Scripting Language

UNIT-V

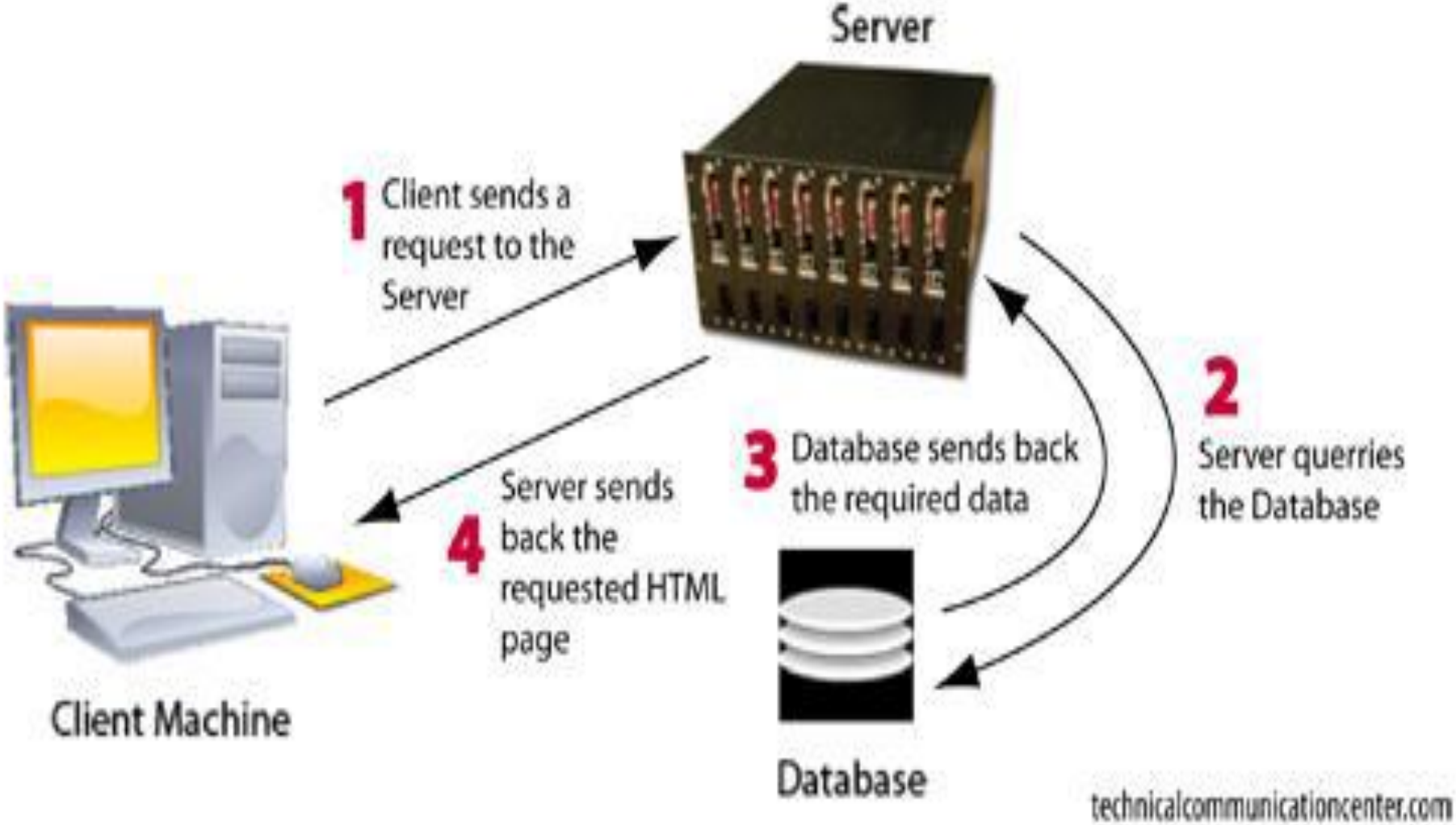
K.RaviKanth, Ast.Prof., Dept. of CSE, IIIT- RGUKT- Basara, T.S, IND.

- Networking Programming using python
- Debugging with pdb
- Python Testing – Unit Testing

python
Programming Language

Networking Programming using Python

- **Computer Network**: Computer networking is the practice of interfacing two or more computing devices with each other for the purpose of sharing data. Computer networks are built with a combination of hardware and software components.
- **Client**: A client is a single-user workstation that provides presentation services, database services and connectivity along with an interface for user interaction to acquire business needs.
- **Server**: A server is one or more multi-user processors with a higher capacity of shared memory which provides connectivity and the database services along with interfaces relevant to the business procedures.
- Client/Server computing provides an environment that enhances business procedures by appropriately synchronizing the application processing between the client and the server.


