



MAWLANA BHASHANI SCIENCE AND TECHNOLOGY UNIVERSITY

SANTOSH, TANGAIL-1902

Department of ICT

Course Code : ICT – 3207
Course Title : Computer Networks

Lab Report No : 05
Report Name : Programming with Python

Submitted By: Name : Ishrath Mostakin ID : IT-17018 Session : 2016-2017	Submitted To: Nazrul Islam Assistant Professor, Department 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.

- **TCP:** TCP stands for transmission control protocol. It is implemented in the transport layer of the IP/TCP model and is used to establish reliable connections. TCP is one of the protocols that encapsulate data into packets. It then transfers these to the remote end of the connection using the methods available on the lower layers. On the other end, it can check for errors, request certain pieces to be resent, and reassemble the information into one logical piece to send to the application layer.

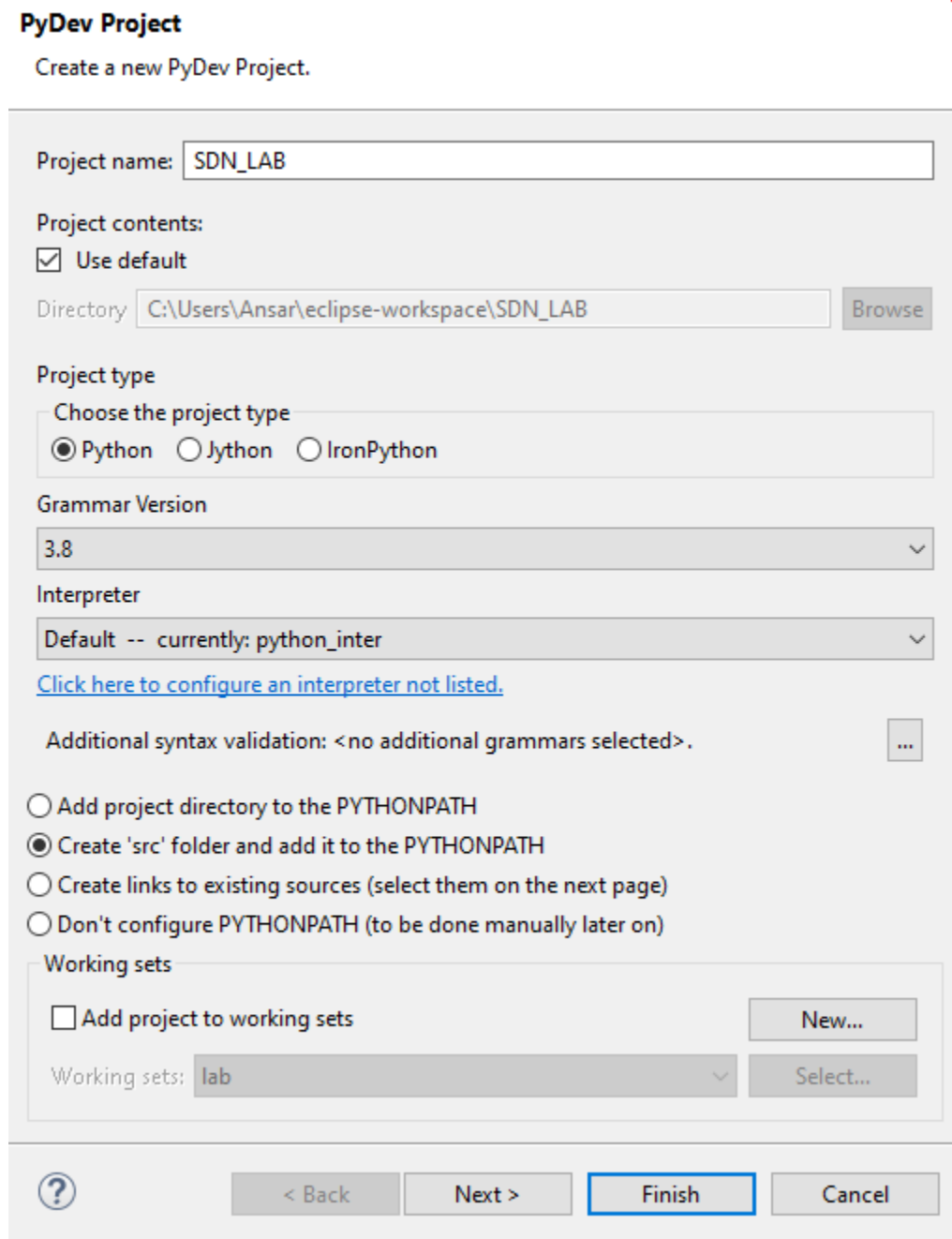
- **UDP:** UDP stands for user datagram protocol. It is a popular companion protocol to TCP and is also implemented in the transport layer.

