Python Networking Cheat Sheet (Exercises 1-3)

This guide provides the core functions and logic patterns you'll need for the TCP/UDP proxy and service discovery exercises.

1. Basic Socket Creation

Every program starts by creating a socket. The type determines its behavior.

- **TCP (Stream) Socket:** For reliable, connection-oriented communication (like the calculator client, TCP server, and HTTP proxy).

```
# For a client or server
tcp_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- **UDP (Datagram) Socket:** For fast, connectionless messages (like the UDP address server).

```
# For a client or server
udp_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

2. TCP Operations: The Connection Lifecycle

TCP requires a formal connection. The roles of client and server are distinct.

TCP Server Workflow (Listening for clients)

1. `bind()`: Claim a port on your machine so you can listen for connections.

```
server_address = ('localhost', 9000)
sock.bind(server_address)
```

2. `listen()`: Put the socket into server mode to listen for incoming connection requests.

```
sock.listen(5) # The number is the backlog of allowed waiting connections.
```

3. `accept()`: Block and wait for a client to connect. **This is crucial:** it returns a **new socket** just for communicating with that one client.

```
# connection is the new socket for this client
# client_info is the client's ('ip', port)
connection, client_info = sock.accept()
```

TCP Client Workflow (Connecting to a server)

1. **`connect()`:** Establish a connection to a listening server.
   ```
   server_address = ('localhost', 9000)
   sock.connect(server_address)
   ```
2. **`sendall()`:** Send data over the established connection. `sendall` is preferred for TCP as it ensures the entire message is sent.
   ```
   sock.sendall(b"some data")
   ```
3. **`recv()`:** Read data from the connection. It blocks until data is available. If `recv()` returns empty bytes (`b''`), the other side has closed the connection.
   ```
   data = sock.recv(1024) # 1024 is the max buffer size
   if not data:
       print("Connection closed by server.")
   ```

3. UDP Operations: Connectionless Messages

UDP is simpler. There is no `connect()` or `listen()`. You just send and receive.

- **`sendto()`:** Send a datagram. You must specify the destination address **every time**.
  ```
  server_address = ('localhost', 10001)
  udp_sock.sendto(b"my message", server_address)
  ```
- **`recvfrom()`:** Wait for a datagram. It returns **both** the data and the address of the sender.
  ```
  # data is the message, sender_addr is who sent it
  data, sender_addr = udp_sock.recvfrom(200)
  ```
- **`bind()` (For UDP Servers):** A UDP server still needs to `bind()` to a port to "claim" it and be able to receive messages sent to that port.

4. Handling Multiple Clients with `select`

The `select` module lets a server monitor multiple sockets at once without freezing. This is key for the proxy and calculator server.

The `select` Pattern:

1. **Create a list of sockets to watch:**

```
# Start with just the main listening socket
inputs = [listening_socket]
```

2. **Call `select.select()` in a loop:** This blocks until at least one socket has data to be read.

```
# readables is a list of sockets that are ready
readables, _, _ = select.select(inputs, [], [])
```

3. **Loop through the `readables` list and act accordingly:**

```
for s in readables:
    # Case 1: The main server socket is ready -> a new client is connecting.
    if s is listening_socket:
        connection, client_info = s.accept()
        inputs.append(connection) # Add the new client to the list to be watched!

    # Case 2: A client socket is ready -> an existing client sent data.
    else:
        data = s.recv(1024)
        if data:
            # Process the data...
        else:
            # Empty data means client disconnected. Clean up!
            s.close()
            inputs.remove(s)
```

5. Special Topic: HTTP Proxy Logic

The HTTP proxy combines these patterns. For each client that connects:

1. **Receive from Browser:** Use `browser_socket.recv()`.
2. **Connect to Web Server:** Create a *new* TCP client socket and `connect()` to the target web server.
3. **Forward Request:** Use `server_socket.sendall()` to send the data you just received from the browser.
4. **Receive Response:** Use `server_socket.recv()` to get the web page.
5. **Filter and Forward Response:**

```
response_data = server_socket.recv(4096)

# The core filtering logic
if b"SzamHalo" in response_data:
    browser_socket.sendall(HTTP_404_RESPONSE)
else:
    browser_socket.sendall(response_data)
```