Server
1 Client sends a request to the Server
2 Server querries the Database
3 Database sends back the required data
4 Server sends back the requested HTML page
Client Machine
Database
technicalcommunicationcenter.com

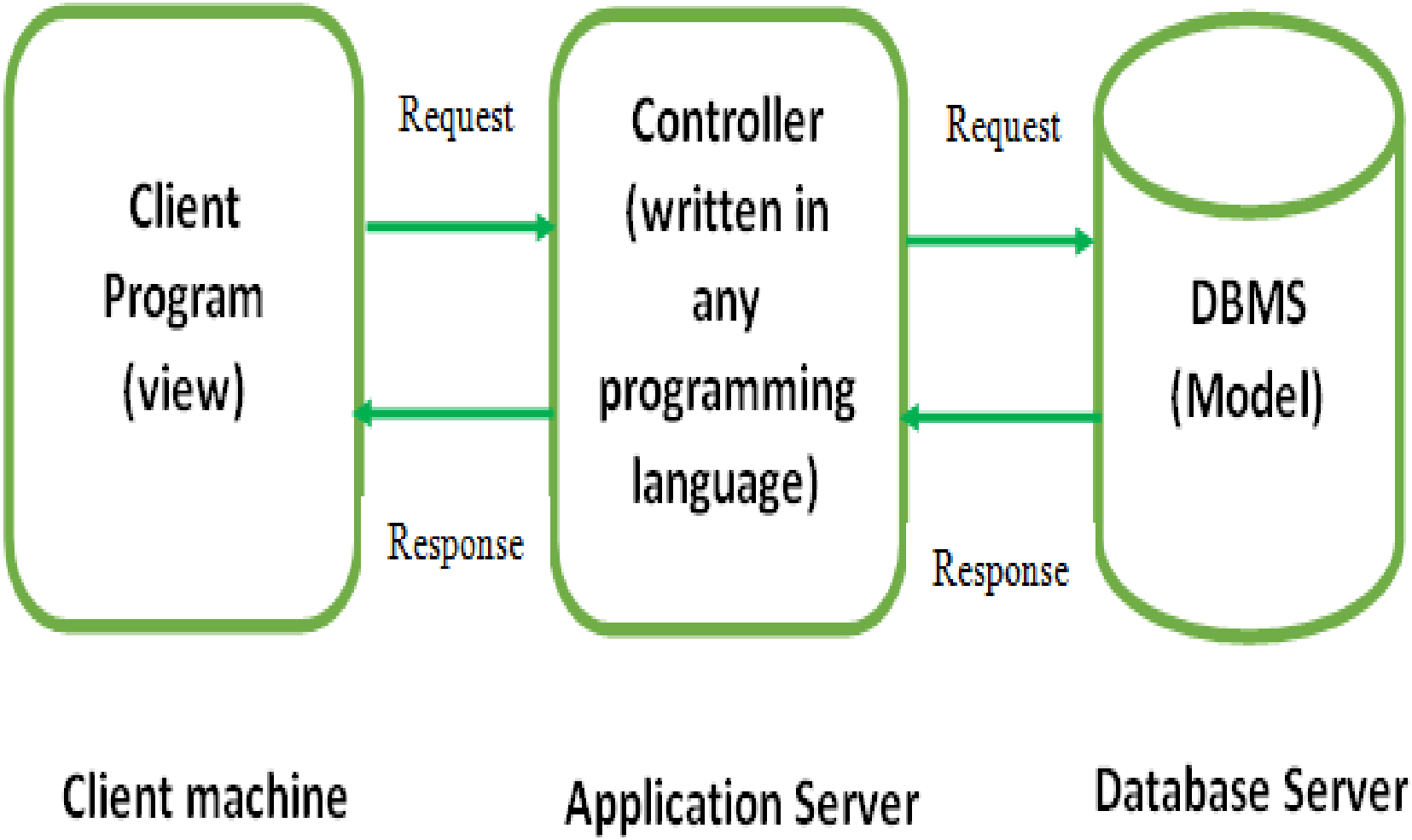
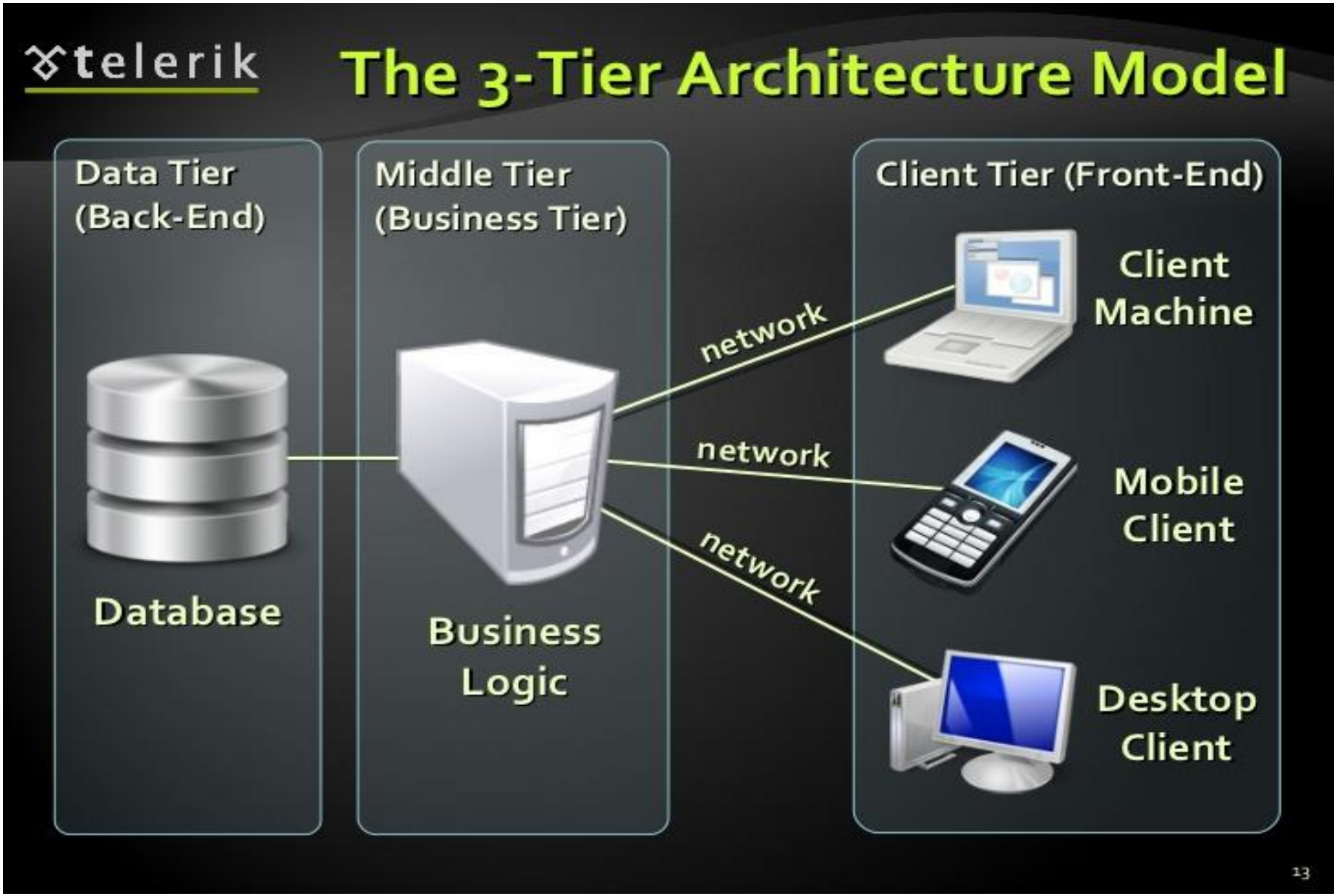


