

CSE3214: Socket Programming

Getting Started with Python

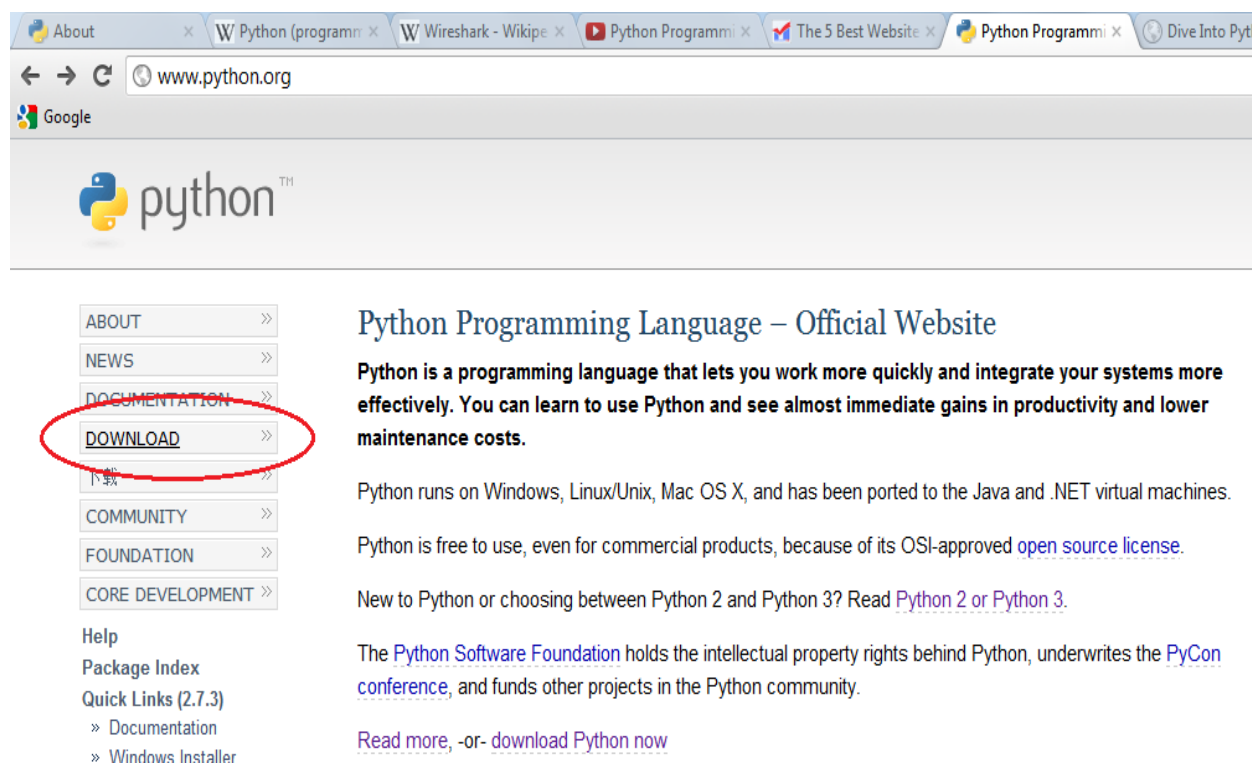
Python is a general purpose, high level programming language that is used in a variety of application domains. The Python language has a very clear and expressive syntax as well as a large and comprehensive library. Although Python is often used as a scripting language, it can also be used in a wide 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