The fundamental difference between UDP and TCP is that UDP offers unreliable data transfer. It does not verify that data has been received on the other end of the connection. This might sound like a bad thing, and for many purposes, it is. However, it is also extremely important for some functions. Because it is not required to wait for confirmation that the data was received and forced to resend data, UDP is much faster than TCP. It does not establish a connection with the remote host, it simply fires off the data to that host and doesn't care if it is accepted or not. Because it is a simple transaction, it is useful for simple communications like querying for network resources. It also doesn't maintain a state, which makes it great for

transmitting data from one machine to many real-time clients. This makes it ideal for VOIP, games, and other applications that cannot afford delays.

Exercise : Create a python project using with SDN_LAB

Ans:



PyDev Project
Create a new PyDev Project.

Project name:

Project contents:
☒ Use default

Directory:

Project type
Choose the project type
☒ Python ☐ Jython ☐ IronPython

Grammar Version

Interpreter

[Click here to configure an interpreter not listed.](#)

Additional syntax validation: <no additional grammars selected>

☐ Add project directory to the PYTHONPATH
☒ Create 'src' folder and add it to the PYTHONPATH
☐ Create links to existing sources (select them on the next page)
☐ Don't configure PYTHONPATH (to be done manually later on)

Working sets
☐ Add project to working sets

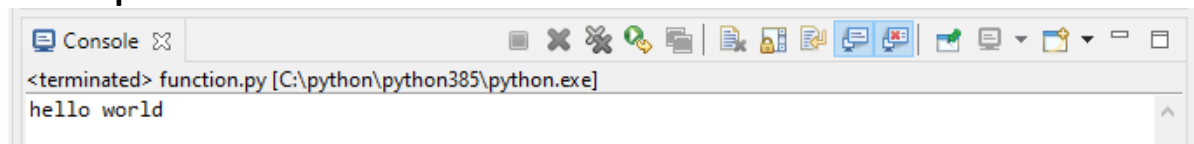
Working sets:

Exercise : Python function (save as function.py)

Create python scrip using the syntax provided below.

```
def say_hello():  
    print('hello world')  
  
if __name__ == '__main__':  
    say_hello()
```

The output of this code is:



Exercise : Python function (save as function_2.py) Create python scrip using the syntax provided below.

```
def print_max(a, b):  
    if a > b:  
        print(a, 'is maximum')  
    elif a == b:  
        print(a, 'is equal to', b)  
    else:  
        print(b, 'is maximum')  
if __name__ == '__main__':  
    pass  
    print_max(3, 4)  # directly pass literal values  
    x = 5  
    y = 7  # pass variables as arguments  
    print_max(x, y)
```

Which is the output of this function? Does the function need any parameter?

Ans: The code does not show any output. May be there is some problem in this code.

```
1
2 def print_max(a, b):
3
4     if a > b:
5         print(a, 'is maximum')
6
7     elif a == b:
8         print('is equal to', b)
9
10    else:
11        print(b, 'is maximum')
12
13    if __name__ == '__main__':
14        pass
15        print_max(3, 4)    # directly pass literal values
16        x = 5
17        y = 7    # pass variables as arguments
18        print_max(x, y)
19        print_max(x,y)
20
21
```

Console [C:\python\python385\python.exe]

This function does not need any parameter .

Exercise : Local variable (save as function_local.py) Create python scrip using the syntax provided below.