Figure: 3-tier MVC architecture


telerik
The 3-Tier Architecture Model
Data Tier
(Back-End)
Database
Middle Tier
(Business Tier)
Business
Logic
network
network
network
Client Tier (Front-End)
Client
Machine
Mobile
Client
Desktop
Client
13

Presentation tier

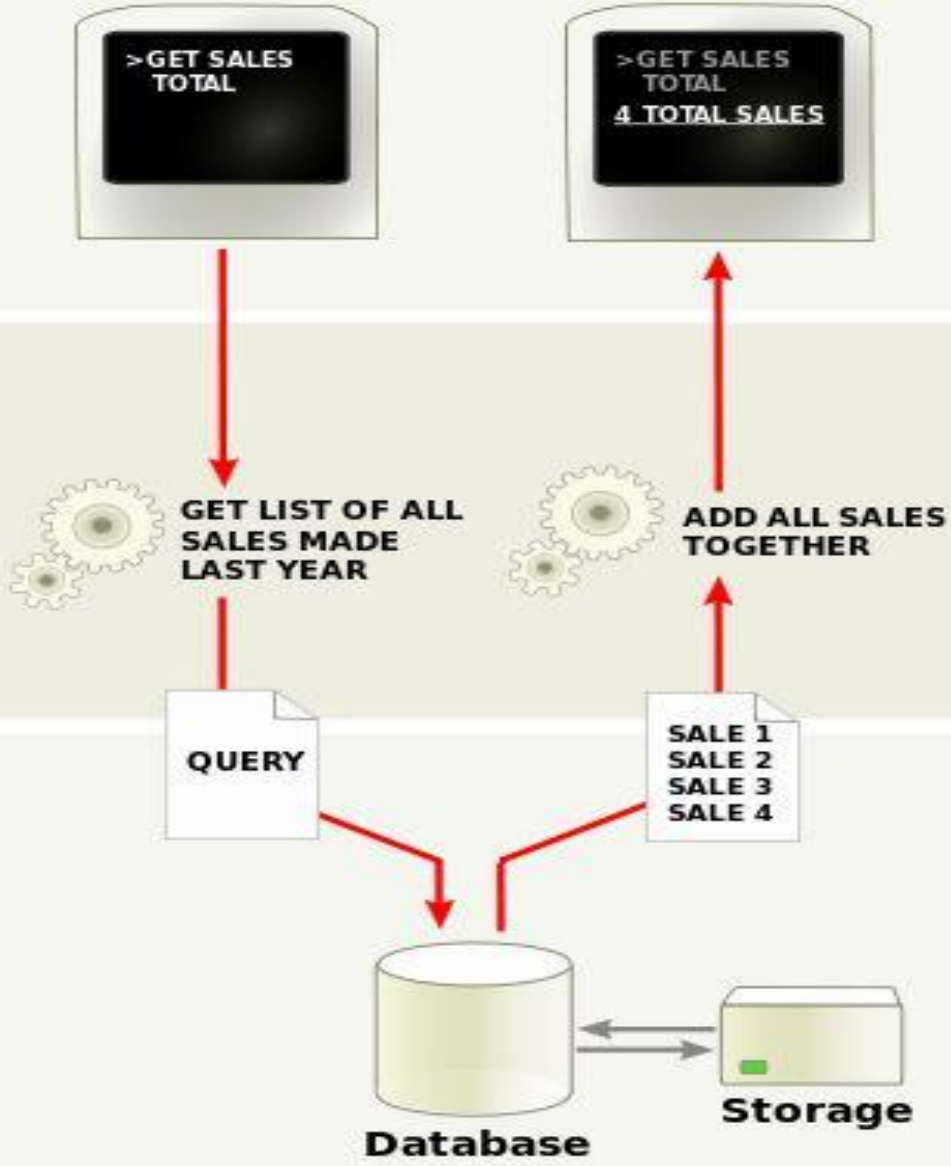
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.



- **Socket:** A socket is an endpoint for communication between two machines. Yes as the name suggests socket is nothing but a pipe that will deliver the stuff from one end to another end. The medium can be the Local Area Newtork, Wide Area Network or the Internet. Sockets can use both TCP and UDP protocols.
- The W3C website has a very good detailed description with example code for sockets here
- **Server:** A server is a machine that waits for client requests and serves or processes them.
- **Client:** A client on the other hand is the requester of the service.
- We are building a simple server and a client where server will open a socket and wait for clients to connect. Once client is connected he can send a message and server will process the message and reply back the same message but converted into upper case to demonstrate the server side processing and transmission. Conceptually this sounds fairly simple and so it is implementation-wise also. Without further ado let's look at the code first. Then we will go through what they are doing

The socket Module

- To create a socket, you must use the socket.socket() function available insocket module, which has the general syntax −
- s = socket.socket (socket_family, socket_type, protocol=0)
- Here is the description of the parameters −
- socket_family: This is either AF_UNIX or AF_INET, as explained earlier.
- socket_type: This is either SOCK_STREAM or SOCK_DGRAM.
- protocol: This is usually left out, defaulting to 0.
- Once you have socket object, then you can use required functions to create your client or server program. Following is the list of functions required −

Server Socket Methods

Method	Description
s.bind()	This method binds address (hostname, port number pair) to socket.
s.listen()	This method sets up and start TCP listener.
s.accept()	This passively accept TCP client connection, waiting until connection arrives (blocking).

