1 Web Technologies

Lab Objective: *The Internet is a term for the collective grouping of all publicly accessible computer networks in the world. This network can be traversed to access services such as social communication, maps, video streaming, and large datasets, all of which are hosted on computers across the world. Using these technologies requires an understanding of data serialization, data transportation protocols, and how programs such as servers, clients, and APIs are created to facilitate this communication.*

Data Serialization

Serialization is the process of packaging data in a form that makes it easy to transmit the data and quickly reconstruct it on another computer or in a different programming language. Many serialization metalanguages exist, such as Python's `pickle`, YAML, XML, and JSON. JSON, which stands for *JavaScript Object Notation*, is the dominant format for serialization in web applications. Despite having "JavaScript" in its name, JSON is a language-independent format and is frequently used for transmitting data between different programming languages. It stores information about objects as a specially formatted string that is easy for both humans and machines to read and write. *Deserialization* is the process of reconstructing an object from the string.

JSON is built on two types of data structures: a collection of key/value pairs similar to Python's built-in `dict`, and an ordered list of values similar to Python's built-in `list`.

```
{                                   # A family's info written in JSON format.
    "lastname": "Smith",            # The outer dictionary has two keys:
    "children": [                   # "lastname" and "children".
        {                           # The "children" key maps to a list of
            "name": "Timmy",        # two dictionaries, one for each of the
            "age": 8                # two children.
        },
        {
            "name": "Missy",
            "age": 5
        }
    ]
}
```

> **Note**
>
> To see a longer example of what JSON looks like, try opening a Jupyter Notebook (a `.ipynb` file) in a plain text editor. The file lists the Notebook cells, each of which has attributes like `"cell_type"` (usually code or markdown) and `"source"` (the actual code in the cell).

The JSON libraries of various languages have a fairly standard interface. The Python standard library module for JSON is called `json`. If performance speed is critical, consider using the `ujson` or `simplejson` modules that are written in C. A string written in JSON format that represents a piece of data is called a *JSON message*. The `json.dumps()` function generates the JSON message for a single Python object, which can be stored and used within the Python program. Alternatively, the json encoder `json.dump()` generates the same object, but writes it directly to a file. To load a JSON string or file, use the json decoder `json.loads()` or `json.load()`, respectively.

```
>>> import json

# Store info about a car in a nested dictionary.
>>> my_car = {
...     "car": {
...         "make": "Ford",
...         "color": [255, 30, 30]  },
...     "owner": "me" }

# Get the JSON message corresponding to my_car.
>>> car_str = json.dumps(my_car)
>>> car_str
'{"car": {"make": "Ford", "color": [255, 30, 30]}, "owner": "me"}'

# Load the JSON message into a Python object, reconstructing my_car.
>>> car_object = json.loads(car_str)
>>> for key in car_object:          # The loaded object is a dictionary.
...     print(key + ':', car_object[key])
...
car: {'make': 'Ford', 'color': [255, 30, 30]}
owner: me

# Write the car info to an external file.
>>> with open("my_car.json", 'w') as outfile:
...     json.dump(my_car, outfile)
...
# Read the file to check that it saved correctly.
>>> with open("my_car.json", 'r') as infile:
...     new_car = json.load(infile)
...
>>> print(new_car.keys())           # This loaded object is also a dictionary.
dict_keys(['car', 'owner'])
```

Problem 1. The file `nyc_traffic.json` contains information about 1000 traffic accidents in New York City during the summer of 2017.[a] Each entry lists one or more reasons for the accident, such as "Unsafe Speed" or "Fell Asleep."

Write a function that loads the data from the JSON file. Make a readable, sorted bar chart showing the total number of times that each of the 7 most common reasons for accidents are listed in the data set.
(Hint: the `collections.Counter` data structure and use `plt.tight_layout()` may be useful here.)

To check your work, the 6th most common reason is "Backing Unsafely," listed 59 times.

[a] See `https://opendata.cityofnewyork.us/`.

Custom Encoders and Decoders for JSON

The default JSON encoder and decoder do not support serialization for every kind of data structure. For example, a `set` cannot be serialized using only `json` functions. However, the default JSON encoder can be subclassed to handle sets or custom data structures. A custom encoder must organize the information in an object as nested lists and dictionaries. The corresponding custom decoder uses the way that the encoder organizes the information to reconstruct the original object.