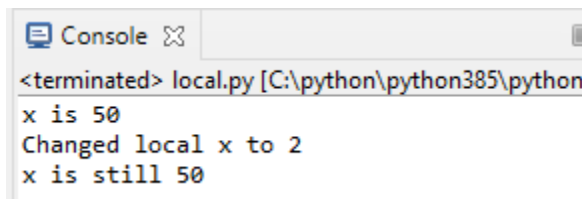
```
x = 50
def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)
    if __name__ == '__main__':
```

```
func(x)
print('x is still', x)
```

Which is the final value of variable x? Why variable x does not change to 2?

Ans:

Output is:



```
<terminated> local.py [C:\python\python385\python
x is 50
Changed local x to 2
x is still 50
```

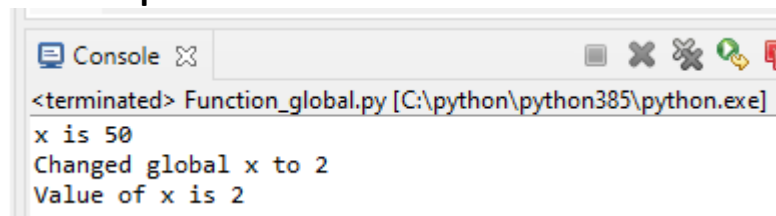
The final value of variable x is 50. It does not change because it is a global variable.

Exercise : Global variable (save as function_global.py) Create python scrip using the syntax provided below.

```
x = 50
def func():
    global x
    print('x is', x)
    x = 2
    print('Changed global x to', x)
if __name__ == '__main__':
    func()
    print('Value of x is', x)
```

Which is the final value of variable x? Why variable x change this time?

Ans: Output is:



```
<terminated> Function_global.py [C:\python\python385\python.exe]
x is 50
Changed global x to 2
Value of x is 2
```

This time variable x is declared as global inside the function .So the variable x is changed.

Exercise : Python modules Create python scrip using the syntax provided below (save as mymodule.py).

```
def say_hi():
```

```
print('Hi, this is mymodule speaking.')
```

```
__version__ = '0.1'
```

Create python scrip using the syntax provided below (save as module_demo.py).

```
import mymodule
```

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

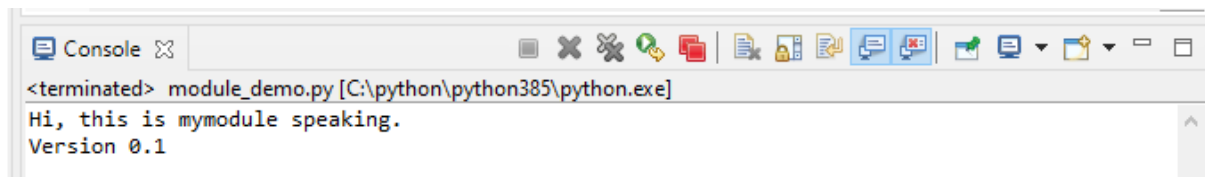
```
mymodule.say_hi()
```

```
print('Version', mymodule.__version__)
```

Run the script, which is the role of import?

Ans:

Output is:

A screenshot of a Windows-style console window titled 'Console'. The window shows the command prompt running 'module_demo.py [C:\python\python385\python.exe]'. The output displayed is 'Hi, this is mymodule speaking.' followed by 'Version 0.1' on the next line. The console window has a standard toolbar with icons for file operations and window management.

Python modules can get access to code from another module by importing the file/function using import. The import statement is the most common way of invoking the import machinery, but it is not the only way. When import is used, it searches for the module initially in the local scope by calling `__import__()` function

Create python scrip using the syntax provided below (save as module_demo2.py).

```
from mymodule import say_hi, __version__
```

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

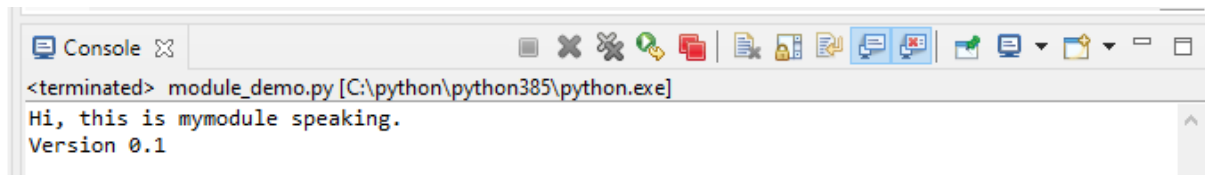
```
say_hi()
```

```
print('Version', __version__)
```

Run the script, which is the role of from, import?