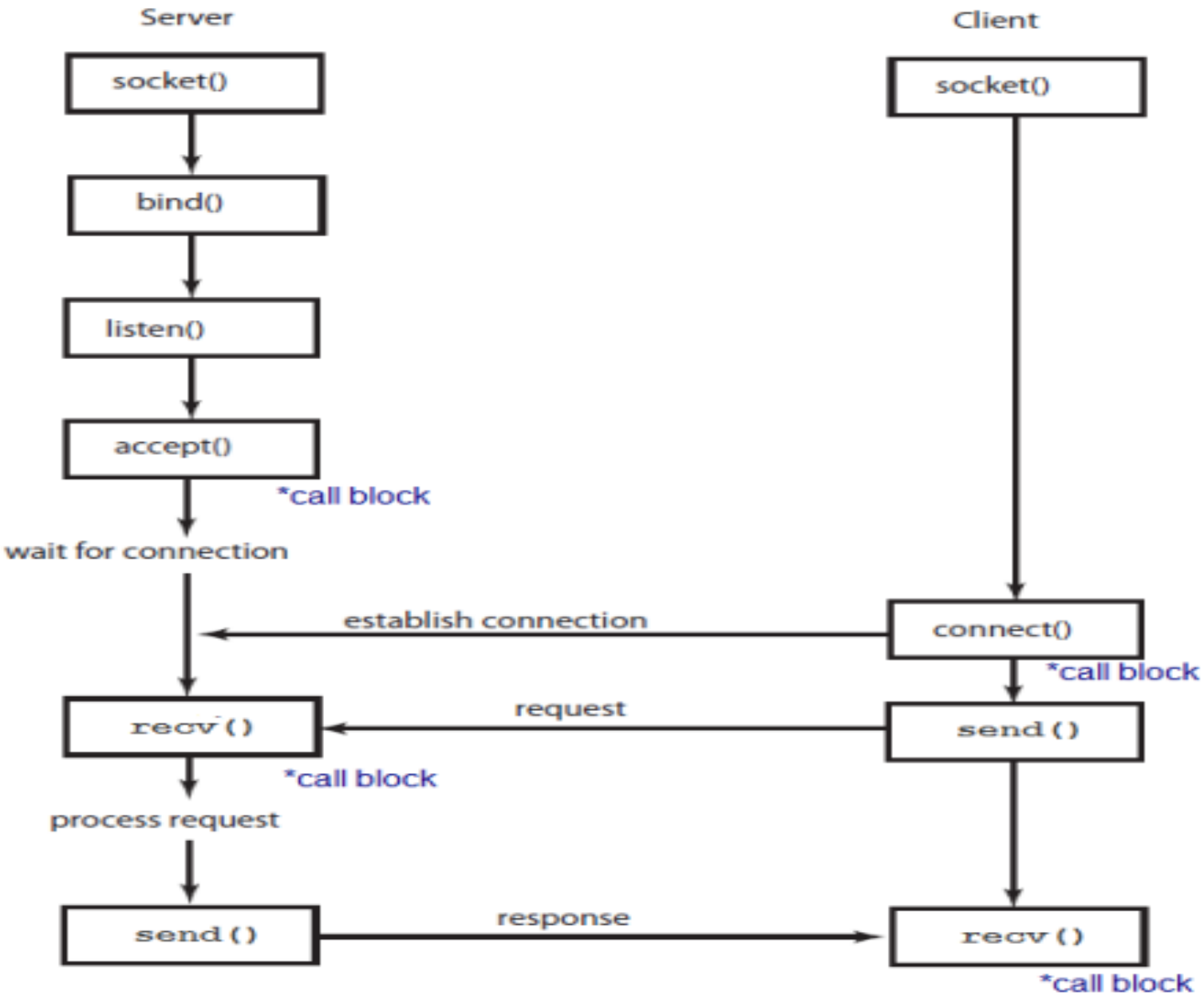
Client Socket Methods

Method	Description
s.connect()	This method actively initiates TCP server connection.

K.RaviKanth, Ast.Prof., Dept. of CSE, IIIT- RGUKT- Basara, T.S, IND.

General Socket Methods

Method	Description
s.recv( )	This method receives TCP message
s.send( )	This method transmits TCP message
s.recvfrom( )	This method receives UDP message
s.sendto( )	This method transmits UDP message
s.close( )	This method closes socket
socket.gethostname( )	Returns the hostname.


Server
Client
socket()
socket()
bind()
listen()
accept()
*call block
wait for connection
establish connection
connect()
*call block
recv()
request
send()
*call block
process request
send()
response
recv()
*call block

A Simple Server

- To write Internet servers, we use the socket function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.
- Now call bind(hostname, port) function to specify a port for your service on the given host.
- Next, call the accept method of the returned object. This method waits until a client connects to the port you specified, and then returns a connection object that represents the connection to that client.

```
#!/usr/bin/python # This is server.py file
import socket # Import socket module
s = socket.socket() # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345 # Reserve a port for your service.
s.bind((host, port)) # Bind to the port
s.listen(5) # Now wait for client connection.
while True:
    c, addr = s.accept() # Establish connection with client.
    print 'Got connection from', addr
    c.send('Thank you for connecting')
    c.close() # Close the connection
```

A Simple Client

- Let us write a very simple client program which opens a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's socket module function.
- The socket.connect(hosname, port) opens a TCP connection to hostnameon the port. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.
- The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits −

```
#!/usr/bin/python #This is client.py file
import socket # Import socket module
s = socket.socket() # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345 # Reserve a port for your service.
s.connect((host, port))
print s.recv(1024)
s.close # Close the socket when done
```

Now run this server.py in background and then run above client.py to see the result.

```
# Following would start a server in background.
$ python server.py &
# Once server is started run client as follows:
$ python client.py
```

This would produce following result −

Got connection from ('127.0.0.1', 48437)

