Module 4 Question and answers

1. What is web scraping ? Explain the step involved in web scraping with an example .

- Web scraping is a technique used to extract data from websites. It involves fetching and parsing HTML content to gather information.
- The main purpose of web scraping is to collect and analyze data from websites for various applications, such as research, business intelligence, or creating datasets.
- Steps involved in web scraping:
  - Send an HTTP request to URL
  - Parse the data which is accessed from the url
  - Navigate and search the parse tree

Send an HTTP request to URL

- The first step in web scraping is to send an HTTP request to the URL of the website from which you want to extract data. This is typically done using Python libraries like requests or urllib to access the HTML content of the webpage.

  Ex:
```
import requests
url = "https://example.com"
response = requests.get(url)
```

- The requests module allows you to send HTTP requests using Python.
- The HTTP request returns a response object with all the response data (content, encoding, status, etc).

Parse the data which is accessed from the url

- Once the HTML content is retrieved from the website, the next step is to parse this data. Parsing involves structuring the raw HTML into a format that is more easily navigable and understandable. This is commonly done using libraries like BeautifulSoup or lxml in Python.

  Ex :
```
from bs4 import BeautifulSoup
soup = BeautifulSoup(response.content,'html.parser')
```

- we will simply parse some HTML input and extract links using the BeautifulSoup library.
- BeautifulSoup tolerates highly flawed HTML and still lets you easily extract the data you need.

- BeautifulSoup library is one of the simplest libraries available for parsing. To use this, download and install the BeautifulSoup code from:

  https://pypi.python.org/pypi/beautifulsoup4

Navigate and search the parse tree

After parsing the HTML content, you navigate and search the resulting parse tree to locate specific elements or information on the webpage. This involves using methods provided by the parsing library to find tags, attributes, and content within the HTML structure.

Ex:
```
# Find the title tag of the webpage
  title_tag = soup.find('title')
# Find all links in the webpage
  all_links = soup.find_all('a')
```

Example Program

```
from bs4 import BeautifulSoup
import requests
# sample web page
sample_web_page = 'https://www.python.org'
# call get method to request that page
page = requests.get(sample_web_page)
# with the help of beautifulSoup and html parser create soup
soup = BeautifulSoup(page.content, "html.parser")
child_soup = soup.find_all('strong')
#print(child_soup)
text = """Notice:"""
# we will search the tag with in which text is same as given text
for i in child_soup:
  if(i.string == text):
    print(i)
```

Output:

```
<strong>Notice:</strong>
```

2. What are the different libraries used for fetching web pages? Explain with example for each.

Fetching web pages

Fetching web pages involves retrieving the HTML content of a webpage programmatically. You can achieve this using libraries like *requests* or *urllib* in Python.

Here's a basic example

using the requests library:

1. Install the requests Library: If you don't have the **requests** library installed, you can install it using: `pip install requests`

2. Write Python Code: Use the requests.get() method to send a GET request to the desired URL and retrieve the HTML content.

```
import requests

url = "https://example.com"
response = requests.get(url)

# Check if the request was successful (status code 200)
if response.status_code == 200:
    # Access the HTML content of the webpage
    html_content = response.text

    # Process or analyze the HTML content (e.g., using BeautifulSoup)
else:
      print(f"Failed to fetch the webpage. Status code:
      {response.status_code}")
```

3. Process the HTML Content (Optional): You can use additional libraries like BeautifulSoup to parse and extract information from the HTML content.

```
from bs4 import BeautifulSoup

# Parse the HTML content using BeautifulSoup
soup = BeautifulSoup(html_content, 'html.parser')

# Extract information from the parsed HTML (e.g., find elements by tag, class, etc.)
```

BeautifulSoup provides methods for navigating and searching the HTML tree, making it easier to extract specific data.

Using Urllib library

We will use urllib to read the page and then use BeautifulSoup to extract the *href* attributes from the anchor (a) tags.

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

url = input('Enter the url')

html = urlopen(url).read()
soup = BeautifulSoup(html, "html.parser")
tags = soup('a')
for tag in tags:
    print('TAG:', tag)
    print('URL:', tag.get('href', None))
    print('Contents:', tag.contents[0])
    print('Attrs:', tag.attrs)
```

```
Output:

Enter the urlhttp://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Contents:
Second Page
Attrs: {'href': 'http://www.dr-chuck.com/page2.htm'}
```

3. Write a program for downloading web pages through form submission.

