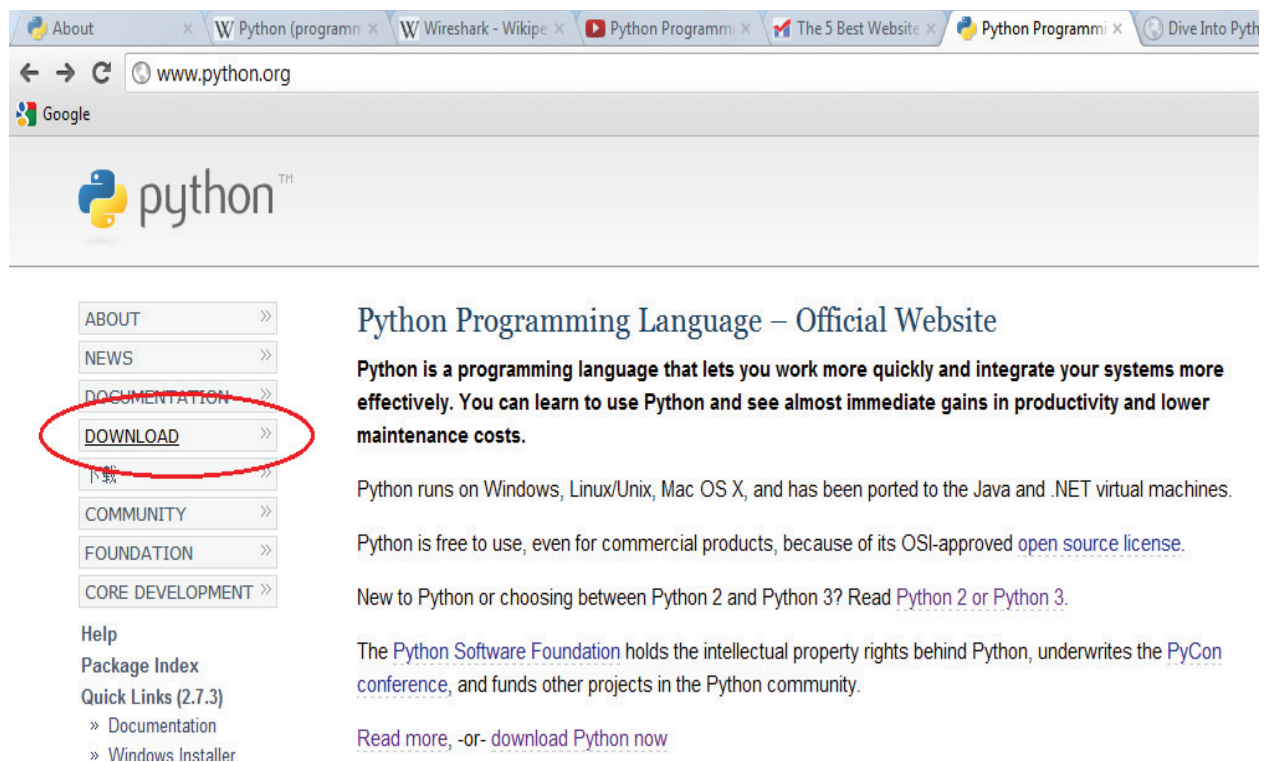
Python is a general purpose, high level programming language that is used in a variety of application domains. The Python language has a very clear and expressive syntax as well as a large and comprehensive library. Although Python is often used as a scripting language, it can also be used in a wide range of non-scripting contexts. It's available for all major Operating Systems: Windows, Linux/Unix, OS/2, Mac, Amiga, among others. Python is free to use, even for commercial products, because of its OSI-approved open source license.

Python 2 or Python 3?

Python has two standard versions, Python 2 and Python 3. The current production versions (July 2012) are Python 2.7.3 and Python 3.2.3. *Python 2.7 is the status quo.* We recommend you use Python 2.7 for completing the assignments.

Installing Python

Python can be downloaded directly from the official website <http://www.python.org/>. This is the program that is used to write all your python code. On the left side of the website there is a download section



The screenshot shows a web browser window with the URL www.python.org. The page features the Python logo and a navigation menu on the left. The 'DOWNLOAD' link in the menu is circled in red. The main content area is titled 'Python Programming Language – Official Website' and contains the following text:

Python is a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs.

Python runs on Windows, Linux/Unix, Mac OS X, and has been ported to the Java and .NET virtual machines.

Python is free to use, even for commercial products, because of its OSI-approved [open source license](#).

New to Python or choosing between Python 2 and Python 3? Read [Python 2 or Python 3](#).

The [Python Software Foundation](#) holds the intellectual property rights behind Python, underwrites the [PyCon conference](#), and funds other projects in the Python community.

