Python Sockets & `select` Cheatsheet

1. Socket Setup & Basic Commands

Creating a Socket (Same for Client & Server)

```
import socket
# For IPv4 TCP protocol
my_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Important Server Setup Commands

```
# Allow reusing the address (prevents "Address already in use" error)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Attach the socket to a specific address and port
server_socket.bind( ("localhost", 10000) )

# Listen for incoming connections (5 is the backlog)
server_socket.listen(5)
```

Important Client Setup Command

```
# Connect to the server
client_socket.connect( ("localhost", 10000) )
```

Closing a Socket (Crucial for cleanup!)

```
my_socket.close()
```

---

2. Sending & Receiving Data

Strings vs. Bytes

- Sockets send **bytes**, not strings.

- You must **encode** strings before sending and **decode** bytes after receiving.

```
# Encoding a string to bytes
message = "hello"
bytes_to_send = message.encode() # Default is 'utf-8'
client_socket.sendall(bytes_to_send)

# Receiving bytes and decoding to a string
received_bytes = client_socket.recv(1024) # 1024 is buffer size
received_message = received_bytes.decode()
```

A `recv(1024)` call returns:

- **Bytes of data:** `b'some data'` if the other side sent something.
- **An empty bytestring:** `b''` if the other side has closed the connection. **This is how you detect a disconnect!**

---

3. Packing & Unpacking Binary Data (`struct`)

- Used when you need to send fixed-size, non-string data (like integers).

```python
import struct

# 1. Create a packer object with a format string
# 'i i 1s' = integer, integer, 1-char string
packer = struct.Struct("i i 1s")

# 2. Pack data into bytes
operand_a = 50
operand_b = 10
operator = b"+" # Must be bytes
packed_data = packer.pack(operand_a, operand_b, operator)
sock.sendall(packed_data)

# 3. Unpack received bytes back into data
received_bytes = sock.recv(packer.size)
unpacked_data = packer.unpack(received_bytes)
# unpacked_data is a tuple: (50, 10, b'+')
result = unpacked_data[0]
```

---

4. The `select` Loop (The Heart of the Server)

Use `select` to manage multiple clients without blocking.

The Logic:

1. Keep a list of all sockets you want to monitor (server + all connected clients).
2. Call `select.select()` on that list.
3. `select` returns a list of sockets that are ready to be read from.
4. Loop through the ready sockets and take action.

Code Pattern:

```
import select

# Sockets to watch: start with just the main server socket
sockets_to_monitor = [server_socket]

while True:
    # This call blocks until at least one socket is ready
    readable_sockets, _, _ = select.select(sockets_to_monitor, [], [])

    for sock in readable_sockets:
        # Case 1: The ready socket is the main listening socket
        if sock is server_socket:
            # A new client is connecting! Accept it.
            client_socket, client_address = server_socket.accept()
            # Add the new client to the list to be monitored
            sockets_to_monitor.append(client_socket)
            print("New connection!")

        # Case 2: The ready socket is a client that sent data
        else:
            data = sock.recv(1024)
            if data:
                # Client sent a message, process it
                print(f"Received: {data.decode()}")
                sock.sendall(b"OK")
            else:
                # Client disconnected (recv returned b'')
                print("Client disconnected.")
                sockets_to_monitor.remove(sock)
                sock.close()
```

Key `select` idea: The listening socket's only job is to `.accept()`. Client sockets are for `.recv()` and `.sendall()`.