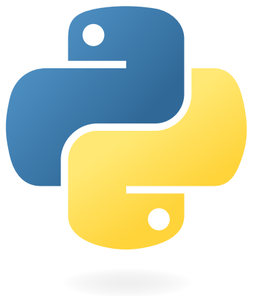
NETWORK PROGRAMMING

University of Tripoli
Faculty of Information Technology
Department of Networking

Assistant Lecturer / Marwa Ebrahim Elwakedy



python

What is Network Programming?

- Network programs: Programs that use network in some way to do their work.
  - Send/receive data across a network
  - Provide/invoke services over a network
  - Mobile computing through wireless networks
  - Cloud/edge computing

- Network programming is the discipline of designing and implementing network programs.

Python Networking

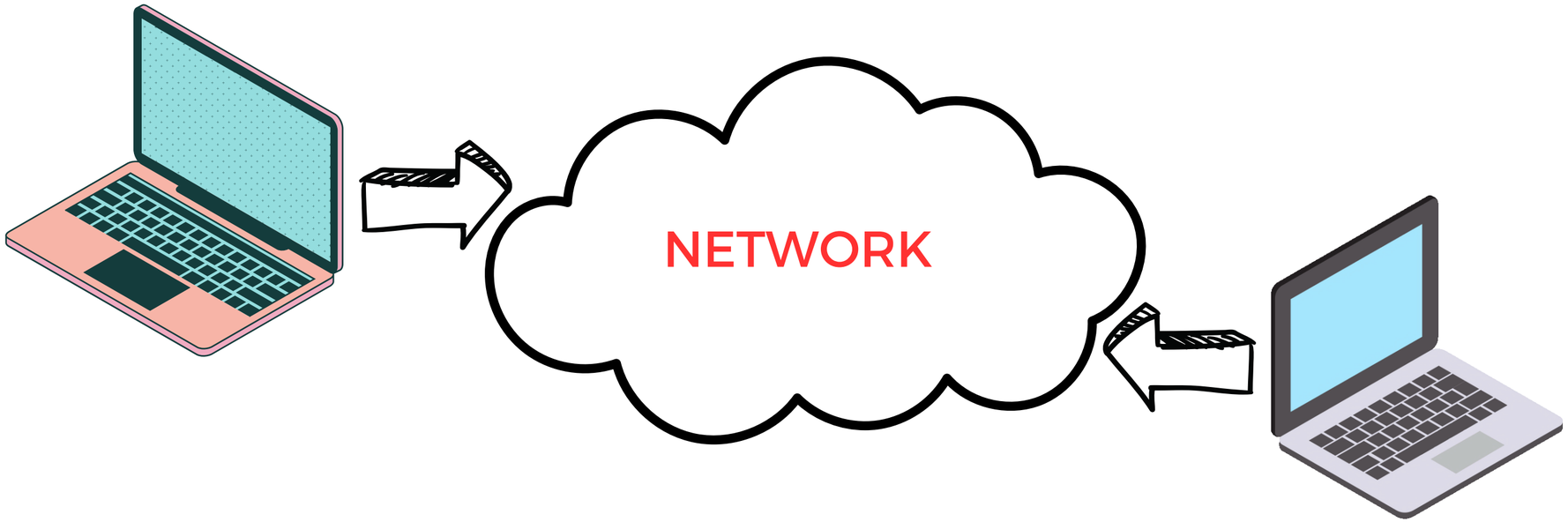
- Network programming is a major use of Python.
- Python standard library has wide support for network protocols, data encoding/decoding, and other things you need to make it work.
- Writing network programs in Python tends to be substantially easier than in C/C++.

Python Networking Levels

- Python provides two levels of access to network services.
- Low level: can access the basic socket support in the underlying OS
  - connection-oriented
  - connectionless
- High level (protocol level): libraries for various application level network protocols
  - FTP, HTTP, POP3, SMTP, ...

