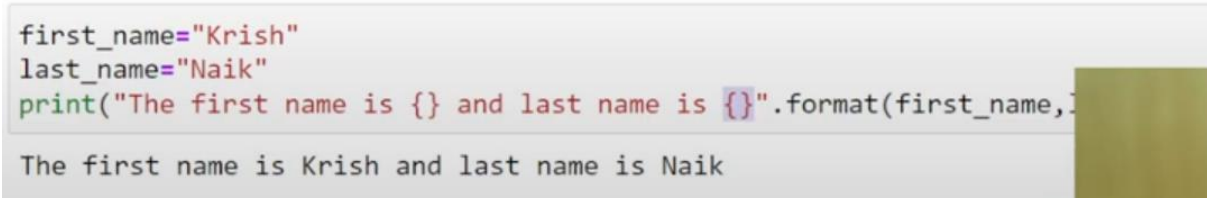
AI/ML

TOPIC:	Python	START DATE:	19/06/2024	↓

1.) String formatting



Kyuki, by default concatenation karta hai,

```
## input type
a=input("Enter the number A ")
b=input("enter the number B ")
print(int(a)+int(b))
```

```
Enter the number A 12
enter the number B 21
33
```

2.) Loops And Control Statements

```
In [27]: ##break
         x=1
         while(x<7):

             if x==4:
                 break
             print(x)
             x=x+1
```

```
1
2
3
```

```
In [1]: ## continue
        x=0
        while x<7:
            x=x+1
            if x==4:
                continue
            print(x)
```

```
1
2
3
5
6
7
```

3.) Operators

The **id() function** in Python returns the unique memory address of an object, which can be shared by multiple variables referencing the same object.

```
In [38]: lst=[1,2,3]
         lst1=[1,2,3]
         print(id(lst))
         print(id(lst1))

2575100854848
2575100861184

In [40]: lst is lst1

Out[40]: False

In [41]: lst is not lst1

Out[41]: True

In [42]: "Krish"!="Krish1"

Out[42]: True
```

The expression lst is lst1 returns **False** because even though lst and lst1 have the same contents, they are separate list objects in memory. **BUT**,

```
In [36]: a="Krish"
         b="Krish"
         print(id(a))
         print(id(b))

2575134525872
2575134525872

In [37]: a is b

Out[37]: True
```

4.) In-built functions

Python Number Methods:

abs(x) will return the absolute value of a number x which we pass in argument. The number x can be integer, float, complex.

ceil(x) will return the ceiling value of a number x which we pass in argument. The ceiling value of a number x will be the smallest integer not less than x.
Note :- ceil(x) function will not be accessible directly using ceil() method. Math module will be required to access this method.

The **math.floor()** function in Python returns the floor value of a given number. The floor value of a number is the largest integer that is not greater than the number.
example:
math.floor(42.1) returns 42 because 42 is the largest integer that is not greater than 42.1.
math.floor(-44.5) returns -45 because -45 is the largest integer that is not greater than -44.5

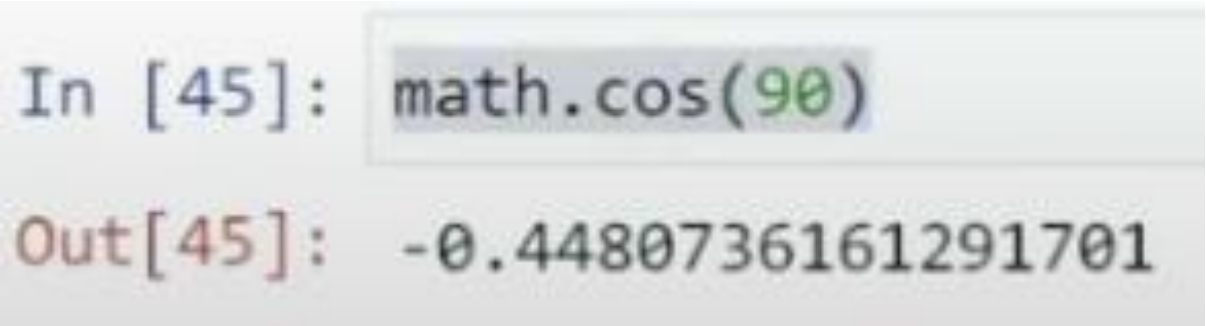
math.exp(x) will return e to the power of x

math.fabs(x) will return the absolute value of a number is its distance from zero, without considering whether it's positive or negative.
example:
math.fabs(10.53) returns 10.53 because the absolute value of 10.53 is 10.53.

math.fabs(-10) returns 10.0 because the absolute value of -10 is 10.0.

math.modf() function in Python returns a tuple containing the fractional and integer parts of a given number,
example:
math.modf(3.14) returns (0.14, 3.0).

Because, it is taking input in radians,



5.) Some basic programs

Program to find max of 3 numbers:

In [28]:
```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))

max_num = max(a, b, c)
print(f"The maximum of {a}, {b}, and {c} is {max_num}.")
```

```
Enter first number: 4
Enter second number: 5
Enter third number: 6
The maximum of 4, 5, and 6 is 6.
```

or

In [29]:
```
def max_of_three(a, b, c):
    return max(a, b, c)

a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))

max_num = max_of_three(a, b, c)
print(f"The maximum of {a}, {b}, and {c} is {max_num}.")
```

```
Enter first number: 4
Enter second number: 5
Enter third number: 6
The maximum of 4, 5, and 6 is 6.
```

Program to find if a number is prime or not:

In [32]:
```python
def is_prime(n, i=2):
    if n <= 1:
        return False
    if n == 2:
        return True
    if n % i == 0:
        return False
    if i * i > n:
        return True
    return is_prime(n, i + 1)

num = int(input("Enter a number: "))

if is_prime(num):
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
```

```
Enter a number: 4
4 is not a prime number.
```

6.) Python Data Structures and Boolean

- **Boolean**

In [8]:
```
my_str='Krish123'
```

In [10]:
```
my_str.istitle()
```

Out[10]: True

In [11]:
```
print(my_str.isalnum()) #check if all char are numbers
print(my_str.isalpha()) #check if all char in the string are alphabetic
print(my_str.isdigit()) #test if string contains digits
print(my_str.istitle()) #test if string contains title words
print(my_str.isupper()) #test if string contains upper case
print(my_str.islower()) #test if string contains lower case
print(my_str.isspace()) #test if string contains spaces
print(my_str.endswith('k')) #test if string endswith a d
print(my_str.startswith('K')) #test if string startswith H
```

```
True
False
False
True
False
False
False
False
True
```

- **Boolean and Logical Operators**

In [13]:
```
str_example='Hello World'
my_str='Krish'
```

In [14]:
```
my_str.isalpha() or str_example.isnum()
```

Out[14]: True

Data Structures: Python vs C++

Python	C++	Description
List my_list = [1, 2, 3, 4, 5]	**std::vector** std::vector<int> my_vector = {1, 2, 3, 4, 5};	Mutable collection of items
Set my_set = {1, 2, 3, 4, 5}	**std::unordered_set** std::unordered_set<int> my_set = {1, 2, 3, 4, 5};	Unordered collection of unique elements
Tuple my_tuple = (1, 2, 3, 4, 5)	**std::tuple** std::tuple<int, int, int, int, int> my_tuple(1, 2, 3, 4, 5);	Immutable collection of elements
Dictionary my_dict = {'name': 'John', 'age': 30}	**std::unordered_map** std::unordered_map<string, variant<string, int>> my_map = {{"name", "John"}, {"age", 30}};	Key-value pairs

- **Lists (IMPORTANT)**

In [8]:
```
str1="Naik"
print(str1)
```

```
Naik
```

In [12]:
```
##string are immutable
str1[2]="jk"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_22692/266264539.py in <module>
----> 1 str1[2]="jk"

TypeError: 'str' object does not support item assignment
```

BUT,

In [41]:
```
##Lists
##mutable
lst=[1,2,3,4,"Krish","Helllo"]
print(lst)
lst[4]="Naik"
print(lst[4])
print(lst)
```

```
[1, 2, 3, 4, 'Krish', 'Helllo']
Naik
[1, 2, 3, 4, 'Naik', 'Helllo']
```

In [37]:
```
lst.append(["John","Bala"])
lst
```

Out[37]: [1, 2, 3, 4, 5, 'Krish', ['John', 'Bala']]

In [37]:
```
lst.append(["John","Bala"])
lst
```

Out[37]: [1, 2, 3, 4, 5, 'Krish', ['John', 'Bala']]

In [44]:
```
lst[2:6]
```

Out[44]: [3, 4, 5, 'Krish']

In [41]:
```
lst
```

Out[41]: [1, 2, 3, 4, 5, 'Krish', ['John', 'Bala']]

In [23]:
```
##Indexing in List
lst[6]
```

Out[23]: 'Krish'

In [26]:
```
lst[1:6]
```

Out[26]: ['chemistry', 100, 200, 300, 204]

```
In [37]: lst.append(["John","Bala"])
         lst

Out[37]: [1, 2, 3, 4, 5, 'Krish', ['John', 'Bala']]
```

```
In [40]: lst[6][1]

Out[40]: 'Bala'
```

```
In [46]: ## insert in a specific order

         lst.insert(2,"Naik")

In [47]: lst

Out[47]: [1, 2, 'Naik', 3, 4, 5, 'Krish', ['John', 'Bala']]
```

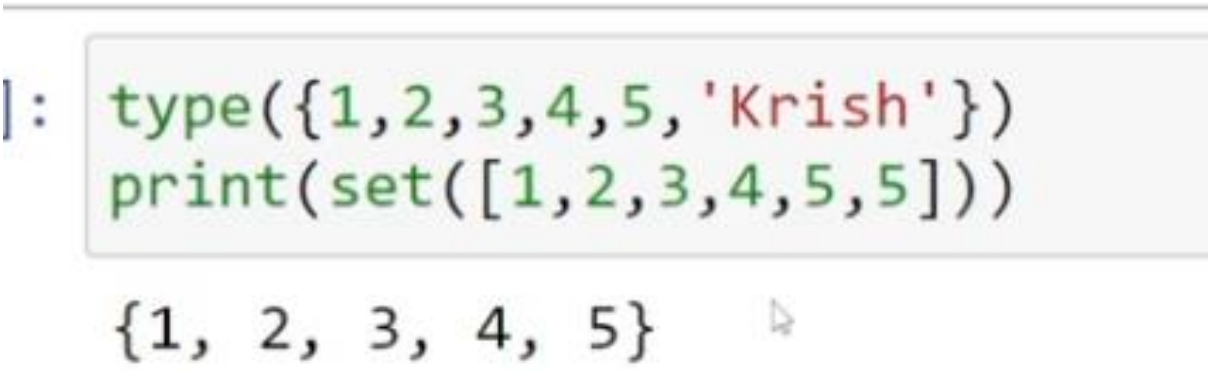
```
In [53]: lst=[1,2,3,4,5]

In [54]: sum(lst)

Out[54]: 15

In [58]: lst*5

Out[58]: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

count():Calculates total occurrence of given element of List

```
In [41]: lst=[1,1,2,3,4,5]
         lst.count(1)

Out[41]: 2
```

- **Sets**

A Set is an **unordered** collection data type that is iterable, mutable, and has no duplicate elements. Python's set class represents the mathematical notion of a set.This is based on a data structure known as a **hash table.**



DOESN'T come in sorted order,

In Python, a set is an **unordered** collection of unique elements. This means that the order of elements in a set is not guaranteed and can vary between different Python implementations or even between different runs of the same program.

However, in Python 3.7 and later, the `set` type maintains the insertion order of elements, which can sometimes give the illusion that sets are ordered. This is because the underlying implementation of sets uses a hash table, and the iteration order is based on the hash values of the elements.

```
In [44]: {3, 1, 2, 1, 4}

Out[44]: {1, 2, 3, 4}
```

```
In [50]: set1={"IronMan","Avengers", 'Hitman'}
         print(set1)

         {'Avengers', 'Hitman', 'IronMan'}
```

SETS me objects are not subscriptable, matlab unlike list, we can't access them using their index,

```
set_var[]

---------------------------------------------------------
TypeError                                 Traceback (mos
~\AppData\Local\Temp/ipykernel_22692/511998048.py in <mo
----> 1 set_var[2]

TypeError: 'set' object is not subscriptable
```

Although, iteration kar sakte hai,

```
In [80]: for i in set_var:
             print(i)

Hitman
Avengers
IronMan
```

INTERSECTION se bas o/p aa raha hai, update nahi ho raha hai,

```
In [89]: set1={"Avengers","IronMan",'Hitman'}
         set2={"Avengers","IronMan",'Hitman','Hulk2'}

In [ ]:

In [90]: set2.intersection(set1)
Out[90]: {'Avengers', 'Hitman', 'IronMan'}
```

BUT, what if we want to update also?

