

Network Programming

#14



Serdar ARITAN

Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

Python provides two levels of access to network programming.

Low-Level Access: At the low level, you can access the basic socket support of the operating system. You can implement client and server for both connection-oriented and connectionless protocols.


High-Level Access: At the high level allows to implement protocols like HTTP, FTP, etc.

Consider a bidirectional communication channel, the sockets are the endpoints of this communication channel. These sockets (endpoints) can **communicate** within a process, **between** processes on **the same machine**, or between processes on **different machines**.

Sockets use different protocols for determining the connection type for port-to-port communication between clients and servers.

What is socket?

Sockets act as bidirectional communications channel where they are endpoints of it.



Layer	Protocols
Application	HTTP, DNS
Transport	TCP, UDP
Internet	IP
Network Access	Ethernet

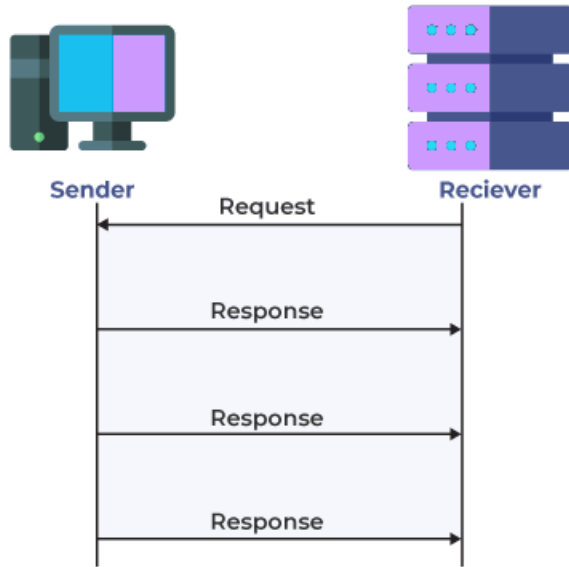
Port #	Application Layer Protocol	Type	Description
20	FTP	TCP	File Transfer Protocol - data
21	FTP	TCP	File Transfer Protocol - control
22	SSH	TCP/UDP	Secure Shell for secure login
23	Telnet	TCP	Unencrypted login
25	SMTP	TCP	Simple Mail Transfer Protocol
53	DNS	TCP/UDP	Domain Name Server
67/68	DHCP	UDP	Dynamic Host
80	HTTP	TCP	HyperText Transfer Protocol
123	NTP	UDP	Network Time Protocol
161,162	SNMP	TCP/UDP	Simple Network Management Protocol
389	LDAP	TCP/UDP	Lightweight Directory Authentication Protocol
443	HTTPS	TCP/UDP	HTTP with Secure Socket Layer

Sockets act as bidirectional communications channel where they are endpoints of it.

Term	Description
Domain	The set of protocols used for transport mechanisms like AF_INET, PF_INET, etc.
Type	Type of communication between sockets
Protocol	Identifies the type of protocol used within domain and type. Typically it is zero
Port	The server listens for clients calling on one or more ports. it can be a string containing a port number, a name of the service, or a Fixnum port
Hostname	Identifies a network interface. It can be a <ul style="list-style-type: none">• a string containing hostname, IPv6 address, or a double-quad address.• an integer• a zero-length string• a string "<broadcast>"



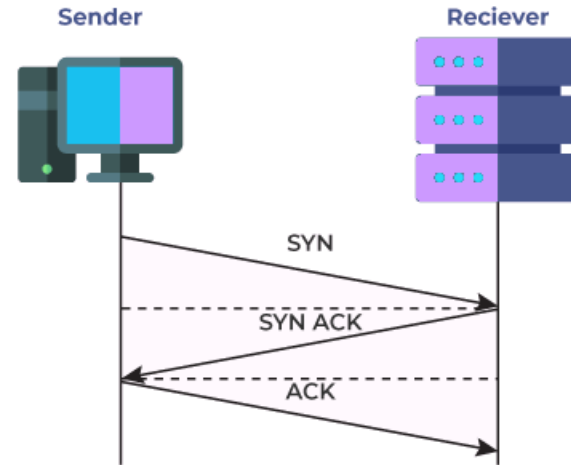
UDP



Where UDP is Used?

Gaming
Video Streaming
Online Video Chats

TCP



Where TCP is Used?

Sending Emails
Transferring Files
Web Browsing

Sockets Programming

Socket programming is a way of **connecting two nodes** on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while the other socket reaches out to the other to form a connection.

```
import socket
```

to create a socket we have to use the `socket.socket()` method.

Syntax:

```
socket.socket(socket_family, socket_type, protocol=0)
```

Where,

`socket_family`: Either `AF_UNIX` or `AF_INET`

`socket_type`: Either `SOCK_STREAM` or `SOCK_DGRAM`.

`protocol`: Usually left out, defaulting to 0.



Socket Server Methods

Function Name	Description
s.bind()	Binds address to the socket. The address contains the pair of hostname and the port number.
s.listen()	Starts the TCP listener
s.accept()	Passively accepts the TCP client connection and blocks until the connection arrives

Socket Client Methods

Function Name	Description
s.connect()	Actively starts the TCP server connection



Socket General Methods

Function Name	Description
<code>s.send()</code>	Sends the TCP message
<code>s.sendto()</code>	Sends the UDP message
<code>s.recv()</code>	Receives the TCP message
<code>s.recvfrom()</code>	Receives the UDP message
<code>s.close()</code>	Close the socket
<code>socket.gethostname()</code>	Returns the host name

```
import socket

s = socket.socket()
print ("Socket successfully created")
# reserve a port on your computer
port = 40674
# Bind to the port which have not typed any ip in the ip field this makes the server listen
to requests coming from other computers on the network
s.bind('', port)
print (f"socket binded to {port}")

# put the socket into listening mode
s.listen(5)
print ("socket is listening")

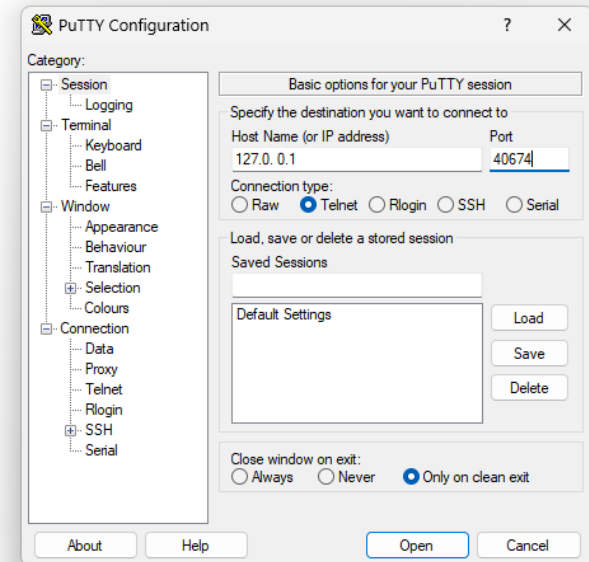
while True:
    # Establish connection with client.
    c, addr = s.accept()
    print ('Got connection from', addr )

    # send a thank you message to the client.
    c.send(b'Thank you for connecting')

    # Close the connection with the client
    c.close()
```




PuTTY is an SSH and telnet client, developed originally by Simon Tatham for the Windows platform. PuTTY is open source software that is available with source code and is developed and supported by a group of volunteers.