The Problem

- Communication between computers



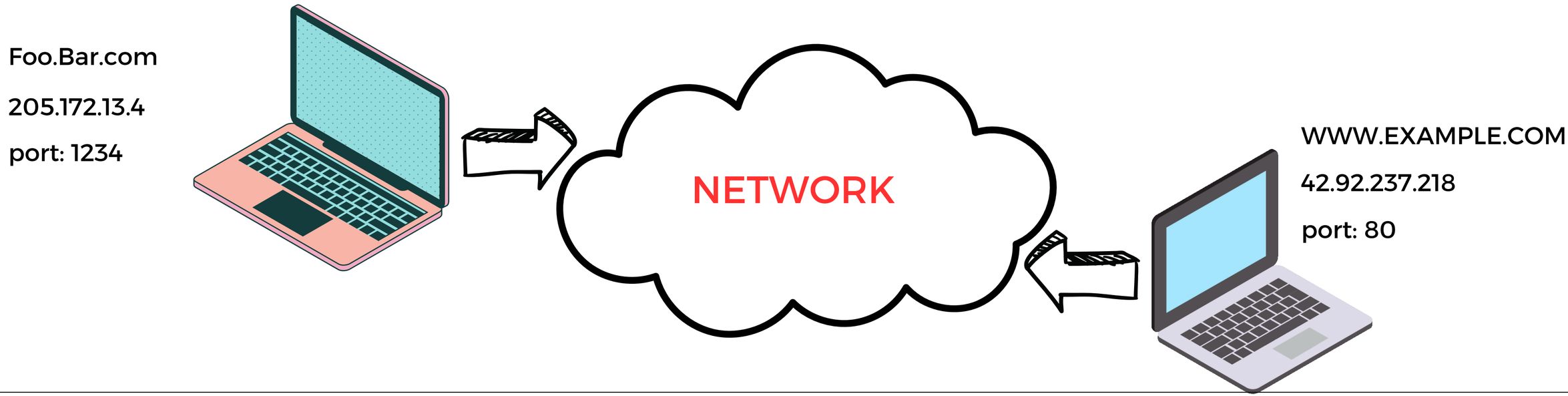
- It’s just sending/receiving bits

Three Main Issues:

- Addressing
  - Specifying a remote computer and service
- Data transport
  - Moving data(bits) back and forth
- Meaningful conversation
  - Conversation in proper order
  - Understand each other

Networking Address:

- Addressing
  - Specifying a remote computer and service



Standard Ports:

- Ports for common services are preassigned

  21 FTP

  22 SSH

  23 Telnet

  25 SMTP (Mail)

  80 HTTP (Web) 1

  10 POP3 (Mail)

  119 NNTP (News)

  443 HTTPS (web)

- Other port numbers may just be randomly assigned to programs by the OS

Using netstat

- Use 'netstat' to view active network connections

```
shell % netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 0 *:imaps *:* LISTEN
tcp 0 0 *:pop3s *:* LISTEN
tcp 0 0 localhost:mysql *:* LISTEN
tcp 0 0 *:pop3 *:* LISTEN
tcp 0 0 *:imap2 *:* LISTEN
tcp 0 0 *:8880 *:* LISTEN
tcp 0 0 *:www *:* LISTEN
```

- Note: Must execute from the command shell on both Unix and Windows

Connections:

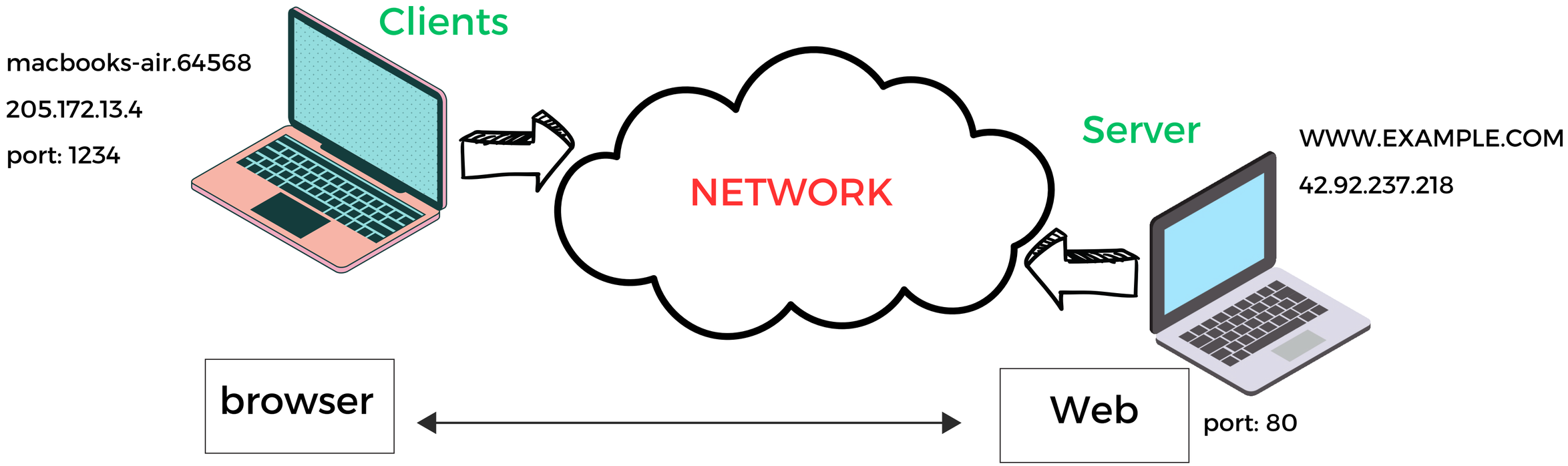
- Each endpoint of a network connection is always represented by a host and port #
- In Python you write it out as a tuple (host, port)

```
("www.python.org", 80)
"205.172.13.4", 443)
```

- In almost all of the network programs you'll write, you use this convention to specify a network address

Client/Server Concept:

- Each endpoint is a running program
- Servers wait for incoming connections and provide a service (e.g., web, mail, etc.)
- Clients make connections to servers



