Unit-2
Working With Data In Python

Presented By:

Sindhu Patchigolla

Asst.Professor, Dept.of CSE

RGUKT ONGOLE

Contents

- File Operation
- Regular Expressions
- Pandas
- NumPys
- Web Scraping

File Operations

- File handling refers to the process of performing operations on a file, such as creating, opening, reading, writing and closing it through a programming interface. It involves managing the data flow between the program and the file system on the storage device, ensuring that data is handled safely and efficiently.

Why do we need File Handling

- To store data permanently, even after the program ends.
- To access external files like .txt, .csv, .json, etc.
- To process large files efficiently without using much memory.
- To automate tasks like reading configs or saving outputs.

Contd...

- Python has several functions for creating, reading, updating, and deleting files.

File Handling

- The key function for working with files in Python is the open() function.
- The open() function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

- "r" - Read - Default value. Opens a file for reading, error if the file does not exist
- "a" - Append - Opens a file for appending, creates the file if it does not exist
- "w" - Write - Opens a file for writing, creates the file if it does not exist
- "x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

- "t" - Text - Default value. Text mode
- "b" - Binary - Binary mode (e.g. images)

Syntax

- To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

- The code above is the same as:

```
f = open("demofile.txt", "rt")
```

- Because "r" for read, and "t" for text are the default values, you do not need to specify them.
- **Note:** Make sure the file exists, or else you will get an error.

Contd...

Open a File on the Server

- Assume we have the following file, located in the same folder as Python:

  demofile.txt

  ```
  Hello! Welcome to demofile.txt
  This file is for testing purposes.
  Good Luck!
  ```

- To open the file, use the built-in open() function.
- The open() function returns a file object, which has a read() method for reading the content of the file:
- Example:

```
f = open("demofile.txt")
print(f.read())
```

- If the file is located in a different location, you will have to specify the file path, like this:
- Example
- Open a file on a different location:

```
f = open("D:\\myfiles\welcome.txt")
print(f.read())
```

Using the with statement

- You can also use the with statement when opening a file:

Example

- Using the with keyword:

```
with open("demofile.txt") as f:
print(f.read())
```

- Then you do not have to worry about closing your files, the with statement takes care of that.
-

Close Files

- It is a good practice to always close the file when you are done with it.
- If you are not using the with statement, you must write a close statement in order to close the file:

Example

- Close the file when you are finished with it:

```
f = open("demofile.txt")
print(f.readline())
f.close()
```

- **Note:** You should always close your files. In some cases, due to buffering, changes made to a file may not show until you close the file.

Read Only Parts of the File

- By default the read() method returns the whole text, but you can also specify how many characters you want to return:

Example

- Return the 5 first characters of the file:

```
with open("demofile.txt") as f:
print(f.read(5))
```

Write to an Existing File

- To write to an existing file, you must add a parameter to the open() function:
- "a" - Append - will append to the end of the file
- "w" - Write - will overwrite any existing content
- Example: Open the file "demofile.txt" and append content to the file:

```
with open("demofile.txt", "a") as f:
f.write("Now the file has more content!")
```

#open and read the file after the appending:

```
with open("demofile.txt") as f:
print(f.read())
```

Overwrite Existing Content

- To overwrite the existing content to the file, use the w parameter:

Example

- Open the file "demofile.txt" and overwrite the content:

```
with open("demofile.txt", "w") as f:
f.write("Woops! I have deleted the content!")
```

#open and read the file after the overwriting:

```
with open("demofile.txt") as f:
print(f.read())
```

- **Note:** the "w" method will overwrite the entire file.

Create a New File

- To create a new file in Python, use the open() method, with one of the following parameters:
- "x" - Create - will create a file, returns an error if the file exists
- "a" - Append - will create a file if the specified file does not exists
- "w" - Write - will create a file if the specified file does not exists

Example

- Create a new file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

- Result: a new empty file is created.
- **Note:** If the file already exist, an error will be raised.