Ports can be scanned to check which ports are engaged and which ports are open or free. In Python “Socket” module provides access to the BSD socket interface, which is available on all platforms.

To scan the ports, the following steps can be implemented:

- 1] Recognize the host's IP address
 - 2] Create new socket
 - 3] Forming a connection with port
 - 4] Checks whether data is received or not
 - 5] Close connection
- To use the socket module, we have to import it :

Simple Port Scanner with Sockets

```
import socket
```

```
#getting ip-address of host
```

```
ip = socket.gethostbyname (socket.gethostname())
```

```
#check for all available ports
```

```
for port in range(65535):
```

```
    try:
```

```
        # create a new socket
```

```
serv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
# bind socket with address
```

```
serv.bind((ip,port))
```

```
except:
```

```
    #print open port number
```

```
    print(' [OPEN] Port open :',port)
```

```
serv.close() #close connection
```

```
socket.gethostname()
```

```
'SA-LENOVO'
```

```
[OPEN] Port open : 135
```

```
[OPEN] Port open : 139
```

```
[OPEN] Port open : 445
```

```
[OPEN] Port open : 5040
```

```
[OPEN] Port open : 49664
```

```
[OPEN] Port open : 49665
```

```
[OPEN] Port open : 49666
```

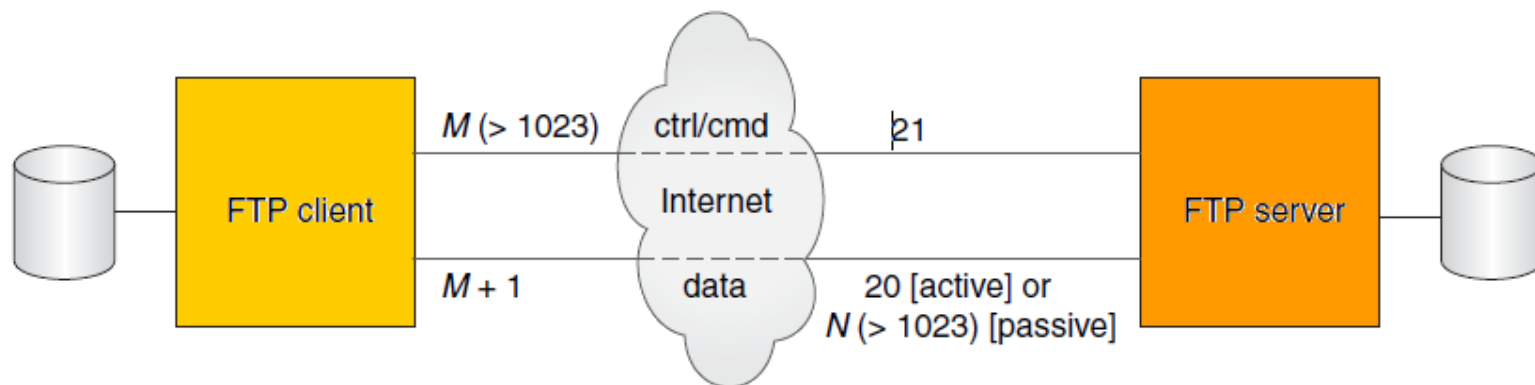
```
[OPEN] Port open : 49667
```

```
[OPEN] Port open : 49668
```

```
[OPEN] Port open : 49670
```

```
[OPEN] Port open : 56480
```

The File Transfer Protocol (FTP) was developed by the late Jon Postel and Joyce Reynolds in the Internet Request for Comment (RFC) 959 document and published in October 1985. It is primarily used to download publicly accessible files in an anonymous fashion. You must have a login/password to access the remote host running the FTP server. The exception is anonymous logins, which are designed for guest downloads



1. Client contacts the FTP server on the remote host
2. Client logs in with username and password (or anonymous and e-mail address)
3. Client performs various file transfers or information requests
4. Client completes the transaction by logging out of the remote host and FTP server

Python and FTP

Under the hood, it is good to know that **FTP** uses only **TCP**. —it does **not** use **UDP** in any way. Also, FTP can be seen as a more unusual example of client/server programming because both the clients and the servers use a pair of sockets for communication: one is the control or command port (port 21), and the other is the data port (sometimes port 20). We say sometimes because there are two FTP modes: Active and Passive, and the server's data port is only 20 for Active mode. After the server sets up 20 as its data port, it “actively” initiates the connection to the client's data port. For Passive mode, the server is only responsible for letting the client know where its random data port is; the client must initiate the data connection. As you can see in this mode, the FTP server is taking a more passive role in setting up the data connection. Finally, there is now support for a new Extended Passive Mode to support version 6 Internet Protocol (IPv6) addresses.


```
>>> from ftplib import FTP
>>> f = FTP('lidy.hacettepe.edu.tr')
>>> f.login('serdar.aritan', '*****')
'230 User saritan logged in'
>>> f.dir()
drwxr-xr-x    2 saritan  akd          3 Dec 21 15:06 dev
drwxr-xr-x    2 saritan  akd          5 Nov  3 06:19 mail
-rw-r--r--    1 saritan  akd         22 Nov  3 06:19 pass
drwxr-xr-x   16 saritan  akd         57 Nov 26 18:56 public_html
drwxr-xr-x    2 saritan  akd          3 Nov  3 06:19 temp
>>> f.quit()
'221 Goodbye.'
```

```
>>> f.dir()  
drwxr-xr-x    2 saritan  akd  
drwxr-xr-x    2 saritan  akd  
-rw-r--r--    1 saritan  akd  
drwxr-xr-x   26 saritan  akd  
drwxr-xr-x    2 saritan  akd  
  
>>> f.cwd
```

- connect
- cwd**
- debug
- debugging
- delete
- dir
- encoding
- file
- getline
- getmultiline

```
-  
3 Dec 21 2015 dev  
5 Nov 3 2015 mail  
22 Nov 3 2015 pass  
83 Apr 25 12:40 public_html  
2 Feb 6 2017 temp
```