```
In [92]: set2.intersection_update(set1)

In [93]: set2
Out[93]: {'Avengers', 'Hitman', 'IronMan'}
```

```
In [51]: set1={"Avengers","IronMan",'Hitman'}
         set2={"Avengers","IronMan",'Hitman','Hulk2'}

In [53]: set1.difference(set2)
Out[53]: set()
```

(set 1 − set 2)

```
In [55]: set2.difference_update(set1)

In [56]: print(set2)
         {'Hulk2'}
```

- **Dictionaries**

A dictionary is a collection which is **unordered, changeable and indexed**. In Python dictionaries are written with curly brackets, and they have keys and values.

```
In [72]: ## Let create a dictionary
         my_dict={"Car1": "Audi", "Car2":"BMW","Car3":"Mercidies Benz"}

In [64]: type(my_dict)
Out[64]: dict

In [73]: ##Access the item values based on keys
         my_dict['Car1']
Out[73]: 'Audi'
```

In [66]:
```
# We can even loop throught the dictionaries keys

for x in my_dict:
    print(x)
```

```
Car1
Car2
Car3
```

In [68]:
```
# We can even loop throught the dictionaries values

for x in my_dict.values():
    print(x)
```

```
Audi
BMW
Mercidies Benz
```

In [69]:
```
# We can also check both keys and values
for x in my_dict.items():
    print(x)
```

```
('Car1', 'Audi')
('Car2', 'BMW')
('Car3', 'Mercidies Benz')
```

```
## Adding items in Dictionaries

my_dict['car4']='Audi 2.0'
```

Nested Dictionary

In [76]:
```
car1_model={'Mercedes':1960}
car2_model={'Audi':1970}
car3_model={'Ambassador':1980}

car_type={'car1':car1_model,'car2':car2_model,'car3':car3_model}
```

In [77]:
```
print(car_type)
```

```
{'car1': {'Mercedes': 1960}, 'car2': {'Audi': 1970}, 'car3': {'Ambassador': 1980}}
```

In [80]:
```
## Accessing the items in the dictionary

print(car_type['car1'])
```

```
{'Mercedes': 1960}
```

In [81]:
```
print(car_type['car1']['Mercedes'])
```

```
1960
```

In [64]:
```
car1_model={'Mercedes':1960}
car2_model={'Audi':1970}
car3_model={'Ambassador':1980}

car_type={'car1':car1_model,'car2':car2_model,'car3':car3_model}
```

In [69]:
```
print([car_type]) # typecasted to a list
```

```
[{'car1': {'Mercedes': 1960}, 'car2': {'Audi': 1970}, 'car3': {'Ambassador': 1980}}]
```

- **Tuples**

In [75]:
```python
# Creating Tuples
my_tuple = ("apple", "banana", "cherry")
print(my_tuple)  # Output: ('apple', 'banana', 'cherry')

# Create a tuple with a single element
single_element_tuple = ("apple",)
print(single_element_tuple)  # Output: ('apple',)

# Create an empty tuple
empty_tuple = ()
print(empty_tuple)  # Output: ()

# Indexing and Slicing
print(my_tuple[0])  # Output: apple
print(my_tuple[1])  # Output: banana
print(my_tuple[2])  # Output: cherry

print(my_tuple[1:2])  # Output: ('banana',)
print(my_tuple[1:])  # Output: ('banana', 'cherry')
print(my_tuple[:2])  # Output: ('apple', 'banana')

# Tuple Methods
print(my_tuple.count("apple"))  # Output: 1
print(my_tuple.index("banana"))  # Output: 1
print(len(my_tuple))  # Output: 3
```

```python
# Tuple Operations
tuple1 = ("apple", "banana")
tuple2 = ("cherry", "date")
print(tuple1 + tuple2)  # Output: ('apple', 'banana', 'cherry', 'date')
print(tuple1 * 2)  # Output: ('apple', 'banana', 'apple', 'banana')
print("apple" in tuple1)  # Output: True
print("cherry" in tuple1)  # Output: False

# Tuple Unpacking
fruit1, fruit2, fruit3 = my_tuple
print(fruit1)  # Output: apple
print(fruit2)  # Output: banana
print(fruit3)  # Output: cherry

# Converting Between Tuples and Lists
my_list = list(my_tuple)
print(my_list)  # Output: ['apple', 'banana', 'cherry']
my_tuple = tuple(my_list)
print(my_tuple)  # Output: ('apple', 'banana', 'cherry')

# Immutable Nature of Tuples
try:
    my_tuple[0] = "date"
except TypeError as e:
    print(e)  # Output: 'tuple' object does not support item assignment
```

6.) Numpy Arrays

```
Out[14]: array([1, 2, 3, 5])

In [20]: arr[-1]
Out[20]: 5

In [21]: arr[:-1]
Out[21]: array([1, 2, 3])

In [23]: arr[::-1]
Out[23]: array([5, 3, 2, 1])

In [24]: arr[::-2]
Out[24]: array([5, 2])
```

```
Out[26]: array([[1, 2, 3, 4, 5],
                [2, 3, 4, 5, 6],
                [3, 4, 5, 6, 7]])
```

```
In [28]: arr1[:,1]

Out[28]: array([2, 3, 4])
```

Returns number of elements<2,

```
In [33]: ##EDA
         arr

Out[33]: array([1, 2, 3, 5])

In [35]: arr[arr<2]

Out[35]: array([1])
```

1. `np.arange`

python Open In Editor

```
1  np.arange(start, stop, step, dtype=None)
```

- `start`: The starting value of the sequence.
- `stop`: The end value of the sequence.
- `step`: The increment between values in the sequence.
- `dtype`: The data type of the output array.

2. `np.linspace`

```python
np.linspace(start, stop, num, endpoint=True, retstep=False, dtype=None)
```

- `start`: The starting value of the sequence.
- `stop`: The end value of the sequence.
- `num`: The number of samples to generate.
- `endpoint`: If `True`, `stop` is included in the sequence.
- `retstep`: If `True`, return the step size as well.
- `dtype`: The data type of the output array.

np.logspace and np.geomspace are also there...

Here is the syntax for the `reshape` function in NumPy:

```python
np.reshape(arr, newshape, order='C')
```

- `arr`: The input array to be reshaped.
- `newshape`: The new shape of the array. This can be a tuple of integers, or an integer if the new shape is one-dimensional.
- `order`: The order in which to store the data in memory. The default is `'C'`, which means row-major (C-style) ordering. Other options are `'F'` for column-major (Fortran-style) ordering, and `'A'` for preserving the original ordering.

2. Reshaping a 2D array to a 1D array

```python
arr = np.array([[1, 2, 3], [4, 5, 6]])
arr_reshaped = np.reshape(arr, (6,))  # [1, 2, 3, 4, 5, 6]
```

3. Reshaping a 3D array to a 2D array

```python
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
arr_reshaped = np.reshape(arr, (4, 2))  # [[1, 2], [3, 4], [5, 6], [7, 8]]
```

4. Using the `-1` placeholder

```python
arr = np.array([1, 2, 3, 4, 5, 6])
arr_reshaped = np.reshape(arr, (2, -1))  # [[1, 2, 3], [4, 5, 6]]
```

In this example, the `-1` placeholder tells NumPy to infer the size of the second dimension based on the size of the original array.

Note that when reshaping an array, the total number of elements must remain the same. If the new shape is incompatible with the original array, NumPy will raise a `ValueError`.

NumPy's **random** module,

np.random.rand()

np.random.randn()

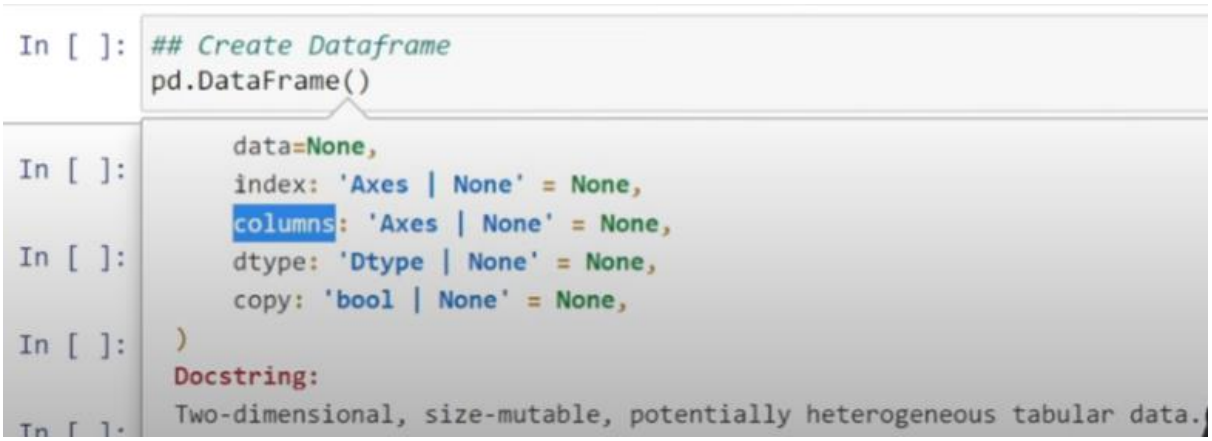
np.random.randint()

np.random.choice()

np.random.seed()

https://numpy.org/doc/stable/reference/random/index.html

7.) Pandas




```
In [100]: ## Create Dataframe
          df=pd.DataFrame(data=np.arange(0,20).reshape(5,4),index=["Row1",
                                                                   "Row2","Row3",
                                                                   "Row4","Row5"],columns=["Column1",
                                                                                           "Column2",
                                                                                           "Column3",
                                                                                           "Column4"])
```

```
In [101]: df.head()
```

Out[101]:

	Column1	Column2	Column3	Column4
Row1	0	1	2	3
Row2	4	5	6	7
Row3	8	9	10	11
Row4	12	13	14	15
Row5	16	17	18	19

`df.head()`: The `head()` method returns the first `n` rows of a DataFrame, where `n` is a default value of 5. It's useful for quickly inspecting the top rows of a dataset. You can specify a different value for `n` by passing an integer argument, e.g., `df.head(10)` to display the first 10 rows.

`df.tail()`: The `tail()` method returns the last `n` rows of a DataFrame, where `n` is a default value of 5. It's useful for quickly inspecting the bottom rows of a dataset. You can specify a different value for `n` by passing an integer argument, e.g., `df.tail(10)` to display the last 10 rows.

Here's a concise comparison of pandas DataFrame vs Series in Python:

DataFrame:

- 2-dimensional labeled data structure
- Multiple columns, each can be a different type
- Like a spreadsheet or SQL table

Series:

- 1-dimensional labeled array
- Can hold data of any type
- Like a single column of a DataFrame

Key differences:

1. Dimensions: DataFrame is 2D, Series is 1D
2. Structure: DataFrame can have multiple columns, Series is a single column
3. Use cases: DataFrames for complex data analysis, Series for simpler data representations

`df.loc` (Label-based indexing)

`df.loc` is used to access rows and columns by their labels. It's primarily label-based, which means you need to specify the row and column labels to select the desired data.

Syntax:

python Editor

```python
df.loc[row_labels, column_labels]
```

`df.iloc` (Integer-based indexing)

`df.iloc` is used to access rows and columns by their integer positions. It's primarily integer-based, which means you need to specify the row and column indices to select the desired data.

Syntax:

python Editor