[Read more, -or- download Python now](#)

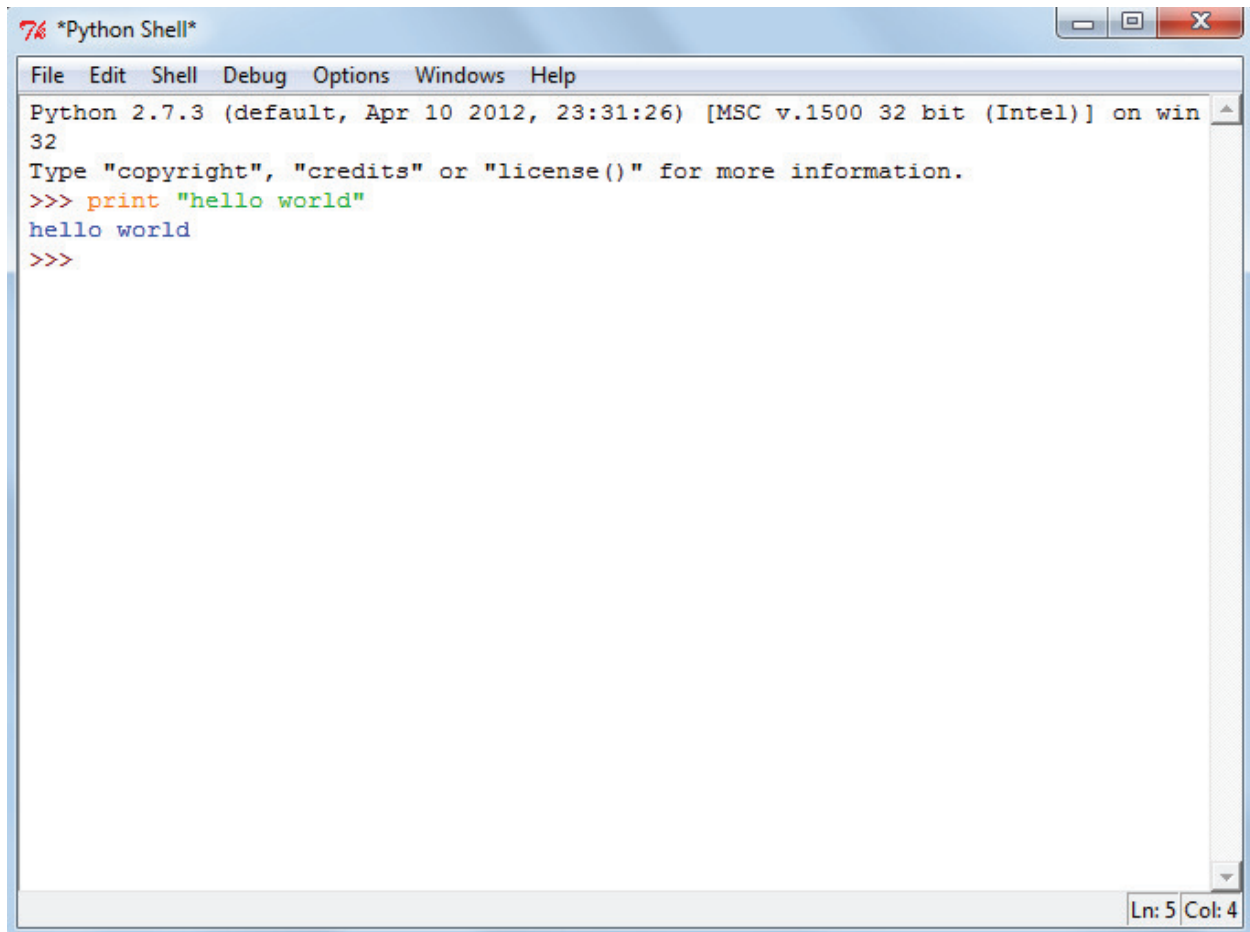
Clicking the download link will present you with two versions of Python, namely *python 2* and *python 3*; you can also choose the version specific to your operating system. Most popular Linux distributions come with Python in the default installation. Mac OS X 10.2 and later includes a command-line version of Python, although you'll probably want to install a version that includes a more Mac-like graphical interface.

Installing Python on Windows

1. Double-click the installer, Python-2.xxx.yyy.exe. The name will depend on the version of Python available when you read this.
2. Select run.
3. Step through the installer program.
4. If disk space is tight, you can deselect the HTMLHelp file, the utility scripts (Tools/), and/or the test suite (Lib/test/).
5. If you do not have administrative rights on your machine, you can select Advanced Options, then choose Non-Admin Install. This just affects where Registry entries and Start menu shortcuts are created.
6. If you see the following that means the installation is complete.



7. After the installation is complete, close the installer and select Start->Programs->Python 2.3->IDLE (Python GUI). You'll see something like the following:

A screenshot of a Windows-style application window titled "*Python Shell*". The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area contains the following text:

```
Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> print "hello world"
hello world
>>>
```

The status bar at the bottom right shows "Ln: 5 Col: 4".

Other Window Installation Options

ActiveState makes a Windows installer for Python called ActivePython, which includes a complete version of Python, an IDE with a Python-aware code editor, plus some Windows extensions for Python that allow complete access to Windows-specific services, APIs, and the Windows Registry. ActivePython is freely downloadable, although it is not open source. You recommend you use this for writing more complicated programs.

Download ActivePython from <http://www.activestate.com/Products/ActivePython/>. If you are using Windows 95, Windows 98, or Windows ME, you will also need to download and install [Windows Installer 2.0](#) before installing ActivePython.