PYTHON PROGRAMMING

Python and FTP

login(user='anonymous', passwd='', acct='') Log in to FTP server; all arguments are optional

pwd() Current working directory

cwd(path) Change current working directory to path

dir([path[,...[,cb]]) Displays directory listing of path; optional callback cb passed to **retrlines**()

nlst([path[,...]]) Like **dir**() but returns a list of filenames instead of displaying

retrlines(cmd [, cb]) Download text file given FTP cmd, for example, RETR filename; optional callback cb for processing each line of file

retrbinary(cmd, cb[, bs=8192[, ra]]) Similar to **retrlines**() except for binary file; callback cb for processing each block (size bs defaults to 8K) downloaded required

storlines(cmd, f) Upload text file given FTP cmd, for example, STOR filename; open file object f required

storbinary(cmd, f[, bs=8192]) Similar to **storlines**() but for binary file; open file object f required, upload blocksize bs defaults to 8K

rename(old, new) Rename remote file from old to new

delete(path) Delete remote file located at path




mkd(directory) Create remote directory

rmd(directory) Remove remote directory

quit() Close connection and quit

```
>>> from ftplib import FTP
>>> f = FTP('lidy.hacettepe.edu.tr')
>>> f.login('saritan', '*****')
'230 User saritan logged in'
>>> f.pwd()
'/'
>>> f.cwd('public_html')
'250 CWD command successful'
>>> f.pwd()
'/public_html'
>>> f.retrlines('LIST')
drwxr-xr-x    2 saritan   akd           13 Dec 10 14:42 bco601
-rw-r--r--    1 saritan   akd          2283 Dec 10 14:40 bco601.htm
'226 Transfer complete'
>>> f.quit()
'221 Goodbye.'
```

```
from ftplib import FTP
f = FTP('lidya.hacettepe.edu.tr')
f.login('saritan', '*****')
f.cwd('public_html')
f.retrlines('LIST', open('dir.txt', 'w').write)
f.retrbinary('RETR bca607.htm', open('web.htm', 'wb').write)
f.quit()
```

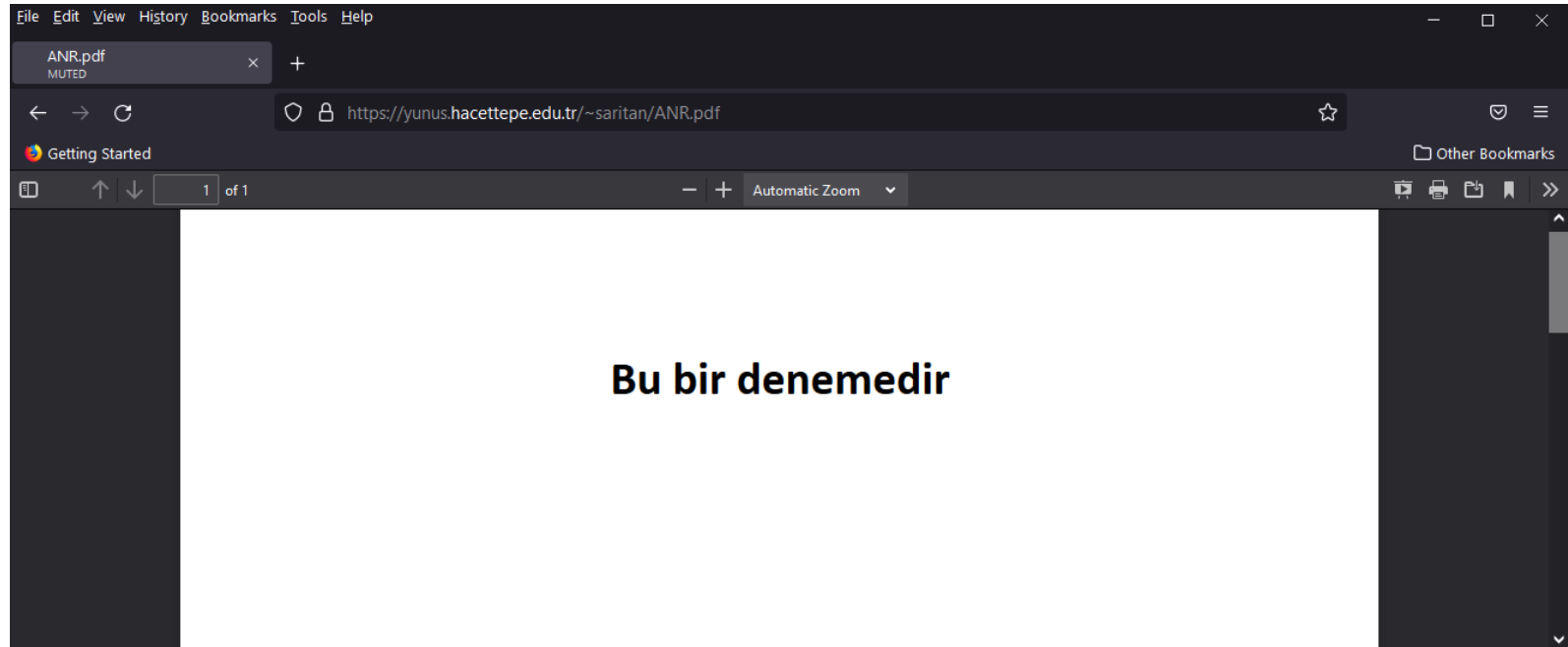
Name	Date modified	Type	Size
 dir	29-Dec-21 11:35 AM	Text Document	5 KB
 ftp_1	29-Dec-21 11:35 AM	Python File	1 KB
 web	29-Dec-21 11:35 AM	Microsoft Edge ...	4 KB

```
import ftplib
import os

def upload(ftp, file):
    ext = os.path.splitext(file)[1]
    if ext in (".txt", ".htm", ".html"):
        ftp.storlines("STOR " + file, open(file, 'rb'))
    else:
        ftp.storbinary("STOR " + file, open(file, 'rb'), 1024)

ftp = ftplib.FTP('lidya.hacettepe.edu.tr')
ftp.login('saritan', '*****')

upload(ftp, "SBA.pdf")
upload(ftp, "web.htm")
upload(ftp, "msFTP.jpg")
```