```python
df.iloc[row_indices, column_indices]
```

```
In [26]: ##convert dataframe into arrays
         df.iloc[:,1:].values

Out[26]: array([[ 1,  2,  3],
                [ 5,  6,  7],
                [ 9, 10, 11],
                [13, 14, 15],
                [17, 18, 19]])
```

.csv is a 'comma-separated-file'

In [109]:
```
### Reading Different Data sources with the help of pandas
from io import StringIO
```

In [110]:
```
import pandas as pd
```

In [111]:
```
df=pd.read_csv('mercedesbenz.csv')
df.head()
```

Out[111]:

	ID	y	X0	X1	X2	X3	X4	X5	X6	X8	...	X375	X376	X377	X378	X379	X380	X382	X
0	0	130.81	k	v	at	a	d	u	j	o	...	0	0	1	0	0	0	0	
1	6	88.53	k	t	av	e	d	y	l	o	...	1	0	0	0	0	0	0	
2	7	76.26	az	w	n	c	d	x	j	x	...	0	0	0	0	0	0	1	
3	9	80.62	az	t	n	f	d	x	l	e	...	0	0	0	0	0	0	0	
4	13	78.02	az	v	n	f	d	h	d	n	...	0	0	0	0	0	0	0	

5 rows × 378 columns

In [15]:
```
data=('col1,col2,col3\n'
      'x,y,1\n'
      'a,b,2\n'
      'c,d,3')
```

In [16]:
```
type(data)
```

Out[16]:
```
str
```

In [17]:
```
##in memeory file format object
StringIO(data)
```

Out[17]:
```
<_io.StringIO at 0x1a1eb0b5e50>
```

In [18]:
```
pd.read_csv(StringIO(data))
```

Out[18]:

	col1	col2	col3
0	x	y	1
1	a	b	2
2	c	d	3

display particular columns,

In [21]:
```
import pandas as pd
df=pd.read_csv('mercedesbenz.csv',usecols=['X0','X1','X2','X3','X4','X5'])
df.head()
```

Out[21]:

	X0	X1	X2	X3	X4	X5
0	k	v	at	a	d	u
1	k	t	av	e	d	y
2	az	w	n	c	d	x
3	az	t	n	f	d	x
4	az	v	n	f	d	h

In [120]:
```
##datatypes in csv
data = ('a,b,c,d\n'
            '1,2,3,4\n'
            '5,6,7,8\n'
            '9,10,11')
```

In [121]:
```
df=pd.read_csv(StringIO(data),dtype={'a':int,'b':float,'c':int})
```

In [122]:
```
df
```

Out[122]:

	a	b	c	d
0	1	2.0	3	4.0
1	5	6.0	7	8.0
2	9	10.0	11	NaN

```
In [136]: pd.read_csv('cu.item.txt',sep='\t')
```

Out[136]:

	item_code	item_name	display_level	selectable	sort_sequence
0	AA0	All items - old base	0	T	2
1	AA0R	Purchasing power of the consumer dollar - old ...	0	T	400
2	SA0	All items	0	T	1
3	SA0E	Energy	1	T	375
4	SA0L1	All items less food	1	T	359
...	...	...	...	...	...
395	SSEA011	College textbooks	3	T	314
396	SSEE041	Smartphones	4	T	335
397	SSFV031A	Food at elementary and secondary schools	3	T	122
398	SSGE013	Infants' equipment	3	T	356
399	SSHJ031	Infants' furniture	3	T	165

400 rows × 5 columns

By specifying `sep='\t'`, you're telling pandas to parse the file as a **tab-separated values (TSV) file**, rather than a traditional comma-separated values (CSV) file.

This is useful when working with files that use tabs as separators, like the `cu.item.txt` file in your example.

Python Pandas Working With JSON:-

```
In [169]: import pandas as pd
          df = pd.DataFrame([['a', 'b'], ['c', 'd']],
                            index=['row 1', 'row 2'],
                            columns=['col 1', 'col 2'])
```

```
In [170]: df
```

Out[170]:

	col 1	col 2
row 1	a	b
row 2	c	d

```
In [171]: df.to_json()
```

```
Out[171]: '{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Different operations on this JSON data,

```
In [13]: df.to_json(orient='index')

Out[13]: '{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'

In [14]: df.to_json(orient='columns')

Out[14]: '{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'

In [15]: df.to_json(orient='records')

Out[15]: '[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'

In [16]: df.to_json(orient='split')

Out[16]: '{"columns":["col 1","col 2"],"index":["row 1","row 2"],"data":[["a","b"],["c","d"]]}'

In [17]: df.to_json(orient='table')

Out[17]: '{"schema":{"fields":[{"name":"index","type":"string"},{"name":"col 1","type":"string"},{"name":"col 2","t
         ype":"string"}],"primaryKey":["index"],"pandas_version":"0.20.0"},"data":[{"index":"row 1","col 1":"a","co
         l 2":"b"},{"index":"row 2","col 1":"c","col 2":"d"}]}'

In [18]: schema='{"schema":{"fields":[{"name":"index","type":"string"},{"name":"col 1","type":"string"},{"name":"co

In [20]: pd.read_json(schema,orient='table')

Out[20]:
```

	col 1	col 2

Normalise me kya karega ki agar, for eg 2 job profiles hai to dono ka alag column bana dega, (ie conversion in dictionary)

```
In [185]: import pandas as pd

          data = [{"employee_name": "James", "email": "james@gmail.com", "job_profile": {"

          df = pd.DataFrame(data)
```

```
In [186]: df
```

Out[186]:

	employee_name	email	job_profile
0	James	james@gmail.com	{'title1': 'Team Lead', 'title2': 'Sr. Develop...

```
In [ ]:
```

```
In [183]: import pandas as pd

          data = [{"employee_name": "James", "email": "james@gmail.com", "job_profile": {"

          df = pd.json_normalize(data)
```

```
In [184]: df
```

Out[184]:

	employee_name	email	job_profile.title1	job_profile.title2
0	James	james@gmail.com	Team Lead	Sr. Developer

Playing with JSON,

In [32]:
```
data = [
    {
        "id": 1,
        "name": "Cole Volk",
        "fitness": {"height": 130, "weight": 60},
    },
    {"name": "Mark Reg", "fitness": {"height": 130, "weight": 60}},
    {
        "id": 2,
        "name": "Faye Raker",
        "fitness": {"height": 130, "weight": 60},
    },
]
```

In [33]:
```
pd.json_normalize(data)
```

Out[33]:

	id	name	fitness.height	fitness.weight
0	1.0	Cole Volk	130	60
1	NaN	Mark Reg	130	60
2	2.0	Faye Raker	130	60

In [34]:
```
pd.json_normalize(data,max_level=0)
```

Out[34]:

	id	name	fitness
0	1.0	Cole Volk	{'height': 130, 'weight': 60}
1	NaN	Mark Reg	{'height': 130, 'weight': 60}
2	2.0	Faye Raker	{'height': 130, 'weight': 60}

max_level ya ye matlab hai ki wo utni baar nested ke andar jayega,

In [35]:
```
pd.json_normalize(data,max_level=1)
```

Out[35]:

	id	name	fitness.height	fitness.weight
0	1.0	Cole Volk	130	60
1	NaN	Mark Reg	130	60
2	2.0	Faye Raker	130	60

example:

In [40]:

```
data = [
    {
        "state": "Florida",
        "shortname": "FL",
        "info": {"governor": "Rick Scott"},
        "counties": [
            {"name": "Dade", "population": 12345},
            {"name": "Broward", "population": 40000},
            {"name": "Palm Beach", "population": 60000},
        ],
    },
    {
        "state": "Ohio",
        "shortname": "OH",
        "info": {"governor": "John Kasich"},
        "counties": [
            {"name": "Summit", "population": 1234},
            {"name": "Cuyahoga", "population": 1337},
        ],
    },
]
```

pd.json_normalise(data, “counties”), iska matlab ye hai ki ham counties wale key data list ko hi normalise kar rahe hai

In [188]:

```
pd.json_normalize(data)
```

Out[188]:

	state	shortname	counties	info.governor
0	Florida	FL	[{'name': 'Dade', 'population': 12345}, {'name...	Rick Scott
1	Ohio	OH	[{'name': 'Summit', 'population': 1234}, {'nam...	John Kasich

In [190]:

```
pd.json_normalize(data, "counties")
```

Out[190]:

	name	population
0	Dade	12345
1	Broward	40000
2	Palm Beach	60000
3	Summit	1234
4	Cuyahoga	1337

Yaha pe, we are normalizing the data counties (which has name and population), state, info (jiske andar abhi ek hi field hai ie governor)

```
In [44]: pd.json_normalize(data, "counties",["state", "shortname","info"])
```

Out[44]:

	name	population	state	shortname	info
0	Dade	12345	Florida	FL	{'governor': 'Rick Scott'}
1	Broward	40000	Florida	FL	{'governor': 'Rick Scott'}
2	Palm Beach	60000	Florida	FL	{'governor': 'Rick Scott'}
3	Summit	1234	Ohio	OH	{'governor': 'John Kasich'}
4	Cuyahoga	1337	Ohio	OH	{'governor': 'John Kasich'}

To agar hame specifically governor ki hi details nikalni hai to aise likhna padega, ie ek alag se list banake,

```
In [45]: pd.json_normalize(data, "counties",["state", "shortname",["info",'governor']])
```

Out[45]:

	name	population	state	shortname	info.governor
0	Dade	12345	Florida	FL	Rick Scott
1	Broward	40000	Florida	FL	Rick Scott
2	Palm Beach	60000	Florida	FL	Rick Scott
3	Summit	1234	Ohio	OH	John Kasich
4	Cuyahoga	1337	Ohio	OH	John Kasich

Or, directly we can do by meta,

```
In [199]: df = pd.json_normalize(
              data,
              record_path='counties',
              meta=['state', 'shortname', ['info', 'governor']]
          )
```

```
In [200]: df
```

Out[200]:

	name	population	state	shortname	info.governor
0	Dade	12345	Florida	FL	Rick Scott
1	Broward	40000	Florida	FL	Rick Scott
2	Palm Beach	60000	Florida	FL	Rick Scott
3	Summit	1234	Ohio	OH	John Kasich
4	Cuyahoga	1337	Ohio	OH	John Kasich

Reading HTML in json:-

```
In [201]: import pandas as pd
          html=pd.read_html("https://en.wikipedia.org/wiki/Mobile_country_code")

In [202]: type(html)

Out[202]: list

In [203]: html

Out[203]: [   MCC  MNC Brand      Operator       Status Bands (MHz)  \
           0    1    1  TEST  Test network  Operational         any
           1    1    1  TEST  Test network  Operational         any
           2  999   99   NaN  Internal use  Operational         any
           3  999  999   NaN  Internal use  Operational         any

                                 References and notes
           0                                       NaN
           1                                       NaN
           2  Internal use in private networks, no roaming[6]
           3  Internal use in private networks, no roaming[6]  ,
                Mobile country code                                  Country ISO 3166  \
           0                    289                              A Abkhazia    GE-AB
           1                    412                             Afghanistan       AF
           2                    276                                 Albania       AL
           3                    603                                 Algeria       DZ
           4                    544  American Samoa (United States of America)       AS
           ..                   ...                                     ...      ...
           247                  452                                 Vietnam       VN

In [204]: html[0]

Out[204]:
```

	MCC	MNC	Brand	Operator	Status	Bands (MHz)	References and notes
0	1	1	TEST	Test network	Operational	any	NaN
1	1	1	TEST	Test network	Operational	any	NaN
2	999	99	NaN	Internal use	Operational	any	Internal use in private networks, no roaming[6]
3	999	999	NaN	Internal use	Operational	any	Internal use in private networks, no roaming[6]

```
In [18]: type(html)

Out[18]: list

In [22]: type(html[1])

Out[22]: pandas.core.frame.DataFrame
```

When we want to access the data of a particular row,

```
In [217]: html = pd.read_html("https://en.wikipedia.org/wiki/Mobile_country_code")
          first_row = html[1].iloc[0]
          print(first_row)
```

```
Mobile country code                                    289
Country                                          A Abkhazia
ISO 3166                                             GE-AB
Mobile network codes      List of mobile network codes in Abkhazia
National MNC authority                                  NaN
Remarks                                 MCC is not listed by ITU
Name: 0, dtype: object
```

```
In [209]: type(html)
Out[209]: list
```

```
In [215]: html[1].iloc[0]
```

Out[215]:

	Mobile country code	Country	ISO 3166	Mobile network codes	National MNC authority	Remarks
0	289	A Abkhazia	GE-AB	List of mobile network codes in Abkhazia	NaN	MCC is not listed by ITU
1	412	Afghanistan	AF	List of mobile network codes in Afghanistan	NaN	NaN
2	276	Albania	AL	List of mobile network codes in Albania	NaN	NaN

Suppose we want to search for a particular table. then we will use match,

en.wikipedia.org/wiki/Minnesota

ral habitats and many species that live in them. For example, the lakes' water is ning, which affects cold-water fish. Trout, for example, is a cold-water fish that is losing its habitat, while the habitat of bass, a warm-water fish, is growing.[78]

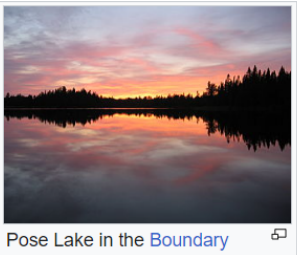
Köppen climate types of Minnesota

Average daily maximum and minimum temperatures for selected cities in Minnesota[79]