For example, one way to serialize a `set` is to express it as a dictionary with one key that indicates its data type, and another key mapping to the actual data.

```
>>> class SetEncoder(json.JSONEncoder):
...     """A custom JSON encoder for Python sets."""
...     def default(self, obj):
...         if not isinstance(obj, set):
...             raise TypeError("expected a set for encoding")
...         return {"dtype": "set", "data": list(obj)}
...
# Use the custom encoder to convert a set to its custom JSON message.
>>> set_message = json.dumps(set('abca'), cls=SetEncoder)
>>> set_message
'{"dtype": "set", "data": ["a", "b", "c"]}'

# Define a custom decoder for JSON messages generated by the SetEncoder.
>>> def set_decoder(item):
...     if "dtype" in item:
...         if item["dtype"] != "set" or "data" not in item:
...             raise ValueError("expected a JSON message from SetEncoder")
...         return set(item["data"])
...     raise ValueError("expected a JSON message from SetEncoder")
...
# Use the custom decoder to convert a JSON message to the original object.
>>> json.loads(set_message, object_hook=set_decoder)
{'a', 'b', 'c'}
```

It is good practice to check for errors to ensure that custom encoders and decoders are only used when intended.

Problem 2. The following class facilitates a regular 3×3 game of tic-tac-toe, where the boxes in the board have the following coordinates.

$(0,0)$	$(0,1)$	$(0,2)$
$(1,0)$	$(1,1)$	$(1,2)$
$(2,0)$	$(2,1)$	$(2,2)$

Write a custom encoder and decoder for the `TicTacToe` class. If the custom encoder receives anything other than a `TicTacToe` object, raise a `TypeError`.

```
class TicTacToe:
    def __init__(self):
        """Initialize an empty board. The O's go first."""
        self.board = [[' ']*3 for _ in range(3)]
        self.turn, self.winner = "O", None

    def move(self, i, j):
        """Mark an O or X in the (i,j)th box and check for a winner."""
        if self.winner is not None:
            raise ValueError("the game is over!")
        elif self.board[i][j] != ' ':
            raise ValueError("space ({},{}) already taken".format(i,j))
        self.board[i][j] = self.turn

        # Determine if the game is over.
        b = self.board
        if any(sum(s == self.turn for s in r)==3 for r in b):
            self.winner = self.turn     # 3 in a row.
        elif any(sum(r[i] == self.turn for r in b)==3 for i in range(3)):
            self.winner = self.turn     # 3 in a column.
        elif b[0][0] == b[1][1] == b[2][2] == self.turn:
            self.winner = self.turn     # 3 in a diagonal.
        elif b[0][2] == b[1][1] == b[2][0] == self.turn:
            self.winner = self.turn     # 3 in a diagonal.
        else:
            self.turn = "O" if self.turn == "X" else "X"

    def empty_spaces(self):
        """Return the list of coordinates for the empty boxes."""
        return [(i,j) for i in range(3) for j in range(3)
                                        if self.board[i][j] == ' ' ]
    def __str__(self):
        return "\n---------\n".join(" | ".join(r) for r in self.board)
```

Servers and Clients

The Internet has specific protocols that allow for standardized communication within and between computers. The most common communication protocols in computer networks are contained in the Internet Protocol Suite. Among these is *Transmission Control Protocol* (TCP), which is used to establish a connection between two computers, exchange bits of information called *packets*, and then close the connection. TCP creates the connection via network *socket* objects that are used to send and receive data packets from a computer.

Essentially, this can be thought of as a PO box at a post office. The socket is like a PO box owned by a particular program, which checks it periodically for updates. The computer can be thought of as the post office which houses the PO boxes. PO boxes, or sockets, can send mail to each other within the same post office, or computer, easily, but more work is needed when the PO boxes send mail to each other from one post office, or computer, to another.

A *server* is a program that interacts with and provides functionality to client programs. This can be thought of as the PO box which sends the mail. A *client* program contacts a server to receive some sort of response that assists it in fulfilling its function. This can be thought of as the PO box which receives the mail. Unlike with physical mail, in which the sender can send mail to himself, a socket being used as a server in a computer cannot also serve as a client at the same time. Servers are fundamental to modern networks and provide services such as file sharing, authentication, webpage information, databases, etc.