Delete a File

- To delete a file, you must import the OS module, and run its os.remove() function:
- Example: Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

- To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

- Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
print("The file does not exist")
```

Delete Folder

- To delete an entire folder, use the os.rmdir() method:
- Example
- Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

- **Note:** You can only remove *empty* folders.

Example:

```
f = open("geek.txt", "r")
print("Filename:", f.name)
print("Mode:", f.mode)
print("Is Closed?", f.closed)
f.close()
print("Is Closed?", f.closed)
```

Output:

Filename: geek.txt
Mode: r
Is Closed? False
Is Closed? True

Example:

```
try:
    file = open("geek.txt", "r")
    content = file.read()
    print(content)
finally:
    file.close()
```

Output:

Hello, World!

Regular Expression

- A Regular Expression or RegEx is a special sequence of characters that uses a search pattern to find a string or set of strings.
- It can detect the presence or absence of a text by matching it with a particular pattern and also can split a pattern into one or more sub-patterns.

Regex Module in Python

- Python has a built-in module named "**re**" that is used for regular expressions in Python. We can import this module by using import statement.
- Importing re module in Python using following command:

import re

How to Use RegEx in Python?

- You can use RegEx in Python after importing re module.

Example:

- This Python code uses regular expressions to search for the word **"portal"** in the given string and then prints the start and end indices of the matched word within the string.

```
import re
s = 'GeeksforGeeks: A computer science portal for geeks'
match = re.search(r'portal', s)
print('Start Index:', match.start())
print('End Index:', match.end())
```

Output

Start Index: 34
End Index: 40

Function	Description
re.findall()	finds and returns all matching occurrences in a list
re.compile()	Regular expressions are compiled into pattern objects
re.split()	Split string by the occurrences of a character or a pattern.
re.sub()	Replaces all occurrences of a character or patter with a replacement string.
resubn	It's similar to re.sub() method but it returns a tuple: (new_string, number_of_substitutions)
re.escape()	Escapes special character
re.search()	Searches for first occurrence of character or pattern

Before starting with the Python regex module let's see how to actually write regex using metacharacters or special sequences.

RegEx Functions

The re module in Python provides various functions that help search, match, and manipulate strings using regular expressions.

Below are main functions available in the re module:

1. re.findall()

- Returns all non-overlapping matches of a pattern in the string as a list. It scans the string from left to right.
- **Example:** This code uses regular expression **\d+** to find all sequences of one or more digits in the given string.

```
import re

string = """Hello my Number is 123456789 and
        my friend's number is 987654321"""


regex = '\d+'

match = re.findall(regex, string)

print(match)
```

Output

- ['123456789', '987654321']

2. re.compile()

- Compiles a regex into a pattern object, which can be reused for matching or substitutions.
- **Example 1:** This pattern **[a-e]** matches all lowercase letters between '**a**' and '**e**', in the input string **"Aye, said Mr. Gibenson Stark".** The output should be **['e', 'a', 'd', 'b', 'e']**, which are matching characters.

```
import re

p = re.compile('[a-e]')

print(p.findall("Aye, said Mr. Gibenson Stark"))
```

Output

- ['e', 'a', 'd', 'b', 'e', 'a']

Explanation:

- First occurrence is 'e' in "Aye" and not 'A', as it is Case Sensitive.
- Next Occurrence is 'a' in "said", then 'd' in "said", followed by 'b' and 'e' in "Gibenson", the Last 'a' matches with "Stark".
- Metacharacter backslash '\' has a very important role as it signals various sequences. If the backslash is to be used without its special meaning as metacharacter, use'\\'

3. re.split()

- Splits a string wherever the pattern matches. The remaining characters are returned as list elements.

Syntax:

- *re.split(pattern, string, maxsplit=0, flags=0)*
- **pattern:** Regular expression to match split points.
- **string:** The input string to split.
- **maxsplit (optional):** Limits the number of splits. Default is 0 (no limit).
- **flags (optional):** Apply regex flags like re.IGNORECASE.

Example 1: Splitting by non-word characters or digits

- This example demonstrates how to split a string using different patterns like non-word characters (\W+), apostrophes, and digits (\d+).

```
from re import split

print(split('\W+', 'Words, words , Words'))

print(split('\W+', "Word's words Words"))

print(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))

print(split('\d+', 'On 12th Jan 2016, at 11:02 AM'))
```

Output

```
['Words', 'words', 'Words']

['Word', 's', 'words', 'Words']

['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']

['On ', 'th Jan ', ', at ', ':', ' AM']
```

Example 2: Using maxsplit and flags

- This example shows how to limit the number of splits using maxsplit, and how flags can control case sensitivity.