Installing Python On Mac

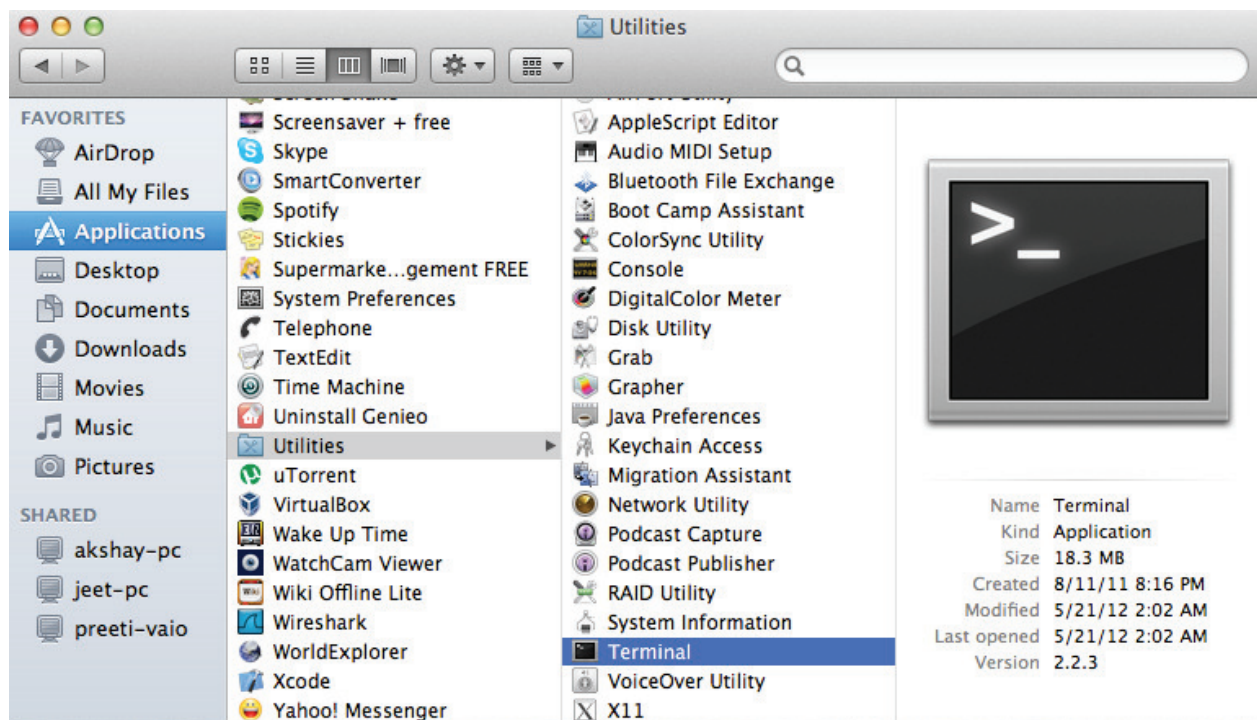
The latest version of Mac OS X, Lion, comes with a command line version preinstalled. This version is great for learning but is not good for development. The preinstalled version may be slightly out of date, it does not come with an XML parser, also Apple has made significant changes that can cause hidden bugs.

Rather than using the preinstalled version, you'll probably want to install the latest version, which also comes with a graphical interactive shell.

Running the Preinstalled Mac Version

Follow these steps in order to use the preinstalled version.

1. Go to Finder->Applications->Utilities.
2. Double click Terminal to get a command line.



3. Type **python** at the command prompt
4. Now you can try out some basic codes here

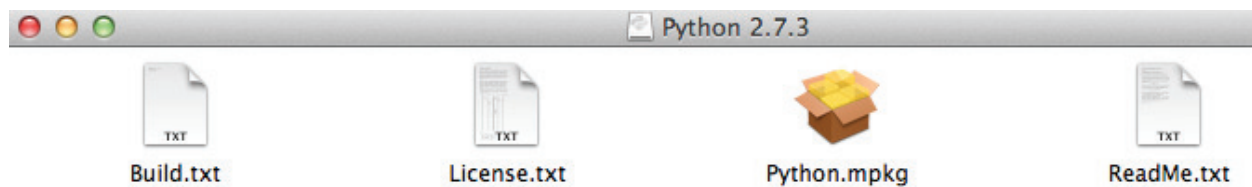
```
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr 9 2012, 20:32:06)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> print "hello world"
hello world
>>> _
```

Installing the Latest Version on the Mac

As said earlier Python comes preinstalled on Mac OS X , but due to Apple’s release cycle, its often a year or two old. The “MacPython” community highly recommends you to upgrade your Python by downloading and installing a newer version.

Go to <http://www.python.org/download/> and download the version suitable for your system from among a list of options.

