Advanced Python Programming

Chapter 10 & 11

Concurrent Image Processing

Roadmap

- Image processing fundamentals
- Applying concurrency to image processing
- Good concurrent image processing practices

Image processing fundamentals

Applications of Image Processing

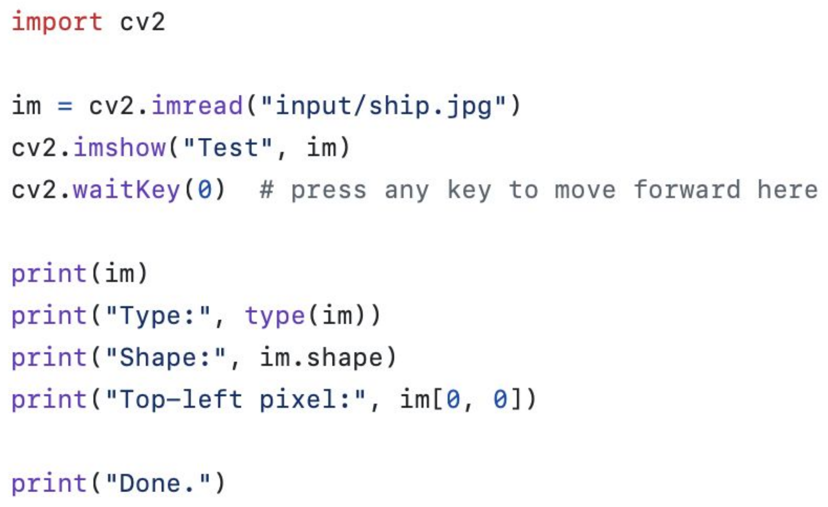
- **Pattern Recognition**: Facial detection and object recognition.
- **Classification**: Categorizing the content of images.
- **Feature Extraction**: Extracting critical information from images.

Why Python is Ideal for Image Processing

- **Matrix Representation**:
  - Images are represented as 2D or 3D matrices.
  - Python excels in matrix computation with powerful libraries.

Key Libraries for Image Processing

1. **OpenCV**:
   - Offers professional tools for image and video processing.
2. **NumPy**:
   - Efficiently handles arrays and matrix computations.

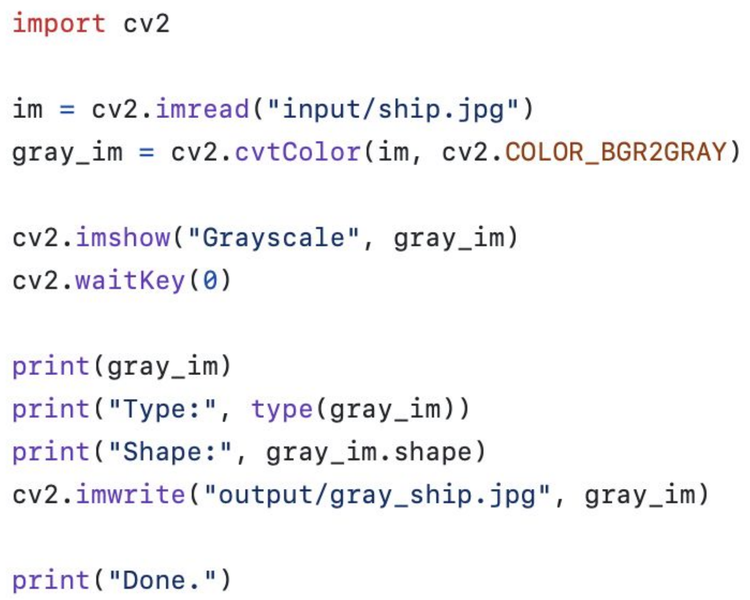


```python
import cv2

im = cv2.imread("input/ship.jpg")
cv2.imshow("Test", im)
cv2.waitKey(0)  # press any key to move forward here

print(im)
print("Type:", type(im))
print("Shape:", im.shape)
print("Top-left pixel:", im[0, 0])

print("Done.")
```