Thank you for connecting

We will create two files:

server.py

client.py

As the name suggests the server.py will hold the server code that will listen for client connection and client.py will hold the client code

```
import socket    # server.py
import time
# create a socket object
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# get local machine name
host = socket.gethostname()
port = 9999
# bind to the port
serversocket.bind((host, port))
# queue up to 5 requests
serversocket.listen(5)
while True:
    # establish a connection
    clientsocket,addr = serversocket.accept()
    print("Got a connection from %s" % str(addr))
    currentTime = time.ctime(time.time()) + "\r\n"
    clientsocket.send(currentTime.encode('ascii'))
    clientsocket.close()
```

- 1. First we import the python socket library.
- 2. Then we define a main function
- 3. We define two variables, a host and a port, here the local machine is the host thus ip address of 127.0.0.1 & I have chosen randomly port 5000 and it is advised to use anything above 1024 as upto 1024 core services use the ports
- 4. Then we define a variable mySocket which is an instance of a Python socket
- 5. In server it is important to bind the socket to host and port thus we bind it, tip: bind takes a Tuple, thus we have double brackets surrounding it
- 6. Then we call the listen method and pass 1 to it so that it will perpetually listen till we close the connection
- 7. Then we have two variables conn and addr which will hold the connection from client and the address of the client
- 8. Then we print the clients address and create another variable data which is receiving data from connection and we have to decode it, this step is necessary for Python 3 as normal string with str won't pass through socket and str does not implement buffer interface anymore.
- 9. We run all this in a while true loop so unless the the connection is closed we run this and server's keeps on listening when the data is received server transforms in into uppercase by calling upper method and sends the string back to client and we encode too as normal string will fail to transmit properly
- So that's all that is there to the server.py file and I hope I was able to clarify each step.

- Now let's have a look at the client.py file

```
import socket # client.py
# create a socket object
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# get local machine name
host = socket.gethostname()
port = 9999
# connection to hostname on the port.
s.connect((host, port))
# Receive no more than 1024 bytes
tm = s.recv(1024)
s.close()
print("The time got from the server is %s" % tm.decode('ascii'))
```

- 1. Well similarly we import the socket module and create similar variable mySocket and connect via passing server's address 127.0.0.1 and port 5000
- 2. However, one noticeable difference is we do not have to add bind as client does not need to bind, this is a very basic yet important concept of network programming
- 3. Then we use the input method to ask for input , see Python 3 has replaced raw_input by simply input
- 4. Then while the character typed is not "q" we keep running the loop and send the message by encoding it and when we received the processed data we decode it and print
- 5. One common thing is the main method of Python we run on both the files which is wrapped in the magic __main__ method to be run

Debugging with pdb

- As a programmer, one of the first things that you need for serious program development is a debugger.
- Python has a debugger, which is available as a module called pdb (for “Python DeBugger”)
- The module pdb defines an interactive source code debugger for Python programs.
- It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame.
- It also supports post-mortem debugging and can be called under program control.
- The debugger is extensible – it is actually defined as the class Pdb. The extension interface uses the modules bdb and cmd.

- Getting started — pdb.set_trace()
- To start, we will look at simplest way to use the Python debugger.
- 1. Let's start with a simple program, exp_pdb1.py.
- # exp_pdb1.py -- experiment with the Python debugger, pdb

```
a = "aaa"
b = "bbb"
c = "ccc"
final = a + b + c
print final
```

- 2. Insert the following statement at the beginning of your Python program. This statement imports the Python debugger module, pdb.

```
import pdb
```

3. Now find a spot where you would like tracing to begin, and insert the following code:

```
pdb.set_trace()
```

- So now your program looks like this.
- # exp_pdb1.py -- experiment with the Python debugger, pdb import pdb a = "aaa" pdb.set_trace() b = "bbb" c = "ccc" final = a + b + c print final
- 4. Now run your program from the command line as you usually do.
- When your program encounters the line with pdb.set_trace() it will start tracing.
- That is, it will
- (1) stop,
- (2) display the “current statement” (that is, the line that will execute next) and
- (3) wait for your input. You will see the pdb prompt, which looks like this: (Pdb)

- **Debugger commands:**
- The commands recognized by the debugger are listed below. Most commands can be abbreviated to one or two letters as indicated; e.g. h(elp) means that either h or help can be used to enter the help command (but not he or hel, nor H or Help or HELP). Arguments to commands must be separated by whitespace (spaces or tabs). Optional arguments are enclosed in square brackets ([]) in the command syntax; the square brackets must not be typed. Alternatives in the command syntax are separated by a vertical bar (|).

- **Execute the next statement… with "n" (next)**
- At the (Pdb) prompt, press the lower-case letter "n" (for "next") on your keyboard, and then press the ENTER key. This will tell pdb to execute the current statement. Keep doing this — pressing "n", then ENTER. Eventually you will come to the end of your program, and it will terminate and return you to the normal command prompt.