Creating a Server

One simple way to create a server in Python is via the `socket` module. The server socket must first be initialized by specifying the type of connection and the address at which clients can find the server. The server socket then listens and waits for a connection from a client, receives and processes data, and eventually sends a response back to the client. After exchanges between the server and the client are finished, the server closes the connection to the client.

Name	Description
`socket`	Create a new socket using the given address family, socket type and protocol number.
`bind`	Bind the socket to an address. The socket must not already be bound.
`listen`	Enable a server to accept connections.
`accept`	Accept a connection. Must be bound to an address and listening for connections.
`connect`	Connect to a remote socket at address.
`sendall`	Send data to the socket. The socket must be connected to a remote socket. Continues to send data until either all data has been sent or an error occurs.
`recv`	Receive data from the socket. Must be given a buffer size; use 1024.
`close`	Mark the socket closed.

Table 1.1: Socket method descriptions

The `socket.socket()` method receives two parameters, which specify the socket type. The server address is a (host, port) tuple. The host is the IP address, which in this case is `"localhost"` or `"0.0.0.0"`—the default address that specifies the local machine and allows connections on all interfaces. The port number is an integer from 0 to 65535. About 250 port numbers are commonly used, and certain ports have pre-defined uses. Only use port numbers greater than 1023 to avoid interrupting standard system services, such as email and system updates.

After setting up the server socket, the server program waits for a client to connect. The `accept()` method returns a new socket object and the client's address. Data is received through the connection socket's `recv()` method, which takes an integer specifying the number of bits of data to receive. The data is transferred as a raw byte stream (of type `bytes`), so the `decode()` method is necessary to translate the data into a string. Likewise, data that is sent back to the client through the connection socket's `sendall()` method must be encoded into a byte stream via the `encode()` method.

Finally, `try-finally` blocks in the server ensure that the connection is always closed securely. Put these blocks within an infinite `while(True)` block to ensure that your server will be ready for any client request. Note that the `accept()` method does not return until a connection is made with a client. Therefore, this server program cannot be executed in its entirety without a client. To stop a server, raise a `KeyBoardInterrupt` (press `ctrl+c`) in the terminal where it is running.

Note that server-client communication is the reason that JSON serialization and deserialization is so important. For example, information such as an image or a family tree could be sent more simply using serialized objects.

```
def mirror_server(server_address=("0.0.0.0", 33333)):
    """A server for reflecting strings back to clients in reverse order."""
    print("Starting mirror server on {}".format(server_address))

    # Specify the socket type, which determines how clients will connect.
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_sock.bind(server_address)     # Assign this socket to an address.
    server_sock.listen(1)                # Start listening for clients.

    while True:
        # Wait for a client to connect to the server.
        print("\nWaiting for a connection...")
        connection, client_address = server_sock.accept()
        try:
            # Receive data from the client.
            print("Connection accepted from {}.".format(client_address))
            in_data = connection.recv(1024).decode()     # Receive data.
            print("Received '{}' from client".format(in_data))

            # Process the received data and send something back to the client.
            out_data = in_data[::-1]
            print("Sending '{}' back to the client".format(out_data))
            connection.sendall(out_data.encode())        # Send data.

        finally:    # Make sure the connection is closed securely.
            connection.close()
            print("Closing connection from {}".format(client_address))
```

Achtung!

It often takes some time for a computer to reopen a port after closing a server connection. This is due to the timeout functionality of specific protocols that check connections for errors and disruptions. While testing code, wait a few seconds before running the program again, or use different ports for each test.

Problem 3. Write a function that accepts a (host, port) tuple and starts up a tic-tac-toe server at the specified location. Wait to accept a connection, then while the connection is open, repeat the following operations.

1. Receive a JSON serialized `TicTacToe` object (serialized with your custom encoder from Problem 2) from the client.
2. Deserialize the `TicTacToe` object using your custom decoder from Problem 2.
3. If the client has just won the game, send `"WIN"` back to the client and close the connection.
4. If there is no winner but board is full, send `"DRAW"` to the client and close the connection.
5. If the game still isn't over, make a random move on the tic-tac-toe board and serialize the updated `TicTacToe` object. If this move wins the game, send `"LOSE"` to the client, then send the serialized object separately (as proof), and close the connection. Otherwise, send only the updated `TicTacToe` object back to the client but keep the connection open.