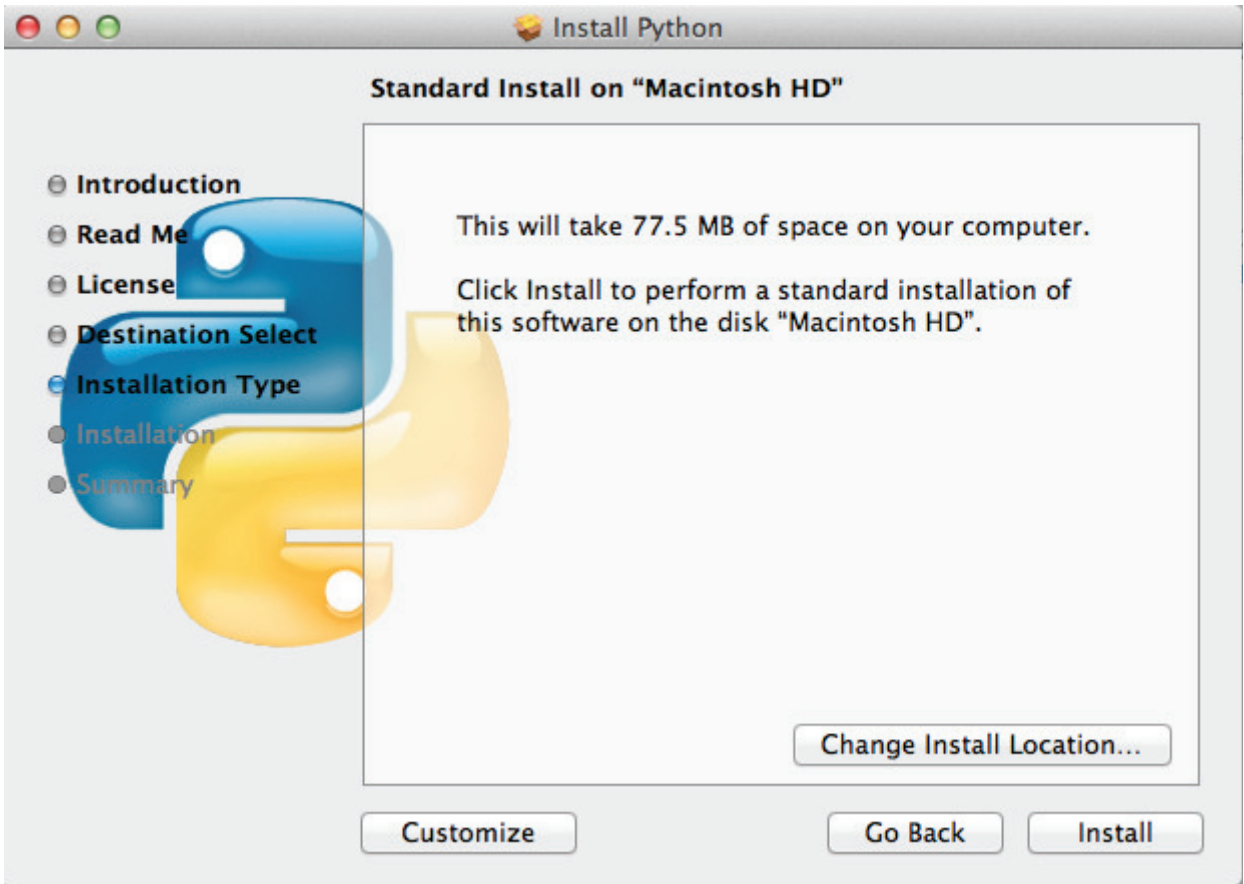
The downloaded file should look like this



Double click the “Python.mpkg” file. The installer may prompt you for your administrative username and password.

Step through the installer program.

You can choose the location at which it is to be installed.





After the installation is complete, close the installer and open the Applications folder , search for Python and you'll see the Python IDLE i.e. the standard GUI that comes with the package.

Alternative Packages for Mac OS X

[ActiveState ActivePython](#) (commercial and community versions, including scientific computing modules). ActivePython also includes a variety of modules that build on the solid core.

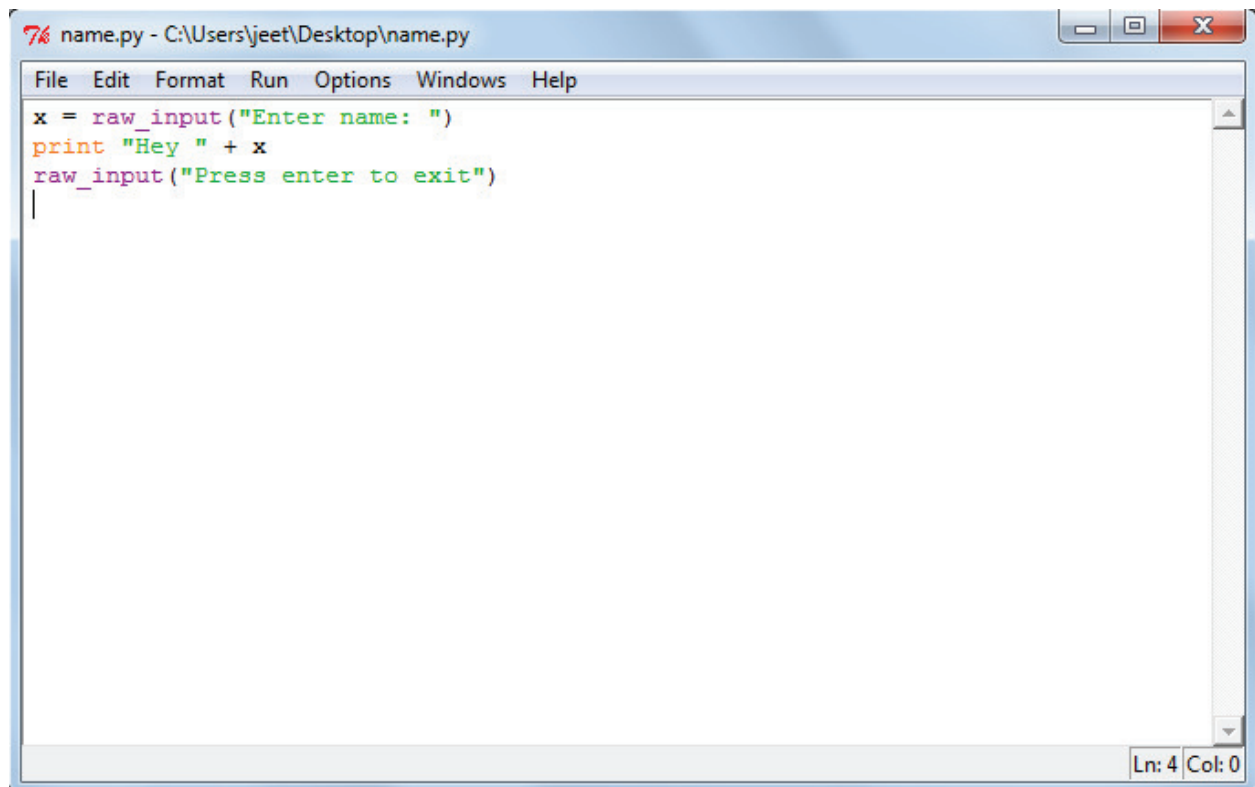
[Enthought Python Distribution](#) The Enthought Python Distribution provides scientists with a comprehensive set of tools to perform rigorous data analysis and visualization.