Location	July (°F)	July (°C)	January (°F)	January (°C)
Minneapolis	83/64	28/18	23/7	−4/−13
Saint Paul	83/63	28/17	23/6	−5/−14
Rochester	82/63	28/17	23/3	−5/−16
Duluth	76/55	24/13	19/1	−7/−17
St. Cloud	81/58	27/14	18/−1	−7/−18
Mankato	86/62	30/16	23/3	−5/−16
International Falls	77/52	25/11	15/−6	−9/−21

Protected lands [edit]

Minnesota's first state park, Itasca State Park, was established in 1891, and is the source of the Mississippi River.[80] Today Minnesota has 72 state parks and recreation areas, 58 state forests covering about four million acres (16,000 km^2), and numerous state wildlife preserves, all managed by the Minnesota Department of Natural Resources. The Chippewa and Superior national forests comprise 5.5 million acres (22,000 km^2). The Superior National Forest in the northeast contains the Boundary Waters Canoe Area Wilderness, which encompasses over a million acres (4,000 km^2) and a thousand lakes. To its west is Voyageurs National Park. The Mississippi National River and Recreation Area (MNRRA) is a 72-mile-long (116 km) corridor along the Mississippi River through the Minneapolis–St. Paul Metropolitan Area connecting a variety of sites of historic,



Pose Lake in the Boundary Waters Canoe Area Wilderness

```
In [221]: html=pd.read_html("https://en.wikipedia.org/wiki/Minnesota",match="Average daily
```

```
In [223]: html[0]
```

Out[223]:

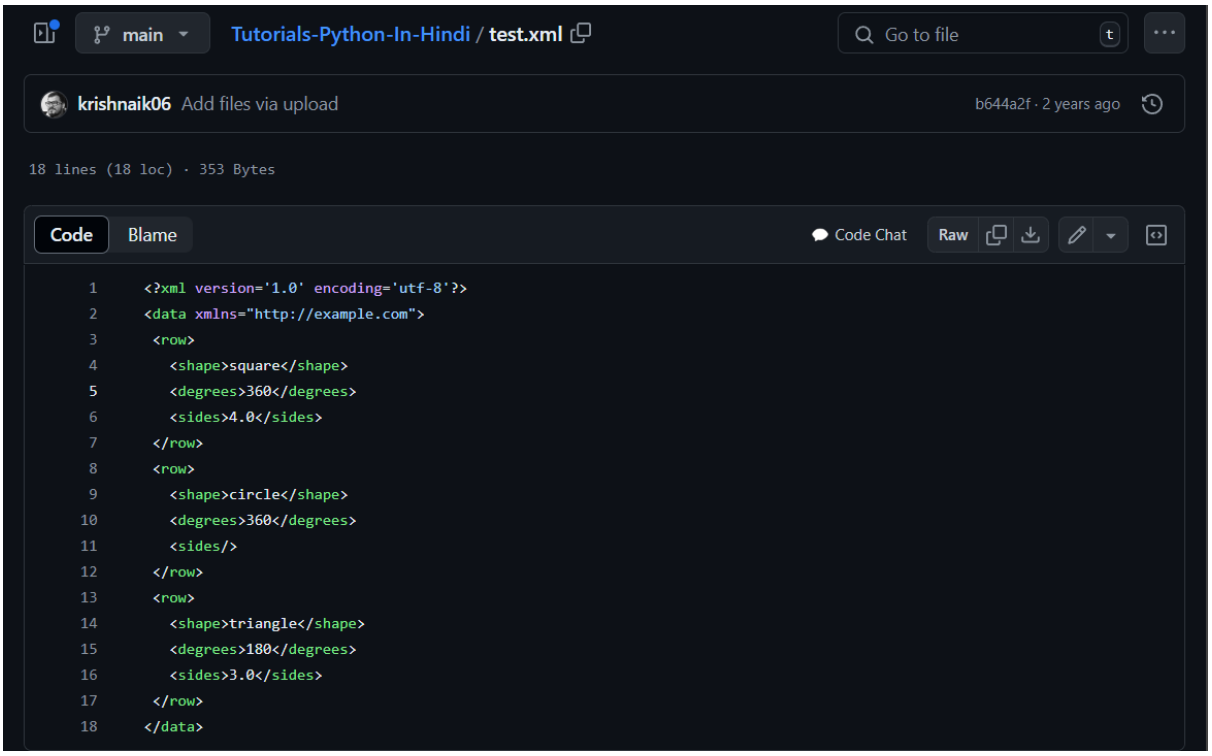
	Location	July (°F)	July (°C)	January (°F)	January (°C)
0	Minneapolis	83/64	28/18	23/7	−4/−13
1	Saint Paul	83/63	28/17	23/6	−5/−14
2	Rochester	82/63	28/17	23/3	−5/−16
3	Duluth	76/55	24/13	19/1	−7/−17
4	St. Cloud	81/58	27/14	18/−1	−7/−18
5	Mankato	86/62	30/16	23/3	−5/−16
6	International Falls	77/52	25/11	15/−6	−9/−21

Python Pandas Working With XML:-

1. Read XML and get Dataframe
2. Convert Dataframe to XML

What is XML?

1. XML stands for eXtensible Markup Language
2. XML is a markup language much like HTML
3. XML was designed to store and transport data
4. XML was designed to be self-descriptive
5. XML is a W3C Recommendation



In [34]:
```
import pandas as pd

pd.read_xml('test.xml')
```

Out[34]:

	shape	degrees	sides
0	square	360	4.0
1	circle	360	NaN
2	triangle	180	3.0

and this is the XML file hardcoded, which mean

In [228]:
```
xml = '''<?xml version='1.0' encoding='utf-8'?>
<data xmlns="http://example.com">
 <row>
   <shape>square</shape>
   <degrees>360</degrees>
   <sides>4.0</sides>
   <firstname>Krish</firstname>
 </row>
 <row>
   <shape>circle</shape>
   <degrees>360</degrees>
   <sides/>
   <firstname/>
 </row>
 <row>
   <shape>triangle</shape>
   <degrees>180</degrees>
   <sides>3.0</sides>
   <firstname/>
 </row>
</data>'''
```

In [229]:
```
pd.read_xml(xml)
```

Out[229]:

	shape	degrees	sides	firstname
0	square	360	4.0	Krish
1	circle	360	NaN	None
2	triangle	180	3.0	None

And this is the 2nd format of XMLs (in which we are writing the key attributes in the row name itself),

```
In [39]: xml = '''<?xml version='1.0' encoding='utf-8'?>
<data>
  <row shape="square" degrees="360" sides="4.0"/>
  <row shape="circle" degrees="360"/>
  <row shape="triangle" degrees="180" sides="3.0"/>
</data>'''
```

```
In [40]: pd.read_xml(xml)
```

Out[40]:

	shape	degrees	sides
0	square	360	4.0
1	circle	360	NaN
2	triangle	180	3.0

```
In [43]: xml = '''<?xml version='1.0' encoding='utf-8'?>
<data>
  <row shape="square" degrees="360" sides="4.0" firstname="Krish"/>
  <row shape="circle" degrees="360"/>
  <row shape="triangle" degrees="180" sides="3.0" lastname="Naik"/>
</data>'''
```

```
In [44]: pd.read_xml(xml,xpath=".//row")
```

Out[44]:

	shape	degrees	sides	firstname	lastname
0	square	360	4.0	Krish	None
1	circle	360	NaN	None	None
2	triangle	180	3.0	None	Naik

BE AWARE of namespaces whenever using XMLNs,

In [230]:
```
xml = '''<?xml version='1.0' encoding='utf-8'?>
<doc:data xmlns:doc="https://example.com">
  <doc:row>
    <doc:shape>square</doc:shape>
    <doc:degrees>360</doc:degrees>
    <doc:sides>4.0</doc:sides>
  </doc:row>
  <doc:row>
    <doc:shape>circle</doc:shape>
    <doc:degrees>360</doc:degrees>
    <doc:sides/>
  </doc:row>
  <doc:row>
    <doc:shape>triangle</doc:shape>
    <doc:degrees>180</doc:degrees>
    <doc:sides>3.0</doc:sides>
  </doc:row>
</doc:data>'''
```

In [233]:
```
df=pd.read_xml(xml,xpath=".//doc:row",namespaces={"doc": "https://example.com"})
```

In [234]:
```
df
```

Out[234]:

	shape	degrees	sides
0	square	360	4.0
1	circle	360	NaN
2	triangle	180	3.0

1. **CSV**: `pd.read_csv()` - Read a comma-separated values (csv) file.
2. **HTML**: `pd.read_html()` - Read HTML tables into a list of DataFrame objects.
3. **Excel**: `pd.read_excel()` - Read an Excel file into a pandas DataFrame.
4. **JSON**: `pd.read_json()` - Read a JSON file into a pandas DataFrame.
5. **SQL**: `pd.read_sql()` - Read a SQL query or database table into a pandas DataFrame.
6. **XML**: `pd.read_xml()` - Read an XML file into a pandas DataFrame (new in pandas 1.3.0).
7. **Stata**: `pd.read_stata()` - Read a Stata file into a pandas DataFrame.
8. **SAS**: `pd.read_sas()` - Read a SAS file into a pandas DataFrame.
9. **Pickled**: `pd.read_pickle()` - Read a pickled pandas object from a file.
10. **Feather**: `pd.read_feather()` - Read a feather-format file into a pandas DataFrame.
11. **HDF5**: `pd.read_hdf()` - Read a HDF5 file into a pandas DataFrame.
12. **MsgPack**: `pd.read_msgpack()` - Read a MessagePack file into a pandas DataFrame.
13. **GBQ**: `pd.read_gbq()` - Read a Google BigQuery table into a pandas DataFrame.
14. **Oracle**: `pd.read_sql_query()` with `oracle` engine - Read an Oracle database table into a pandas DataFrame.
15. **Parquet**: `pd.read_parquet()` - Read a Parquet file into a pandas DataFrame.

8.) Python Pickling And Unpickling

The tutorial itself explains the concept of Pickling and Unpickling in Python. Pickling is the process of converting a Python object hierarchy into a byte stream, which can be stored in a file, database, or transmitted over a network. Unpickling is the inverse operation, where a byte stream is converted back into a Python object hierarchy.

Pickle in Python is primarily used in serializing and deserializing a Python object structure. In other words, it's the process of converting a Python object into a byte stream to store it in a file/database, maintain program state across sessions, or transport data over the network

In [12]:
```
import seaborn as sns
```

In [13]:
```
df=sns.load_dataset('tips')
```

In [14]:
```
df.head()
```

Out[14]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

1. `import seaborn as sns`: This line imports the Seaborn library and assigns it the alias `sns`. Seaborn is a Python data visualization library based on matplotlib.
2. `df = sns.load_dataset('tips')`: This line loads a dataset called "tips" from Seaborn's built-in datasets. The dataset is stored in a Pandas DataFrame object called `df`.
3. `df.head()`: This line displays the first few rows of the `df` DataFrame using the `head()` method. By default, `head()` shows the first 5 rows of the data.

In [15]:
```
import pickle
```

In [16]:
```
filename='file.pkl'
```

In [17]:
```
##serialize process
pickle.dump(df,open(filename,'wb'))
```

In [19]:
```
##unsereliaze
df=pickle.load(open(filename,'rb'))
```

In [21]:
```
df.head()
```

Out[21]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

The data comes inside the file.pkl using the seaborn library,

The data in `df` came from the Seaborn library's `load_dataset` function, which loaded a dataset called "tips" into a Pandas DataFrame object called `df`.

Here's the code that loaded the data:

```
import seaborn as sns
df = sns.load_dataset('tips')
```

The `load_dataset` function retrieved the "tips" dataset from Seaborn's built-in datasets and stored it in the `df` variable. This dataset contains information about tips given to waiters in a restaurant, with columns for total bill, tip amount, sex, smoker, day, time, and size.

After loading the data, the code serialized the `df` object using `pickle.dump()` and wrote it to a file called `file.pkl`.

1. `import pickle`: This line imports the `pickle` module, which provides a way to serialize and deserialize Python objects.
2. `filename='file.pkl'`: This line sets a variable `filename` to the string `'file.pkl'`, which will be used as the file name for storing the pickled object.
3. `pickle.dump(df, open(filename, 'wb'))`: This line serializes the Pandas DataFrame object `df` using the `dump()` function from the `pickle` module. The `open()` function is used to open a file in binary write mode (`'wb'`) with the specified `filename`. The `dump()` function writes the serialized object to the file.
4. `df = pickle.load(open(filename, 'rb'))`: This line deserializes the pickled object from the file using the `load()` function from the `pickle` module. The `open()` function is used to open the file in binary read mode (`'rb'`) with the same `filename` as before. The `load()` function reads the serialized object from the file and returns it as a Python object, which is assigned to the `df` variable.