```
from selenium import webdriver
import time
from selenium.webdriver.common.by import By

# Create a new instance of the Chrome driver
driver = webdriver.Chrome()
driver.maximize_window()
time.sleep(3)

# Navigate to the form page
driver.get('https://www.confirmtkt.com/pnr-status')

# Locate form elements
pnr_field = driver.find_element("name", "pnr")
submit_button = driver.find_element(By.CSS_SELECTOR, '.col-xs-4')
```

```
# Fill in form fields
pnr_field.send_keys('4358851774')

# Submit the form
submit_button.click()
welcome_message = driver.find_element(By.CSS_SELECTOR,".pnr-card")

# Print or use the scraped values
print(type(welcome_message))
html_content = welcome_message.get_attribute('outerHTML')

# Print the HTML content
print("HTML Content:", html_content)

# Close the browser
driver.quit()
```

4. What are CSS selectors ? describe different CSS selectors with examples.

CSS selectors are patterns used to select and style elements in a document.
There are different CSS selectors:

- **Id selector (#)** :The ID selector targets a specific HTML element based on its unique identifier attribute (id). An ID is intended to be unique within a webpage, so using the ID selector allows you to style or apply CSS rules to a particular element with a specific ID.

```
#header {
color: blue;
font-size: 16px;
}
```

- **Class selector (.) :** The class selector is used to select and style HTML elements based on their class attribute. Unlike IDs, multiple elements can share the same class, enabling you to apply the same styles to multiple elements throughout the document.

```
.highlight {
 background-color: yellow;
 font-weight: bold;
}
```

- **Universal Selector (*) :**The universal selector selects all HTML elements on the webpage. It can be used to apply styles or rules globally, affecting every element. However, it is important to use the universal selector judiciously to avoid unintended consequences.

```
* {
    margin: 0;
    padding: 0;
  }
```

- **Element Selector (tag) :** The element selector targets all instances of a specific HTML element on the page. It allows you to apply styles universally to elements of the same type, regardless of their class or ID.

```
p {
 color: green;
 font-size: 14px;
}
```

- **Grouping Selector(,) :** The grouping selector allows you to apply the same styles to multiple selectors at once. Selectors are separated by commas, and the styles specified will be applied to all the listed selectors.

```
h1, h2, h3 {
 font-family: 'Arial', sans-serif;
 color: #333;
}
```

- These selectors are fundamental to CSS and provide a powerful way to target and style different elements on a webpage.

5. Illustrate with an example, how are CSS selectors used in web scraping?

Let us first create a basic HTML page

```
<!DOCTYPE html>
<html>
 <head>
      <title>Sample Page</title>
</head>
 <body>
        <div id="content">
            <h1>Heading 1</h1>
            <p class="paragraph">This is a sample paragraph.</p>
            <ul>
                <li>Item 1</li>
                <li>Item 2</li>
                <li>Item 3</li>
            </ul>
            <a href="https://example.com">Visit Example</a>
        </div>
</body>
</html>
```

Now we can Scrape the above html using CSS selectors

```
from bs4 import BeautifulSoup
Html=request.get((“web.html”)
soup = BeautifulSoup(Html, 'html.parser')

# 1. Select by tag name
heading = soup.select('h1')
print("1. Heading:", heading[0].text)

# 2. Select by class
paragraph = soup.select('.paragraph')
print("2. Paragraph:", paragraph[0].text)

# 3. Select by ID
div_content = soup.select('#content')
print("3. Div Content:", div_content[0].text)

# 4. Select by attribute
link = soup.select('a[href="https://example.com"]')
print("4. Link:", link[0]['href'])

# 5. Select all list items
list_items = soup.select('ul li')
print("5. List Items:")
for item in list_items:
   print("-", item.text)
```

6. What are Universal functions? Discuss the need for Ufuncs.

- Vectorized operations in NumPy are implemented via **ufuncs**, whose main purpose is to quickly execute repeated operations on values in NumPy arrays.
- Ufuncs are extremely flexible—operation between a scalar and an array, or operations between two arrays:

```
In[5]: np.arange(5) / np.arange(1, 6)

Out[5]: array([ 0.        ,  0.5       ,  0.66666667,  0.75      ,  0.8       ])
```

- ufunc operations are not limited to one-dimensional arrays—they can act on multidimensional arrays as well.
- Ufuncs exist in two flavors:
  - *unary ufuncs*, which operate on a single input
  - *binary ufuncs*, which operate on two inputs.

- Different types of universal functions are
  - Array arithmetic
  - Absolute value
  - Trigonometric functions
  - Exponents and logarithms

Array arithmetic

NumPy's ufuncs feel very natural to use because they make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

```
In[7]: x = np.arange(4)
       print("x     =", x)
       print("x + 5 =", x + 5)
       print("x - 5 =", x - 5)
       print("x * 2 =", x * 2)
       print("x / 2 =", x / 2)
       print("x // 2 =", x // 2)
```