Example of a Basic Python Program

The interface “Idle” that we opened so far is only useful for testing out basic python commands or can otherwise be used as a calculator, it basically means the program cannot be saved this way.

To save a program and execute it we need to follow the following instruction:
On the top left corner of “Idle” select File -> New Window.

The new window that pops out will allow you to save and execute your python programs. You can write your python code in this window. Try the following:



The screenshot shows a window titled "name.py - C:\Users\jeet\Desktop\name.py". The window contains a menu bar with "File", "Edit", "Format", "Run", "Options", "Windows", and "Help". The main text area contains the following Python code:

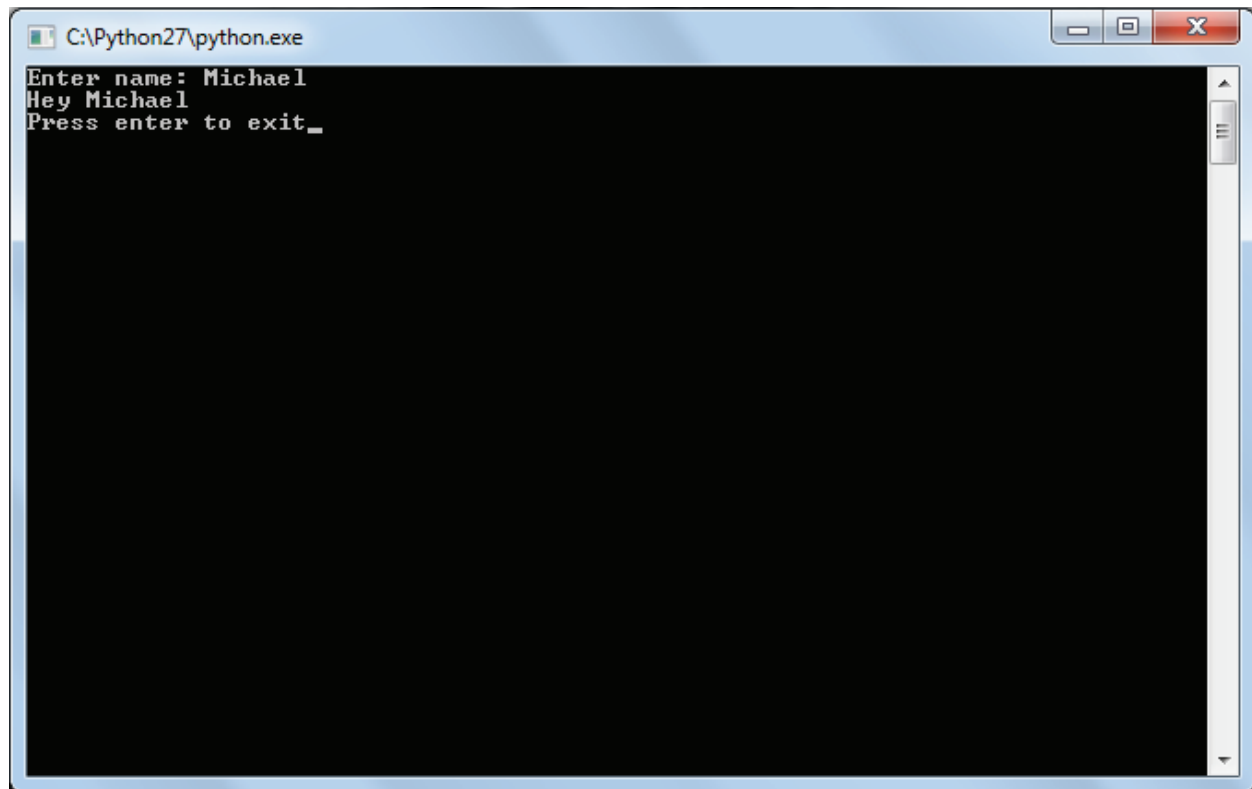
```
x = raw_input("Enter name: ")
print "Hey " + x
raw_input("Press enter to exit")
```

The status bar at the bottom right of the window shows "Ln: 4 Col: 0".

We cannot run this program without saving it. A saved python file has an icon that looks like this



You run the program by pressing F5 or Run-> Run Module. You can also run the program by simply double clicking the file icon. When you open a saved file to run the program, you should see:



```
C:\Python27\python.exe
Enter name: Michael
Hey Michael
Press enter to exit_
```

Learning Python Programming

Python is easy to learn, easy to use and very powerful. There are a lot of web resources for learning the language, most of which are entirely free. We recommend *Sturlow.com's* A Beginner's Python Tutorial:

<http://www.sturlow.com/python/>

Socket Programming Assignment 1: Web Server

In this lab, you will learn the basics of socket programming for TCP connections in Python: how to create a socket, bind it to a specific address and port, as well as send and receive a HTTP packet. You will also learn some basics of HTTP header format.