Simple Mail Transfer Protocol (SMTP) is a protocol, which handles sending e-mail and routing e-mail between mail servers.

```
import smtplib
```

```
smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

host: This is the host running your SMTP server. You can specify IP address of the host or a domain name like hacettepe.edu.tr. This is optional argument.

port: If you are providing host argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.

local_hostname: If your SMTP server is running on your local machine, then you can specify just localhost as of this option

An `smtplib` and an `smtplib.SMTP` class to instantiate.

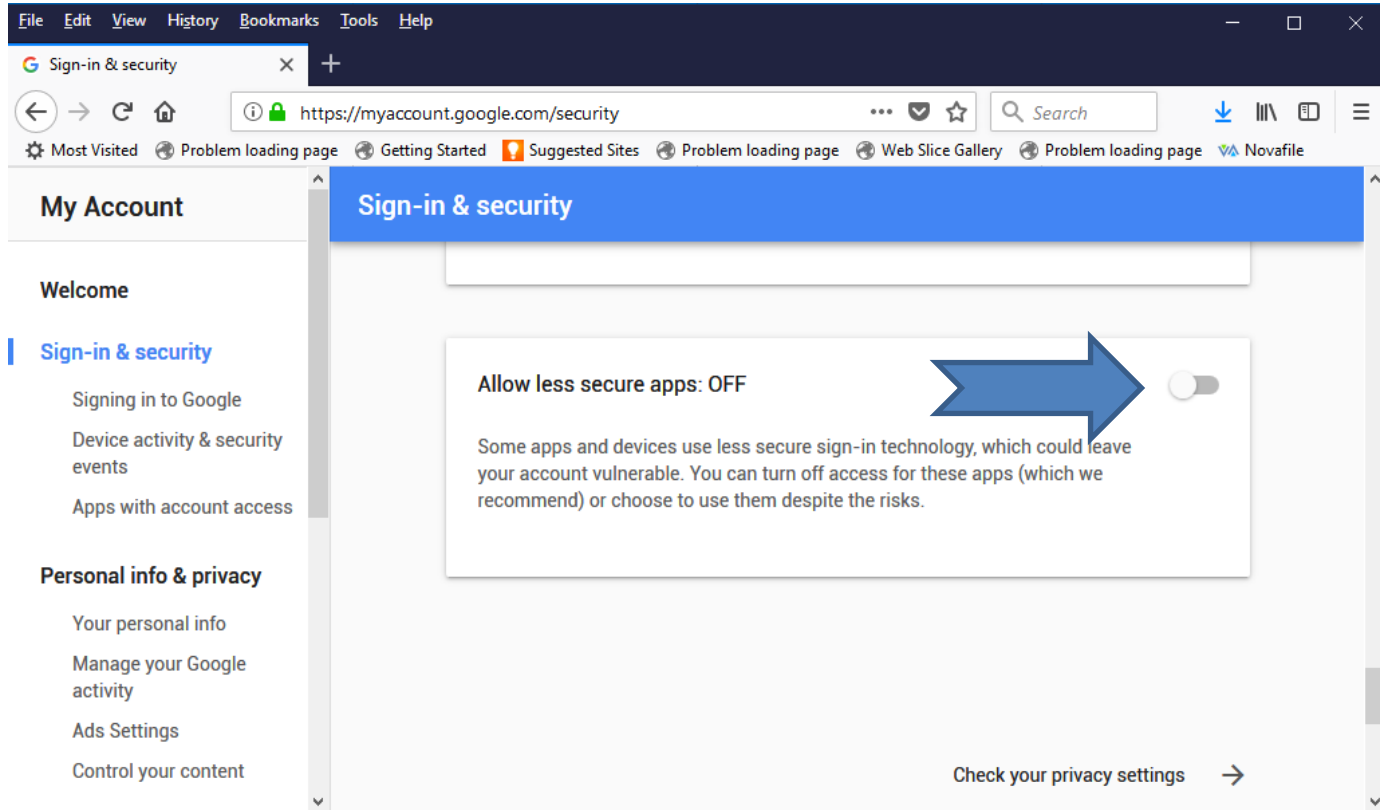
1. Connect to server
2. Log in (if applicable)
3. Make service request(s)
4. Quit

In addition to the `smtplib.SMTP` class, Python 2.6 introduced another pair. `SMTP_SSL`. If omitted, the default port for `SMTP_SSL` is 465. For most e-mail sending applications, only two are required `sendmail()` and `quit()`.

Python and SMTP

Method	Description
<code>sendmail(<i>from</i>, <i>to</i>, <i>msg</i>[, <i>mopts</i>, <i>ropts</i>])</code>	Sends <i>msg</i> from <i>from</i> to <i>to</i> (list/tuple) with optional ESMTP mail (<i>mopts</i>) and recipient (<i>ropts</i>) options.
<code>ehlo()</code> or <code>helo()</code>	Initiates a session with an SMTP or ESMTP server using EHLO or HELO, respectively. Should be optional because <code>sendmail()</code> will call these as necessary.
<code>starttls(<i>keyfile</i>=None, <i>certfile</i>=None)</code>	Directs server to begin Transport Layer Security (TLS) mode. If either <i>keyfile</i> or <i>certfile</i> are given, they are used in the creation of the secure socket.
<code>set_debuglevel(<i>level</i>)</code>	Sets the debug level for server communication.
<code>quit()</code>	Closes connection and quits.
<code>login(<i>user</i>, <i>passwd</i>)^a</code>	Log in to SMTP server with <i>user</i> name and <i>passwd</i> .

a. SMTP-AUTH only.



<https://myaccount.google.com/security>

PYTHON PROGRAMMING

Python and SMTP

```
import smtplib
from email.mime.text import MIMEText
from sys import platform

def send_email(subject, body, sender, recipient, password):
    msg = MIMEText(body)
    msg['Subject'] = subject
    msg['From'] = sender
    msg['To'] = recipient
    smtp_server = smtplib.SMTP_SSL('smtp.gmail.com', 465)
    smtp_server.login(sender, password)
    smtp_server.sendmail(sender, recipient, msg.as_string())
    smtp_server.quit()

subject = "BC0601 Ara Sınav"
body = """Merhabalar,
Ara sınavlarınızı e-posta ile gönderirken mutlaka konu "Subject" kısmına
dersin kodunu "BC0601" yazmayı unutmayın.
"""
sender = "serdar.aritan@gmail.com"

if platform == "darwin": # OS X
    password = "xxxxxxxxxxxxxx"
elif platform == "win32": # Windows
    password = "xxxxxxxxxxxxxx"
```



```
# Using readlines()
file1 = open('eMail_bco601.txt', 'r')
Lines = file1.readlines()

# Strips the newline character
for line in Lines:
    recipient = line.strip()
    send_email(subject, body, sender, recipient, password)
    print(f"eMail has been sent to : {recipient}")
```

the Internet Message Access Protocol, or IMAP. (IMAP has also been known by various other names: “Internet Mail Access Protocol,” “**Interactive Mail Access Protocol**” and “Interim Mail Access Protocol.”). The current version of IMAP in use today is IMAP4, that is widely used.

1. Connect to server
2. Log in
3. Make service request(s)
4. Quit

Method	Description
<code>close()</code>	Closes the current mailbox. If access is not set to read-only, any deleted messages will be discarded.
<code>fetch(message_set, message_parts)</code>	Retrieve e-mail messages (or requested parts via <i>message_parts</i>) stated by <i>message_set</i> .
<code>login(user, password)</code>	Logs in <i>user</i> by using given <i>password</i> .
<code>logout()</code>	Logs out from the server.
<code>noop()</code>	Ping the server but take no action (“no operation”).
<code>search(charset, *criteria)</code>	Searches mailbox for messages matching at least one piece of <i>criteria</i> . If <i>charset</i> is False, it defaults to US-ASCII.
<code>select(mailbox= 'INBOX' , read-only=False)</code>	Selects a <i>mailbox</i> (default is INBOX); user not allowed to modify contents if <i>readonly</i> is set.