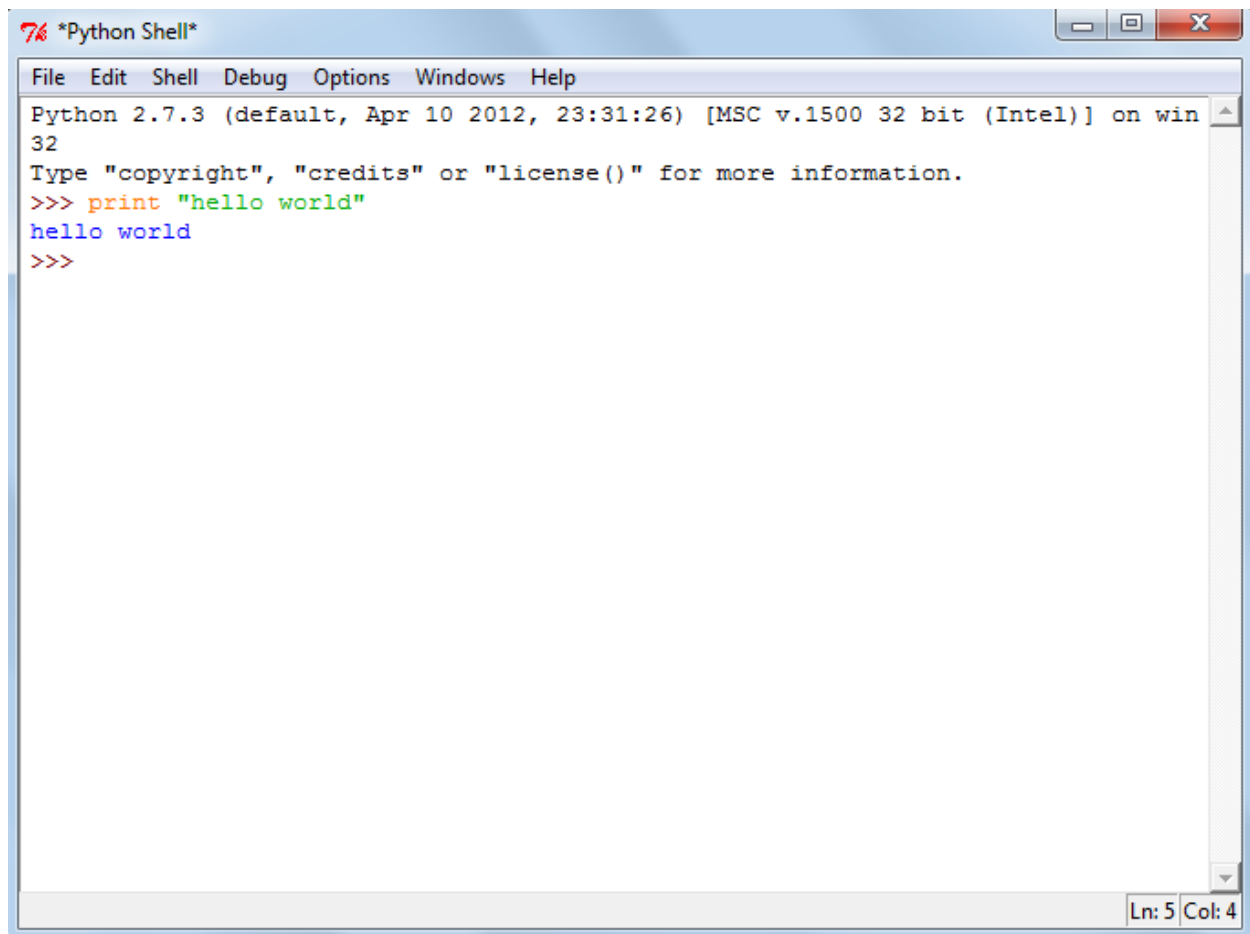
Clicking the download link will present you with two versions of Python, namely *python 2* and *python 3*; you can also choose the version specific to your operating system. Most popular Linux distributions come with Python in the default installation. Mac OS X 10.2 and later includes a command-line 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 application window titled '*Python Shell*'. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Windows', and 'Help'. The main text area displays the following text: 'Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win32', 'Type "copyright", "credits" or "license()" for more information.', '>>> print "hello world"', 'hello world', and '>>>'. 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

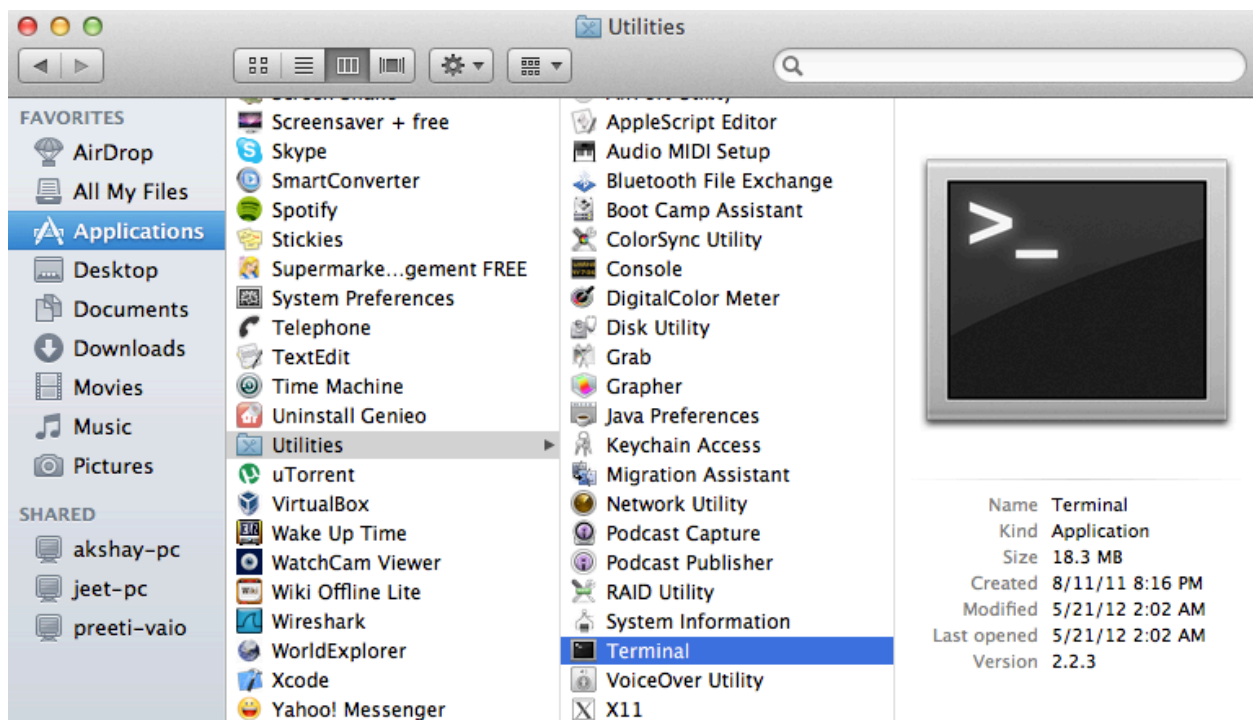
The latest version of Mac OS X, Lion, comes with a command line version preinstalled. This version is great for learning but is not good for development. The preinstalled version may be slightly out of date, it does not come with an XML parser, also Apple has made significant changes that can cause hidden 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