```
import re
print(re.split('\d+', 'On 12th Jan 2016, at 11:02 AM', 1))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here',
flags=re.IGNORECASE))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here'))
```

Output

```
['On ', 'th Jan 2016, at 11:02 AM']
['', 'y, ', 'oy oh ', 'oy, ', 'om', ' h', 'r', '']
['A', 'y, Boy oh ', 'oy, ', 'om', ' h', 'r', '']
```

- ***Note:*** *In the second and third cases of the above , [a-f]+ splits the string using any combination of lowercase letters from 'a' to 'f'. The re.IGNORECASE flag includes uppercase letters in the match.*

4. re.sub()

- The re.sub() function replaces all occurrences of a pattern in a string with a replacement string.
- **Syntax:**
- *re.sub(pattern, repl, string, count=0, flags=0)*
- **pattern**: The regex pattern to search for.
- **repl**: The string to replace matches with.
- **string**: The input string to process.
- **count** (optional): Maximum number of substitutions (default is 0, which means replace all).
- **flags (optional)**: Regex flags like re.IGNORECASE.
- **Example 1:** The following examples show different ways to replace the pattern '**ub**' with '~*', using various flags and count values.

```
import re

# Case-insensitive replacement of all 'ub'

print(re.sub('ub', '~*', 'Subject has Uber booked already', flags=re.IGNORECASE))

# Case-sensitive replacement of all 'ub'

print(re.sub('ub', '~*', 'Subject has Uber booked already'))

# Replace only the first 'ub', case-insensitive

print(re.sub('ub', '~*', 'Subject has Uber booked already', count=1, flags=re.IGNORECASE))

# Replace "AND" with "&", ignoring case

print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE))
```

Output

```
S~*ject has ~*er booked already
S~*ject has Uber booked already
S~*ject has Uber booked already
Baked Beans & Spam
```

5. re.subn()

- re.subn() function works just like **re.sub()**, but instead of returning only the modified string, it returns a tuple: **(new_string, number_of_substitutions)**
- **Syntax:**
- *re.subn(pattern, repl, string, count=0, flags=0)*
- **Example: Substitution with count**
- This example shows how re.subn() gives both the replaced string and the number of times replacements were made.

```
import re

# Case-sensitive replacement

print(re.subn('ub', '~*', 'Subject has Uber booked already'))

# Case-insensitive replacement

t = re.subn('ub', '~*', 'Subject has Uber booked already',
flags=re.IGNORECASE)

print(t)

print(len(t))      # tuple length

print(t[0])        # modified string
```

Output:

```
('S~*ject has Uber booked already', 1)

('S~*ject has ~*er booked already', 2)

2

S~*ject has ~*er booked already
```

6. re.escape()

- re.escape() function adds a backslash (\) before all special characters in a string. This is useful when you want to match a string literally, including any characters that have special meaning in regex (like ., *, [,], etc.).

Syntax:

re.escape(string)

Example: Escaping special characters

- This example shows how re.escape() treats spaces, brackets, dashes, and tabs as literal characters.

```
import re
print(re.escape("This is Awesome even 1 AM"))
print(re.escape("I Asked what is this [a-9], he said \t ^WoW"))
```

Output

```
This\ is\ Awesome\ even\ 1\ AM
I\ Asked\ what\ is\ this\ \[a\-9\]\,\ he\ said\ \        \ \^WoW
```

7. re.search()

- The re.search() function searches for the first occurrence of a pattern in a string. It returns a **match object** if found, otherwise **None**.
- ***Note:*** *Use it when you want to check if a pattern exists or extract the first match.*
- **Example: Search and extract values**
- This example searches for a date pattern with a month name (letters) followed by a day (digits) in a sentence.

