

MAWLANA BHASHANI SCIENCE AND TECHNOLOGY UNIVERSITY

Santosh, Tangail – 1902



Course Title : Computer Networks Lab

Lab Report Name : Programming with Python

Lab Report No : 02

Submitted by,

Name: Tazneen Akter &

Jannatul Ferdush Dhina

ID: IT-18056 & 18012

Session: 2017-18

Dept. of ICT, MBSTU

Submitted to,

NAZRUL ISLAM

Assistant Professor

Dept. of ICT, MBSTU

Theory:

Python functions: Functions are reusable pieces of programs. They allow you to give a name to a block of statements, allowing you to run that block using the specified name anywhere in the program and any number of times. This is known as calling the function.

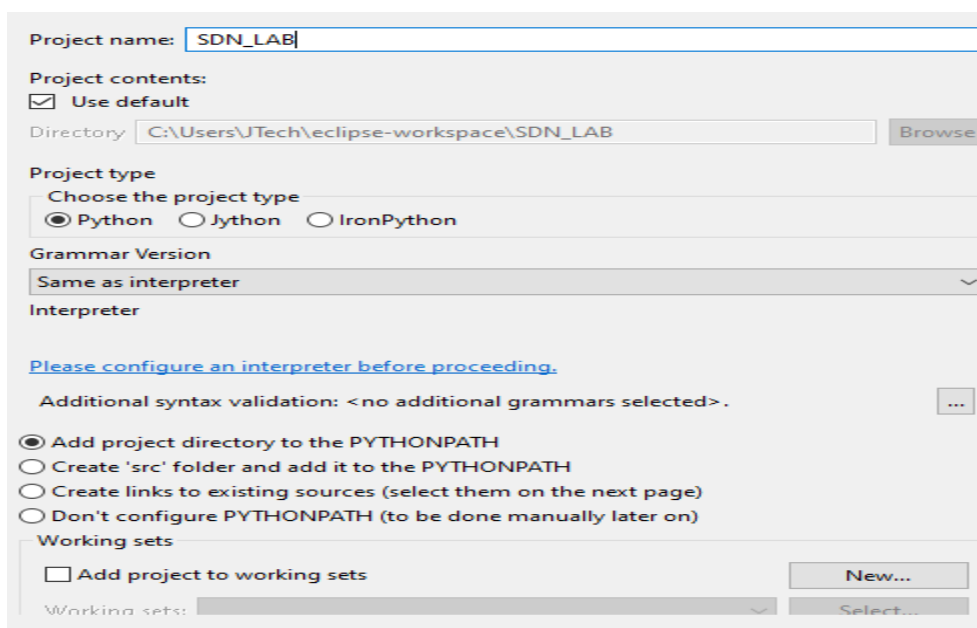
Local Variables: Variables declared inside a function definition are not related in any way to other variables with the same names used outside the function (variable names are local to the function). This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

The global statement: Variables defined at the top level of the program are intended global. Global variables are intended to be used in any functions or classes). Global statement allows defining global variables inside functions as well.

Modules: Modules allow reusing a number of functions in other programs.