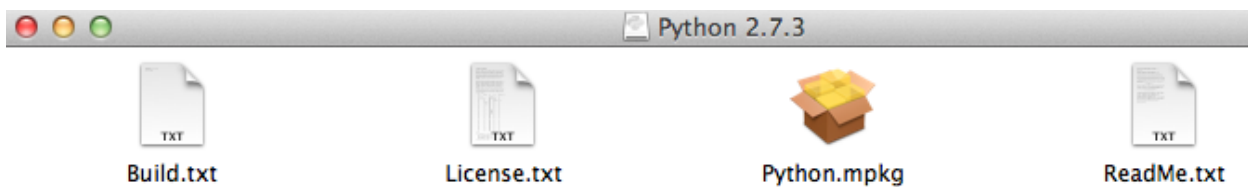
```
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012, 20:32:06)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> print "hello world"
hello world
>>> _
```

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.

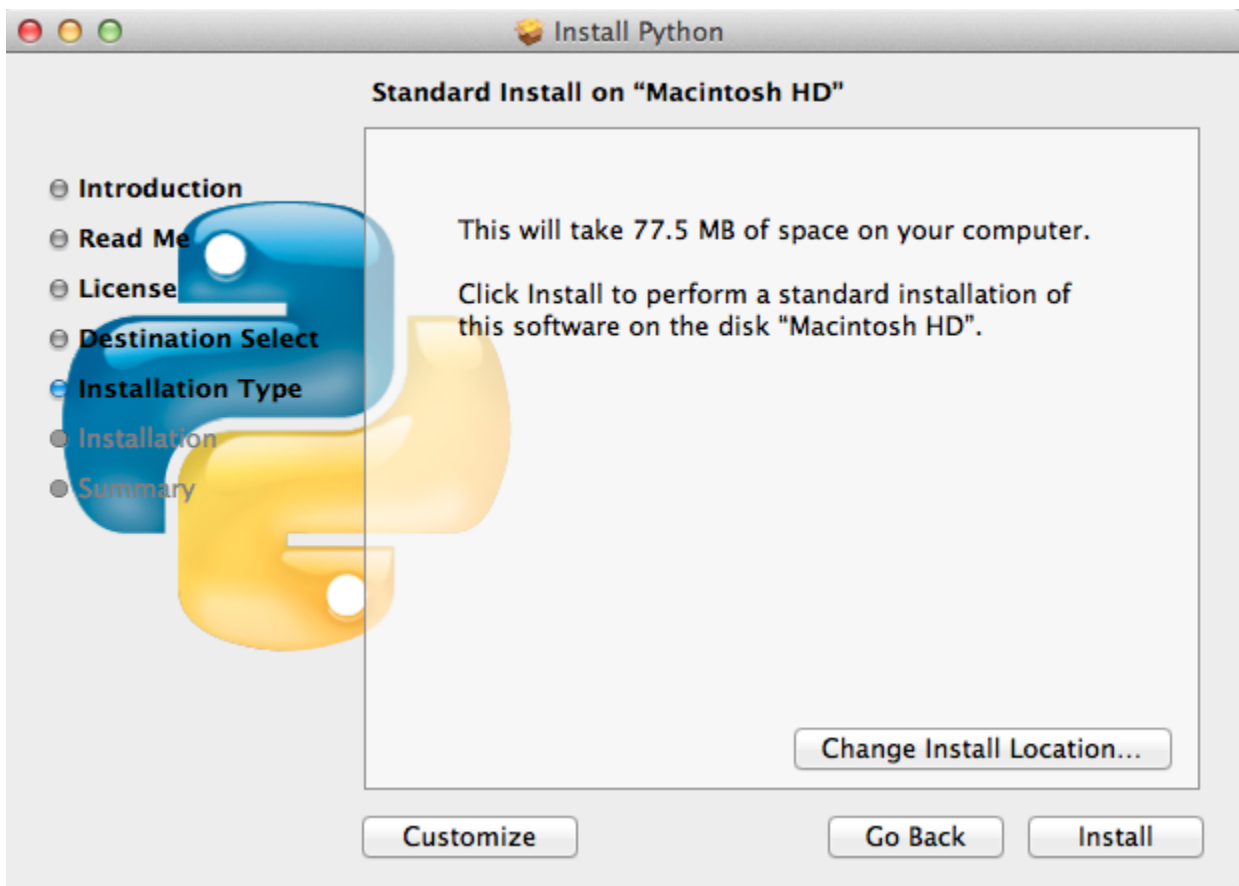
The downloaded file should look like this