Image processing fundamentals

Grayscaling

- **Definition**:
  - Simplifies image pixel data by retaining only the brightness information (light intensity).
- **Transformation**:
  - Converts a 3D image (RGB) into a 2D representation.
  - Grayscale values range from **0 (black)** to **255 (white)**.
- **Advantages**:
  - Reduces computational complexity.
  - Preserves essential brightness information for analysis.



```python
import cv2

im = cv2.imread("input/ship.jpg")
gray_im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

cv2.imshow("Grayscale", gray_im)
cv2.waitKey(0)

print(gray_im)
print("Type:", type(gray_im))
print("Shape:", gray_im.shape)
cv2.imwrite("output/gray_ship.jpg", gray_im)

print("Done.")
```

Image processing fundamentals

Simple Thresholding

Definition:

- For each pixel:
  - If the brightness value exceeds a specified threshold, set it to **white**.
  - If the brightness value is below the threshold, set it to **black**.

Purpose:

- Enhance the contrast between high-intensity and low-intensity pixels.
- Extract **key features and patterns** from the image.

Choosing the Right Threshold:

  - Select an appropriate threshold to divide the image pixels into regions.
  - Make different parts of the image more prominent and easier to analyze.

```python
import cv2

im = cv2.imread("input/ship.jpg")
gray_im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

ret, custom_thresh_im = cv2.threshold(gray_im, 127, 255, cv2.THRESH_BINARY)
cv2.imwrite("output/custom_thresh_ship.jpg", custom_thresh_im)

print("Done.")
```

Image processing fundamentals

Adaptive Thresholding

Definition:

- Dynamically calculates a threshold rather than using a fixed global threshold.

Techniques:

1. **Adaptive Mean Thresholding**:
   - Computes the threshold based on the **mean intensity** of pixels in the neighborhood.
   - Subtracts a constant value from the computed mean.
2. **Adaptive Gaussian Thresholding**:
   - Similar to Adaptive Mean Thresholding, but uses a **Gaussian-weighted sum**.
   - Assigns higher weights to pixels closer to the center of the neighborhood.

```python
import cv2

im = cv2.imread("input/ship.jpg")
im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

mean_thresh_im = cv2.adaptiveThreshold(
    im, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 11, 2
)
cv2.imwrite("output/mean_thresh_ship.jpg", mean_thresh_im)

gauss_thresh_im = cv2.adaptiveThreshold(
    im, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2
)
cv2.imwrite("output/gauss_thresh_ship.jpg", gauss_thresh_im)

print("Done.")
```

Applying concurrency to image processing

Overview example 5

- **Task**: Process 400 cropped images using thresholding techniques.
- **Goal**: Compare performance of **sequential** vs **concurrent processing**.

Methodology

- Used `multiprocessing.Pool`:
  - `starmap()` to map images and parameters to `process_threshold()` function.
- **Thresholding**: Applied adaptive thresholding for feature extraction.
- Processed with 1 to 6 processes for comparison.

Results

- **Execution Time**:
  - **1 process**: 0.6590 seconds
  - **2 processes**: 0.3190 seconds
  - **3-6 processes**: Negligible improvement due to overhead.

Key Insights

- Significant speedup from 1 to 2 processes.
- Overhead limits benefits with >2 processes.
- **Optimal concurrency depends on task size and system resources.**

Applying concurrency to image processing

Overview example 6

- **Task**: Process 400 cropped images using **adaptive thresholding techniques**.
- **Goal**: Optimize performance by integrating **I/O operations** with processing.

Methodology

- Combined file reading and image thresholding in `process_threshold()` function.
- Processed with 1 to 6 processes for comparison.

Results

- **Execution Time**:
  - **1 process**: 0.6590 seconds
  - **2 processes**: 0.3190 seconds
  - **3-6 processes**: Negligible improvement due to overhead.

Key Insights

- **Significant speedup**: From 1 to 3 processes.
- **Diminishing returns**: Overhead limits performance improvement beyond 3 processes.
- **Optimal concurrency**: Depends on the size of input tasks and system resources.

Good concurrent image processing practices

False Positives: Detected objects mistaken for faces due to Variations in face sizes within the image.