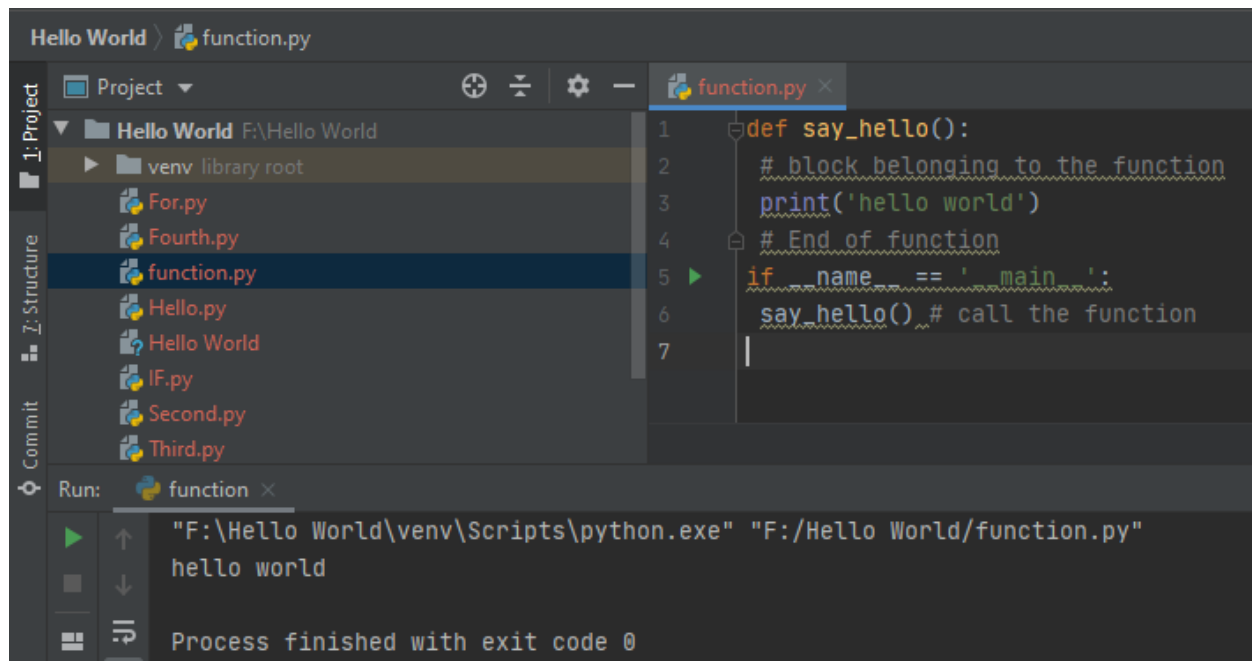
Exercises:

Exercise 4.1.1: Create a python project using with SDN_LAB



The screenshot shows the 'New Project' dialog in the Eclipse IDE. The 'Project name' field is filled with 'SDN_LAB'. Under 'Project contents', the 'Use default' checkbox is checked. The 'Directory' field shows the path 'C:\Users\JTech\eclipse-workspace\SDN_LAB' with a 'Browse' button. In the 'Project type' section, 'Python' is selected with a radio button. The 'Grammar Version' is set to 'Same as interpreter'. Below this, a message states 'Please configure an interpreter before proceeding.' with a link to the configuration page. The 'Additional syntax validation' section shows '<no additional grammars selected>'. Under the 'Interpreter' section, four options are listed: 'Add project directory to the PYTHONPATH' (selected), 'Create 'src' folder and add it to the PYTHONPATH', 'Create links to existing sources (select them on the next page)', and 'Don't configure PYTHONPATH (to be done manually later on)'. At the bottom, the 'Working sets' section has an unchecked checkbox 'Add project to working sets' and a 'New...' button. A 'Working sets:' dropdown and a 'Select...' button are also visible.

Exercise 4.1.2: Python function (save as function.py)