(Hint: print information at each step so you can see what the server is doing.)

Ensure that the connection closes securely even if an exception is raised. Note that you will not be able to fully test your server until you have written a client (see Problem 4).

Creating a Client

The `socket` module also has tools for writing client programs. First, create a socket object with the same settings as the server socket, then call the `connect()` method with the server address as a parameter. Once the client socket is connected to the server socket, the two sockets can transfer information between themselves.

Unlike the server socket, the client socket sends and reads the data itself instead of creating a new connection socket. When the client program is complete, close the client socket. The server will keep running, waiting for another client to serve.

To see a client and server communicate, open a terminal and run the server. Then run the client in a separate terminal. Try this with the provided examples.

```
def mirror_client(server_address=("0.0.0.0", 33333)):
    """A client program for mirror_server()."""
    print("Attempting to connect to server at {}...".format(server_address))

    # Set up the socket to be the same type as the server.
```

```
    client_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_sock.connect(server_address)     # Attempt to connect to the server.

    # Send some data from the client user to the server.
    out_data = input("Type a message to send to the server: ")
    client_sock.sendall(out_data.encode())       # Send data.

    # Wait to receive a response back from the server.
    in_data = client_sock.recv(1024).decode()    # Receive data.
    print("Received '{}' from the server".format(in_data))

    # Close the client socket.
    client_sock.close()
```

Problem 4. Write a client function that accepts a (host, port) tuple and connects to the tic-tac-toe server at the specified location. Start by initializing a new `TicTacToe` object, then repeat the following steps until the game is over.

1. Print the board and prompt the player for a move. Continue prompting the player until they provide valid input.
2. Update the board with the player's move, then serialize it using your custom encoder from Problem 2, and send the serialized version to the server.
3. Receive a response from the server. If the game is over, congratulate or mock the player appropriately. If the player lost, receive a second response from the server (the final game board), deserialize it, and print it out.

Close the connection once the game ends.

APIs

An *Application Program Interface* (API) is a particular kind of server that listens for requests from authorized users and responds with data. For example, a list of locations can be sent with the proper request syntax to a Google Maps API, and it will respond with the calculated driving time from start to end, including each location. Every API has *endpoints* where clients send their requests. Though standards exist for creating and communicating with APIs, most APIs have a unique syntax for authentication and requests that is documented by the organization providing the service.

The `requests` module is the standard way to send a download request to an API in Python.

```
>>> import requests
>>> requests.get(endpoint).json()   # Download and extract the data.
```

Achtung!

Each website and API has a policy that specifies appropriate behavior for automated data retrieval and usage. If data is requested without complying with these requirements, there can be severe legal consequences. Most websites detail their policies in a file called *robots.txt* on their main page. See, for example, `https://www.google.com/robots.txt`.

Additional Material

Other Internet Protocols

There are many protocols in the Internet Protocol Suite other than TCP that are used for different purposes. The Protocol Suite can be divided into four categorical layers:

1. **Application**: Software that utilizes transport protocols to move information between computers. This layer includes protocols important for email, file transfers, and browsing the web.
2. **Transport**: Protocols that assist in basic high level communication between two computers in areas such as data-streaming, reliability control, and flow control.
3. **Internet**: Protocols that handle routing, assignment of addresses, and movement of data on a network.
4. **Link**: Protocols that communicate with local networking hardware such as routers and switches.

Although these examples are simple, every data transfer with TCP follows a similar pattern. For basic connections, these interactions are simple processes. However, requesting a webpage would require management of possibly hundreds of connections. In order to make this more feasible, there are higher level protocols that handle smaller TCP/IP details. The most predominant of these protocols is HTTP.

HTTP

HTTP stands for Hypertext Transfer Protocol, which is an application layer networking protocol. It is a higher level protocol than TCP but uses TCP protocols to manage connections and provide network capabilities. The protocol is centered around a request and response paradigm in which a client makes a request to a server and the server replies with response. There are several methods, or *requests*, defined for HTTP servers, the three most common of which are GET, POST, and PUT. GET requests request information from a server, POST requests modify the state of the server, and PUT requests add new pieces of data to the server.