Ans:

Output is :

A screenshot of a Python console window. The title bar shows 'Console' and a close button. The command prompt shows '<terminated> module_demo.py [C:\python\python385\python.exe]'. The output text is 'Hi, this is mymodule speaking.' followed by 'Version 0.1' on the next line.

```
<terminated> module_demo.py [C:\python\python385\python.exe]
Hi, this is mymodule speaking.
Version 0.1
```

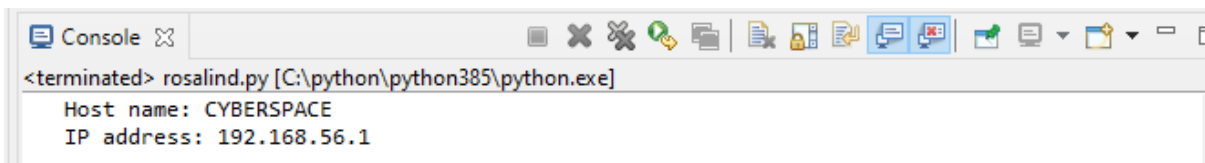
Using **'from'** we say the module name and then using **'import'** we say what we are importing from the module.

Exercise : Printing your machine's name and IPv4 address Create python scrip using the syntax provided below (save as local_machine_info.py):
import socket

```
def print_machine_info():
    host_name = socket.gethostname()
    ip_address = socket.gethostbyname(host_name)
    print (" Host name: %s" % host_name)
    print (" IP address: %s" % ip_address)
    if __name__ == '__main__':
        print_machine_info()
```

Run the script, which module the program uses? Provide two additional functions of socket? ?

Ans:

A screenshot of a Python console window. The title bar shows 'Console' and a close button. The command prompt shows '<terminated> rosaland.py [C:\python\python385\python.exe]'. The output text is 'Host name: CYBERSPACE' followed by 'IP address: 192.168.56.1' on the next line.

```
<terminated> rosaland.py [C:\python\python385\python.exe]
Host name: CYBERSPACE
IP address: 192.168.56.1
```

The *type* argument specifies the socket type, which determines the semantics of communication over the socket. The following socket types are defined; implementations may specify additional socket types:

SOCK_STREAM

Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.

SOCK_DGRAM

Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.

SOCK_SEQPACKET

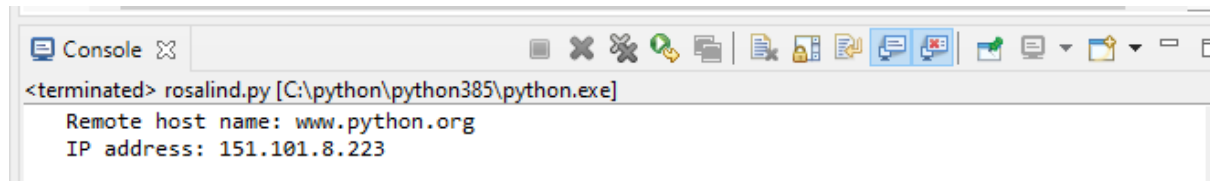
Provides sequenced, reliable, bidirectional, connection-mode transmission paths for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

Exercise: Retrieving a remote machine's IP address Create python scrip using the syntax provided below (save as remote_machine_info.py):

```
import socket
def get_remote_machine_info():
    remote_host = 'www.python.org'
    try:
        print (" Remote host name: %s" % remote_host)
        print (" IP address: %s" %socket.gethostbyname(remote_host))
    except socket.error as err_msg:
        print ("Error accesing %s: error number and detail %s" %(remote_host,
        err_msg))
if __name__ == '__main__':
    get_remote_machine_info()
```

Run the script, which is the output? Modify the code for getting the RMIT website info.

Ans:

A screenshot of a Python console window. The title bar shows 'Console' and a close button. The command prompt shows '<terminated> rosaland.py [C:\python\python385\python.exe]'. The output of the script is displayed in two lines: 'Remote host name: www.python.org' and 'IP address: 151.101.8.223'.