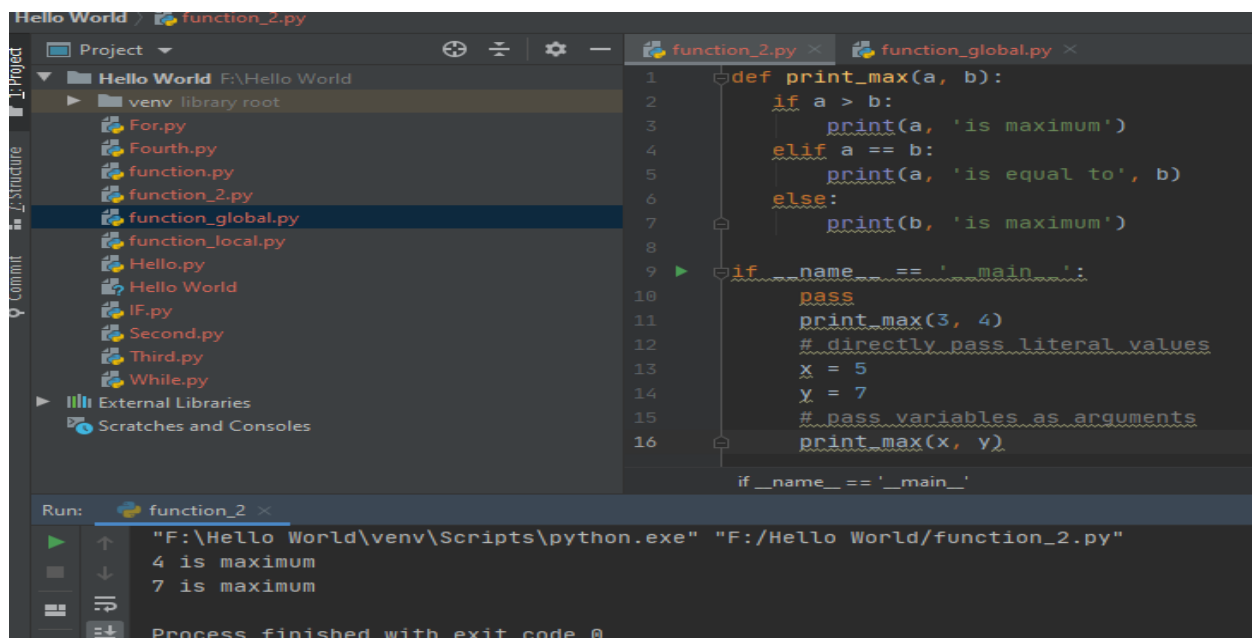


The screenshot shows an IDE with a project named 'Hello World'. The file explorer on the left lists several Python files, with 'function.py' selected. The main editor displays the code for 'function.py':

```
1 def say_hello():
2     # block belonging to the function
3     print('hello world')
4     # End of function
5 if __name__ == '__main__':
6     say_hello() # call the function
7
```

The Run console at the bottom shows the command executed: `"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/function.py"`, the output: `hello world`, and the status: `Process finished with exit code 0`.

Exercise 4.1.3: Python function (save as function_2.py)



The screenshot shows an IDE with a project named 'Hello World'. The file explorer on the left lists several Python files, with 'function_2.py' selected. The main editor displays the code for 'function_2.py':

```
1 def print_max(a, b):
2     if a > b:
3         print(a, 'is maximum')
4     elif a == b:
5         print(a, 'is equal to', b)
6     else:
7         print(b, 'is maximum')
8
9 if __name__ == '__main__':
10     pass
11     print_max(3, 4)
12     # directly pass literal values
13     x = 5
14     y = 7
15     # pass variables as arguments
16     print_max(x, y)
```

The Run console at the bottom shows the command executed: `"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/function_2.py"`, the output: `4 is maximum` and `7 is maximum`, and the status: `Process finished with exit code 0`.

Exercise 4.1.4: Local variable (save as function_local.py)

```
1 x = 50
2 def func(x):
3     print('x is', x)
4     x = 2
5     print('Changed local x to', x)
6
7 if __name__ == '__main__':
8     func(x)
9     print('x is still', x)
```

Run: function_local ×

```
"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/function_local.py"
x is 50
Changed local x to
x is still 50
Process finished with exit code 0
```

Exercise 4.1.5: Global variable (save as function_global.py)

```
1 x = 50
2 def func():
3     global x
4     print('x is', x)
5     x = 2
6     print('Changed global x to', x)
7
8 if __name__ == '__main__':
9     func()
10    print('Value of x is', x)
```

Run: function_global ×

```
"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/function_global.py"
x is 50
Changed global x to
Value of x is 2
Process finished with exit code 0
```

Exercise 4.1.6: Python modules

```
function_2.py x mymodule.py x function_global.py x
1 def say_hi():
2     print('Hi, this is mymodule speaking.')
3     __version__ = '0.1'
```

```
Project venv library root
  For.py
  Fourth.py
  function.py
  function_2.py
  function_global.py
  function_local.py
  Hello.py
  Hello World
  IF.py
  module_demo.py
  mymodule.py

1 import mymodule
2 if __name__ == '__main__':
3     mymodule.say_hi()
4     print('Version', mymodule.__version__)
5
6     from mymodule import say_hi, __version__
7
8 if __name__ == '__main__':
9     say_hi()
10    print('Version', __version__)

if __name__ == '__main__': if __name__ == '__main__'
```

```
Run: module_demo x
"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/module_demo.py"
Hi, this is mymodule speaking.
Version 0.1
Hi, this is mymodule speaking.
Version 0.1
Process finished with exit code 0
```