```
import re
regex = r"([a-zA-Z]+) (\d+)"
match = re.search(regex, "I was born on June 24")
if match:
    print("Match at index %s, %s" % (match.start(), match.end()))
    print("Full match:", match.group(0))
    print("Month:", match.group(1))
    print("Day:", match.group(2))
else:
    print("The regex pattern does not match.")
```

Output

```
Match at index 14, 21
Full match: June 24
Month: June
Day: 24
```

Metacharacters

Metacharacters are characters with a special meaning:

Character	Description	Example	Try it
[]	A set of characters	"[a-m]"	
\	Signals a special sequence (can also be used to escape special characters)	"\d"	
.	Any character (except newline character)	"he..o"	
^	Starts with	"^hello"	
$	Ends with	"planet$"	
*	Zero or more occurrences	"he.*o"	
+	One or more occurrences	"he.+o"	
?	Zero or one occurrences	"he.?o"	
{}	Exactly the specified number of occurrences	"he.{2}o"	
|	Either or	"falls|stays"	
()	Capture and group		

Flags

You can add flags to the pattern when using regular expressions.

Flag	Shorthand	Description	Try it
re.ASCII	re.A	Returns only ASCII matches	
re.DEBUG		Returns debug information	
re.DOTALL	re.S	Makes the . character match all characters (including newline character)	
re.IGNORECASE	re.I	Case-insensitive matching	
re.MULTILINE	re.M	Returns only matches at the beginning of each line	
re.NOFLAG		Specifies that no flag is set for this pattern	
re.UNICODE	re.U	Returns Unicode matches. This is default from Python 3. For Python 2: use this flag to return only Unicode matches	
re.VERBOSE	re.X	Allows whitespaces and comments inside patterns. Makes the pattern more readable	

Special Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

		Example	Try it
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"	
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"	
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"	
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"	
\D	Returns a match where the string DOES NOT contain digits	"\D"	
\s	Returns a match where the string contains a white space character	"\s"	
\S	Returns a match where the string DOES NOT contain a white space character	"\S"	
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"	
\W	Returns a match where the string DOES NOT contain any word characters	"\W"	
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"	

Sets

A set is a set of characters inside a pair of square brackets `[]` with a special meaning:

Set	Description	Try it
[arn]	Returns a match where one of the specified characters (`a`, `r`, or `n`) is present	
[a-n]	Returns a match for any lower case character, alphabetically between `a` and `n`	
[^arn]	Returns a match for any character EXCEPT `a`, `r`, and `n`	
[0123]	Returns a match where any of the specified digits (`0`, `1`, `2`, or `3`) are present	
[0-9]	Returns a match for any digit between `0` and `9`	
[0-5][0-9]	Returns a match for any two-digit numbers from `00` and `59`	
[a-zA-Z]	Returns a match for any character alphabetically between `a` and `z`, lower case OR upper case	
[+]	In sets, `+`, `*`, `.`, `|`, `()`, `$`,`{}` has no special meaning, so `[+]` means: return a match for any `+` character in the string	

Match Object

- A Match Object is an object containing information about the search and the result.
- **Note:** If there is no match, the value None will be returned, instead of the Match Object.

Example

- Do a search that will return a Match Object:
- ```
  import re
  txt = "The rain in Spain"
  x = re.search("ai", txt)
  print(x) #this will print an object
  ```

The Match object has properties and methods used to retrieve information about the search, and the result:

- .span() returns a tuple containing the start-, and end positions of the match.
  .string returns the string passed into the function
  .group() returns the part of the string where there was a match

Example

- Print the position (start- and end-position) of the first match occurrence.
- The regular expression looks for any words that starts with an upper case "S":

```
import re
txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

.

Example

- Print the string passed into the function:

```
import re
txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

Example

- Print the part of the string where there was a match.
- The regular expression looks for any words that starts with an upper case "S":