Adjust Parameters:

Use `scaleFactor` and `minNeighbors` in `detectMultiScale()` to reduce false positives

```python
import cv2

face_cascade = cv2.CascadeClassifier("input/haarcascade_frontalface_default.xml")

for filename in ["obama1.jpeg", "obama2.jpg"]:
    im = cv2.imread("input/" + filename)
    gray_im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(im)

    for (x, y, w, h) in faces:
        cv2.rectangle(im, (x, y), (x + w, y + h), (0, 255, 0), 2)

    cv2.imshow("%i face(s) found" % len(faces), im)
    cv2.waitKey(0)

    # cv2.imwrite('output/' + filename, im)

print("Done.")
```

Question list

What is an image processing task? Analyzing and manipulating digital images to extract information or create modified versions of the images.

What is the smallest unit of digital imaging? How is it represented in computers? Pixel: Represented as a numerical value indicating color intensity, typically in RGB (red, green, blue) or grayscale.

What is grayscaling? What purpose does this technique serve? Simplifies images by retaining only brightness information, converting them to grayscale (values between 0–255). Reduces computational complexity and focuses on intensity information.

What is thresholding? What purpose does this technique serve? Binarize an image by setting pixels above a threshold to white and below it to black.Enhances contrast and isolates key regions for feature extraction.

Why should image processing be made concurrent? Image processing often involves computationally intensive tasks and large datasets.

What are some good practices for concurrent image processing? Choose suitable methods for the problem (e.g., adaptive vs simple thresholding).

Building Communication Channels with asyncio

Question list

1. What is a communication channel? What is its connection to asynchronous programming?
2. What are the two main parts of the OSI model protocol layers? What purpose does each of them serve?
3. What is the transport layer? Why is it crucial to communication channels?
4. How does asyncio facilitate the implementation of server-side communication channels?
5. How does asyncio facilitate the implementation of client-side communication channels?
6. What is aiofiles?

Roadmap

- The ecosystem of communication channels
- Getting started with Python and Telnet
- Client-side communication with aiohttp

The ecosystem of communication channels

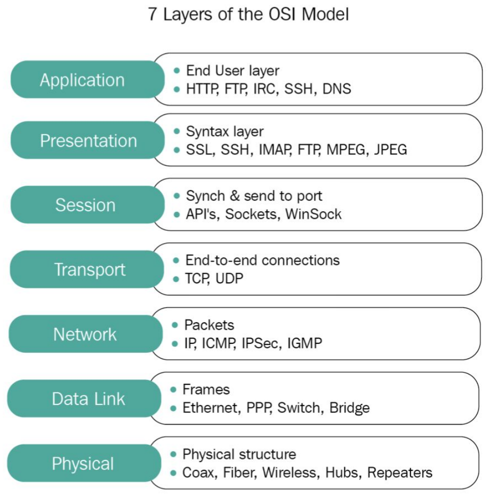
Communication Channels Overview

- **Definition**:
    - Logical: Data communication enabling computer networks.
- **Focus of this Chapter**:
    - Logical communication related to **asynchronous programming**.

The ecosystem of communication channels

OSI Model and Protocol Layers

- **OSI Model**:
  1. Framework defining 7 layers of communication processes.
  2. Layers: Physical, Data Link, Network, Transport, Session, Presentation, Application.
- **Two Key Parts**:
  1. **Media Layers**:
     - **Physical Layer**: Bit-level data transmission (coding schemes, synchronization).
     - **Data Link Layer**: Error detection and correction, framing.
     - **Network Layer**: Packet forwarding and routing between systems.
  2. **Host Layers**:
     - Focused on data handling, interpretation, and user interaction.



The ecosystem of communication channels

Role of Transport Layer

- **What It Does**:
  - Bridges **media** and **host layers**.
  - Ensures reliable **end-to-end (E2E) connections**.
  - Handles packet loss or corruption with error detection methods.
- **Importance for asyncio**:
  - Directly utilized in implementing asynchronous communication channels.