```
import imaplib

mailserver = imaplib.IMAP4_SSL('mail.hacettepe.edu.tr', 993)
username = 'saritan'
password = '*****'

mailserver.login(username, password)
status, count = mailserver.select('Inbox')
status, data = mailserver.fetch(count[0], '(UID BODY[TEXT])')
print (data[0][1])
mailserver.close()
mailserver.logout()
```

PYTHON PROGRAMMING

Python and URL

The `urllib.request` module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')

>>> print(f.read(300)) #displays the first 300 bytes
b'<!doctype html>\n<!--[if lt IE 7]>    <html class="no-js ie6
lt-ie7 lt-ie8 lt-ie9">    <![endif]-->\n<!--[if IE 7]>    <html
class="no-js ie7 lt-ie8 lt-ie9">    <![endif]-->\n<!--[if
IE 8]>    <html class="no-js ie8 lt-ie9">
<![endif]-->\n<!--[if gt IE 8]><!--><html class="no-js"'
```

```
>>> with urllib.request.urlopen('http://www.hacettepe.edu.tr/') as f:  
    print(f.read(100).decode('utf-8'))
```

```
<!doctype html>
```

```
<html>
```

```
<head>
```

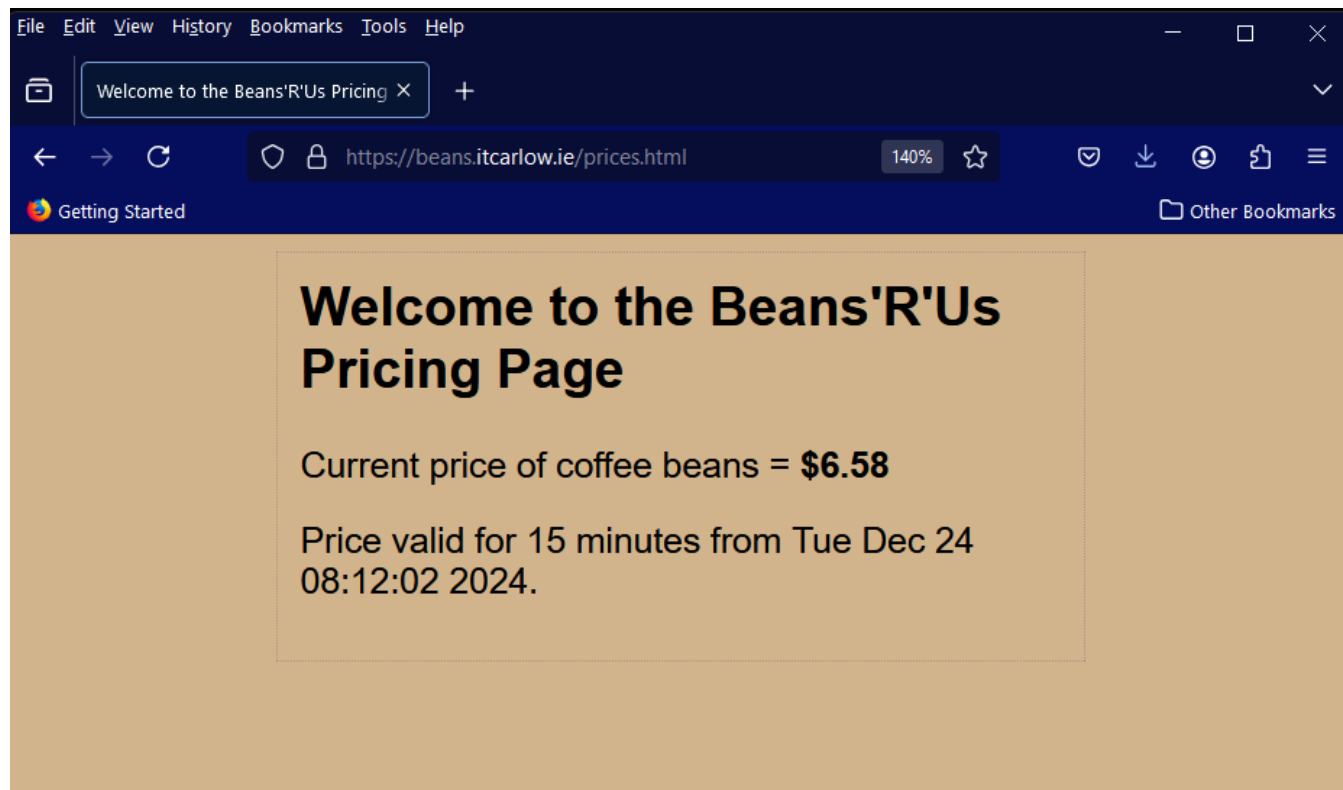
```
    <meta charset="UTF-8">
```

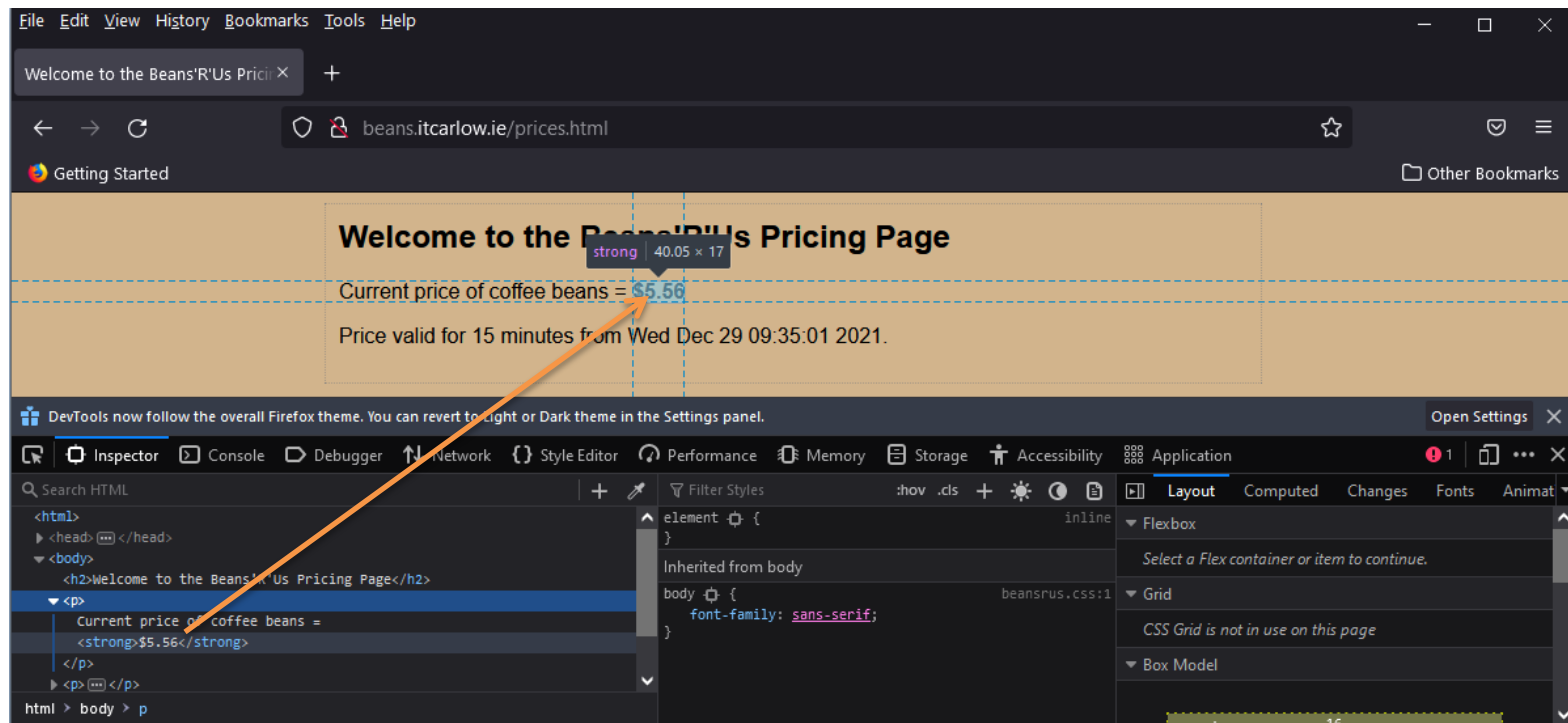
```
    <meta name="viewport" content="width=de
```

The URL parsing functions focus on splitting a URL string into its components, or on combining URL components into a URL string

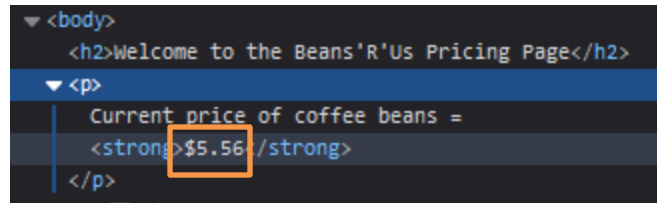
```
>>> from urllib.parse import urlparse
>>> o=urlparse('https://yunus.hacettepe.edu.tr/~saritan/bco601.htm')
>>> o
ParseResult(scheme='https', netloc='yunus.hacettepe.edu.tr',