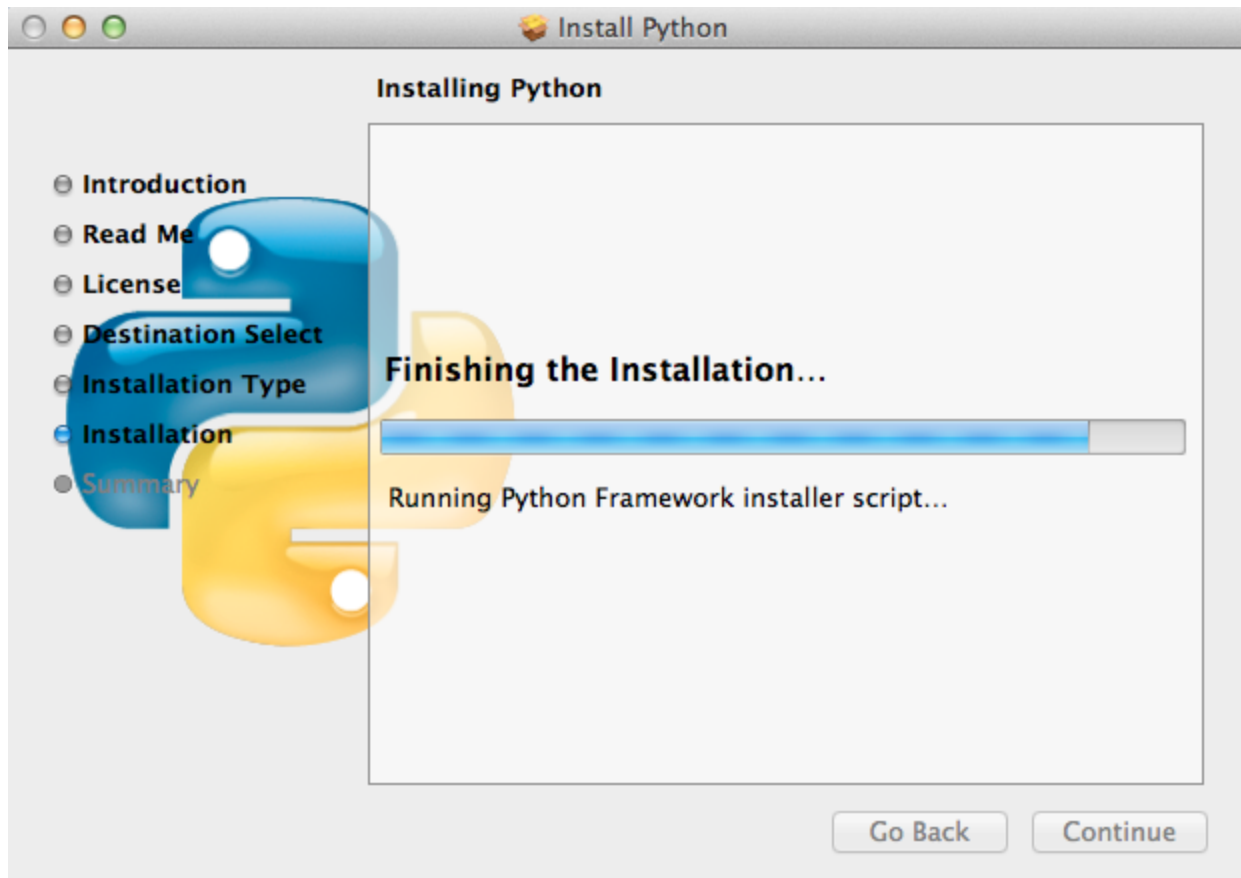


Double click the "Python.mpkg" file. The installer may prompt you for your administrative username and password.

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.