Exercise 4.2.1: Printing your machine's name and IPv4 address

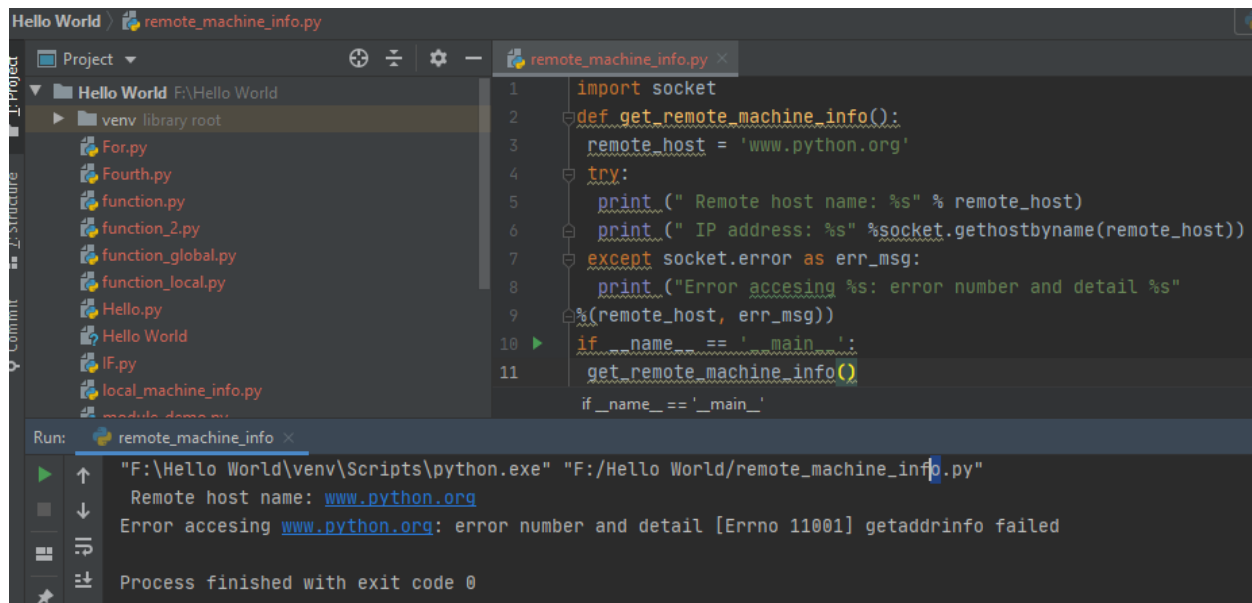
```
Hello World local_machine_info.py
Project venv library root
  For.py
  Fourth.py
  function.py
  function_2.py
  function_global.py
  function_local.py
  Hello.py
  Hello World
  IF.py
  local_machine_info.py

1 import socket
2 def print_machine_info():
3     host_name = socket.gethostname()
4     ip_address = socket.gethostbyname(host_name)
5     print(" Host name: %s" % host_name)
6     print(" IP address: %s" % ip_address)
7 if __name__ == '__main__':
8     print_machine_info()

if __name__ == '__main__'
```

```
Run: local_machine_info x
"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/local_machine_info.py"
Host name: DESKTOP-FB5S452
IP address: 127.0.0.1
Process finished with exit code 0
```