```
import re
txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

- **Note:** If there is no match, the value None will be returned, instead of the Match Object.
```

Pandas

Pandas (stands for Python Data Analysis) is an open-source software library designed for **data manipulation** and **analysis**.

•Revolves around two primary Data structures: Series (1D) and DataFrame (2D)

•Built on top of NumPy, efficiently manages large datasets, offering tools for data cleaning, transformation, and analysis.

•Tools for working with time series data, including date range generation and frequency conversion. For example, we can convert date or time columns into pandas' datetime type using pd.to_datetime(), or specify parse_dates=True during CSV loading.

•Seamlessly integrates with other Python libraries like NumPy, Matplotlib, and scikit-learn.

•Provides methods like .dropna() and .fillna() to handle missing values seamlessly

Important Facts to Know :

- ***DataFrames:*** *It is a two-dimensional data structure constructed with rows and columns, which is more similar to Excel spreadsheet.*
- ***pandas:*** *This name is derived for the term "panel data" which is econometrics terms of data sets.*

Pandas Basic Operations

Pandas simplifies data handling by enabling efficient preprocessing, cleaning, transformation, and visualization.

Data Preprocessing

Handle missing values and clean raw data for analysis.
Before (Table with Missing Values

Before (Table with Missing Values)

Name	Age	Salary_Amount
Aryan	25	NaN
Harsh	NaN	5000
Kunal	30	7000

After (Missing Values Filled)

Name	Age	Salary
Aryan	25	0
Harsh	28	5000
Kunal	30	7000

Operation Applied: Filled missing values with 0 / average using fillna()

Data Cleaning

Remove duplicates and fix column inconsistencies.
Before (Duplicate Rows & Bad Column Names):

Before

Name	Age	Salary_Amount
Aryan	25	5000
Harsh	30	7000
Aryan	25	5000

After

Name	Age	Salary
Aryan	25	5000
Harsh	30	7000

Operation Applied: Removed duplicates using drop_duplicates(), renamed columns using rename()

Data Transformation

Filter and sort data for better insights.

Before (unfiltered Data)

Product	Category	Price
Laptop	Electronics	800
Hoodie	Clothing	20
Smartphone	Electronics	500
Jeans	Clothing	40

After (Filtered & Sorted Data)

Product	Category	Price
Smartphone	Electronics	500
Laptop	Electronics	800

Operation Applied: Filtered Electronics using query(), sorted by price using sort_values()

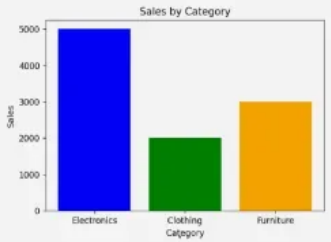
Data Visualization

Represent data graphically for insights

Before (Raw Data Table)

Category	Price
Electronics	5000
Clothing	2000
Furniture	3000

After (Bar Chart Visual)



Operation Applied: Plotted sales data using matplotlib

What is Pandas Used for?

- Reading and writing data from various file formats like CSV, Excel and SQL databases.
- Cleaning and preparing data (handling missing values, filtering, removing duplicates).
- Merging, joining, and reshaping datasets.
- Performing statistical analysis and descriptive statistics.
- Visualizing data quickly.

NumPy

- NumPy is a core Python library for numerical computing, built for handling large arrays and matrices efficiently. It is significantly faster than Python's built-in lists because it uses optimized C-based operations and supports vectorized computations.
- **ndarray object:** Stores homogeneous data in n-dimensional arrays for fast processing.
- **Vectorized operations:** Perform element-wise calculations without explicit loops.
- **Broadcasting**: Apply operations across arrays of different shapes.
- **Linear algebra functions**: Matrix multiplication, inversion, eigenvalues, etc.
- **Statistical tools**: Mean, median, standard deviation, and more.
- **Fourier transforms**: Fast computation for signal and image processing.
- **Integration with other libraries**: Works seamlessly with Pandas, SciPy, and scikit-learn.

Numpy Fundamentals

Numpy provides efficient numerical computing with features like Array Manipulation, mathematical Operation, Linear Algebra and Random number Generation.

01 ARRAY MANIPULATION

Create and reshape arrays

np.array()
reshape()