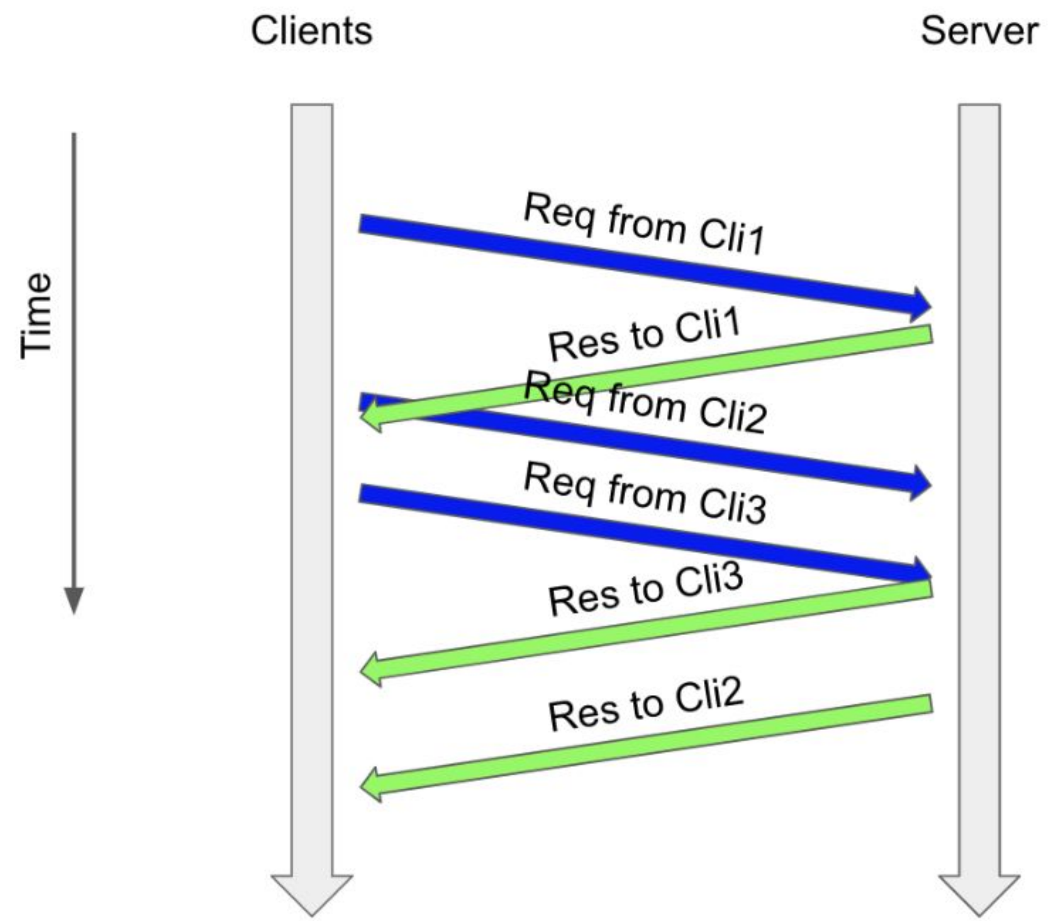


Figure 11.2 – Asynchronous, interleaved HTTP requests

The ecosystem of communication channels

Core Components of asyncio Communication

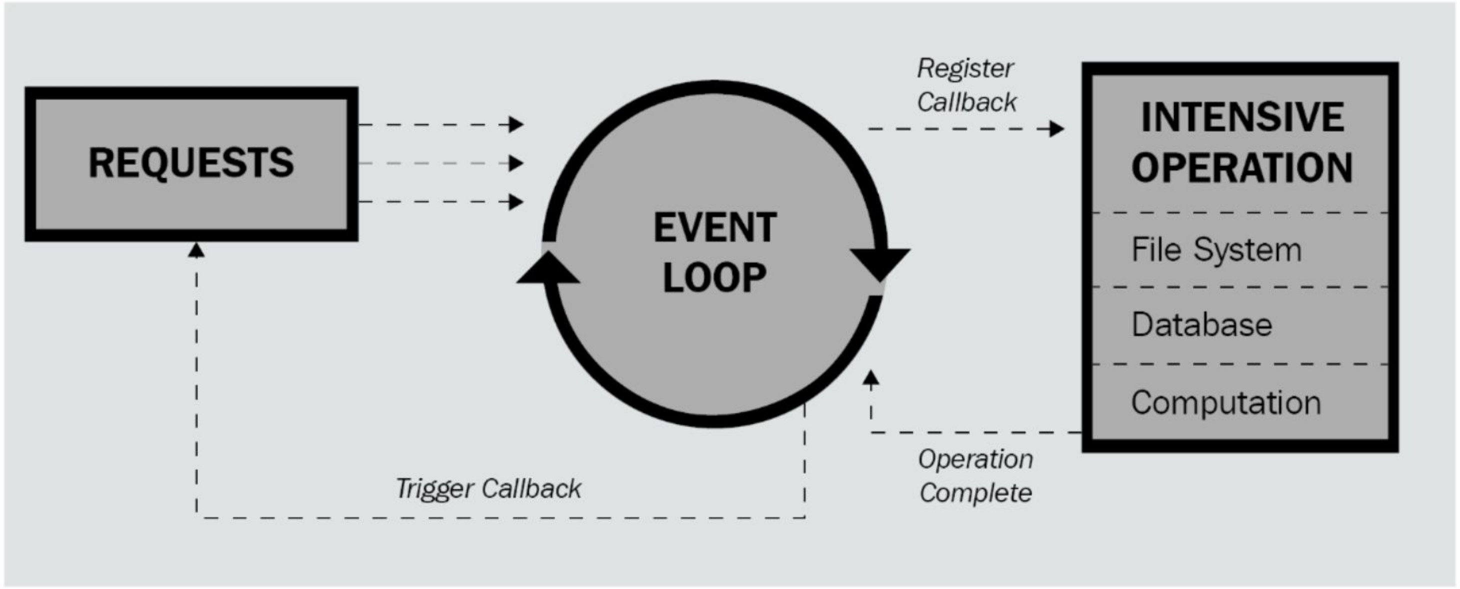
1. Transport Classes:

- Represent low-level communication channels.
- Key methods:
  - **`BaseTransport.get_extra_info()`**:
    - Provides channel-specific details (e.g., socket, remote address).
  - **`BaseTransport.close()`**:
    - Closes the transport and triggers the associated protocol's `connection_lost()`.

2. Protocol Classes:

- Define communication behavior by subclassing `asyncio.Protocol`.
- Key methods:
  - **`connection_made(transport)`**:
    - Called when a connection is established.
    - Stores the transport object for sending data.
  - **`data_received(data)`**:
    - Called when data is received.
    - Processes data (typically in bytes) and sends responses.
    -