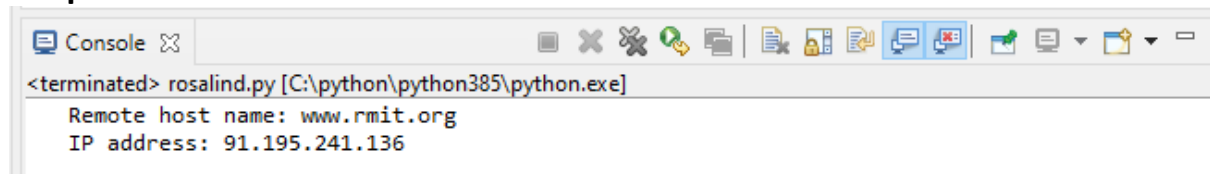
RMIT website info:

Code:

```
import socket
```

```
def get_remote_machine_info():  
    __remote_host = 'www.rmit.org'  
    __try:  
        print (" Remote host name: %s" % remote_host)  
        print (" IP address: %s" %socket.gethostbyname(remote_host))  
    __except socket.error as err_msg:  
        print ("Error accesing %s: error number and detail %s" %(remote_host,  
err_msg))  
  
if __name__ == '__main__':  
    get_remote_machine_info()
```

Output:

A screenshot of a Python console window. The title bar says 'Console'. The command prompt shows '<terminated> rosaland.py [C:\python\python385\python.exe]'. The output of the script is displayed below: 'Remote host name: www.rmit.org' and 'IP address: 91.195.241.136'.

```
<terminated> rosaland.py [C:\python\python385\python.exe]  
Remote host name: www.rmit.org  
IP address: 91.195.241.136
```

Exercise : Converting an IPv4 address to different formats Create python scrip using the syntax below (save as ip4_address_conversion.py):

```
import socket from binascii import hexlify  
def convert_ip4_address():  
    for ip_addr in ['127.0.0.1', '192.168.0.1']:  
        packed_ip_addr = socket.inet_aton(ip_addr)  
        unpacked_ip_addr = socket.inet_ntoa(packed_ip_addr)  
        print (" IP Address: %s => Packed: %s, Unpacked: %s" %(ip_addr,  
hexlify(packed_ip_addr), unpacked_ip_addr))  
    if __name__ == '__main__':  
        convert_ip4_address()
```

Run the script, which is the output? How binascii works?

Ans:

```
Console [X]
<terminated> rosaland.py [C:\python\python385\python.exe]
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
```

Binascii:

The **binascii** module contains a number of methods to convert between binary and various ASCII-encoded binary representations. ... Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of data should be at most 45.

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

```
import socket
```

```
def find_service_name():
```

```
    protocolname = 'tcp'
```

```
    for port in [80, 25]:
```

```
        print ("Port: %s => service name: %s" %(port,
```

```
socket.getservbyport(port, protocolname)))
```

```
        print ("Port: %s => service name: %s" %(53, socket.getservbyport(53,
'udp'))))
```

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

```
    find_service_name()
```

Run the script, which is the output? Modify the code for getting complete the table:

Output:

```
Console [X]
<terminated> rosaland.py [C:\python\python385\python.exe]
Port: 80 => service name: http
Port: 53 => service name: domain
Port: 25 => service name: smtp
Port: 53 => service name: domain
```