Commonly used types of picking and unpicking (EXTRA):-

Pickling (Serializing)

1. `pickle.dump(obj, file)`: Serialize an object `obj` and write it to a file-like object `file`.
    - `obj`: The object to be serialized.
    - `file`: A file-like object (e.g., a file opened in binary write mode (`'wb'`)).

Example: `pickle.dump(df, open('file.pkl', 'wb'))`

2. `pickle.dumps(obj)`: Serialize an object `obj` and return it as a bytes object.
    - `obj`: The object to be serialized.

Example: `pickled_data = pickle.dumps(df)`

Unpickling (Deserializing)

1. `pickle.load(file)`: Deserialize an object from a file-like object `file` and return it.
    - `file`: A file-like object (e.g., a file opened in binary read mode (`'rb'`)).

Example: `df = pickle.load(open('file.pkl', 'rb'))`

2. `pickle.loads(data)`: Deserialize an object from a bytes object `data` and return it.
    - `data`: A bytes object containing the serialized data.

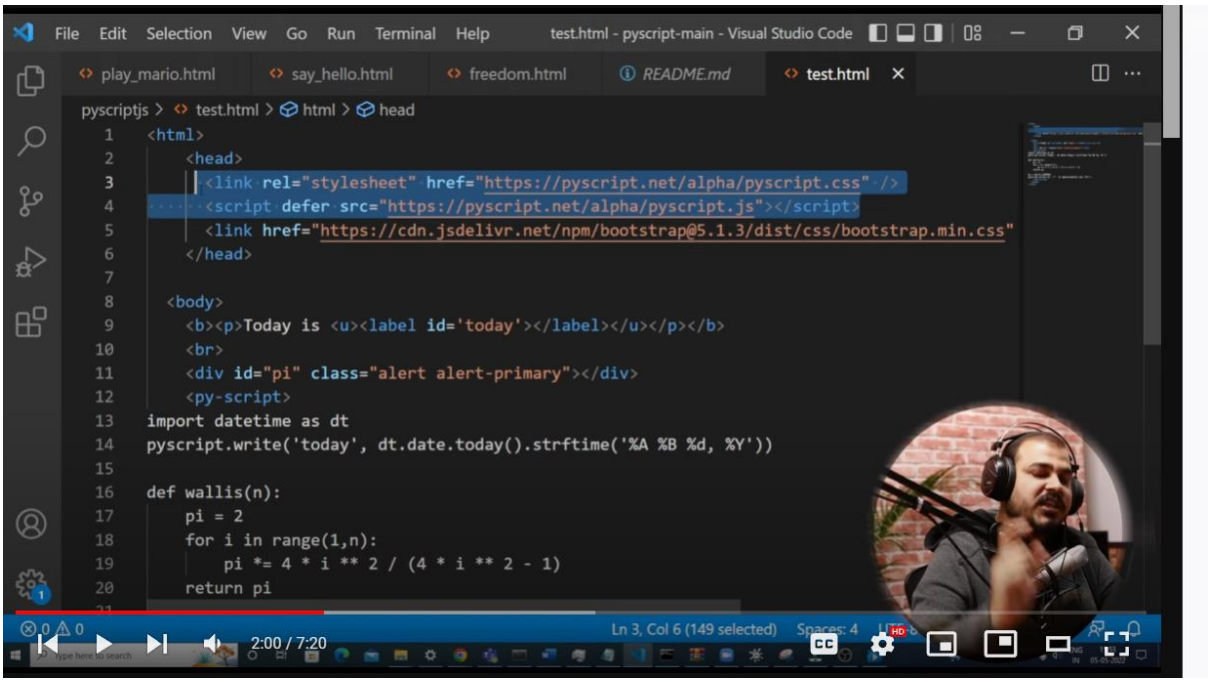
Example: `df = pickle.loads(pickled_data)`

one more **example**,

```
In [245]: dic_example={'first_name':'Krish','last_name':'Naik'}

In [246]: pickle.dump(dic_example,open('test2.pkl','wb'))

In [247]: pickle.load(open('test2.pkl','rb'))
Out[247]: {'first_name': 'Krish', 'last_name': 'Naik'}
```

9.) PyScript

To enable, ye dono cheeze to honi hi chahiye



PyScript is a new web framework that allows you to write Python code directly in your HTML files, without the need for a backend server or a separate Python environment. It's a way to use Python on the client-side, similar to how JavaScript is used.

Here are some key features of PyScript:

1. **Python in HTML**: You can write Python code directly in your HTML files, using the `<py-script>` tag.

2. **Client-side execution**: PyScript code is executed on the client-side, in the user's web browser, rather than on a server.

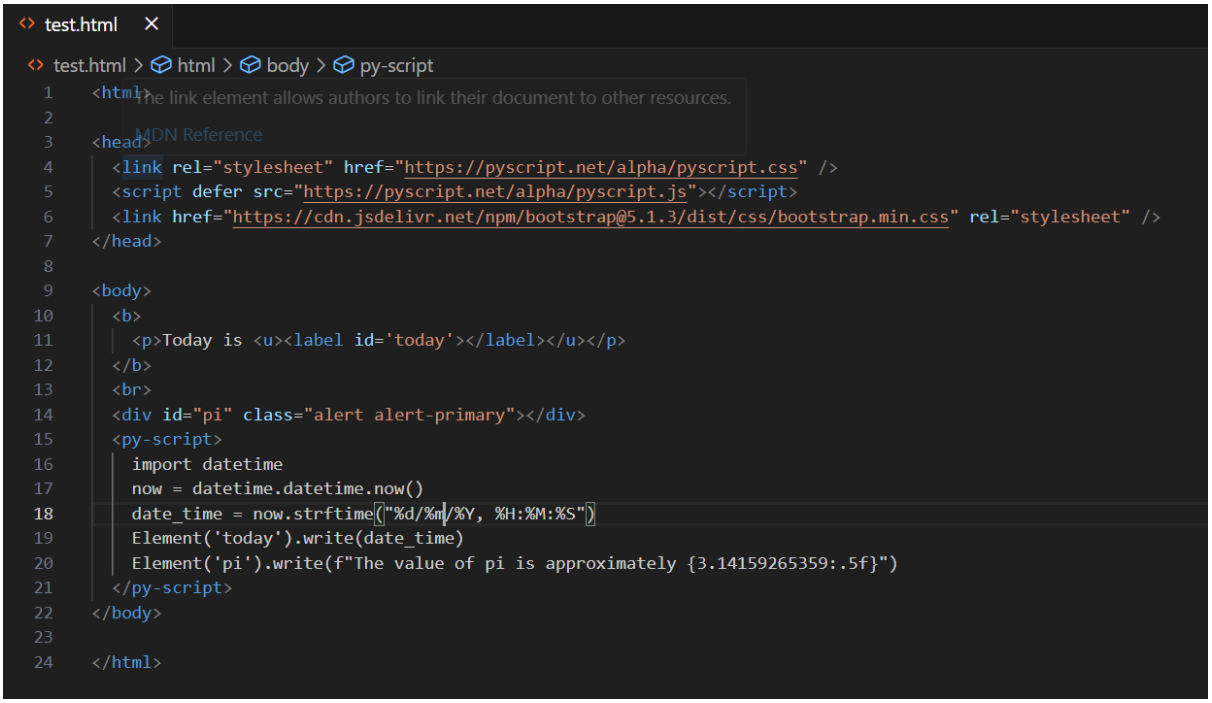
3. **No backend required**: You don't need to set up a backend server or API to use PyScript. It's a self-contained solution.

4. **Interactive**: PyScript allows for interactive coding, so you can write and run code in real-time, without needing to refresh the page.

5. **Support for popular libraries**: PyScript supports popular Python libraries like NumPy, Pandas, and Matplotlib, making it easy to use them in your web applications.

PyScript is still in its alpha stage, but it has the potential to revolutionize the way we build web applications, making it easier to use Python for web development.

example:



10.) Python Functions

```python
## why functions?(Interview Question)
# 1. to make code more readable
# 2. to make code more efficient
# 3. to make code more maintainable
# 4. to make code more reusable
# 5. to make code more extensible


# function

def welcome(msg)->str:
    """
    Description: This function will show a welcome message


    Return : This function will return the welcome message
    """



    return msg



msg=welcome("Welcome all")
print(msg + "Please subscribe")
```

Positional Arguments

Positional arguments are the values that are passed to a function in the order they are defined in the function's parameter list. They are assigned to the corresponding parameters in the order they are defined.

Example:

```python
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("John", 30)  # Output: Hello, John! You are 30 years old.
```

Keyword Arguments

Keyword arguments are values that are passed to a function using the parameter name as a keyword. They are assigned to the corresponding parameters by matching the keyword with the parameter name.

Example:

```python
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet(age=30, name="John")  # Output: Hello, John! You are 30 years old.
```

Mixing Positional and Keyword Arguments

You can mix positional and keyword arguments when calling a function. Positional arguments must be passed before keyword arguments.

Example:

```
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("John", age=30)  # Output: Hello, John! You are 30 years old.
```

But, **positional argument can't appear after keyword argument**,

Default Values

You can also specify default values for function parameters. If a keyword argument is not provided, the function will use the default value.

Example:

```
def greet(name, age=25):
    print(f"Hello, {name}! You are {age} years old.")

greet("John")  # Output: Hello, John! You are 25 years old.
```

*args and **kwargs:-

***args**:

*args is used to capture a variable number of non-keyword arguments. It allows you to pass a tuple of arguments to a function. When you use *args, you can pass any number of arguments to the function, and they will be collected into a tuple.

In your example, `hello("Krish","Naik","1","2","1989")` is equivalent to `hello(*("Krish","Naik","1","2","1989"))`. The * operator unpacks the tuple and passes each element as a separate argument to the function.

When you print `args` inside the `hello` function, you'll see a tuple containing all the arguments: `('Krish', 'Naik', '1', '2', '1989')`.

****kwargs**:

**kwargs is used to capture a variable number of keyword arguments. It allows you to pass a dictionary of keyword arguments to a function. When you use **kwargs, you can pass any number of keyword arguments to the function, and they will be collected into a dictionary.

In your example, `hello(age=10,dob=1990)` is equivalent to `hello(**{'age': 10, 'dob': 1990})`. The ** operator unpacks the dictionary and passes each key-value pair as a separate keyword argument to the function.

When you print `kwargs` inside the `hello` function, you'll see a dictionary containing all the keyword arguments: `{'age': 10, 'dob': 1990}`.

You can combine `*args` and `**kwargs` in the same function signature. This allows you to handle both non-keyword and keyword arguments.

In your example, `hello("Krish","NAik","1","2","1989",age=10,dob=1990)` is equivalent to `hello(*("Krish","NAik","1","2","1989"), **{'age': 10, 'dob': 1990})`. The * operator unpacks the tuple and passes each element as a separate non-keyword argument, while the ** operator unpacks the dictionary and passes each key-value pair as a separate keyword argument.

When you print `args` and `kwargs` inside the `hello` function, you'll see:

- `args: ('Krish', 'NAik', '1', '2', '1989')`
- `kwargs: {'age': 10, 'dob': 1990}`

example:

```
def hello(*args,**kwargs):
    print(args)
    print(kwargs)

#hello("Krish","Naik",age=32,dob=1989)

lst=list(('Krish', 'Naik'))
dict_val={'age': 32, 'dob': 1989}

#hello(*lst,**dict_val)

hello("Krish","NAik","1","2","1989",age=10,dob=1990)
```

11.) OOPS Basics

self is the word jiski madad se hum jo bhi attributes aage likhe hai unhe access kar sakte hai.

```python
class Car:
    # constructor
    def __init__(self,windows,tyres,engine):
        self.windows=windows
        self.tyres=tyres
        self.engine=engine

    def self_driving(self,engine):
        print("The car type is {} engine ".format(engine))


car1=Car(4,4,"petrol")
print("The no of tyres in object car1 is {}".format(car1.tyres))
print("The no of windows in object car1 is {}".format(car1.windows))
car1.self_driving("electric")
```

Python in-built methods:

1. `__init__`: This is a special method that is automatically called when an object of a class is instantiated. It is used to initialize the attributes of the class.

