4 Web Technologies

Lab Objective: *The Internet is an umbrella term for the collective grouping of all publicly accessible computer networks in the world. This collective network can be traversed to access services such as social communication, maps, video streaming, and accessibility to 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. One of the most prevalent serialization metalanguages is *JSON*, which stands for *JavaScript Object Notation*.[1] 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 and an ordered list of values, similar to Python's built-in `dict` and `list` structures, respectively.

```
{                                   # 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
        }
    ]
}
```

[1] Despite having "JavaScript" in its name, JSON is a language-independent format. In fact, JSON is frequently used for transmitting data between different programming languages.

> **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*. To generate the JSON message for a single Python object, use `json.dumps()`. Alternatively, the function `json.dump()` generates the JSON message of an object and writes it to an open file. To load a JSON string or file, use `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. Look at the first few entries of the dataset and decide how to gather information about the cause(s) of each accident. 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 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'}
```

Checks for errors like in the previous example are good practice 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.

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

```
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)
```

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

NOTE

JSON is a good option for transferring data between two different languages. Python's `pickle` module is particularly good for serialization when the stored object will only be unpacked in Python. The main functions are almost identical to `json.dumps()` and its companions.

```
>>> import pickle

>>> item = pickle.dumps([1, 2, 3, 4, 5, 6])
>>> item
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04K\x05K\x06e.'

>>> pickle.loads(item)
[1, 2, 3, 4, 5, 6]
```

In addition, there are many other serialization formats such as YAML and XML. However, JSON is the dominant format for serialization in web applications.

Servers and Clients

The Internet is like a network of roads connecting the buildings of a city where each building represents a computer and each road represents the physical wires or wireless pathways that allow for intercommunication. Navigating the road properly, requires using the correct kinds of vehicles and following the established laws for road travel. There are also various kinds of vehicles for different purposes and with different capabilities that are used to transport items from building to building. In a similar fashion, 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), used to establish a connection between two computers, exchange bits of information called *packets*, and then close the connection. More specifically, TCP creates network *socket* objects that are used to send and receive data packets from a computer. A socket is basically an address within a computer on which a program can send or receive data, like a P.O. box within a post office. The post office is the computer that receives mail, but the mail is distributed to individual programs that check the P.O. box for their personal communications.

Creating a Server

A *server* is a program that interacts with and provides functionality to *client* programs. A client program contacts a server to receive some sort of response that assists it in fulfilling its function. Servers are fundamental to modern networks and provide services such as file sharing, authentication, webpage information, databases, etc.

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 that clients can find the server at. The server socket then listens and waits for a connection from a client, receives and processes data, then 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.

```
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))
```

The two parameters for `socket.socket()` specify the socket type.[2] The server address is a tuple (host, port). 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.

> **Achtung!**
>
> Only use port numbers greater than 1023 to avoid interrupting standard system services.

After setting up the server socket, it waits for a client to connect. The `accept()` method returns a new socket object (`connection`) 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, the `try-finally` blocks ensure that the connection is always closed securely. To stop a server, raise a `KeyBoardInterrupt` (press `ctrl+c`) in the terminal where it is running.

[2] See `https://docs.python.org/3/library/socket.html` for details on these parameters.

NOTE

When running `mirror_server()`, the program hangs on the following line.

```
connection, client_address = server_sock.accept()
```

This is because 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. Client creation is addressed in the next section.

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.

```
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()
```

Note that 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.

```
# TERMINAL 1
>>> mirror_server()                 # First start up the server.
Starting mirror server on ('0.0.0.0', 33333)

Waiting for a connection...         # At this point, start the client.
Connection accepted from ('127.0.0.1', 50679).
Received 'racecars and lollipops' from client
Sending 'spopillol dna sracecar' back to the client
Closing connection from ('127.0.0.1', 50679)

Waiting for a connection...
# The client program is over, but the server waits to keep serving clients.
```

```
# TERMINAL 2
>>> mirror_client()                 # Start the client after the server.
Attempting to connect to server at ('0.0.0.0', 33333)...
Connected!
Type a message to send: racecars and lollipops
Received 'spopillol dna sracecar' from the server
```

Problem 4. Write a 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 twenty different locations can be sent with the proper request syntax to a Google Maps API, and it will respond with the calculated driving time from each location to every other. 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.

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`.

Problem 5. The `requests` module is the standard way to send a download request to an API.

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

Write a function that makes requests to download data from the following API endpoints managed by New York City.

- Recycling bin locations: https://data.cityofnewyork.us/api/views/sxx4-xhzg/rows.json?accessType=DOWNLOAD
- Residential addresses: https://data.cityofnewyork.us/api/views/7823-25a9/rows.json?accessType=DOWNLOAD

Save the recycling data as `nyc_recycling.json` and the address data as `nyc_addresses.json`.

Problem 6. Write a function that loads the data files generated in Problem 5 but **does not** call the actual function from Problem 5. Determine how close the residential addresses in New York City are to the nearest recycling bin.

1. Retrieve the latitude and longitude of each recycling bin and, separately, the latitude and longitude of each residential address (ignore entries without these coordinates). Note carefully that the coordinates for the recycling data are in (latitude, longitude) format, but the coordinates for the address data are in (longitude, latitude) format.
   (Hint: Both datasets are, at the highest level, dictionaries with two keys: `"meta"`, which has information about the data; and `"data"`, which has the actual data. All of the information needed is contained in the `"data"` key value.)
2. Load the recycling bin data into a k-d tree.
3. For each address, query the tree to find the distance to the nearest recycling bin, in terms of the coordinates.
4. Plot a histogram of the distances.

For steps 2–3, recall the following syntax for using a k-d tree in SciPy.

```
from scipy.spatial import KDTree

tree = KDTree(data)                          # Initialize the tree.
min_distance, index = tree.query(target)     # Query for a point.
```