For the given port the code will be:

```
import socket
```

```
def find_service_name():
```

```
    protocolname = 'tcp'
```

```
    for port in [21,22,110]:
```

```

    print ("Port: %s => service name: %s" %(port,
socket.getservbyport(port, protocolname)))
    print ("Port: %s => service name: %s" %(53, socket.getservbyport(53,
'udp'))))

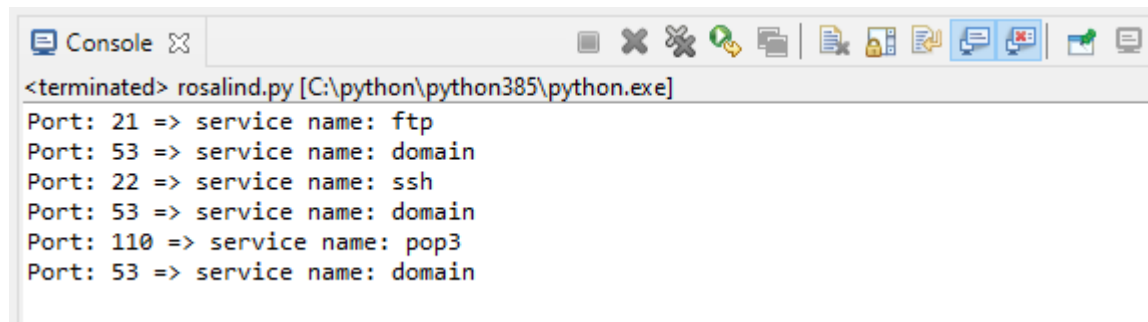
```

```

if __name__ == '__main__':
    find_service_name()

```

Output:



```

<terminated> rosaling.py [C:\python\python385\python.exe]
Port: 21 => service name: ftp
Port: 53 => service name: domain
Port: 22 => service name: ssh
Port: 53 => service name: domain
Port: 110 => service name: pop3
Port: 53 => service name: domain

```

Exercise : Setting and getting the default socket timeout

```
import socket
```

```

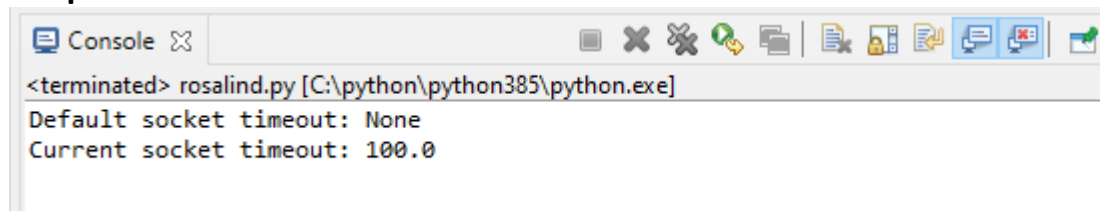
def test_socket_timeout():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print ("Default socket timeout: %s" %s.gettimeout())
    s.settimeout(100)
    print ("Current socket timeout: %s" %s.gettimeout())

```

```
if __name__ == '__main__': test_socket_timeout()
```

Run the script, which is the role of socket timeout in real applications?

Output:



```

<terminated> rosaling.py [C:\python\python385\python.exe]
Default socket timeout: None
Current socket timeout: 100.0

```

A *socket timeout* implementation should allow for setting the timeout at ... For *example*, this is how we connect to a local HTTP server on port 80 ... It can be implemented as a method that we add to `IO::Socket::INET` class,

possibly by using a *Role*. ... The *real* version handles EINTR and other corner cases.

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

Ans:

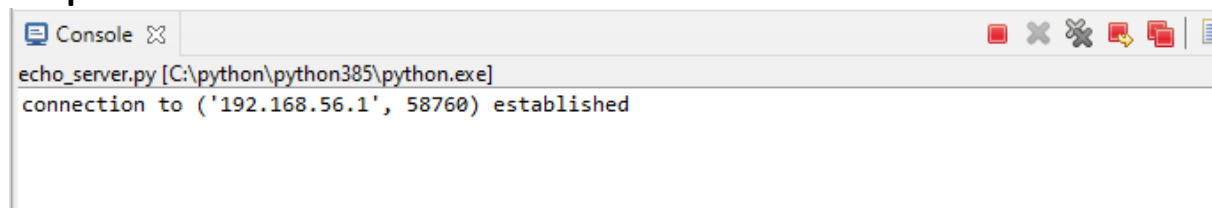
Server:

```
import socket
s= socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.bind((socket.gethostname(),1022))
s.listen(5)
while True:
    clt,adr = s.accept()
    print(f"connection to {adr} established")
    clt.send(bytes("Socket programming in python","utf-8"))
```

Client:

```
import socket
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((socket.gethostname(),1022))
msg = s.recv(1022)
print(msg.decode("utf-8"))
```

Output:



We have to run the server program first ,then client program.

Conclusion :

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. In this lab, we faced some problem.