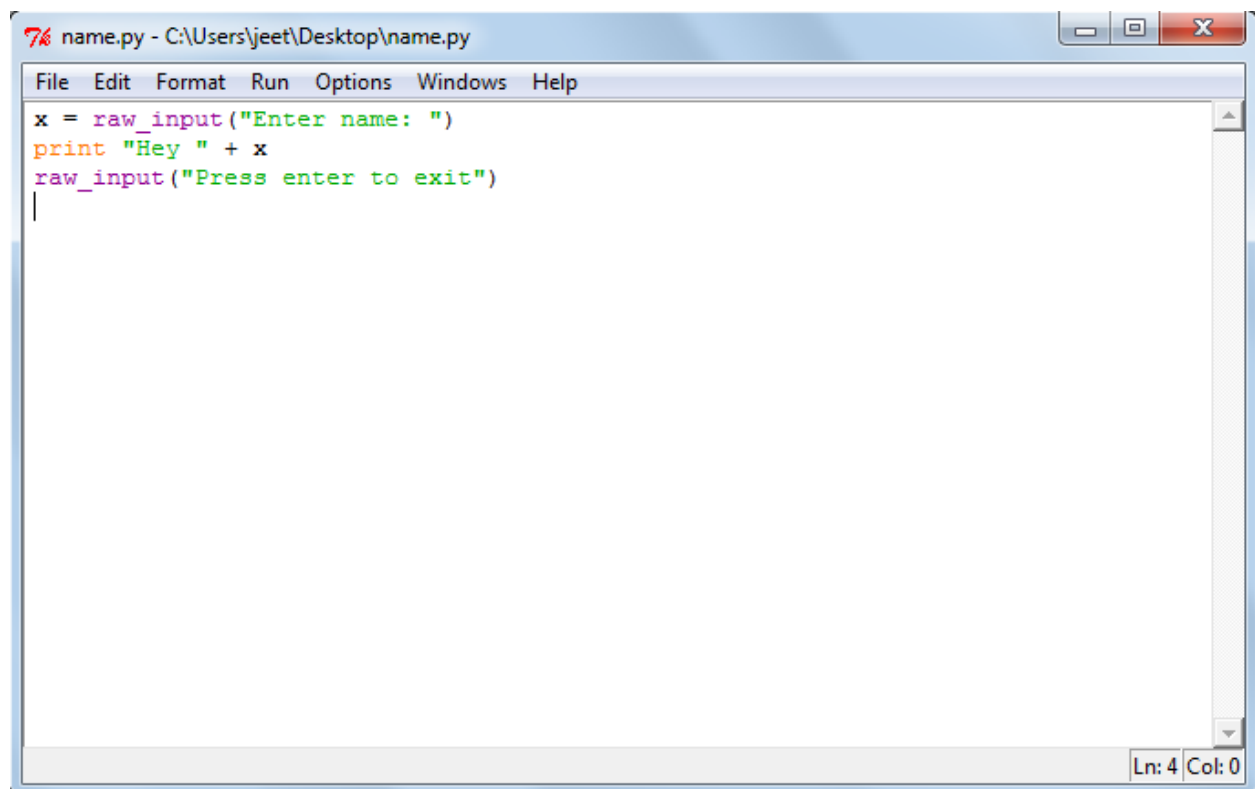
[Enthought Python Distribution](#) The Enthought Python Distribution provides scientists with a comprehensive set of tools to perform rigorous data analysis and visualization.

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:

A screenshot of the Python Idle IDE. The window title is '7% name.py - C:\Users\jeet\Desktop\name.py'. The menu bar includes File, Edit, Format, Run, Options, Windows, and Help. The code editor contains the following Python code:

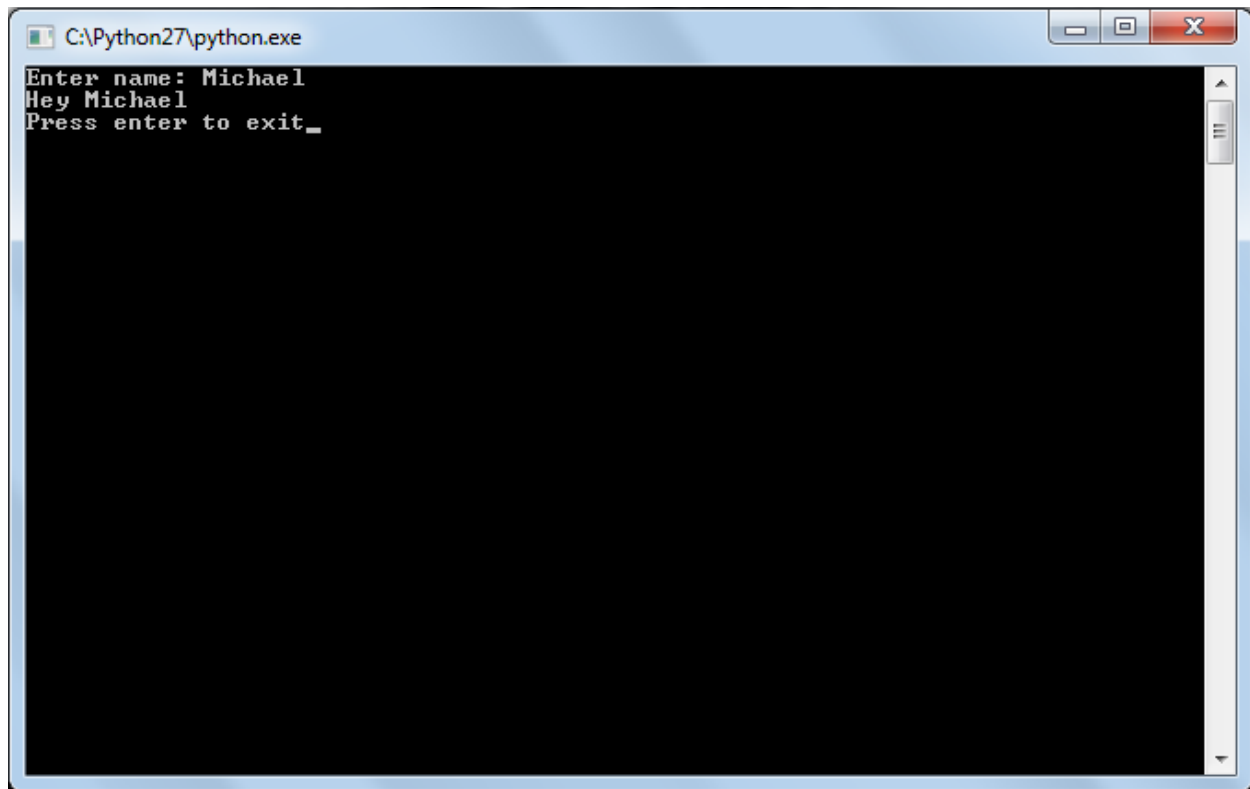
```
x = raw_input("Enter name: ")
print "Hey " + x
raw_input("Press enter to exit")
```