The big picture of asyncio's server client



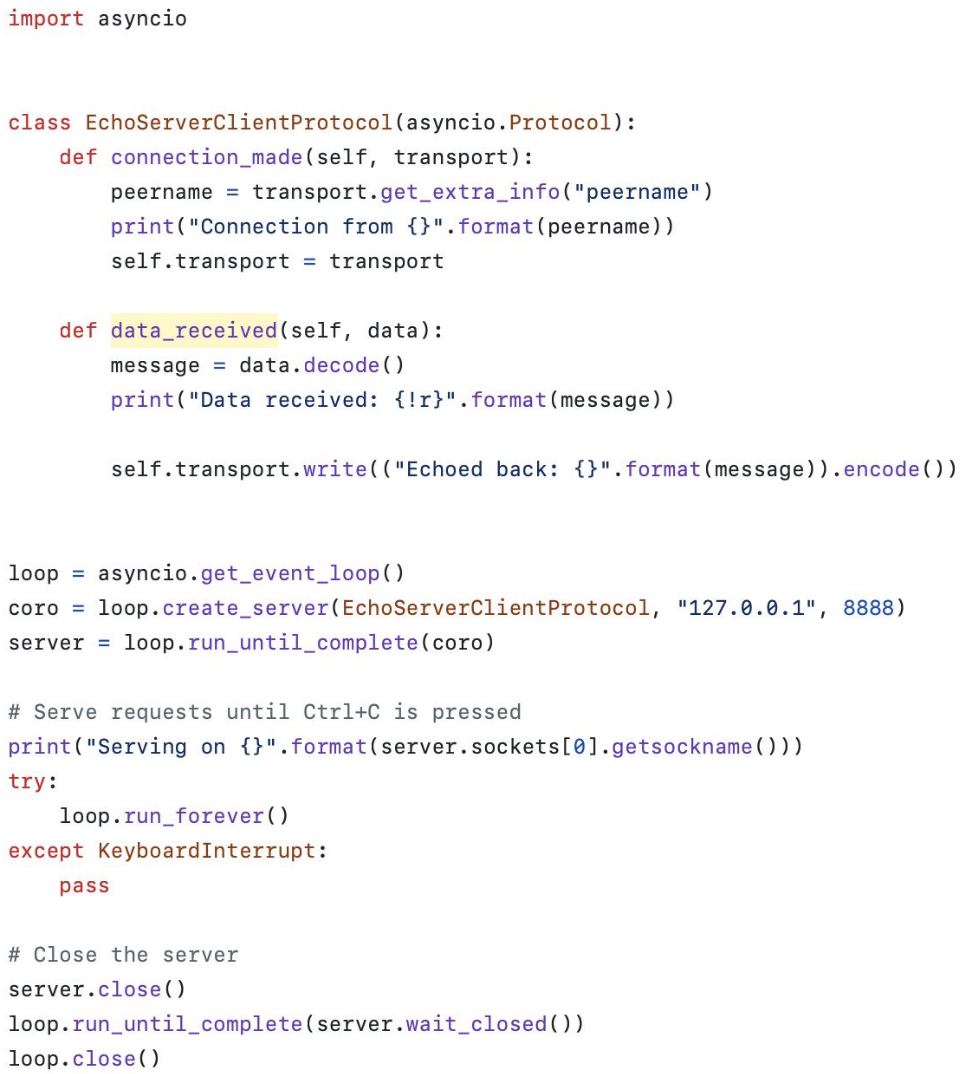
Getting started with Python and Telnet

Starting a server

```python
import asyncio


class EchoServerClientProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info("peername")
        print("Connection from {}".format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print("Data received: {!r}".format(message))