You will develop a web server that handles one HTTP request at a time. Your web server should accept and parse the HTTP request, get the requested file from the server's file system, create an HTTP response message consisting of the requested file preceded by header lines, and then send the response directly to the client. If the requested file is not present in the server, the server should send an HTTP "404 Not Found" message back to the client.

Code

Below you will find the skeleton code for the Web server. You are to complete the skeleton code. The places where you need to fill in code are marked with `#Fill in start` and `#Fill in end`. Each place may require one or more lines of code.

Running the Server

Put an HTML file (e.g., HelloWorld.html) in the same directory that the server is in. Run the server program. Determine the IP address of the host that is running the server (e.g., 128.238.251.26). From another host, open a browser and provide the corresponding URL. For example:

```
http://128.238.251.26:6789/HelloWorld.html
```

'HelloWorld.html' is the name of the file you placed in the server directory. Note also the use of the port number after the colon. You need to replace this port number with whatever port you have used in the server code. In the above example, we have used the port number 6789. The browser should then display the contents of HelloWorld.html. If you omit ":6789", the browser will assume port 80 and you will get the web page from the server only if your server is listening at port 80.

Then try to get a file that is not present at the server. You should get a "404 Not Found" message.

What to Hand in

You will hand in the complete server code along with the screen shots of your client browser, verifying that you actually receive the contents of the HTML file from the server.

Skeleton Python Code for the Web Server

```
#import socket module

from socket import *

serverSocket = socket(AF_INET, SOCK_STREAM)

#Prepare a sever socket

#Fill in start

#Fill in end

while True:

    #Establish the connection

    print 'Ready to serve...'

    connectionSocket, addr = #Fill in start #Fill in end

    try:

        message = #Fill in start #Fill in end

        filename = message.split()[1]

        f = open(filename[1:])

        outputdata = #Fill in start #Fill in end

        #Send one HTTP header line into socket

        #Fill in start

        #Fill in end

        #Send the content of the requested file to the client

        for i in range(0, len(outputdata)):

            connectionSocket.send(outputdata[i])

        connectionSocket.close()

    except IOError:

        #Send response message for file not found

        #Fill in start

        #Fill in end

        #Close client socket

        #Fill in start

        #Fill in end
```

```
serverSocket.close()
```

Optional Exercises

1. Currently, the web server handles only one HTTP request at a time. Implement a multithreaded server that is capable of serving multiple requests simultaneously. Using threading, first create a main thread in which your modified server listens for clients at a fixed port. When it receives a TCP connection request from a client, it will set up the TCP connection through another port and services the client request in a separate thread. There will be a separate TCP connection in a separate thread for each request/response pair.
2. Instead of using a browser, write your own HTTP client to test your server. Your client will connect to the server using a TCP connection, send an HTTP request to the server, and display the server response as an output. You can assume that the HTTP request sent is a GET method. The client should take command line arguments specifying the server IP address or host name, the port at which the server is listening, and the path at which the requested object is stored at the server. The following is an input command format to run the client.

```
client.py server_host server_port filename
```

Project 1 (Java Manual): Building a Multi-Threaded Web Server

This exercise is due to Kurose and Ross (the authors of the textbook). It is also available via the textbook's companion web site.

In this lab we will develop a Web server in two steps. In the end, you will have built a multi-threaded Web server that is capable of processing multiple simultaneous service requests in parallel. You should be able to demonstrate that your Web server is capable of delivering your home page to a Web browser.

We are going to implement version 1.0 of HTTP, as defined in [RFC 1945](#), where separate HTTP requests are sent for each component of the Web page. The server will be able to handle multiple simultaneous service requests in parallel. This means that the Web server is multi-threaded. In the main thread, the server listens to a fixed port. When it receives a TCP connection request, it sets up a TCP connection through another port and services the request in a separate thread. To simplify this programming task, we will develop the code in two stages. In the first stage, you will write a multi-threaded server that simply displays the contents of the HTTP request message that it receives. After this program is running properly, you will add the code required to generate an appropriate response.

As you are developing the code, you can test your server from a Web browser. But remember that you are not serving through the standard port 80, so you need to specify the port number within the URL that you give to your browser. For example, if your machine's name is `host.someschool.edu`, your server is listening to port 6789, and you want to retrieve the file `index.html`, then you would specify the following URL within the browser:

```
http://host.someschool.edu:6789/index.html
```

If you omit `":6789"`, the browser will assume port 80 which most likely will not have a server listening on it.

When the server encounters an error, it sends a response message with the appropriate HTML source so that the error information is displayed in the browser window.

Web Server in Java: Part A

In the following steps, we will go through the code for the first implementation of our Web Server. Wherever you see `"?"`, you will need to supply a missing detail.

Our first implementation of the Web server will be multi-threaded, where the processing of each incoming request will take place inside a separate thread of execution. This allows the server to service multiple clients in parallel, or to perform multiple file transfers to a single client in parallel. When we create a new thread of execution, we need to pass to the Thread's constructor an instance of some class that implements the `Runnable` interface. This is the reason that we define a separate class called `HttpRequest`. The structure of the Web server is shown below:

```
import java.io.* ;
import java.net.* ;
import java.util.* ;
```

```

public final class WebServer
{
    public static void main(String argv[]) throws Exception
    {
        . . .
    }
}

final class HttpRequest implements Runnable
{
    . . .
}

```

Normally, Web servers process service requests that they receive through well-known port number 80. You can choose any port higher than 1024, but remember to use the same port number when making requests to your Web server from your browser.

```

public static void main(String argv[]) throws Exception
{
    // Set the port number.
    int port = 6789;

    . . .
}