Example:

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p = Person("John", 30)
print(p.name)  # Output: John
print(p.age)   # Output: 30
```

2. __str__: This method returns a string representation of an object. It is often used to provide a human-readable representation of an object.

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

p = Person("John", 30)
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__',
 '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__redu
ce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'doors', 'driving', 'enginetype', '
horsepower', 'selfdriving', 'windows']
```

Multiple Constructors:

In python, it is not allowed but, there is a JUGAAD.

```
class Animal:
    def __init__(self,*args):
        if len(args)==1:
            self.name = args[0]
        elif len(args)==2:
            self.name = args[0]
            self.species = args[1]
        elif len(args)==3:
            self.name = args[0]
            self.species = args[1]
            self.age = args[2]

    def make_sound(self, sound):
        return "The animal is {} and says {}".format(self.name, sound)

dog=Animal("dog","mammals",17)
print(dog.name)
print(dog.species)
print(dog.age)
print(dog.make_sound("woof wwoof"))
```

Inheritance:

In this eg, the Audi is inheriting the properties of Car, with some additional key attribute (horsepower).

super()__init__(****, ****, ****) matlab hum parent ke attributes ko call kar rahe hai.

self.horsepower matlab, horsepower additional attribute hai.

```
class Car:
    def __init__(self,windows,doors,enginetype):
        self.windows=windows
        self.doors=doors
        self.enginetype=enginetype

    def driving(self):
        print("Car is ued for driving")

##Audi car is inherting from Car class
class Audi(Car):
    def __init__(self,windows,doors,enginetype,horsepower):
        super().__init__(windows,doors,enginetype)
        self.horsepower=horsepower

    def selfdriving(self):
        print("IT is a self driving car")


audiq7=Audi(4,5,"Diesel",200)

print(audiq7.horsepower)
print(audiq7.windows)
audiq7.driving()
audiq7.selfdriving()

car1=Car(4,5,"Diesel")
print(car1)
print(audiq7)

print(dir(audiq7))
print(dir(car1))
```

12.) List me Mastery

```python
## Filtering even numbers from a list:
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_number=[n for n in numbers if n%2==0]
print(even_number)
```

```
[2, 4, 6, 8, 10]
```


```python
##Flattening a list of lists: 2 for loops
lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattend_list=[item for sublist in lists for item in sublist ]
```

```python
flattend_list
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```python
# Generating a list of the first letters of words in a list:
words = ['apple', 'banana', 'cherry', 'date']
first_letters=[word[0] for word in words]
print(first_letters)
```

```
['a', 'b', 'c', 'd']
```

```
## Converting a list of strings to a list of integers
strings = ['1', '2', '3', '4', '5']
[int(s) for s in strings]
```

```
[1, 2, 3, 4, 5]
```

```
##Generating a list of all the divisors of a number:

number =36
[i for i in range(1,number+1) if number%i==0]
```

```
[1, 2, 3, 4, 6, 9, 12, 18, 36]
```

example:

```
## Generating a list of all the prime numbers less than a given number:
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

number = 36
primes = [i for i in range(2, number) if is_prime(i)]

print(f"Prime numbers less than {number}:")
print(primes)
```

```
Prime numbers less than 36:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

3. In our specific example: `f"Prime numbers less than {number}:"`
   - The `{number}` part will be replaced with the actual value of the `number` variable (which is 36 in this case).

4. Comparison with older formatting methods:
   - Before f-strings, you might have written this using the `.format()` method: `"Prime numbers less than {}:".format(number)`
   - Or using the older %-formatting: `"Prime numbers less than %d:" % number`

5. Advantages of f-strings:
   - They are more readable and concise.
   - They can include any valid Python expressions inside the curly braces.

13.) Lambda (anonymous) Functions

What is a lambda function?

A lambda function is a small, anonymous function that can be defined using the `lambda` keyword. It's called "anonymous" because it doesn't have a declared name.

The syntax for a lambda function is as follows:

```
lambda arguments: expression
```

Where:

- `arguments` is a comma-separated list of variables that will be passed to the function.
- `expression` is the code that will be executed when the function is called.

Example

Here's a simple example of a lambda function that takes a single argument x and returns its square:

```
square = lambda x: x ** 2
print(square(4))  # Output: 16
```

Common use cases for lambda functions

Lambda functions are often used in the following situations:

1. **Data processing**: Lambda functions can be used to process data in a concise and expressive way.
2. **Event handling**: Lambda functions can be used as event handlers in GUI applications or web development.
3. **Higher-order functions**: Lambda functions can be used as arguments to higher-order functions, such as `map()`, `filter()`, and `reduce()`.

We can make multiple Lambda functions,

No, you can't make only one lambda function per program. There is no such restriction in Python. You can define as many lambda functions as you want in a single program.

Here's an example:

```
square = lambda x: x ** 2
cube = lambda x: x ** 3
print(square(4))  # Output: 16
print(cube(4))  # Output: 64
```

```
## Return the length of a string
string_length=lambda s:len(s)
string_length("Krish Naik")
```

```
10
```

```
## Convert a list of integers to their corresponding square values:
numbers=[1,2,3,4,5,6]
squares=list(map(lambda x:x**2,numbers))
print(squares)
```

```
[1, 4, 9, 16, 25, 36]
```

```
## Filter out even numbers from a list:

numbers=[1,2,3,4,5,6]
list(filter(lambda x:x%2==0,numbers))
```

```
[2, 4, 6]
```

```
## Sort a list of strings based on their alphabetical characters and  length:
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
```

```
sorted(fruits,key=lambda x:len(x))
```

```
['date', 'apple', 'banana', 'cherry', 'elderberry']
```

Complex examples:

1. First, sort the list by the length and then secondly, sort it alphabetically

```
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
sorted_fruits = sorted(fruits, key=lambda x: (x.lower(), len(x)))
print(sorted_fruits)
```

```
['apple', 'banana', 'cherry', 'date', 'elderberry']
```

```
## Sort a list of strings based on their alphabetical characters and  length:
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
sorted(fruits,key=lambda x:len(x))
```

```
['date', 'apple', 'banana', 'cherry', 'elderberry']
```

2. Sorting a list of dictionaries based on a specific key

```
people = [
    {'name': 'Alice', 'age': 25, 'occupation': 'Engineer'},
    {'name': 'Bob', 'age': 30, 'occupation': 'Manager'},
    {'name': 'Charlie', 'age': 22, 'occupation': 'Intern'},
    {'name': 'Dave', 'age': 27, 'occupation': 'Designer'},
]
```

```
sorted(people,key=lambda x:(x['age']))
```

```
[{'name': 'Charlie', 'age': 22, 'occupation': 'Intern'},
 {'name': 'Alice', 'age': 25, 'occupation': 'Engineer'},
 {'name': 'Dave', 'age': 27, 'occupation': 'Designer'},
 {'name': 'Bob', 'age': 30, 'occupation': 'Manager'}]
```

3. Finding max value in a dictionary

```
data = {'a': 10, 'b': 20, 'c': 5, 'd': 15}
max(data,key=lambda x:data[x])
```

```
'b'
```

4. Grouping a list of strings based on their first letter

```
from itertools import groupby

words = ['apple', 'banana', 'cherry', 'date', 'elderberry', 'fig']

groups = groupby(sorted(words), key=lambda x: x[0])

for key, group in groups:
    print(key, list(group))
```

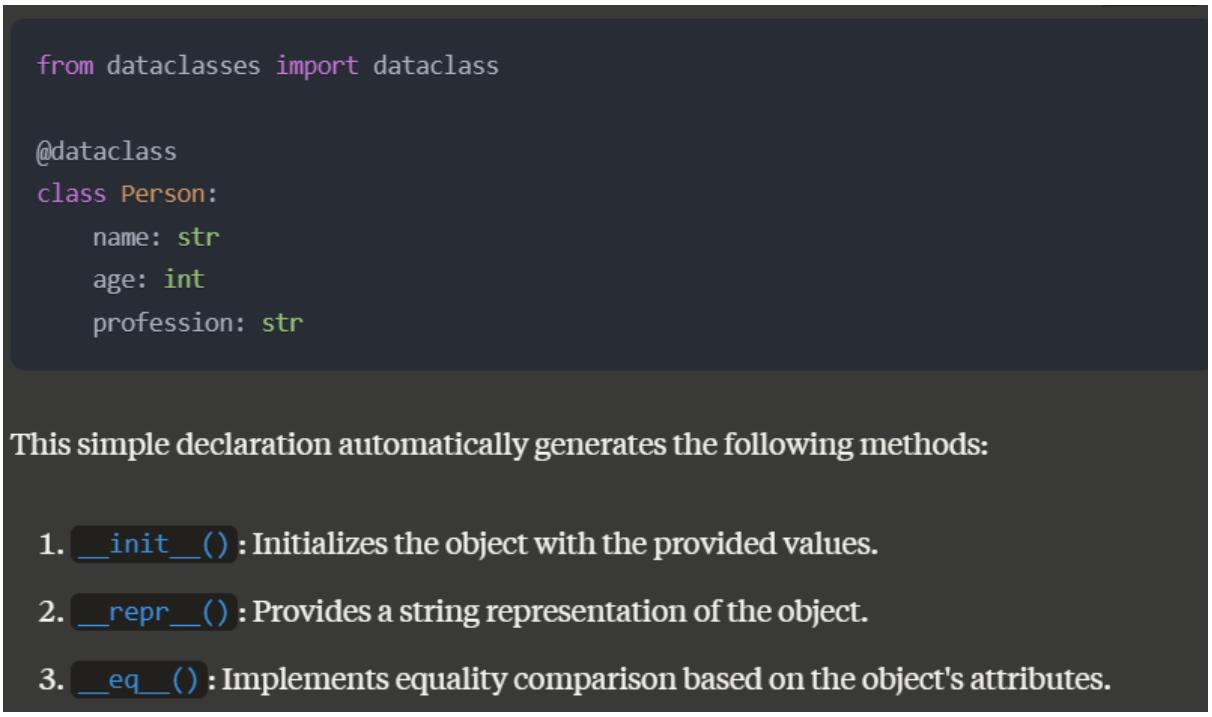
```
a ['apple']
b ['banana']
c ['cherry']
d ['date']
e ['elderberry']
f ['fig']
```

14.) Python Dataclasses

Data classes in Python are a feature introduced in Python 3.7 that provides a concise way to define classes that primarily store data. They automatically generate several special methods, such as `__init__`, `__repr__`, and `__eq__`, based on the class attributes you define.

Here are the **key features** of data classes in Python:

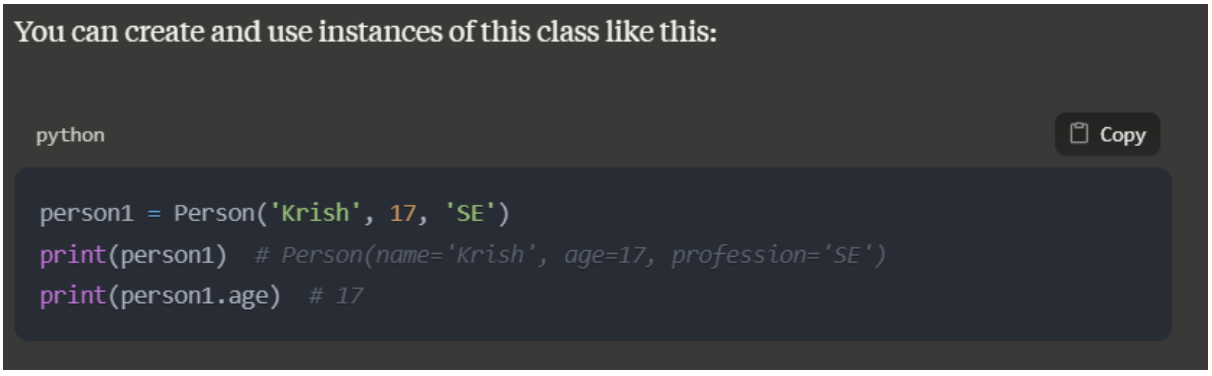
1. **Automatic Generation of Special Methods**: Data classes automatically generate several special methods, including `__init__`, `__repr__`, and `__eq__`, based on the class attributes you define. This simplifies the process of creating and working with data-focused classes.