path='/~saritan/bco601.htm', params='', query='', fragment='')
>>> o.scheme
'https'
>>> o.geturl()
'https://yunus.hacettepe.edu.tr/~saritan/bco601.htm'
```

`http://beans.itcarlow.ie/prices.html`

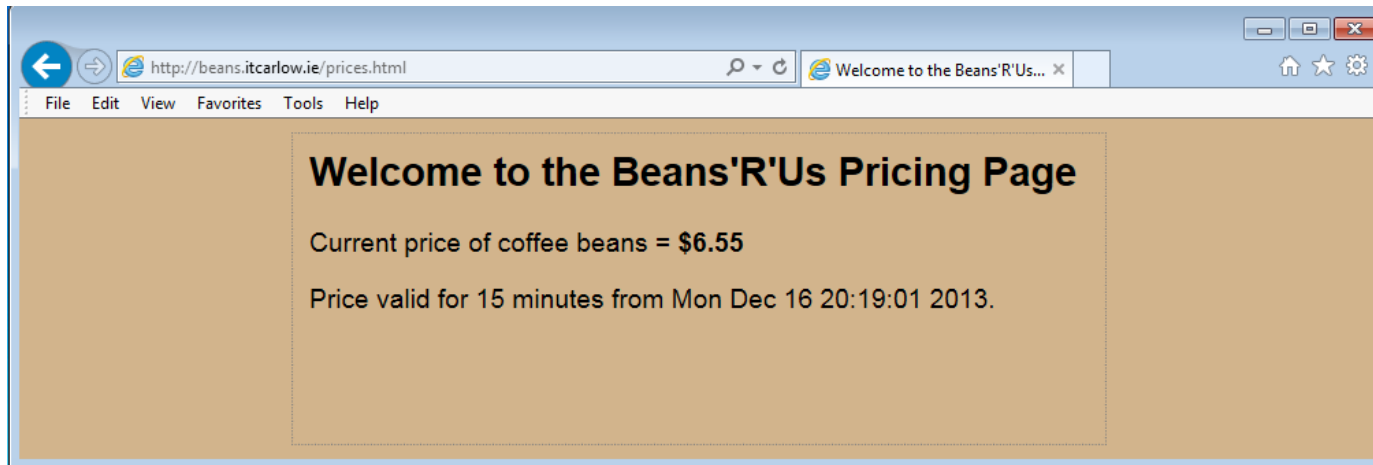




```
import urllib.request
price = 99.99
while price > 4.74:
    page = urllib.request.urlopen("http://beans.itcarlow.ie/prices.html")
    text = page.read().decode("utf8")
    where = text.find('>$')
    start_of_price = where + 2
    end_of_price = start_of_price + 4
    price = float(text[start_of_price:end_of_price])
print ("Buy!")
```



```
<body>
  <h2>Welcome to the Beans'R'Us Pricing Page</h2>
  <p>
    Current price of coffee beans =
    <strong>$5.56</strong>
  </p>
```



The Web was initially developed to be a global online repository or archive of documents (mostly educational and research-oriented). Such pieces of information generally come in the form of static text and usually in HTML. HTML is not as much a language as it is a text formatter, indicating changes in font types, sizes, and styles. The main feature of HTML is in its hypertext capability. This refers to the ability to designate certain text (usually highlighted in some fashion) or even graphic elements as links that point to other “documents” or locations on the Internet and Web that are related in context to the original. Such a document can be accessed by a simple mouse click or other user selection mechanism. These (static) HTML documents live on the Web server and are sent to clients when requested.