- **Repeating the last debugging command… with ENTER**
- This time, do the same thing as you did before. Start your program running. At the (Pdb) prompt, press the lower-case letter “n” (for “next”) on your keyboard, and then press the ENTER key. But this time, after the first time that you press “n” and then ENTER, don’t do it any more. Instead, when you see the (Pdb) prompt, just press ENTER. You will notice that pdb continues, just as if you had pressed “n”. If you press ENTER without entering anything, pdb will re-execute the last command that you gave it. In this case, the command was “n”, so you could just keep stepping through the program by pressing ENTER. Notice that as you passed the last line (the line with the “print” statement), it was executed and you saw the output of the print statement (“aaabbbccc”) displayed on your screen.

- **Quitting it all… with “q” (quit)**
- The debugger can do all sorts of things, some of which you may find totally mystifying. So the most important thing to learn now — before you learn anything else — is how to quit debugging! It is easy. When you see the (Pdb) prompt, just press “q” (for “quit”) and the ENTER key. Pdb will quit and you will be back at your command prompt. Try it, and see how it works.

- **Printing the value of variables… with “p” (print)**
- The most useful thing you can do at the (Pdb) prompt is to print the value of a variable. Here’s how to do it. When you see the (Pdb) prompt, enter “p” (for “print”) followed by the name of the variable you want to print. And of course, you end by pressing the ENTER key. Note that you can print multiple variables, by separating their names with commas (just as in a regular Python “print” statement). For example, you can print the value of the variables a, b, and c this way: `p a, b, c`

- **When does pdb display a line?**
- Suppose you have progressed through the program until you see the line
- final = a + b + c and you give pdb the command p final
- You will get a NameError exception. This is because, although you are seeing the line, it has not yet executed. So the final variable has not yet been created.
- Now press "n" and ENTER to continue and execute the line. Then try the "p final" command again. This time, when you give the command "p final", pdb will print the value of final, which is "aaabbbccc".
- Turning off the (Pdb) prompt… with "c" (continue)
- You probably noticed that the "q" command got you out of pdb in a very crude way — basically, by crashing the program.

- If you wish simply to stop debugging, but to let the program continue running, then you want to use the “c” (for “continue”) command at the (Pdb) prompt. This will cause your program to continue running normally, without pausing for debugging. It may run to completion. Or, if the pdb.set_trace() statement was inside a loop, you may encounter it again, and the (Pdb) debugging prompt will appear once more.
- Seeing where you are… with “l” (list)
- As you are debugging, there is a lot of stuff being written to the screen, and it gets really hard to get a feeling for where you are in your program. That’s where the “l” (for “list”) command comes in. “l” shows you, on the screen, the general area of your program’s souce code that you are executing. By default, it lists 11 (eleven) lines of code. The line of code that you are about to execute (the “current line”) is right in the middle, and there is a little arrow “–>” that points to it

- So a typical interaction with pdb might go like this
- The pdb.set_trace() statement is encountered, and you start tracing with the (Pdb) prompt
- You press “n” and then ENTER, to start stepping through your code.
- You just press ENTER to step again.
- You just press ENTER to step again.
- You just press ENTER to step again. etc. etc. etc.
- Eventually, you realize that you are a bit lost. You’re not exactly sure where you are in your program any more. So…
- You press “l” and then ENTER. This lists the area of your program that is currently being executed.
- You inspect the display, get your bearings, and are ready to start again. So….
- You press “n” and then ENTER, to start stepping through your code.
- You just press ENTER to step again.
- You just press ENTER to step again. etc. etc. etc.

- **Stepping into subroutines… with “s” (step into)**
- Eventually, you will need to debug larger programs — programs that use subroutines. And sometimes, the problem that you’re trying to find will lie buried in a subroutine. Consider the following program.

```
 # epdb2.py -- experiment with the Python debugger, pdb
import pdb
def combine(s1,s2): # define subroutine combine, which...
        s3 = s1 + s2 + s1 # sandwiches s2 between copies of s1, ...
        s3 = "" + s3 +"" # encloses it in double quotes,...
        return s3            # and returns it.
a = "aaa"
pdb.set_trace()
b = "bbb"
c = "ccc"
final = combine(a,b) print final
```

- As you move through your programs by using the “n” command at the (Pdb) prompt, you will find that when you encounter a statement that invokes a subroutine — the final = combine(a,b) statement, for example — pdb treats it no differently than any other statement. That is, the statement is executed and you move on to the next statement — in this case, to print final.
- But suppose you suspect that there is a problem in a subroutine. In our case, suppose you suspect that there is a problem in the combine subroutine. when you encounter the final = combine(a,b) statement — is some way to step into the combine subroutine, and to continue your debugging inside it.
- Well, you can do that too. Do it with the “s” (for “step into”) command. When you execute statements that do not involve function calls, “n” and “s” do the same thing — move on to the next statement. But when you execute statements that invoke functions, “s”, unlike “n”, will step into the subroutine. In our case, if you executed the

```
final = combine(a,b)
```

statement using “s”, then the next statement that pdb would show you would be the first statement in the combine subroutine:

```
def combine(s1,s2):
```

and you will continue debugging from there.