```python
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int
    profession: str
```

This simple declaration automatically generates the following methods:

1. `__init__()`: Initializes the object with the provided values.
2. `__repr__()`: Provides a string representation of the object.
3. `__eq__()`: Implements equality comparison based on the object's attributes.



You can create and use instances of this class like this:

python Copy

```python
person1 = Person('Krish', 17, 'SE')
print(person1)  # Person(name='Krish', age=17, profession='SE')
print(person1.age)  # 17
```

2. **Concise Definition**: Data classes can be defined in a concise manner by using the `@dataclass` decorator and specifying the attributes of the class.

Default Values

You can specify default values for attributes:

python Copy

```python
@dataclass
class Rectangle:
    width: int
    height: int
    color: str = 'white'

rectangle1 = Rectangle(12, 14)  # color will be 'white'
rectangle2 = Rectangle(13, 14, 'red')
print(rectangle2.color)  # 'red'
```

3. **Immutable by Default**: Data classes are immutable by default, meaning that once an object is created, its attributes **cannot be changed**. This can be overridden by setting the `**frozen**` parameter to `**False**` when defining the data class.

Immutable Data Classes

To create an immutable data class, use the `frozen` parameter:

python Copy

```python
@dataclass(frozen=True)
class Point:
    x: int
    y: int

point = Point(3, 4)
# point.x = 12  # This would raise a FrozenInstanceError
```

4. **Inheritance**: Data classes support inheritance, which allows you to create a hierarchy of classes that **share common attributes and behaviour.**

5. **Nested Data Classes**: Data classes can be nested, which allows you to create complex data structures with **multiple levels of nesting**.

Nested Data Classes

You can nest data classes within each other:

python Copy

```python
@dataclass
class Address:
    street: str
    city: str
    zip_code: str

@dataclass
class Person:
    name: str
    age: int
    address: Address

address = Address('123 Main Street', 'Bangalore', '12345')
person = Person("Krish", 31, address)
print(person.address.city)  # 'Bangalore'
```

15.) Encapsulation in Python

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). In Python, it refers to the bundling of data (attributes) and the methods that operate on that data within a single unit or object. The main goals of encapsulation are:

1. To restrict direct access to some of an object's components
2. To bind together the data and the methods that manipulate that data

Python implements encapsulation through the use of classes and access modifiers. Let's break this down:

1. Classes as Units of Encapsulation: In Python, a class serves as the basic unit of encapsulation. It combines attributes (data) and methods (functions that operate on the data) into a single entity.
2. Access Modifiers in Python: Python doesn't have strict access modifiers like some other languages, but it does provide conventions for indicating the intended visibility of attributes and methods. There are three main types: a) Public Members:
   - Accessible from outside the class
   - No special syntax (just a regular name)
   - Example: `self.name`

   b) Protected Members:
   - Intended for use within the class and its subclasses
   - Prefixed with a single underscore `_`
   - Example: `self._name`

   c) Private Members:
   - Intended for use only within the class itself
   - Prefixed with double underscores `__`
   - Example: `self.__name`

example:

```python
class Person:
    def __init__(self, name, age, ssn):
        self.name = name        # Public
        self._age = age         # Protected
        self.__ssn = ssn        # Private

    def display_info(self):
        print(f"Name: {self.name}, Age: {self._age}")

    def _update_age(self, new_age):
        self._age = new_age

    def __get_ssn(self):
        return self.__ssn

class Employee(Person):
    def __init__(self, name, age, ssn, employee_id):
        super().__init__(name, age, ssn)
        self.employee_id = employee_id

    def display_employee_info(self):
        self.display_info()
        print(f"Employee ID: {self.employee_id}")
        # We can access protected members in subclass
        print(f"Age: {self._age}")
        # But we can't access private members of the parent class
        # print(f"SSN: {self.__ssn}")  # This would raise an AttributeError
```

Now, let's break down how encapsulation works in this example:

1. Public Members:
   - `self.name` and `self.employee_id` are public attributes.
   - They can be accessed and modified from anywhere.
2. Protected Members:
   - `self._age` and `self._update_age()` are protected.
   - They can be accessed within the class and its subclasses.
   - By convention, they shouldn't be accessed outside the class hierarchy, but Python doesn't enforce this.
3. Private Members:
   - `self.__ssn` and `self.__get_ssn()` are private.
   - They can only be accessed within the `Person` class.
   - Python uses name mangling for private members, changing `__ssn` to `_Person__ssn` internally.
4. Method Encapsulation:
   - `display_info()` and `display_employee_info()` are public methods that provide controlled access to the object's data.
   - They allow us to display information without directly accessing the attributes.
5. Inheritance and Encapsulation:
   - The `Employee` class inherits from `Person`, demonstrating how encapsulation works with inheritance.
   - It can access public and protected members of the parent class, but not private members.

Benefits of Encapsulation:

1. Data Hiding: It prevents direct access to attributes, protecting them from unintended changes.
2. Flexibility: You can change the internal implementation without affecting the external code that uses the class.
3. Modularity: It bundles data and methods together, making the code more organized and easier to maintain.
4. Abstraction: It allows you to expose only what's necessary, hiding the complex implementation details.

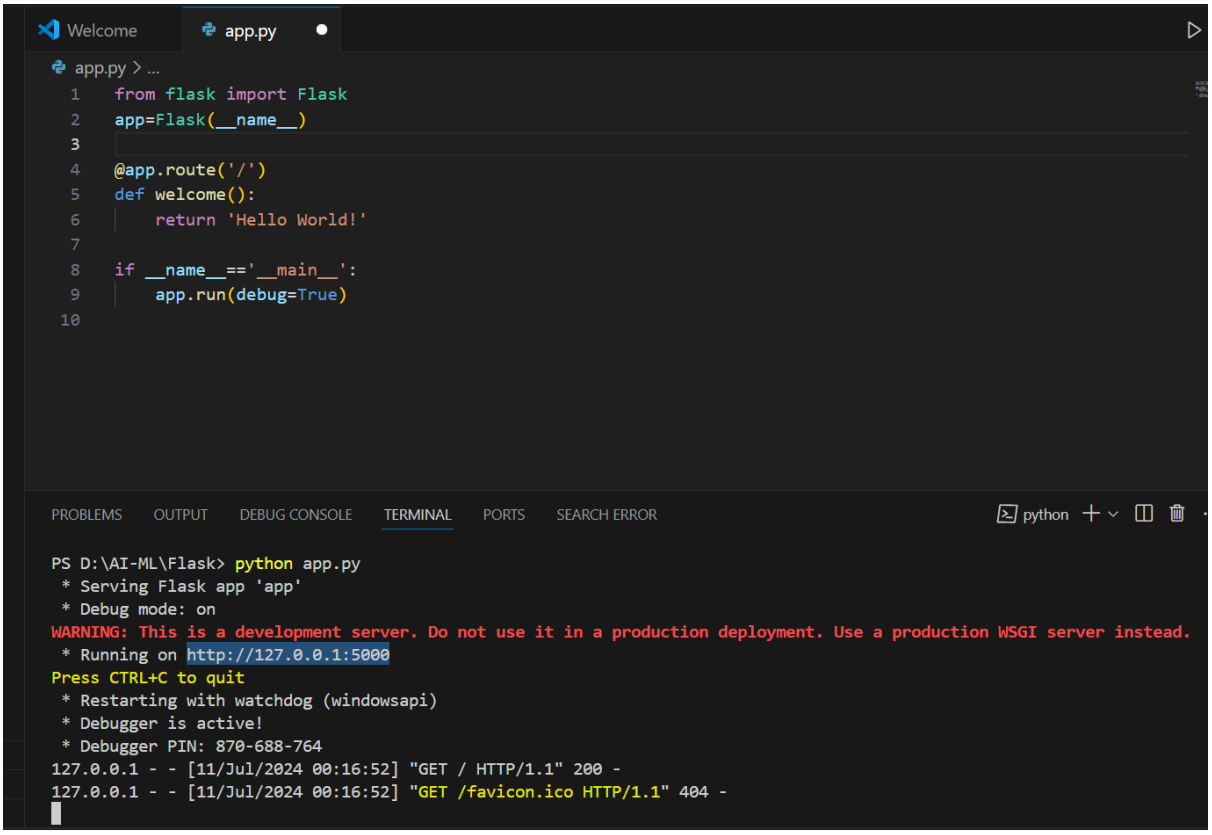
In practice, Python's encapsulation is more about convention than strict enforcement. Developers are expected to respect these conventions:

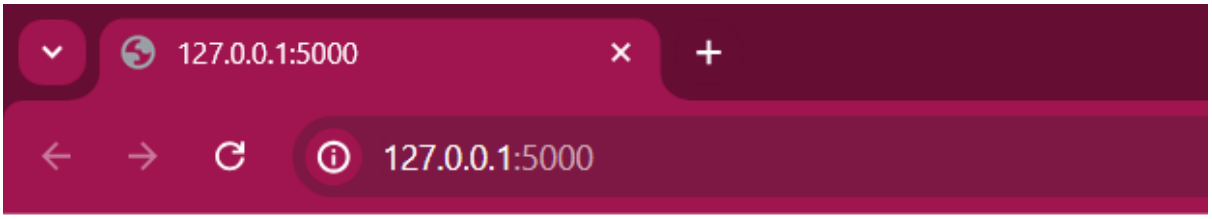
- Treat single underscore members as protected.
- Avoid accessing double underscore members from outside the class.
- Use public methods to interact with an object's data when appropriate.

AI/ML

TOPIC:	Flask	START DATE:	10/07/2024	↓

1.) Flask basics





Hello World!

2.) What is (if name==’__main__’:)?

Certainly, I'll provide an example to illustrate when the `if __name__ == '__main__':` condition is not triggered, and why that's useful. Let's create two Python files to demonstrate this.

First, let's create a file called `my_flask_app.py`:

python Copy

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, World!"

if __name__ == '__main__':
    print("The Flask app is running!")
    app.run()
```

Now, let's create another file called `use_flask_app.py`:

python Copy

```python
import my_flask_app

print("Imported my_flask_app")
print(f"The value of __name__ in my_flask_app is: {my_flask_app.__name__}")

# Use a function from my_flask_app
print(my_flask_app.hello())
```

When you run `use_flask_app.py`, here's what happens:

1. It imports `my_flask_app`.
2. The `if __name__ == '__main__':` condition in `my_flask_app.py` is not true, because `my_flask_app` is being imported, not run directly.
3. Therefore, `app.run()` is not called, and the Flask server doesn't start.
4. However, you can still use functions and objects defined in `my_flask_app`.

If you run `use_flask_app.py`, you'll see output like this:

Copy

```
Imported my_flask_app
The value of __name__ in my_flask_app is: my_flask_app
Hello, World!
```

Notice that "The Flask app is running!" is not printed, and the Flask server doesn't start. This is because `__name__` is set to `'my_flask_app'` when it's imported, not `'__main__'`.

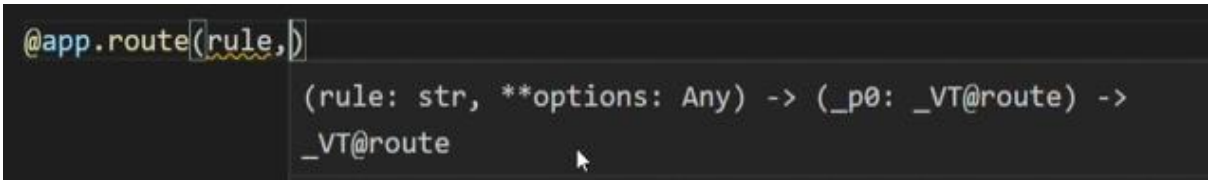
This approach allows you to:

1. Run your Flask app directly when you want to start the server.
2. Import and use parts of your Flask app in other scripts without automatically starting the server.

Is this example helpful in demonstrating when and why the `if __name__ == '__main__':` condition is used? Let me know if you need any clarification or have more questions!

It ensures that the py file will not execute when we are importing some other file.

3.) @app.route() Decorator



Yes, `@app.route()` is a decorator in Python.

In Python, a decorator is a special type of function that can modify or extend the behavior of another function. Decorators are often used to add additional functionality to a function without changing its underlying code.

In the case of `@app.route()`, it's a decorator that tells Flask to associate the function that follows it with a specific URL. When Flask receives a request for that URL, it will call the decorated function to handle the request.

The syntax `@app.route('/')` is called a "decorator syntax" and is used to apply the decorator to the function that follows it. The `@` symbol is used to indicate that the function that follows it is a decorator.

Example 3: A page that takes a parameter

```
from flask import Flask

app = Flask(__name__)

@app.route('/hello/<name>')
def hello(name):
    return f'Hello, {name}!'
```

In this example, when you visit `http://localhost:5000/hello/John` in your web browser, Flask will call the `hello` function and pass the string `'John'` as the `name` parameter. The function will then return a string that says `'Hello, John!'`.

