CIS 112

Intro to Programming Using Python

Module 5 Part 1

Agenda for the Day!

- Network Programming
  - network sockets,
  - client-server communication,
  - JSON and XML files in network communication,

Network Sockets

Intro to Network Sockets

- In Python, network sockets are endpoints used for communication between two machines over a network.
- Sockets allow programs to communicate with each other, either on the same machine or across a network.
- Python provides a built-in module called **socket** which allows you to create, configure, and use sockets in your programs.

Creating a Socket

- Creating a Socket: You can create a **socket object** using the **socket.socket()** function, which takes two parameters: **socket_family** and **socket_type**.
- The **socket_family** parameter specifies the address family, such as **AF_INET** for **IPv4** or **AF_INET6** for **IPv6**.
- The **socket_type** parameter specifies the type of socket, such as **SOCK_STREAM** for **TCP** or **SOCK_DGRAM** for **UDP**.

```
import socket

# Create a TCP socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Binding the Socket

- **Binding the Socket:** Before you can use a socket to send or receive data, you need to bind it to a specific address and port on the local machine using the bind() method.
- This binding step effectively allocates that machine resource to this application, allowing it to send or receive data through that port.

```
host = '127.0.0.1'  # localhost
port = 12345
s.bind((host, port))
```

Listening

- **Listening for Connections (for server sockets):** If you're creating a server socket (e.g., for a TCP server), typically done for applications that will run continuously and await external traffic requests, you'll typically call the **listen()** method to start listening for incoming connections.
  - You specify the maximum number of queued connections as an argument.
  - In this example, 5 is the backlog queue size. It indicates that the server can handle up to 5 pending connections in the backlog queue. If additional connections arrive while the queue is full, they may be rejected or may experience delays until space becomes available in the queue.

```
s.listen(5)
```

Accepting Connections

- **Accepting Connections (for server sockets):** Once a client tries to connect to the server, you use the **accept()** method to accept the incoming connection.
- This method returns a new socket object and the address of the client.
  - The address of the client is returned because in network communication, particularly in server-client architectures, it's often necessary for the server to know the address of the client it's communicating with.
  - The new socket object is a separate socket instance specifically created for handling communication with the client that initiated the connection. This socket object is distinct from the original listening socket that the server uses to accept incoming connections.

```
client_socket, client_address = s.accept()
```

Connecting to External Connections

- **Connecting to a Server** (for client sockets): If you're creating a client socket (e.g., for a TCP client), you use the connect() method to connect to the server. You specify the server's address and port as arguments (can be local or remotely hosted.

```
server_address = ('127.0.0.1', 12345)
s.connect(server_address)
```

Sending/Receiving Data

- **Sending and Receiving Data:** Once the connection is established, you can send and receive data using the **send()** and **recv()** methods.
  - **send()** Method:
    - The send() method is used to send data over a connected socket. It takes a single argument, which is the data to be sent, typically in the form of a bytes-like object (e.g., bytes or bytearray). The method returns the number of bytes sent, which may be less than the length of the data if the underlying network buffer is full.
  - recv() Method:
  - The recv() method is used to receive data from a connected socket. It takes an optional argument specifying the maximum number of bytes to receive at once. If no argument is provided, it will receive up to a default maximum size. It returns a bytes object containing the data received from the socket. If the connection is closed by the peer, recv() will return an empty bytes object (b'').

```
# Example usage of send() method
data = b"Hello, server!"
bytes_sent = client_socket.send(data)
```

```
# Example usage of recv() method
received_data = client_socket.recv(1024)
```

Closing Time!

Closing the Socket: After you're done using the socket, it's essential to close it using the **close()** method.

```
s.close()
```

Example!

```python
# Client side
import socket

# Create a TCP/IP socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the socket to the server's address and port
client_socket.connect(('127.0.0.1', 12345))

# Send data to the server
data = b"Hello, server!"
client_socket.send(data)

# Receive a response from the server
response = client_socket.recv(1024)
print("Response from server:", response.decode())

# Close the connection
client_socket.close()
```

```python
# Server side
import socket

# Create a TCP/IP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address and port
server_socket.bind(('127.0.0.1', 12345))

# Listen for incoming connections
server_socket.listen(5)

# Accept a connection
client_socket, client_address = server_socket.accept()

# Receive data from the client
received_data = client_socket.recv(1024)
print("Received:", received_data.decode())

# Send a response back to the client
response = b"Message received!"
client_socket.send(response)

# Close the connection
client_socket.close()
server_socket.close()
```

Data Exchange and Network Communication

Data Exchange and Network Communication`

- We’ve already explored structured data representations and both **JSON** (JavaScript Object Notation) and **XML** (eXtensible Markup Language) are commonly used formats for representing structured data,
- They can also be used in server-client communication to exchange data.
- Let’s explore how you might incorporate filetypes like JSON and XML files into a server-client connection.

Serving up some JSON!

Let’s look at JSON:

In a server-client scenario, JSON can be used to serialize and deserialize data for transmission between the server and client.

Server-Side:

- The server creates or reads data from a file in JSON format.
- When a client request is received, the server processes the data and serializes it into JSON format using the json module in Python.
- The JSON data is then sent to the client over the network.

Client-Side:

- The client receives the JSON data over the network.
- It deserializes the JSON data into Python objects using the json module.
- The client processes the data as needed.

```python
import json

# Read data from a JSON file
with open('data.json', 'r') as file:
    data = json.load(file)

# Serialize data into JSON format
json_data = json.dumps(data)

# Send JSON data to the client
client_socket.send(json_data.encode())
```

```python
import json

# Receive JSON data from the server
json_data = client_socket.recv(1024).decode()

# Deserialize JSON data into Python objects
data = json.loads(json_data)

# Process the data
print(data)
```