The status bar at the bottom right 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:



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 *Sthurlow.com's* A Beginner's Python Tutorial:

<http://www.sthurlow.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
```

Socket Programming Assignment 2: UDP

In this lab, you will learn the basics of socket programming for UDP in Python. You will learn how to send and receive datagram packets using UDP sockets and also, how to set a proper socket timeout. Throughout the lab, you will gain familiarity with a Ping application and its usefulness in computing statistics such as packet loss rate.

You will first study a simple Internet ping server written in the Python, and implement a corresponding client. The functionality provided by these programs is similar to the functionality provided by standard ping programs available in modern operating systems. However, these programs use a simpler protocol, UDP, rather than the standard Internet Control Message Protocol (ICMP) to communicate with each other. The ping protocol allows a client machine to send a packet of data to a remote machine, and have the remote machine return the data back to the client unchanged (an action referred to as echoing). Among other uses, the ping protocol allows hosts to determine round-trip times to other machines.

You are given the complete code for the Ping server below. Your task is to write the Ping client.

Server Code

The following code fully implements a ping server. You need to compile and run this code before running your client program. *You do not need to modify this code.*

In this server code, 30% of the client's packets are simulated to be lost. You should study this code carefully, as it will help you write your ping client.

```
# UDPPingerServer.py
# We will need the following module to generate randomized lost packets
import random
from socket import *

# Create a UDP socket
# Notice the use of SOCK_DGRAM for UDP packets
serverSocket = socket(AF_INET, SOCK_DGRAM)
# Assign IP address and port number to socket
serverSocket.bind('', 12000)

while True:
    # Generate random number in the range of 0 to 10
    rand = random.randint(0, 10)
    # Receive the client packet along with the address it is coming from
    message, address = serverSocket.recvfrom(1024)
    # Capitalize the message from the client
    message = message.upper()
```

```
# If rand is less is than 4, we consider the packet lost and do not respond
if rand < 4:
    continue
# Otherwise, the server responds
serverSocket.sendto(message, address)
```

The server sits in an infinite loop listening for incoming UDP packets. When a packet comes in and if a randomized integer is greater than or equal to 4, the server simply capitalizes the encapsulated data and sends it back to the client.

Packet Loss

UDP provides applications with an unreliable transport service. Messages may get lost in the network due to router queue overflows, faulty hardware or some other reasons. Because packet loss is rare or even non-existent in typical campus networks, the server in this lab injects artificial loss to simulate the effects of network packet loss. The server creates a variable randomized integer which determines whether a particular incoming packet is lost or not.

Client Code

You need to implement the following client program.

The client should send 10 pings to the server. Because UDP is an unreliable protocol, a packet sent from the client to the server may be lost in the network, or vice versa. For this reason, the client cannot wait indefinitely for a reply to a ping message. You should get the client wait up to one second for a reply; if no reply is received within one second, your client program should assume that the packet was lost during transmission across the network. You will need to look up the Python documentation to find out how to set the timeout value on a datagram socket.

Specifically, your client program should

- (1) send the ping message using UDP (Note: Unlike TCP, you do not need to establish a connection first, since UDP is a connectionless protocol.)
- (2) print the response message from server, if any
- (3) calculate and print the round trip time (RTT), in seconds, of each packet, if server responses
- (4) otherwise, print "Request timed out"

During development, you should run the `UDPPingerServer.py` on your machine, and test your client by sending packets to *localhost* (or, 127.0.0.1). After you have fully debugged your code, you should see how your application communicates across the network with the ping server and ping client running on different machines.

Message Format

The ping messages in this lab are formatted in a simple way. The client message is one line, consisting of ASCII characters in the following format:

Ping *sequence_number* *time*

where *sequence_number* starts at 1 and progresses to 10 for each successive ping message sent by the

client, and *time* is the time when the client sends the message.

What to Hand in

You will hand in the complete client code and screenshots at the client verifying that your ping program works as required.

Optional Exercises

1. Currently, the program calculates the round-trip time for each packet and prints it out individually. Modify this to correspond to the way the standard ping program works. You will need to report the minimum, maximum, and average RTTs at the end of all pings from the client. In addition, calculate the packet loss rate (in percentage).
2. Another similar application to the UDP Ping would be the UDP Heartbeat. The Heartbeat can be used to check if an application is up and running and to report one-way packet loss. The client sends a sequence number and current timestamp in the UDP packet to the server, which is listening for the Heartbeat (i.e., the UDP packets) of the client. Upon receiving the packets, the server calculates the time difference and reports any lost packets. If the Heartbeat packets are missing for some specified period of time, we can assume that the client application has stopped. Implement the UDP Heartbeat (both client and server). You will need to modify the given `UDPPingerServer.py`, and your UDP ping client.

Socket Programming Assignment 3: SMTP

By the end of this lab, you will have acquired a better understanding of SMTP protocol. You will also gain experience in implementing a standard protocol using Python.

Your task is to develop a simple mail client that sends email to any recipient. Your client will need to connect to a mail server, dialogue with the mail server using the SMTP protocol, and send an email message to the mail server. Python provides a module, called `smtplib`, which has built in methods to send mail using SMTP protocol. However, we will not be using this module in this lab, because it hide the details of SMTP and socket programming.

In order to limit spam, some mail servers do not accept TCP connection from arbitrary sources. For the experiment described below, you may want to try connecting both to your university mail server and to a popular Webmail server, such as a AOL mail server. You may also try making your connection both from your home and from your university campus.

Code

Below you will find the skeleton code for the client. 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.

Additional Notes

In some cases, the receiving mail server might classify your e-mail as junk. Make sure you check the junk/spam folder when you look for the e-mail sent from your client.

What to Hand in

In your submission, you are to provide the complete code for your SMTP mail client as well as a screenshot showing that you indeed receive the e-mail message.

Skeleton Python Code for the Mail Client