Request/Response Cycle

```
import requests
def request_response(url):
    response = requests.get(url)
    status_code = response.status_code
    print("Status code:", status_code)
    if status_code == 200:
        content = response.content
        print("Content:", content)
    else: print("Error:", response.reason)
if __name__ == "__main__":
    url = "https://www.google.com"
    request_response(url)
```

Request/Response Cycle

- The first line imports the requests library. This library provides a simple and easy-to-use interface for making HTTP requests.

```
import requests
```

- The request_response() function takes a URL as input and makes a GET request to that URL. The function returns a Response object, which contains information about the response, such as the status code, the content, and the headers

```
def request_response(url):
```

- The requests.get() method makes a GET request to the specified URL and returns a Response object.

```
response = requests.get(url)
```

Request/Response Cycle

- The status_code attribute of the Response object contains the status code of the response. The status code is an integer that indicates the success or failure of the request.

```
status_code = response.status_code
```

- The print() statement prints the status code of the response.

```
print("Status code:", status_code)
```

- The if statement checks if the status code is 200. The status code 200 indicates that the request was successful.

```
if status_code == 200:
```

Request/Response Cycle

- The content attribute of the Response object contains the content of the response. The content is the data that was returned by the server in response to the request.

```
content = response.content
```

- The print() statement prints the content of the response.

```
print("Content:", content)
```

- The else block is executed if the status code is not 200.

```
else: print("Error:", response.reason)
```

Request/Response Cycle

- The print() statement prints the reason for the error. The reason is a string that describes the error that occurred.

```
print("Error:", response.reason)
```

- The if __name__ == "__main__": block is executed if the script is run as the main program. The url variable is set to the URL of the Google website. The request_response() function is called with the url variable as input.

```
if __name__ == "__main__":
 url = "https://www.google.com"
 request_response(url)
```

Request/Response Cycle

- Most network programs use a request/response model based on messages.
- Client sends a request message (e.g., HTTP)

```
GET /index.html HTTP/1.0
```

- Server sends back a response message

```
HTTP/1.0 200 OK Content-
type: text/html Content-
length: 48823 ...
```

- The exact format depends on the application

Using Telnet:

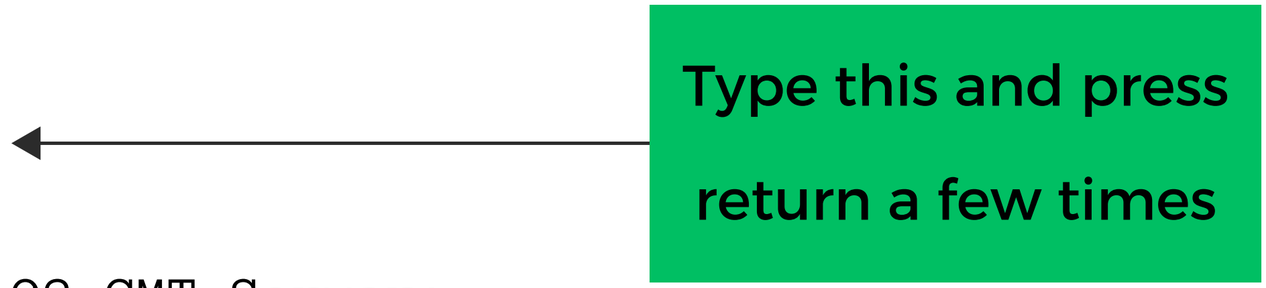
- As a debugging aid, telnet can be used to directly communicate with many services

```
telnet hostname portnum
```

- Example:

```
shell % telnet www.python.org 80
Trying 82.94.237.218...
Connected to www.python.org. Escape
character is '^]'.
GET /index.html HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 31 Mar 2008 13:34:03 GMT Server:
Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2
mod_ssl/2.2.3 OpenSSL/0.9.8c
```



Data Transport

- There are two basic types of communication

- Streams (TCP): Computers establish a connection with each other and read/write data in a continuous stream of bytes---like a file. This is the most common.

- Datagrams (UDP): Computers send discrete packets (or messages) to each other. Each packet contains a collection of bytes, but each packet is separate and self-contained.

Sockets:

- Programming abstraction for network code
- Socket: A communication endpoint

- Supported by socket library module.
- Allows connections to be made and data to be transmitted in either direction

Python Socket Support

- Python supports socket networking through the socket module.
- The module provides the BSD socket interface.
- The socket() function create socket objects.
- Various functions(gethostbyname(), gethostbyaddr() ...) to get commn related info.
- The send()/recv() function send/receive data through the socket.

Python Socket Support

```
import socket

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

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

Protocols

- Send/receive data back and forth is meaningless if we can't understand each other.
- To conduct meaningful conversation:
  - Follow agreed upon rules of message exchange
  - Provide data in proper format
- This is done through protocols.
- Python offers protocol modules for many networking tasks/applications.

Finding service name

```
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()
```

Conclusion

- Python networking support is rich and friendly.
- Use socket programming for low-level or short/simple message exchange.
- Use TCP/UDP for normal client-server programming.
- Use application protocol modules for corresponding services.