loop = asyncio.get_event_loop()
coro = loop.create_server(EchoServerClientProtocol, "127.0.0.1", 8888)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print("Serving on {}".format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

Getting started with Python and Telnet

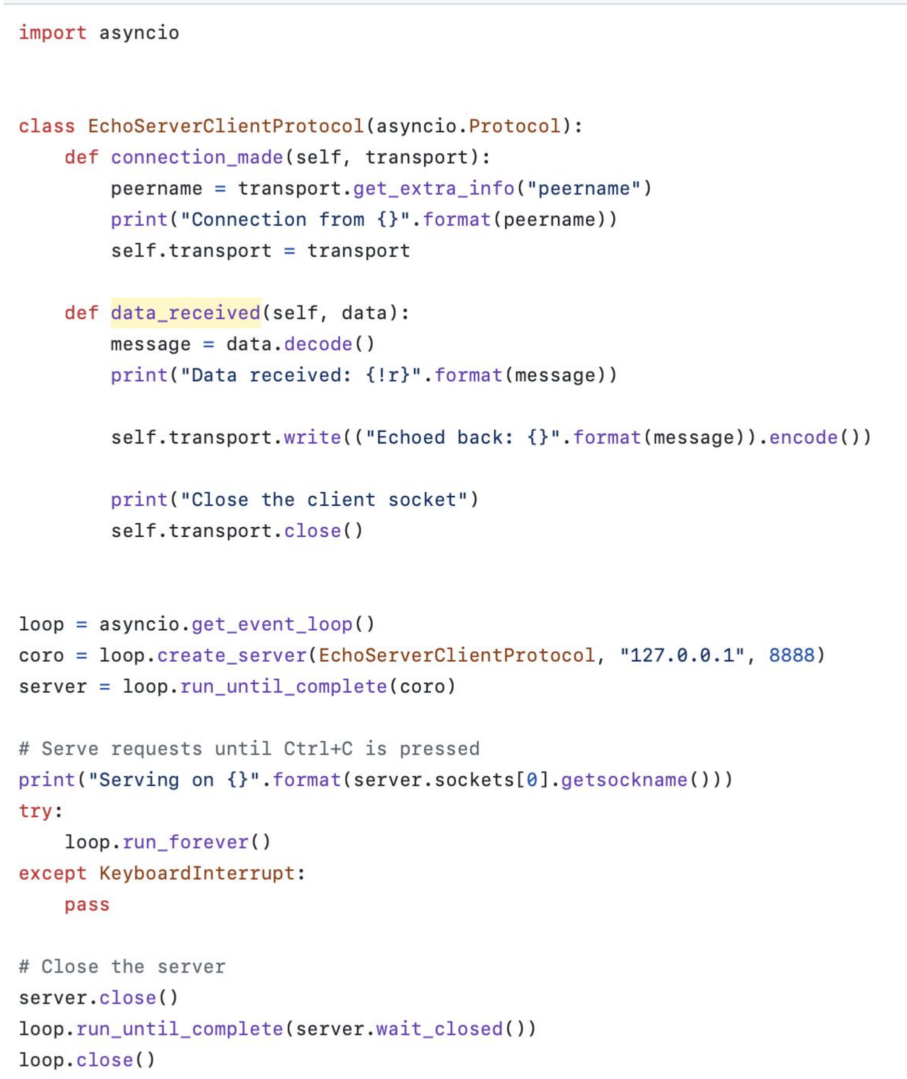
Sending messages back to clients