```
from socket import *
msg = "\r\n I love computer networks!"
endmsg = "\r\n.\r\n"

# Choose a mail server (e.g. Google mail server) and call it mailserver
mailserver = #Fill in start    #Fill in end

# Create socket called clientSocket and establish a TCP connection with mailserver
#Fill in start


#Fill in end
recv = clientSocket.recv(1024)
print recv
if recv[:3] != '220':
    print '220 reply not received from server.'

# Send HELO command and print server response.
heloCommand = 'HELO Alice\r\n'
clientSocket.send(heloCommand)
recv1 = clientSocket.recv(1024)
print recv1
if recv1[:3] != '250':
    print '250 reply not received from server.'

# Send MAIL FROM command and print server response.
# Fill in start


# Fill in end

# Send RCPT TO command and print server response.
# Fill in start


# Fill in end

# Send DATA command and print server response.
# Fill in start


# Fill in end

# Send message data.
# Fill in start


# Fill in end
```

```
# Message ends with a single period.  
# Fill in start  
  
# Fill in end  
  
# Send QUIT command and get server response.  
# Fill in start  
  
# Fill in end
```

Optional Exercises

1. Mail servers like Google mail (address: smtp.gmail.com, port: 587) requires your client to add a Transport Layer Security (TLS) or Secure Sockets Layer (SSL) for authentication and security reasons, before you send MAIL FROM command. Add TLS/SSL commands to your existing ones and implement your client using Google mail server at above address and port.
2. Your current SMTP mail client only handles sending text messages in the email body. Modify your client such that it can send emails with both text and images.

Socket Programming Assignment 4: ICMP Pinger

In this lab, you will gain a better understanding of Internet Control Message Protocol (ICMP). You will learn to implement a Ping application using ICMP request and reply messages.

Ping is a computer network application used to test whether a particular host is reachable across an IP network. It is also used to self-test the network interface card of the computer or as a latency test. It works by sending ICMP “echo reply” packets to the target host and listening for ICMP “echo reply” replies. The “echo reply” is sometimes called a pong. Ping measures the round-trip time, records packet loss, and prints a statistical summary of the echo reply packets received (the minimum, maximum, and the mean of the round-trip times and in some versions the standard deviation of the mean).

Your task is to develop your own Ping application in Python. Your application will use ICMP but, in order to keep it simple, will not exactly follow the official specification in RFC 1739. Note that you will only need to write the client side of the program, as the functionality needed on the server side is built into almost all operating systems.

You should complete the Ping application so that it sends ping requests to a specified host separated by approximately one second. Each message contains a payload of data that includes a timestamp. After sending each packet, the application waits up to one second to receive a reply. If one second goes by without a reply from the server, then the client assumes that either the ping packet or the pong packet was lost in the network (or that the server is down).

Code

Below you will find the skeleton code for the client. 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.

Additional Notes

1. In “receiveOnePing” method, you need to receive the structure ICMP_ECHO_REPLY and fetch the information you need, such as checksum, sequence number, time to live (TTL), etc. Study the “sendOnePing” method before trying to complete the “receiveOnePing” method.
2. You do not need to be concerned about the checksum, as it is already given in the code.
3. This lab requires the use of raw sockets. In some operating systems, you may need administrator/root privileges to be able to run your Pinger program.
4. See the end of this programming exercise for more information on ICMP.

Testing the Pinger

First, test your client by sending packets to localhost, that is, 127.0.0.1.

Then, you should see how your Pinger application communicates across the network by pinging servers in different continents.

What to Hand in

You will hand in the complete client code and screenshots of your Pinger output for four target hosts, each on a different continent.

Skeleton Python Code for the ICMP Pinger

```
from socket import *
import os
import sys
import struct
import time
import select
import binascii