The Python standard library has everything you need to handle the standard internet protocols and to create both clients and servers.

The standard library has a number of modules for writing servers for various network protocols. In many cases, you can create a server in only a few lines of code. Suppose we want to make a particular folder's files freely accessible via HTTP, perhaps to share a few files with coworkers without the hassle of setting up a formal repository or file share. With Python, **we don't need to install and configure a server.**

```
>>> from http.server import HTTPServer, SimpleHTTPRequestHandler
>>> server = HTTPServer(("", 8080), SimpleHTTPRequestHandler)
>>> server.serve_forever()
```

This server will serve the contents of the folder it's run in on port 8080 for all active network interfaces. The tuple `("", 8080)` sets the address and port for the server.



```
import sys
import http.server
import socketserver

Handler = http.server.SimpleHTTPRequestHandler

if sys.argv[1:]:
    PORT = int(sys.argv[1])
else:
    PORT = 8080
httpd = socketserver.TCPServer(("", PORT), Handler)

print("serving at port", PORT)
httpd.serve_forever()
```

`http.server` can also be invoked directly using the `-m` switch of the interpreter with a port number argument. This serves files relative to the current directory.

```
python -m http.server 8080
```

Python and HTTP Server

index.htm

<!doctype html>

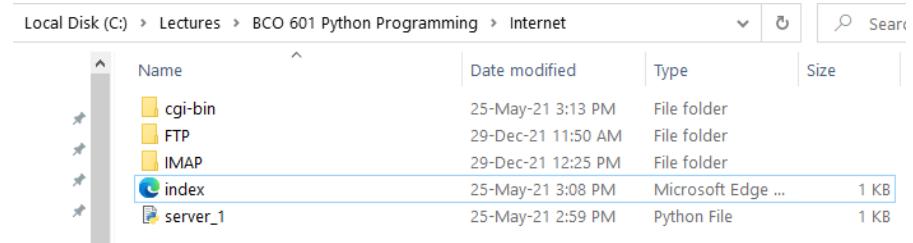
<html>

<body>

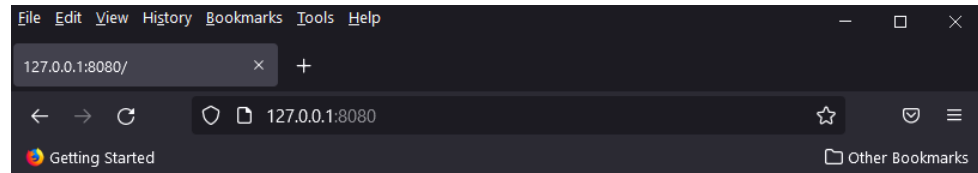
Hello, from Python world!

</body>

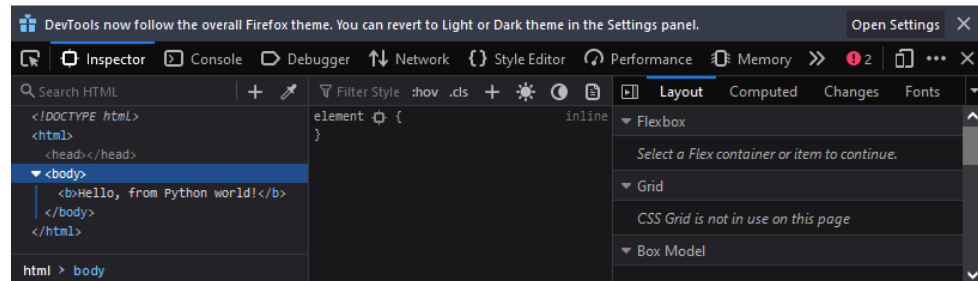
</html>



Local Disk (C:) > Lectures > BCO 601 Python Programming > Internet				
Name	Date modified	Type	Size	
cgi-bin	25-May-21 3:13 PM	File folder		
FTP	29-Dec-21 11:50 AM	File folder		
IMAP	29-Dec-21 12:25 PM	File folder		
index	25-May-21 3:08 PM	Microsoft Edge ...	1 KB	
server_1	25-May-21 2:59 PM	Python File	1 KB	



Hello, from Python world!



Writing Python code to interact with an HTTP server is also quite easy. The `urllib.request` module is designed to enable interaction with URLs in all their real world complexity, including authentication, cookies, redirections, and the like.

```
>>> from urllib.request import urlopen
>>> url_file = urlopen("http://localhost:8080")
>>> print(url_file.geturl())
http://localhost:8080
>>> print(url_file.info())
Server: SimpleHTTP/0.6 Python/3.10.1
Date: Wed, 29 Dec 2021 10:13:29 GMT
Content-type: text/html
Content-Length: 89
Last-Modified: Tue, 25 May 2021 12:08:28 GMT
```

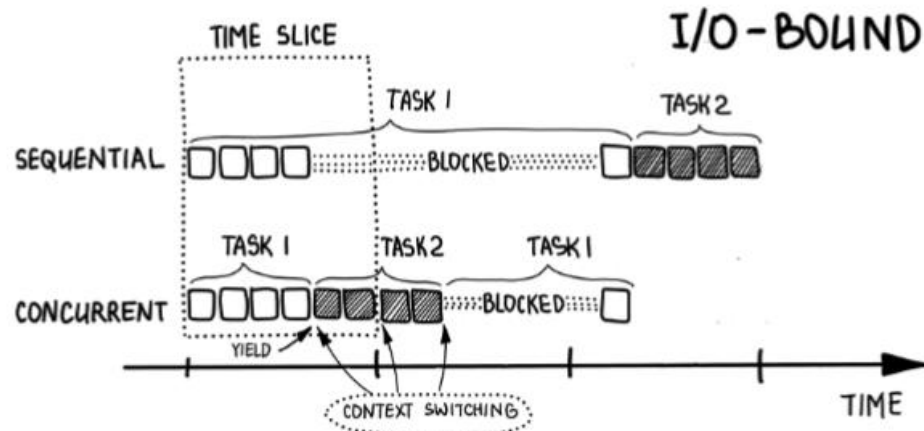
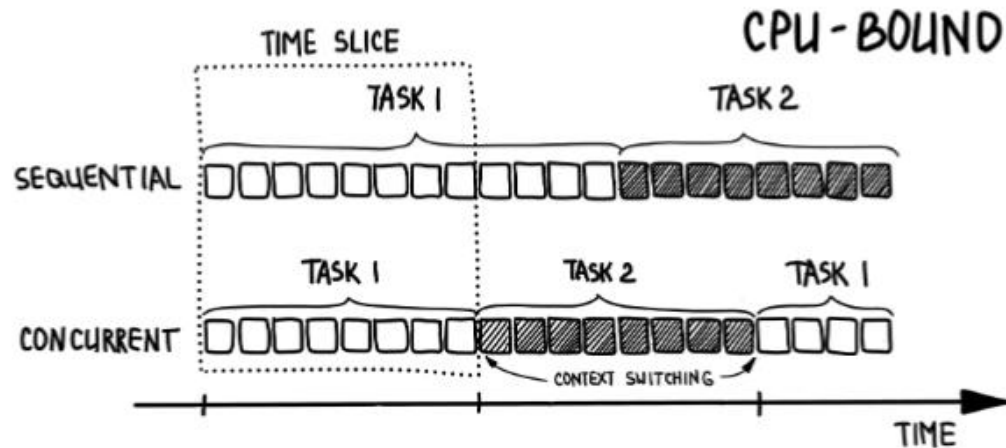