```
import asyncio


class EchoServerClientProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info("peername")
        print("Connection from {}".format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print("Data received: {!r}".format(message))

        self.transport.write(("Echoed back: {}".format(message)).encode())


loop = asyncio.get_event_loop()
coro = loop.create_server(EchoServerClientProtocol, "127.0.0.1", 8888)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print("Serving on {}".format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

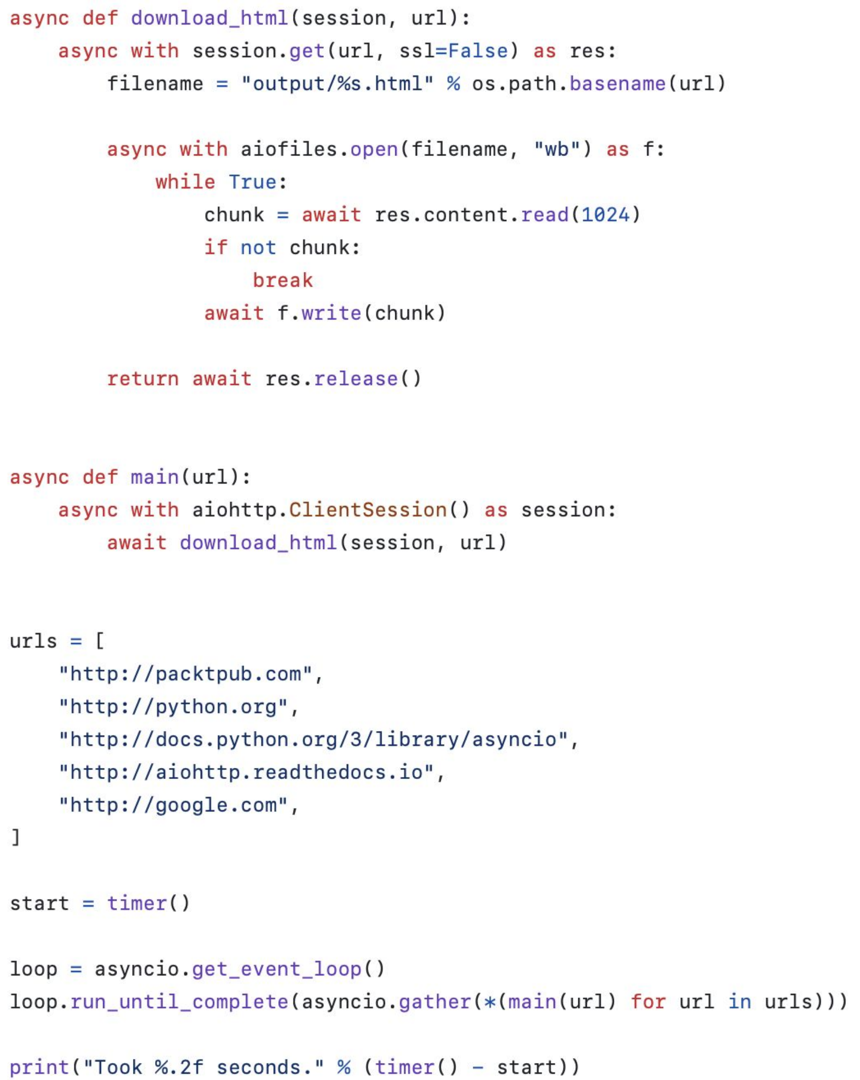
Getting started with Python and Telnet

Closing transports



```python
import asyncio


class EchoServerClientProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info("peername")
        print("Connection from {}".format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print("Data received: {!r}".format(message))

        self.transport.write(("Echoed back: {}".format(message)).encode())

        print("Close the client socket")
        self.transport.close()


loop = asyncio.get_event_loop()
coro = loop.create_server(EchoServerClientProtocol, "127.0.0.1", 8888)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print("Serving on {}".format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

Client-side communication with aiohttp



```
async def download_html(session, url):
    async with session.get(url, ssl=False) as res:
        filename = "output/%s.html" % os.path.basename(url)

        async with aiofiles.open(filename, "wb") as f:
            while True:
                chunk = await res.content.read(1024)
                if not chunk:
                    break
                await f.write(chunk)

        return await res.release()


async def main(url):
    async with aiohttp.ClientSession() as session:
        await download_html(session, url)


urls = [
    "http://packtpub.com",
    "http://python.org",
    "http://docs.python.org/3/library/asyncio",
    "http://aiohttp.readthedocs.io",
    "http://google.com",
]

start = timer()

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.gather(*(main(url) for url in urls)))

print("Took %.2f seconds." % (timer() - start))
```

Back to question list

1. What is a communication channel? What is its connection to asynchronous programming?

- **Communication Channel**:
    - A communication channel is the logical medium through which two systems (like a server and a client) exchange data.
    - Logical channels involve protocols (like HTTP, TCP, etc.).

- **Connection to Asynchronous Programming**:
    - Asynchronous programming allows efficient use of communication channels by enabling non-blocking operations:
        - A server can handle multiple clients simultaneously without waiting for one client's response before serving others.