Exercise 4.2.2: Retrieving a remote machine's IP address



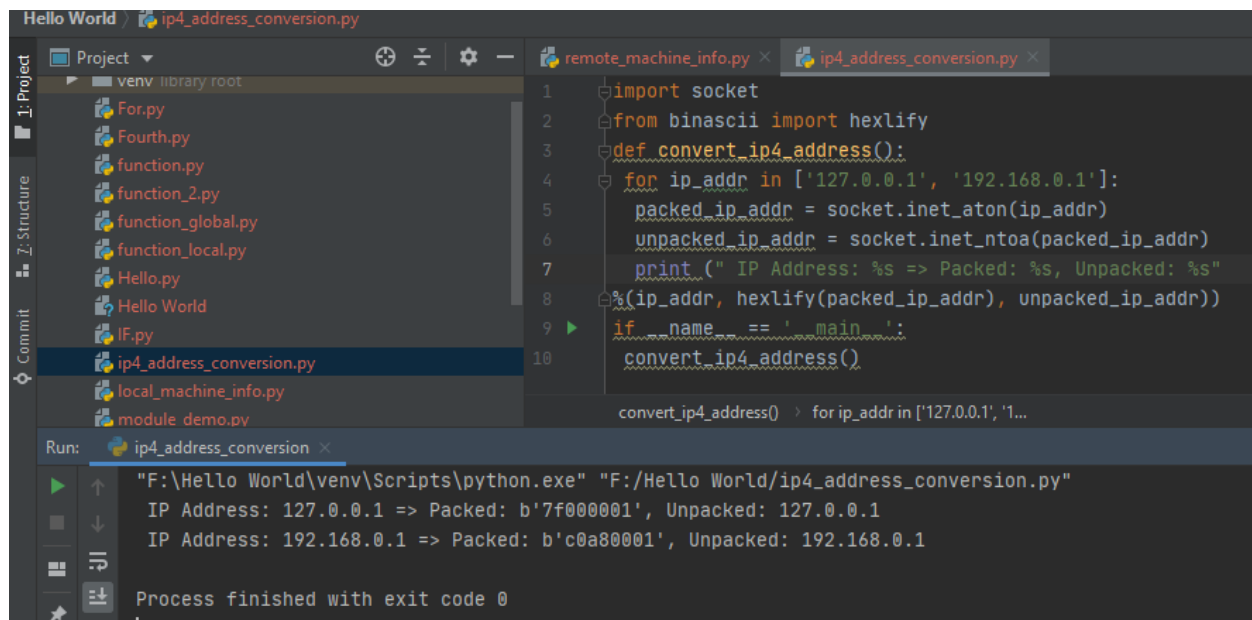
The screenshot shows an IDE with a project named 'Hello World'. The file explorer on the left lists several Python files, including 'remote_machine_info.py'. The main editor displays the code for 'remote_machine_info.py':

```
1 import socket
2 def get_remote_machine_info():
3     remote_host = 'www.python.org'
4     try:
5         print(" Remote host name: %s" % remote_host)
6         print(" IP address: %s" % socket.gethostbyname(remote_host))
7     except socket.error as err_msg:
8         print("Error accessing %s: error number and detail %s"
9               % (remote_host, err_msg))
10 if __name__ == '__main__':
11     get_remote_machine_info()
```

The Run window at the bottom shows the execution of the script:

```
"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/remote_machine_info.py"
Remote host name: www.python.org
Error accessing www.python.org: error number and detail [Errno 11001] getaddrinfo failed
Process finished with exit code 0
```

Exercise 4.2.3: Converting an IPv4 address to different formats



The screenshot shows the same IDE with an additional file, 'ip4_address_conversion.py', open in the editor. The code for 'ip4_address_conversion.py' is:

```
1 import socket
2 from binascii import hexlify
3 def convert_ip4_address():
4     for ip_addr in ['127.0.0.1', '192.168.0.1']:
5         packed_ip_addr = socket.inet_aton(ip_addr)
6         unpacked_ip_addr = socket.inet_ntoa(packed_ip_addr)
7         print(" IP Address: %s => Packed: %s, Unpacked: %s"
8               % (ip_addr, hexlify(packed_ip_addr), unpacked_ip_addr))
9 if __name__ == '__main__':
10     convert_ip4_address()
```

The Run window shows the execution of 'ip4_address_conversion.py':

```
"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/ip4_address_conversion.py"
IP Address: 127.0.0.1 => Packed: b'7f000001', Unpacked: 127.0.0.1
IP Address: 192.168.0.1 => Packed: b'c0a80001', Unpacked: 192.168.0.1
Process finished with exit code 0
```

Exercise 4.2.4: Finding a service name, given the port and protocol

The screenshot shows a code editor with a project named 'Hello World'. The file 'finding_service_name.py' is open and contains the following Python code:

```
1 import socket
2 def find_service_name():
3     protocolname = 'tcp'
4     for port in [80, 25]:
5         print("Port: %s => service name: %s" %(port, socket.getservbyport(port, protocolname)))
6     print("Port: %s => service name: %s" %(53, socket.getservbyport(53, 'udp')))
7 if __name__ == '__main__':
8     find_service_name()
```

The Run window shows the execution of the script:

```
"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/finding_service_name.py"
Port: 80 => service name: http
Port: 53 => service name: domain
Port: 25 => service name: smtp
Port: 53 => service name: domain
Process finished with exit code 0
```