02 MATHEMATICAL FUNCTIONS

Perform element wise Operations

np.add()
np.multiply()

03 LINEAR ALGEBRA

Perform matrix operations

np.dot()
np.linalg.inv()

04 RANDOM NUMBER GENERATION

Generate Random Data

np.random.rand()
np.random.randint()

Numpy Array Manipulation

Create and reshape array easily using Numpy

Creation

np.array ([1,2,3,4,5,6])

1	2	3	4	5	6

Reshape

Original =np.array ([1,2,3,4,5,6])

1	2	3
4	5	6

Random Array Generation

Generate random numbers using Numpy random module

np.random.rand (2,3)

0.63	0.91	0.40
0.27	0.76	0.10

shape (2,3)

np.random.randint ((0,10), (2,2))

3	7
1	9

shape (2,2)

Mathematical Operations in Numpy

Perform element-wise arithmetic operations using Numpy

a = np.array ([[1,2], [3,4]])

1	2
3	4

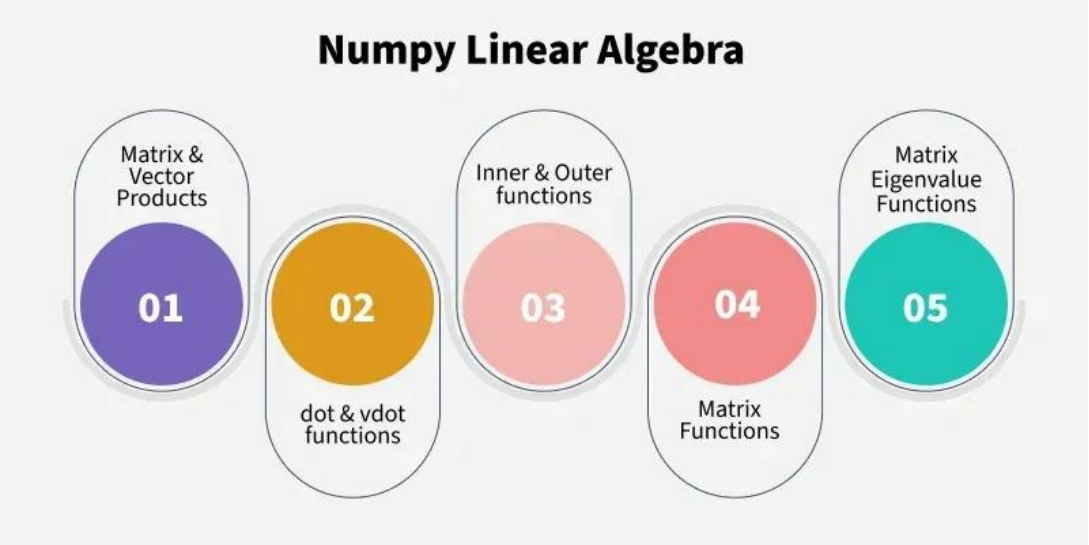
+

b = np.array ([[5,6], [7,8]])

5	6
7	8

=

6	8
10	12



Important Facts to Know :

NumPy arrays are homogeneous, meaning all elements must be the same type, allowing efficient computation.
Vectorized operations in NumPy can be 10 to 100 times faster than equivalent Python loops.

What is NumPy Used for?

- With NumPy, you can perform a wide range of numerical operations, including:
- Creating and manipulating arrays.
- Performing element-wise and matrix operations.
- Generating random numbers and statistical calculations.
- Conducting linear algebra operations.
- Working with Fourier transformations.
- Handling missing values efficiently in datasets.

Why Learn NumPy?

- NumPy speeds up math operations like addition and multiplication on large groups of numbers compared to regular Python..
- It’s good for handling large lists of numbers (arrays), so you don’t have to write complicated loops.
- It gives ready-to-use functions for statistics, algebra and random numbers.
- Libraries like Pandas, SciPy, TensorFlow and many others are built on top of NumPy.
- NumPy uses less memory and stores data more efficiently, which matters when working with lots of data.

Web Scraping

What is Web Scraping?

- The dictionary meaning of word Scrapping implies getting something from the web. Here two questions arise: What we can get from the web and How to get that.
- The answer to the first question is **data**. Data is indispensable for any programmer and the basic requirement of every programming project is the large amount of useful data.
- The answer to the second question is a bit tricky, because there are lots of ways to get data. In general, we may get data from a database or data file and other sources. But what if we need large amount of data that is available online? One way to get such kind of data is to manually search (clicking away in a web browser) and save (copy-pasting into a spreadsheet or file) the required data. This method is quite tedious and time consuming. Another way to get such data is using **web scraping**.
- **Web scraping**, also called **web data mining** or **web harvesting**, is the process of constructing an agent which can extract, parse, download and organize useful information from the web automatically. In other words, we can say that instead of manually saving the data from websites, the web scraping software will automatically load and extract data from multiple websites as per our requirement.

Origin of Web Scraping

- The origin of web scraping is screen scrapping, which was used to integrate non-web based applications or native windows applications. Originally screen scraping was used prior to the wide use of World Wide Web (WWW), but it could not scale up WWW expanded. This made it necessary to automate the approach of screen scraping and the technique called **Web Scraping** came into existence.

Web Crawling v/s Web Scraping

- The terms Web Crawling and Scraping are often used interchangeably as the basic concept of them is to extract data. However, they are different from each other. We can understand the basic difference from their definitions.
- Web crawling is basically used to index the information on the page using bots aka crawlers. It is also called **indexing**. On the hand, web scraping is an automated way of extracting the information using bots aka scrapers. It is also called **data extraction**.