        - A client can handle multiple requests/responses concurrently without waiting for one task to finish before starting another.

2. What are the two main parts of the OSI model protocol layers? What purpose does each of them serve?

Two Main Parts:

1. **Media Layers** (Physical, Data Link, Network layers):
   - These layers handle the low-level transmission of raw data (bits, frames, packets) between systems.
   - They ensure data is properly prepared for transmission and routed to the correct destination.

2. **Host Layers** (Transport, Session, Presentation, Application layers):
   - These layers manage high-level data processing and user interactions.
   - They handle tasks like reliable data delivery, encryption, user authentication, and presenting information in a user-friendly format.

3. What is the transport layer? Why is it crucial to communication channels?

- **Transport Layer**:
  - The transport layer is the fourth layer in the OSI model.
  - It ensures reliable, end-to-end communication between systems by handling:

- **Why Crucial**:
  - It bridges the gap between low-level data transmission (media layers) and high-level application logic (host layers).
  - It ensures data packets are delivered correctly, in the right order, and without duplication or loss.

4. How does asyncio facilitate the implementation of server-side communication channels?

- **Server-Side Communication with asyncio**:
  - `asyncio` uses **transports and protocols** to manage communication.
  - Key features:
    1. **Asynchronous Event Loop**:
       - Handles incoming client connections, reads data, and writes responses without blocking the program.
    2. **`create_server()` Method**:
       - Creates an asynchronous server by associating a protocol class with a listening address and port.
    3. **Protocol Subclass**:
       - Defines server behavior (e.g., handling client connections and processing data).

5. How does asyncio facilitate the implementation of client-side communication channels?

- **Client-Side Communication with asyncio**:
    - `asyncio` simplifies creating asynchronous clients with tools like `asyncio.open_connection()` for TCP communication.

6. What is aiofiles?

- **aiofiles**:
  - `aiofiles` is a Python library used for performing asynchronous file operations (e.g., reading and writing files) with `asyncio`.
  - It provides non-blocking file I/O, allowing programs to read/write files while continuing to process other tasks.
- **Key Features**:
  - Asynchronous file read (`await aiofiles.open(...).read()`).
  - Asynchronous file write (`await aiofiles.open(...).write()`).
  - Useful in applications that need to handle file I/O without slowing down other tasks, such as logging or saving data during network communication.