ICMP_ECHO_REQUEST = 8

def checksum(str):
    csum = 0
    countTo = (len(str) / 2) * 2

    count = 0
    while count < countTo:
        thisVal = ord(str[count+1]) * 256 + ord(str[count])
        csum = csum + thisVal
        csum = csum & 0xffffffffL
        count = count + 2

    if countTo < len(str):
        csum = csum + ord(str[len(str) - 1])
        csum = csum & 0xffffffffL

    csum = (csum >> 16) + (csum & 0xffff)
    csum = csum + (csum >> 16)
    answer = ~csum
    answer = answer & 0xffff
    answer = answer >> 8 | (answer << 8 & 0xff00)
    return answer

def receiveOnePing(mySocket, ID, timeout, destAddr):
    timeLeft = timeout

    while 1:
        startedSelect = time.time()
        whatReady = select.select([mySocket], [], [], timeLeft)
        howLongInSelect = (time.time() - startedSelect)
```

```

        if whatReady[0] == []: # Timeout
            return "Request timed out."

        timeReceived = time.time()
        recPacket, addr = mySocket.recvfrom(1024)

        #Fill in start

        #Fetch the ICMP header from the IP packet

        #Fill in end

        timeLeft = timeLeft - howLongInSelect
        if timeLeft <= 0:
            return "Request timed out."

def sendOnePing(mySocket, destAddr, ID):
    # Header is type (8), code (8), checksum (16), id (16), sequence (16)

    myChecksum = 0
    # Make a dummy header with a 0 checksum.
    # struct -- Interpret strings as packed binary data
    header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
    data = struct.pack("d", time.time())
    # Calculate the checksum on the data and the dummy header.
    myChecksum = checksum(header + data)

    # Get the right checksum, and put in the header
    if sys.platform == 'darwin':
        myChecksum = socket.htons(myChecksum) & 0xffff
        #Convert 16-bit integers from host to network byte order.
    else:
        myChecksum = socket.htons(myChecksum)

    header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, myChecksum, ID, 1)
    packet = header + data

    mySocket.sendto(packet, (destAddr, 1)) # AF_INET address must be tuple, not str
    #Both LISTS and TUPLES consist of a number of objects
    #which can be referenced by their position number within the object

def doOnePing(destAddr, timeout):
    icmp = socket.getprotobyname("icmp")

```

#SOCK_RAW is a powerful socket type. For more details see:
http://sock-raw.org/papers/sock_raw

```
#Fill in start
```

```
#Create Socket here
```

```
#Fill in end
```

```
myID = os.getpid() & 0xFFFF #Return the current process i
sendOnePing(mySocket, destAddr, myID)
delay = receiveOnePing(mySocket, myID, timeout, destAddr)

mySocket.close()
return delay
```

```
def ping(host, timeout=1):
    #timeout=1 means: If one second goes by without a reply from the server,
    #the client assumes that either the client's ping or the server's pong is lost
    dest = socket.gethostbyname(host)
    print "Pinging " + dest + " using Python:"
    print ""
    #Send ping requests to a server separated by approximately one second
    while 1 :
        delay = doOnePing(dest, timeout)
        print delay
        time.sleep(1)# one second
    return delay

ping("www.poly.edu")
```

Optional Exercises

1. Currently, the program calculates the round-trip time for each packet and prints it out individually. Modify this to correspond to the way the standard ping program works. You will need to report the minimum, maximum, and average RTTs at the end of all pings from the client. In addition, calculate the packet loss rate (in percentage).
2. Your program can only detect timeouts in receiving ICMP echo responses. Modify the Pinger program to parse the ICMP response error codes and display the corresponding error results to the user. Examples of ICMP response error codes are 0: Destination Network Unreachable, 1: Destination Host Unreachable.

Internet Control Message Protocol (ICMP)

ICMP Header

The ICMP header starts after bit 160 of the IP header (unless IP options are used).

Bits	160-167	168-175	176-183	184-191
160	Type	Code	Checksum	
192	ID		Sequence	

- **Type** - ICMP type.
- **Code** - Subtype to the given ICMP type.
- **Checksum** - Error checking data calculated from the ICMP header + data, with value 0 for this field.
- **ID** - An ID value, should be returned in the case of echo reply.
- **Sequence** - A sequence value, should be returned in the case of echo reply.

Echo Request

The echo request is an ICMP message whose data is expected to be received back in an echo reply ("pong"). The host must respond to all echo requests with an echo reply containing the exact data received in the request message.

- Type must be set to 8.
- Code must be set to 0.
- The Identifier and Sequence Number can be used by the client to match the reply with the request that caused the reply. In practice, most Linux systems use a unique identifier for every ping process, and sequence number is an increasing number within that process. Windows uses a fixed identifier, which varies between Windows versions, and a sequence number that is only reset at boot time.
- The data received by the echo request must be entirely included in the echo reply.

Echo Reply

The echo reply is an ICMP message generated in response to an echo request, and is mandatory for all hosts and routers.

- Type and code must be set to 0.
- The identifier and sequence number can be used by the client to determine which echo requests are associated with the echo replies.
- The data received in the echo request must be entirely included in the echo reply.