```
x     = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [ 0.   0.5  1.   1.5]
x // 2 = [0 0 1 1]
```

- There is also a unary ufunc for negation, a ** operator for exponentiation, and a % operator for modulus:

```
In[8]: print("-x     = ", -x)
       print("x ** 2 = ", x ** 2)
       print("x % 2  = ", x % 2)

-x     =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2  =  [0 1 0 1]
```

All of these arithmetic operations are simply convenient wrappers around specific functions built into NumPy; for example, the + operator is a wrapper for the add function.

Absolute value

The corresponding NumPy ufunc is np.absolute, which is also available under the alias np.abs:

```
x = np.array([-2, -1, 0, 1, 2])

In[12]: np.absolute(x)

Out[12]: array([2, 1, 0, 1, 2])

In[13]: np.abs(x)

Out[13]: array([2, 1, 0, 1, 2])
```

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions.

```
In[15]: theta = np.linspace(0, np.pi, 3)

In[16]: print("theta      = ", theta)
        print("sin(theta) = ", np.sin(theta))
        print("cos(theta) = ", np.cos(theta))
        print("tan(theta) = ", np.tan(theta))

 theta      = [ 0.         1.57079633  3.14159265]
 sin(theta) = [  0.00000000e+00   1.00000000e+00   1.22464680e-16]
 cos(theta) = [  1.00000000e+00   6.12323400e-17  -1.00000000e+00]
 tan(theta) = [  0.00000000e+00   1.63312394e+16  -1.22464680e-16]
```

Exponents and logarithms

Following are the few examples of Exponents and logarithms ufuncs

```
In[18]: x = [1, 2, 3]
        print("x     =", x)
        print("e^x   =", np.exp(x))
        print("2^x   =", np.exp2(x))
        print("3^x   =", np.power(3, x))

x     = [1, 2, 3]
e^x   = [  2.71828183   7.3890561   20.08553692]
2^x   = [ 2.  4.  8.]
3^x   = [ 3  9 27]
```

```
In[19]: x = [1, 2, 4, 10]
        print("x        =", x)
        print("ln(x)    =", np.log(x))
        print("log2(x)  =", np.log2(x))
        print("log10(x) =", np.log10(x))

x        = [1, 2, 4, 10]
ln(x)    = [ 0.         0.69314718  1.38629436  2.30258509]
log2(x)  = [ 0.         1.          2.          3.32192809]
log10(x) = [ 0.         0.30103     0.60205999  1.        ]
```

7. Discuss any 3 Numpy basic array manipulations with examples

Few categories of basic array manipulations here:

- **Attributes of arrays** : Determining the size, shape, memory consumption, and data types of arrays
- **Indexing of arrays** : Getting and setting the value of individual array elements
- **Slicing of arrays**: Getting and setting smaller subarrays within a larger array

- **Reshaping of arrays**: Changing the shape of a given array
- **Joining and splitting of arrays**: Combining multiple arrays into one, and splitting one array into many.
  Note: refer the examples in Module4 ppt

8. What is broadcasting in Numpy? Explain the rules of broadcasting. Illustrate broadcasting with an example.

Broadcasting in NumPy is a powerful mechanism that allows for the arithmetic operations on arrays of different shapes and sizes, without explicitly creating additional copies of the data. It simplifies the process of performing element-wise operations on arrays of different shapes, making code more concise and efficient.

Here are the key concepts of broadcasting in NumPy:

- ***Shape Compatibility***: Broadcasting is possible when the dimensions of the arrays involved are compatible. Dimensions are considered compatible when they are equal or one of them is 1. NumPy automatically adjusts the shape of smaller arrays to match the shape of the larger array during the operation.

- ***Rules of Broadcasting***: For broadcasting to occur, the sizes of the dimensions must either be the same or one of them must be 1. If the sizes are different and none of them is 1, then broadcasting is not possible, and NumPy will raise a *ValueError*.

- **Automatic Replication:** When broadcasting, NumPy automatically replicates the smaller array along the necessary dimensions to make it compatible with the larger array. This replication is done without actually creating multiple copies of the data, which helps in saving memory.

Example:

Suppose you have a 2D array **A** of shape (3, 1) and another 1D array **B** of shape (3). Broadcasting allows you to add these arrays directly, and NumPy will automatically replicate the second array along the second dimension to match the shape of the first array.

```
import numpy as np
A = np.array([[1], [2], [3]])
B = np.array([4, 5, 6])
result = A + B    # Broadcasting occurs here
```

```
array([[5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])
```

Note : refer the lab program for the example.