- If you are in a subroutine and you enter the “r” (command for return) command at the (Pdb) prompt, pdb will continue executing until the end of the subroutine. At that point — the point when it is ready to return to the
- calling routine — it will stop and show the (Pdb) prompt again, and you can resume stepping through your code.
- One of the nice things about the (Pdb) prompt is that you can do anything at it — you can enter any command that you like at the (Pdb) prompt. So you can, for instance, enter this command at the (Pdb) prompt.

```
(Pdb) var1 = "bbb"
```

Python Testing

- Testing assures correctness under a basic set of conditions. Syntax errors will almost certainly be caught by running tests, and the basic logic of a unit of code can be tested to ensure correctness under certain conditions. It's not about proving the code is correct under any set of conditions. We're simply aiming for a reasonably complete set of possible conditions, but you needn't test all possible strings for each argument.
- Most commonly unit testing is used to check Syntax errors are unintentional misuses of the language and Logical errors are created when the algorithm is not correct.
- All of these errors can be caught through careful testing of the code. Unit testing, specifically tests a single "unit" of code in isolation. A unit could be an entire module, a single class or function, or almost anything in between.
- It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

- To achieve this, unittest supports some important concepts in an object-oriented way: test fixture
- A test fixture - represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process. test case
- A test case - is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.
- test suite - A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.
- test runner - A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.24

K.RaviKanth, Ast.Prof., Dept. of CSE, IIIT- RGUKT-Basara, T.S, IND.

- The Anatomy of A Unit Test
- We'll see how to write and organize unit tests by the example.

```
import unittest
class TestStringMethods(unittest.TestCase):
        def test_upper(self):
                self.assertEqual('foo'.upper(), 'FOO')
        def test_isupper(self):
                self.assertTrue('FOO'.isupper())
                self.assertFalse('Foo'.isupper())
        def test_split(self):
                s = 'hello world'
                self.assertEqual(s.split(), ['hello', 'world'])
# check that s.split fails when the separator is not a string
                with self.assertRaises(TypeError):
                s.split(2)
 if __name__ == '__main__':
        unittest.main()
```

- A testcase is created by subclassing unittest.TestCase. The three individual tests are defined with methods whose names start with the letters test. This naming convention informs the test runner about which methods represent tests.
- The crux of each test is a call to assertEqual() to check for an expected result; assertTrue() or assertFalse() to verify a condition; or assertRaises() to verify that a specific exception gets raised. These methods are used instead of the assert statement so the test runner can accumulate all test results and produce a report.
- The setUp() and tearDown() methods allow you to define instructions that will be executed before and after each test method. They are covered in more detail in the section Organizing test code.
- The final block shows a simple way to run the tests. unittest.main() provides a command-line interface to the test script.

- When run from the command line, the above script produces an output that looks like this:

...

Ran 3 tests in 0.000s

OK

- Passing the -v option to your test script will instruct unittest.main() to enable a higher level of verbosity, and produce the following output:

test_isupper (__main__.TestStringMethods) ... ok

test_split (__main__.TestStringMethods) ... ok

test_upper (__main__.TestStringMethods) ... ok

Ran 3 tests in 0.001s

OK

- The above examples show the most commonly used unittest features which are sufficient to meet many everyday testing needs. The remainder of the documentation explores the full feature set from first principles.

- Making Assertions
- unittest is part of the Python standard library and a good place to start our unit test walk-through. A unit test consists of one or more assertions (statements that assert that some property of the code being tested is true). The word "assert" literally means, "to state as fact." This is what assertions in unit tests do as well.
- self.assertTrue is rather self explanatory, it asserts that the argument passed to it evaluates to True. The unittest.TestCase class contains a number of assert methods, so be sure to check the list and pick the appropriate methods for your tests. Using assertTrue for every test should be considered an anti-pattern as it increases the cognitive burden on the reader of tests. Proper use of assert methods state explicitly exactly what is being asserted by the test (e.g. it is clear what assertIsInstance is saying about its argument just by glancing at the method name).

Case Study

&

Academic Project

K.RaviKanth, Ast.Prof., Dept. of CSE, IIIT- RGUKT-Basara, T.S, IND.

IIIT BASARA

Infosys | Campus Connect

For Details Contact Me @ :

9247448766

ravikanth27787@gmail.com

python

Programming Language