Exercise 4.2.5: Setting and getting the default socket timeout

The screenshot shows a code editor with a project named 'Hello World'. The file 'socket_timeout.py' is open and contains the following Python code:

```
1 import socket
2 def test_socket_timeout():
3     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4     print("Default socket timeout: %s" %s.gettimeout())
5     s.settimeout(100)
6     print("Current socket timeout: %s" %s.gettimeout())
7 if __name__ == '__main__':
8     test_socket_timeout()
```

The Run window shows the execution of the script:

```
"F:\Hello World\venv\Scripts\python.exe" "F:/Hello World/socket_timeout.py"
Default socket timeout: None
Current socket timeout: 100.0
Process finished with exit code 0
```

Exercise 4.2.6: Writing a simple echo client/server application (Tip: Use port 9900)

Server Code:

```
echo_server.py x
1 import socket
2 import sys
3 import argparse
4 import codecs
5 from codecs import encode, decode
6 host = 'localhost'
7 data_payload = 4096
8 backlog = 5
9 def echo_server(port):
10     """ A simple echo server """
11     # Create a TCP socket
12     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     # Enable reuse address/port
14     sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
15     # Bind the socket to the port
16     server_address = (host, port)
17     print ("Starting up echo server on %s port %s" %server_address)
18     sock.bind(server_address)
19     # Listen to clients, backlog argument specifies the max no. of queued connections
20     sock.listen(backlog)
21
22     while True:
23         print ("Waiting to receive message from client")
24         client, address = sock.accept()
25         data = client.recv(data_payload)
26         if data:
27             print ("Data: %s" %data)
28             client.send(data)
29             print ("sent %s bytes back to %s" % (data, address))
30             # end connection
31             client.close()
32
33 if __name__ == '__main__':
34     parser = argparse.ArgumentParser(description='Socket Server Example')
35     parser.add_argument('--port', action="store", dest="port", type=int,
36                         required=True)
37     given_args = parser.parse_args()
38     port = given_args.port
39     echo_server(port)
40
41 echo_server()
```

Client Code:


```
echo_server.py × echo_client.py ×
1  #!/usr/bin/env python
2  import socket
3  import sys
4  import argparse
5  import codecs
6  from codecs import encode, decode
7  host = 'localhost'
8  def echo_client(port):
9      """ A simple echo client """
10     # Create a TCP/IP socket
11     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12     # Connect the socket to the server
13     server_address = (host, port)
14     print("Connecting to %s port %s" % server_address)
15     sock.connect(server_address)
16     # Send data
17     try:
18         #Send data
19         message = "Test message: SDN course examples"
20         print("Sending %s" % message)
21         sock.sendall(message.encode('utf_8'))
22         # Look for the response
23         amount_received = 0
```

```
echo_server.py x echo_client.py x
22     # Look for the response
23     amount_received = 0
24     amount_expected = len(message)
25     while amount_received < amount_expected:
26         data = sock.recv(16)
27         amount_received += len(data)
28         print("Received: %s" % data)
29     except socket.errno as e:
30         print("Socket error: %s" %str(e))
31     except Exception as e:
32         print("Other exception: %s" %str(e))
33     finally:
34         print("Closing connection to the server")
35     sock.close()
36     if __name__ == '__main__':
37         parser = argparse.ArgumentParser(description='Socket Server Example')
38         parser.add_argument('--port', action="store", dest="port", type=int,
39                             required=True)
40         given_args = parser.parse_args()
41         port = given_args.port
42         echo_client(port)
```

Conclusion:

Python plays an essential role in network programming. The standard library of Python has full support for network protocols, encoding, and decoding of data and other networking concepts, and it is simpler to write network programs in Python than that of C++. There are two levels of network service access in Python. These are:

- Low-Level Access
- High-Level Access

In the first case, programmers can use and access the basic socket support for the operating system using Python's libraries, and programmers can implement both connection-less and connection-oriented protocols for programming.

Application-level network protocols can also be accessed using high-level access provided by Python libraries. These protocols are HTTP, FTP, etc.

A socket is the end-point in a flow of communication between two programs or communication channels operating over a network. They are created using a set of programming requests called socket API (Application Programming Interface). Python's socket library offers classes for handling common transports as a generic interface.

Sockets use protocols for determining the connection type for port-to-port communication between client and server machines. The protocols are used for:

- Domain Name Servers (DNS)
- IP addressing
- E-mail
- FTP (File Transfer Protocol) etc.