- To understand the difference between these two terms, let us look into the comparison table given hereunder −

Web Crawling	Web Scraping
Refers to downloading and storing the contents of a large number of websites.	Refers to extracting individual data elements from the website by using a site-specific structure.
Mostly done on large scale.	Can be implemented at any scale.
Yields generic information.	Yields specific information.
Used by major search engines like Google, Bing, Yahoo. **Googlebot** is an example of a web crawler.	The information extracted using web scraping can be used to replicate in some other website or can be used to perform data analysis. For example the data elements can be names, address, price etc.

Uses of Web Scraping

- The uses and reasons for using web scraping are as endless as the uses of the World Wide Web. Web scrapers can do anything like ordering online food, scanning online shopping website for you and buying ticket of a match the moment they are available etc. just like a human can do. Some of the important uses of web scraping are discussed here −
- **E-commerce Websites** − Web scrapers can collect the data specially related to the price of a specific product from various e-commerce websites for their comparison.
- **Content Aggregators** − Web scraping is used widely by content aggregators like news aggregators and job aggregators for providing updated data to their users.
- **Marketing and Sales Campaigns** − Web scrapers can be used to get the data like emails, phone number etc. for sales and marketing campaigns.
- **Search Engine Optimization (SEO)** − Web scraping is widely used by SEO tools like SEMRush, Majestic etc. to tell business how they rank for search keywords that matter to them.
- **Data for Machine Learning Projects** − Retrieval of data for machine learning projects depends upon web scraping.

Data for Research − Researchers can collect useful data for the purpose of their research work by saving their time by this automated process

Components of a Web Scraper

A web scraper consists of the following components –

Web Crawler Module

- A very necessary component of web scraper, web crawler module, is used to navigate the target website by making HTTP or HTTPS request to the URLs. The crawler downloads the unstructured data (HTML contents) and passes it to extractor, the next module.

Extractor

- The extractor processes the fetched HTML content and extracts the data into semistructured format. This is also called as a parser module and uses different parsing techniques like Regular expression, HTML Parsing, DOM parsing or Artificial Intelligence for its functioning.

Data Transformation and Cleaning Module

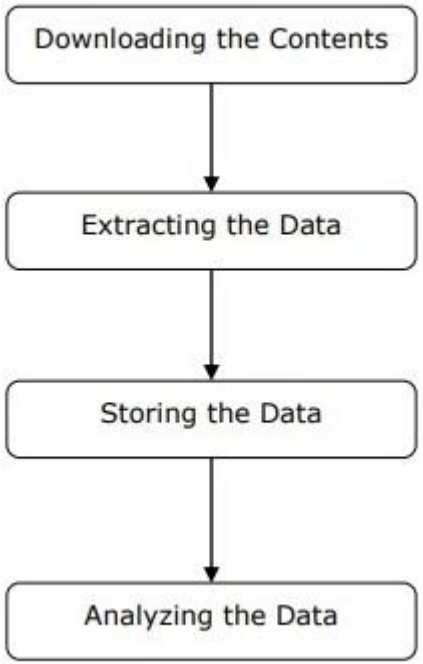
- The data extracted above is not suitable for ready use. It must pass through some cleaning module so that we can use it. The methods like String manipulation or regular expression can be used for this purpose. Note that extraction and transformation can be performed in a single step also.

Storage Module

- After extracting the data, we need to store it as per our requirement. The storage module will output the data in a standard format that can be stored in a database or JSON or CSV format.

Working of a Web Scraper

- Web scraper may be defined as a software or script used to download the contents of multiple web pages and extracting data from it.



We can understand the working of a web scraper in simple steps as shown in the diagram given above.

- Step 1: Downloading Contents from Web Pages
- In this step, a web scraper will download the requested contents from multiple web pages.
- Step 2: Extracting Data
- The data on websites is HTML and mostly unstructured. Hence, in this step, web scraper will parse and extract structured data from the downloaded contents.
- Step 3: Storing the Data
- Here, a web scraper will store and save the extracted data in any of the format like CSV, JSON or in database.
- Step 4: Analyzing the Data
- After all these steps are successfully done, the web scraper will analyze the data thus obtained.