```

Next, we open a socket and wait for a TCP connection request. Because we will be servicing request messages indefinitely, we place the listen operation inside of an infinite loop. This means we will have to terminate the Web server by pressing ^C on the keyboard.

```

// Establish the listen socket.
?

// Process HTTP service requests in an infinite loop.
while (true) {
    // Listen for a TCP connection request.
    ?

    . . .
}

```

When a connection request is received, we create an `HttpRequest` object, passing to its constructor a reference to the `Socket` object that represents our established connection with the client.

```

// Construct an object to process the HTTP request message.
HttpRequest request = new HttpRequest( ? );

// Create a new thread to process the request.
Thread thread = new Thread(request);

// Start the thread.
thread.start();

```


In order to have the `HttpRequest` object handle the incoming HTTP service request in a separate thread, we first create a new `Thread` object, passing to its constructor a reference to the `HttpRequest` object, and then call the thread's `start()` method.

After the new thread has been created and started, execution in the main thread returns to the top of the message processing loop. The main thread will then block, waiting for another TCP connection request, while the new thread continues running. When another TCP connection request is received, the main thread goes through the same process of thread creation regardless of whether the previous thread has finished execution or is still running.

This completes the code in `main()`. For the remainder of the lab, it remains to develop the `HttpRequest` class.

We declare two variables for the `HttpRequest` class: `CRLF` and `socket`. According to the HTTP specification, we need to terminate each line of the server's response message with a carriage return (CR) and a line feed (LF), so we have defined `CRLF` as a convenience. The variable `socket` will be used to store a reference to the connection socket, which is passed to the constructor of this class. The structure of the `HttpRequest` class is shown below:

```
final class HttpRequest implements Runnable
{
    final static String CRLF = "\r\n";
    Socket socket;

    // Constructor
    public HttpRequest(Socket socket) throws Exception
    {
        this.socket = socket;
    }

    // Implement the run() method of the Runnable interface.
    public void run()
    {
        . . .
    }

    private void processRequest() throws Exception
    {
        . . .
    }
}
```

In order to pass an instance of the `HttpRequest` class to the `Thread`'s constructor, `HttpRequest` must implement the `Runnable` interface, which simply means that we must define a public method called `run()` that returns `void`. Most of the processing will take place within `processRequest()`, which is called from within `run()`.

Up until this point, we have been throwing exceptions, rather than catching them. However, we can not throw exceptions from `run()`, because we must strictly adhere to the declaration of `run()` in the `Runnable` interface, which does not throw any exceptions. We will place all the processing code in `processRequest()`, and from there, throw exceptions to `run()`. Within `run()`, we explicitly catch and handle exceptions with a `try/catch` block.

```
// Implement the run() method of the Runnable interface.
```

```

public void run()
{
    try {
        processRequest();
    } catch (Exception e) {
        System.out.println(e);
    }
}

```

Now, let's develop the code within `processRequest()`. We first obtain references to the socket's input and output streams. Then we wrap `InputStreamReader` and `BufferedReader` filters around the input stream. However, we won't wrap any filters around the output stream, because we will be writing bytes directly into the output stream.

```

private void processRequest() throws Exception
{
    // Get a reference to the socket's input and output streams.
    InputStream is = ?;
    DataOutputStream os = ?;

    // Set up input stream filters.
    ?
    BufferedReader br = ?;

    . . .
}

```

Now we are prepared to get the client's request message, which we do by reading from the socket's input stream. The `readLine()` method of the `BufferedReader` class will extract characters from the input stream until it reaches an end-of-line character, or in our case, the end-of-line character sequence CRLF.

The first item available in the input stream will be the HTTP request line. (See Section 2.2 of the textbook for a description of this and the following fields.)

```

// Get the request line of the HTTP request message.
String requestLine = ?;

// Display the request line.
System.out.println();
System.out.println(requestLine);

```

After obtaining the request line of the message header, we obtain the header lines. Since we don't know ahead of time how many header lines the client will send, we must get these lines within a looping operation.

```

// Get and display the header lines.
String headerLine = null;
while ((headerLine = br.readLine()).length() != 0) {
    System.out.println(headerLine);
}

```

We don't need the header lines, other than to print them to the screen, so we use a temporary String variable, `headerLine`, to hold a reference to their values. The loop terminates when the expression

```
(headerLine = br.readLine()).length()
```

evaluates to zero, which will occur when `headerLine` has zero length. This will happen when the empty line terminating the header lines is read. (See the HTTP Request Message diagram in Section 2.2 of the textbook)

In the next step of this lab, we will add code to analyze the client's request message and send a response. But before we do this, let's try compiling our program and testing it with a browser. Add the following lines of code to close the streams and socket connection.

```
// Close streams and socket.
os.close();
br.close();
socket.close();
```

After your program successfully compiles, run it with an available port number, and try contacting it from a browser. To do this, you should enter into the browser's address text box the IP address of your running server. For example, if your machine name is `host.someschool.edu`, and you ran the server with port number 6789, then you would specify the following URL:

```
http://host.someschool.edu:6789/
```

The server should display the contents of the HTTP request message. Check that it matches the message format shown in the HTTP Request Message diagram in Section 2.2 of the textbook.

Web Server in Java: Part B

Instead of simply terminating the thread after displaying the browser's HTTP request message, we will analyze the request and send an appropriate response. We are going to ignore the information in the header lines, and use only the file name contained in the request line. In fact, we are going to assume that the request line always specifies the GET method, and ignore the fact that the client may be sending some other type of request, such as HEAD or POST.