Every HTTP request or response consists of two parts: a header and a body. The headers contain important information about the request including: the type of request, encoding, and a timestamp. Custom headers may be added to any request to provide additional information. The body of the request or response contains the appropriate data or may be empty.

An HTTP connection can be setup in Python by using the standard Python library `http`. Though it is the standard, the process can be greatly simplified by using an additional library called `requests`. The following demonstrates a simple GET request with the `http` library.

```
>>> import http
>>> conn = http.client.HTTPConnection("www.example.net") # Establish connection
>>> conn.request("GET", "/") # Send GET request
>>> resp = conn.getresponse() # Server response message
>>> print(resp.status)
200 # A status of 200 is the standard sign for successful communication
>>> print(resp.getheaders())
[('Cache-Control', 'max-age=604800'), ... , ('Content-Length', '1270')] # ↩
    Header information about request
>>> print(resp.read())
```

```
b'<!doctype html>\n<html> ... n</html>\n'     # Long string with HTML from webpage
>>> conn.close() # When the request is finished, the connection is closed
```

As previously mentioned, this exchange is greatly simplified by the `requests` library:

```
>>> import requests
>>> r = requests.get("http://www.example.net")
>>> print(r.headers)
{'Cache-Control': 'max-age=604800', ... , 'Content-Length': '606'}
>>> print(r.content)
b'<!doctype html>\n<html> ... n</html>\n'
```

This process is how a web browser (a client program) retrieves a webpage. It first sends an HTTP request to the web server (a server program) and receives the HTML, CSS, and other code files for a webpage, which are compiled and run in the web browser.

Requests also often include parameters which are keys to tell the server what is being requested or placed. These parameters can be included in the URL that requests from the server, or in parameters that the `requests` library can implement. For example:

```
>>> r = requests.get("http://httpbin.org/get?key2=value2&key1=value1")
>>> print(r.text)
 {
  "args": {
    "key1": "value1",
    "key2": "value2"
  },
  ...
  },
  "origin": "128.187.116.7",
  "url": "http://httpbin.org/get?key2=value2&key1=value1"
}
>>> r = requests.get("http://httpbin.org/get", params={'key1':'value1','key2':'value2'})
>>> print(r.url)
 http://httpbin.org/get?key2=value2&key1=value1
>>> print(r.text)
 {
  "args": {
    "key1": "value1",
    "key2": "value2"
  },
  ...
  },
  "origin": "128.187.116.7",
  "url": "http://httpbin.org/get?key2=value2&key1=value1"
}
```

A similar format to GET requests can also be used for PUT or POST requests. These special requests alter the state of the server or send a piece of data to the server, respectively. In addition, for PUT and POST requests, a data string or dictionary may be sent as a binary stream attachment. The `requests` library attaches these data objects with the `data` parameter. For example:

```
>>> r = requests.put('http://httpbin.org/put', data='{key1:value1,key2:value2}'↩
    )
>>> print(r.text)
 {
  "args": {},
  "data": "{key1:value1,key2:value2}",
  "files": {},
  "form": {},
  ...
  "json": null,
  "origin": "128.187.116.7",
  "url": "http://httpbin.org/put"
}
```

Note that the `data` parameter accepts input in the form of a JSON string.

Frequently, when these requests arrive at the server, they are in the form of a binary stream, which can be read with similar notation to the Python `open` function. Below is an example of reading the previous PUT request with a data attachment as a binary stream using `read`.

```
>>> data = r.json()['data'] # Retrieve the sent data string
>>> print(data)
 '{key1:value1,key2:value2}'
>>> print(len(data.encode())) # Show the string's length in bytes
 25
>>> with open('request.txt', 'w') as file:
>>>         file.write(data) # Write the string to a file
>>> with open('request.txt', 'rb') as file: # Open the file as a binary stream
>>>         file.read(25) # Read the correct number of bytes
 b'{key1:value1,key2:value2}'
```

For more information on the `requests` library, see the documentation at `http://docs.python-requests.org/`.