Although the `http.server` module has the basics of a web server, you need more than the basics to write a full-featured web application. You need to manage users, authentication, and sessions; you need a way to generate HTML pages. The solution to this is to use a web framework, and over the years many frameworks have been created in Python, leading to the present generation, which includes Zope and Plone, **Django**, TurboGears, web2py, and many more.

Asynchrony in Python

The operating system only sees **your code** as running in a **single process**, with a **single thread**; it is Python itself that manages the multitasking, with some help from you, thereby sidestepping some of the issues that crop up with threading. Still, writing good **asynchronous code in Python** requires some forethought and planning.

It's important to keep in mind that **asynchrony is not parallelism**. In Python, a mechanism called the Global Interpreter Lock (GIL) ensures that any single Python process is constrained to a single CPU core, regardless of how many cores are available to the system.

Asynchrony in Python



The **Collatz conjecture** is one of the most famous unsolved problems in mathematics. The conjecture asks whether repeating two simple arithmetic operations will eventually transform every positive integer into 1.

1. Start with any positive integer n .
2. If n is even, the next term in the sequence should be $n / 2$.
3. If n is odd, the next term in the sequence should be $3 * n + 1$.
4. If n is 1, stop.

Even if you start with a fantastically large number, you'll always wind up at 1 after relatively few steps. Starting with 942 488 749 153 153, for example, the Collatz sequence arrives at 1 in only 262 steps.

Asynchrony in Python

Let's create a simple game that challenges the user to guess how many **Collatz sequences** have a particular length. We will restrict the range of starting numbers to integers between 2 and 1,000,000.

For example, exactly **782** starting numbers will yield **Collatz sequences** with a length of exactly **42** values.

Asynchrony in Python

```
import time

BOUND= 10**6
def collatz(n):
    steps = 0
    while n > 1:
        if n % 2:
            n = n * 3 + 1
        else:
            n = n / 2
        steps += 1
    return steps

def length_counter(target):
    count = 0
    for i in range(2, BOUND):
        if collatz(i) == target:
            count += 1
    return count
```

```
def get_input(prompt):  
    while True:  
        n = input(prompt)  
        try:  
            n = int(n)  
        except ValueError:  
            print("Value must be an integer.")  
            continue  
        if n <= 0:  
            print("Value must be positive.")  
        else:  
            return n
```

```
def main():  
    print("Collatz Sequence Counter")  
    target = get_input("Collatz sequence length to search for: ")  
    print(f"Searching in range 1-{BOUND}...")  
  
    count = length_counter(target)  
    print(f"Your target number {target} occurred {count} times")  
  
if __name__ == "__main__":  
    # Start timer  
    start_time = time.time()  
    main()  
    end_time = time.time()  
  
    # Calculate elapsed time  
    elapsed_time = end_time - start_time  
    print("Elapsed time: ", elapsed_time)
```

Asynchrony in Python

```
===== RESTART: C:\Lectures\BCO 601 Python Programming\asynch\collatz.py  
Collatz Sequence Counter  
Collatz sequence length to search for: 88  
Searching in range 1-1000000...  
Your target number 88 occurred 4091 times  
Elapsed time: 22.951388835906982
```

```
===== RESTART: C:\Lectures\BCO 601 Python Programming\asynch\collatz.py  
Collatz Sequence Counter  
Collatz sequence length to search for: 88  
Searching in range 1-1000000...  
Your target number 88 occurred 4091 times  
Elapsed time: 21.74228024482727
```


Asynchrony in Python

```
import asyncio
```

```
BOUND = 10**6
```

```
def collatz(n):  
    steps = 0  
    while n > 1:  
        if n % 2:  
            n = n * 3 + 1  
        else:  
            n = n / 2  
        steps += 1  
    return steps
```

This function will always **return almost instantly**, so it neither needs to call an awaitable nor run concurrently with another awaitable. It can remain as a normal function..

length_counter() is labor intensive and CPU-bound.

```
async def length_counter(target):
```

```
    count = 0
```

```
    for i in range(2, BOUND):
```

```
        if collatz(i) == target:
```

```
            count += 1
```



```
        # allow other tasks to run for a moment
```

```
        # await asyncio.sleep(0)
```

```
    return count
```

This function turn into a **coroutine function** with **async** and use **await asyncio.sleep(0)** to tell Python where the coroutine function can pause and let something else work.




```
def get_input(prompt):  
    while True:  
        n = input(prompt)  
        try:  
            n = int(n)  
        except ValueError:  
            print("Value must be an integer.")  
            continue  
        if n <= 0:  
            print("Value must be positive.")  
        else:  
            return n
```

```
async def main():
    print("Collatz Sequence Counter")
    target = get_input("Collatz sequence length to search for: ")
    print(f"Searching in range 1-{BOUND}...")

    length_counter_task = asyncio.create_task(length_counter(target))
    count = await length_counter_task
    print(f"Your target number {target} occurred {count} times")

if __name__ == "__main__":
    # Start timer
    start_time = time.time()
    asyncio.run(main())
    end_time = time.time()

    # Calculate elapsed time
    elapsed_time = end_time - start_time
    print("Elapsed time: ", elapsed_time)
```



Asynchrony in Python

```
=== RESTART: C:\Lectures\BCO 601 Python Programming\asynch\asynch_collatz.py
Collatz Sequence Counter
Collatz sequence length to search for: 88
Searching in range 1-1000000...
Your target number 88 occurred 4091 times
Elapsed time: 19.947562217712402
```

```
=== RESTART: C:\Lectures\BCO 601 Python Programming\asynch\asynch_collatz.py
Collatz Sequence Counter
Collatz sequence length to search for: 88
Searching in range 1-1000000...
Your target number 88 occurred 4091 times
Elapsed time: 19.881373643875122
```