We extract the file name from the request line with the aid of the `StringTokenizer` class. First, we create a `StringTokenizer` object that contains the string of characters from the request line. Second, we skip over the method specification, which we have assumed to be "GET". Third, we extract the file name.

```
// Extract the filename from the request line.
StringTokenizer tokens = new StringTokenizer(requestLine);
tokens.nextToken(); // skip over the method, which should be "GET"
String fileName = tokens.nextToken();

// Prepend a "." so that file request is within the current directory.
fileName = "." + fileName;
```

Because the browser precedes the filename with a slash, we prefix a dot so that the resulting pathname starts within the current directory.

Now that we have the file name, we can open the file as the first step in sending it to the client. If the file does not exist, the `FileInputStream()` constructor will throw the `FileNotFoundException`. Instead of throwing this possible exception and terminating the thread, we will use a try/catch construction to set the boolean variable `fileExists` to false. Later in the code, we will use this flag to construct an error response message, rather than try to send a nonexistent file.

```
// Open the requested file.
FileInputStream fis = null;
boolean fileExists = true;
try {
    fis = new FileInputStream(fileName);
} catch (FileNotFoundException e) {
    fileExists = false;
}
```

There are three parts to the response message: the status line, the response headers, and the entity body. The status line and response headers are terminated by the character sequence CRLF. We are going to respond with a status line, which we store in the variable `statusLine`, and a single response header, which we store in the variable `contentTypeLine`. In the case of a request for a nonexistent file, we return *404 Not Found* in the status line of the response message, and include an error message in the form of an HTML document in the entity body.

```
// Construct the response message.
String statusLine = null;
String contentTypeLine = null;
String entityBody = null;
if (fileExists) {
    statusLine = "?";
    contentTypeLine = "Content-type: " +
        contentType(fileName) + CRLF;
} else {
    statusLine = "?";
    contentTypeLine = "?";
    entityBody = "<HTML>" +
        "<HEAD><TITLE>Not Found</TITLE></HEAD>" +
        "<BODY>Not Found</BODY></HTML>";
}
```

When the file exists, we need to determine the file's MIME type and send the appropriate MIME-type specifier. We make this determination in a separate private method called `contentType()`, which returns a string that we can include in the content type line that we are constructing.

Now we can send the status line and our single header line to the browser by writing into the socket's output stream.

```
// Send the status line.
os.writeBytes(statusLine);

// Send the content type line.
os.writeBytes(?);

// Send a blank line to indicate the end of the header lines.
os.writeBytes(CRLF);
```

Now that the status line and header line with delimiting CRLF have been placed into the output stream on their way to the browser, it is time to do the same with the entity body. If the requested file exists, we call a separate method to send the file. If the requested file does not exist, we send the HTML-encoded error message that we have prepared.

```
// Send the entity body.
if (fileExists) {
    sendBytes(fis, os);
    fis.close();
} else {
    os.writeBytes("?");
}
```

After sending the entity body, the work in this thread has finished, so we close the streams and socket before terminating.

We still need to code the two methods that we have referenced in the above code, namely, the method that determines the MIME type, `contentType()`, and the method that writes the requested file onto the socket's output stream. Let's first take a look at the code for sending the file to the client.

```
private static void sendBytes(FileInputStream fis, OutputStream os)
throws Exception
{
    // Construct a 1K buffer to hold bytes on their way to the socket.
    byte[] buffer = new byte[1024];
    int bytes = 0;

    // Copy requested file into the socket's output stream.
    while((bytes = fis.read(buffer)) != -1 ) {
        os.write(buffer, 0, bytes);
    }
}
```

Both `read()` and `write()` throw exceptions. Instead of catching these exceptions and handling them in our code, we throw them to be handled by the calling method.

The variable, `buffer`, is our intermediate storage space for bytes on their way from the file to the output stream. When we read the bytes from the `FileInputStream`, we check to see if `read()` returns minus one, indicating that the end of the file has been reached. If the end of the file has not been reached, `read()` returns the number of bytes that have been placed into `buffer`. We use the `write()` method of the `OutputStream` class to place these bytes into the output stream, passing to it the name of the byte array, `buffer`, the starting point in the array, 0, and the number of bytes in the array to write, `bytes`.

The final piece of code needed to complete the Web server is a method that will examine the extension of a file name and return a string that represents its MIME type. If the file extension is unknown, we return the type `application/octet-stream`.

```
private static String contentType(String fileName)
{
    if(fileName.endsWith(".htm") || fileName.endsWith(".html")) {
        return "text/html";
    }
}
```

```
    if(?) {  
        ?;  
    }  
    if(?) {  
        ?;  
    }  
    return "application/octet-stream";  
}
```

There is a lot missing from this method. For instance, nothing is returned for GIF or JPEG files. You may want to add the missing file types yourself, so that the components of your home page are sent with the content type correctly specified in the content type header line. For GIFs the MIME type is `image/gif` and for JPEGs it is `image/jpeg`.

This completes the code for the second phase of development of your Web server. Run the server from a directory with some sample html files and try viewing the files with a browser from a different computer in the network. Remember to include a port specifier in the URL, so that your browser doesn't try to connect to the default port 80. Once this part is done, make sure that the web server behaves reasonably when the HTTP request includes methods that are not implemented. For example, if client evokes a method other than GET, your server should send back to the client a response message with status code 501 and phrase "Not Implemented." It is enough to support `text/html` as the content type for the purpose of grading.