These are just a few examples of what you can do with Flask and the `@app.route()` decorator. The possibilities are endless!

Example 2: A page that displays a list of items

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    items = ['apple', 'banana', 'orange']
    return render_template('index.html', items=items)
```

In this example, when you visit `http://localhost:5000/` in your web browser, Flask will render an HTML template called `index.html` and pass a list of items to the template. The template can then use this list to display a list of items.

Example 1: A simple "Hello, World!" page

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello, World!'
```

In this example, when you visit `http://localhost:5000/` in your web browser, you'll see the string `'Hello, World!'` displayed.

4.) Dynamic URLs (1. Variable rules and 2. URL Binding)

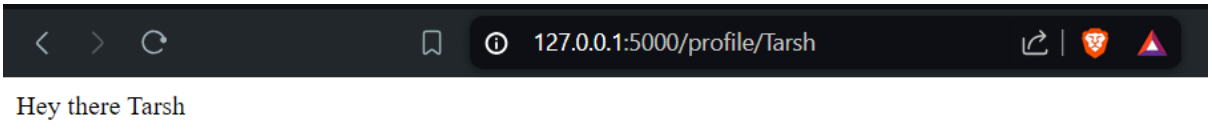
i.) Variable Rules

```
## Variable rules are used to capture the values specified at the URL and pass them to the view function as
arguments.
## The syntax for variable rules is <converter:variable_name>
## The converter is optional and can be string, int, float, path, uuid.
## The default converter is string.
## The variable_name is the name of the variable that the view function will receive.
## The variable_name should be a valid Python identifier.
## The converter and variable_name are separated by a colon.
## The variable_name is passed to the view function as an argument.
## The converter specifies the type of the variable.
## The converter is used to convert the variable to the specified type.
```

example 1:

dynamic-url.py > ...

```python
## Creating Dynamic URLs
## Variable rules and URL binding

from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, World!"


@app.route('/profile/<username>')
def profile(username):
    return "Hey there %s" % username

if __name__ == '__main__':
    app.run(debug=True)
```



example 2:

At the end, concatenation is to be done, so converted to string.

```
@app.route('/success/<int:score>')
def success(score):
    return "The Person has passed and the marks is "+ str(score)
```

example 3:

```
### Result checker
@app.route('/results/<int:score>')
def results(score):
    result=""
    if score<50:
        result='fail'
    else:
        result='success'
    return result
```

Now, when we want to display a different page, then we have to import 2 more libraries:

```python
## Creating Dynamic URLs
## Variable rules and URL binding

from flask import Flask, redirect, url_for

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, World!"

@app.route('/profile/<username>')
def profile(username):
    return "Hey there %s" % username

## Success and Fail routes
@app.route('/success/<int:score>')
def success(score):
    return "The person has passed and the score is %s" % score

@app.route('/fail/<int:score>')
def fail(score):
    return "The person has failed and the score is %s" % score

## Result Checker
@app.route('/result/<int:marks>')
def result(marks):
    result = ""
    if marks < 50:
        result = "fail"
    else:
        result = "success"
    return redirect(url_for(result, score=marks))

if __name__ == '__main__':
    app.run(debug=True)
```

5. Result URL (Pass scenario): Entered: `http://localhost:5000/result/80` What happens:
   - The server processes this request in the `result()` function.
   - It determines that 80 is greater than or equal to 50.
   - It redirects to the 'success' route. Result: The browser is redirected to `http://localhost:5000/success/80` and displays "The person has passed and the score is 80"
6. Result URL (Fail scenario): Entered: `http://localhost:5000/result/45` What happens:
   - The server processes this request in the `result()` function.
   - It determines that 45 is less than 50.
   - It redirects to the 'fail' route. Result: The browser is redirected to `http://localhost:5000/fail/45` and displays "The person has failed and the score is 45"

We can also use HTML,

```
return "<html><body><h1>The Reult is passed</h1></body></html>"
```

5.) Integrating HTML with Flask (JINJA-2 technique)

render_template, request, url_for, redirect are used,

main.py:

```
### Integrate HTML With Flask
### HTTP verb GET And POST

##Jinja2 template engine
'''
{%...%} conditions,for loops
{{    }} expressions to print output
{#....#} this is for comments
'''
from flask import Flask,redirect,url_for,render_template,request

app=Flask(__name__)

@app.route('/')
def welcome():
    return render_template('index.html')

@app.route('/success/<int:score>')
def success(score):
    res=""
    if score>=50:
        res="PASS"
    else:
        res='FAIL'
    exp={'score':score,'res':res}
    return render_template('result.html',result=exp)


@app.route('/fail/<int:score>')
def fail(score):
    return "The Person has failed and the marks is "+ str(score)

### Result checker
@app.route('/results/<int:marks>')
def results(marks):
    result=""
    if marks<50:
        result='fail'
    else:
        result='success'
    return redirect(url_for(result,score=marks))

### Result checker submit html page
@app.route('/submit',methods=['POST','GET'])
def submit():
    total_score=0
    if request.method=='POST':
        science=float(request.form['science'])
        maths=float(request.form['maths'])
        c=float(request.form['c'])
```

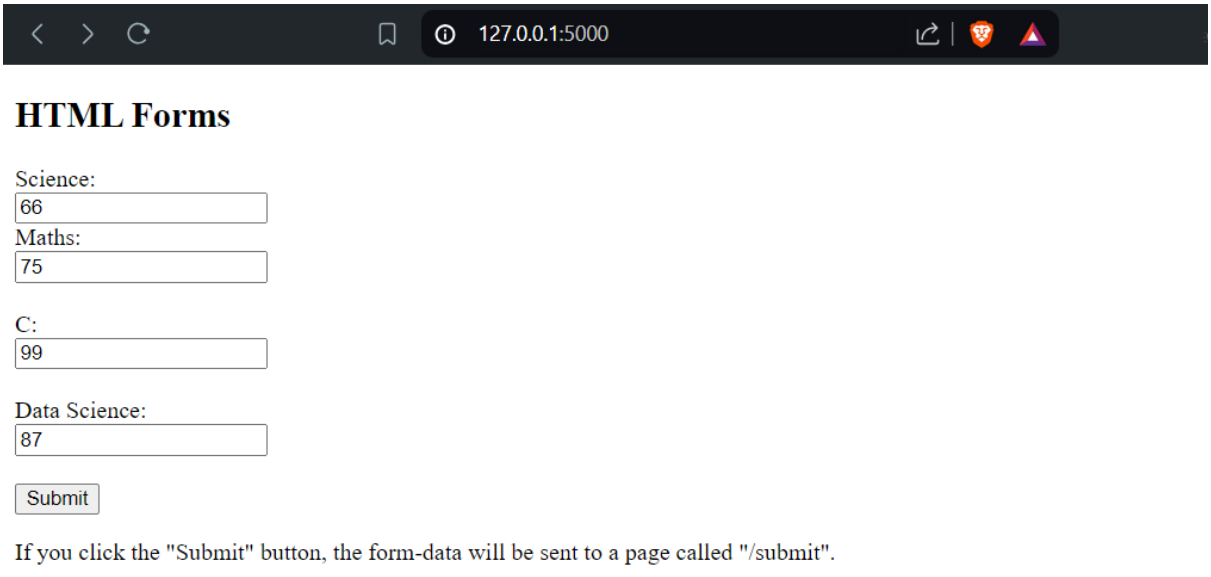
index.html:

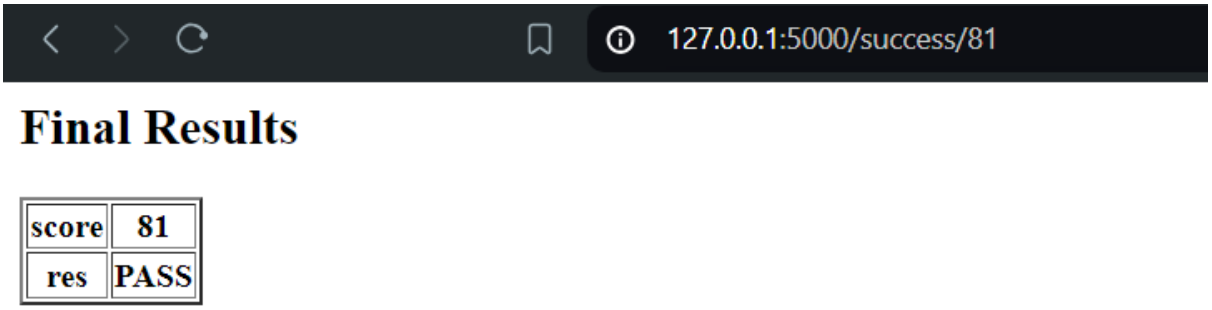
```
<!DOCTYPE html>
<html>

<head>
    <link rel="stylesheet" href="{{ url_for('static',filename='css/style.css') }}">
    <script type="text/javascript" src="{{ url_for('static',filename='script/script.js') }}">

    </script>

</head>

<body>

    <h2>HTML Forms</h2>

    <form action="/submit" method='post'>
        <label for="Science">Science:</label><br>
        <input type="text" id="science" name="science" value="0"><br>
        <label for="Maths">Maths:</label><br>
        <input type="text" id="maths" name="maths" value="0"><br><br>
        <label for="C ">C:</label><br>
        <input type="text" id="c" name="c" value="0"><br><br>
        <label for="datascience">Data Science:</label><br>
        <input type="text" id="datascience" name="datascience" value="0"><br><br>
        <input type="submit" value="Submit">
    </form>

    <p>If you click the "Submit" button, the form-data will be sent to a page called "/submit".</p>

</body>

</html>
```

result.html:

```
<html>

<h2>Final Results</h2>

<body>
<table border=2>
 {% for key,value in result.items() %}
    <tr>
        {# This is the comment sections #}
        <th>{{ key }}</th>
        <th>{{ value }}</th>

    </tr>
  {% endfor %}
</table>
</body>

</html>
```

o/p:

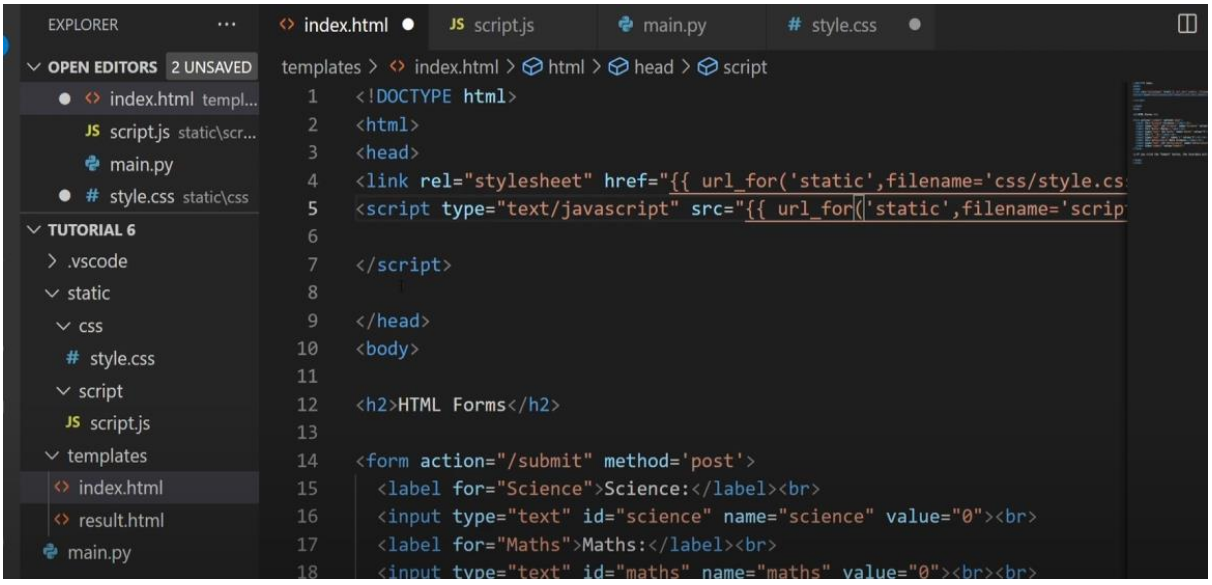




explanation:

- `{% ... %}` : These are used for control structures like conditions and loops in Jinja2 templates.
- `{{ ... }}` : These are used for expressions to print output in Jinja2 templates.
- `{# ... #}` : These are used for comments in Jinja2 templates.

6.) Integrating CSS and JS with Flask



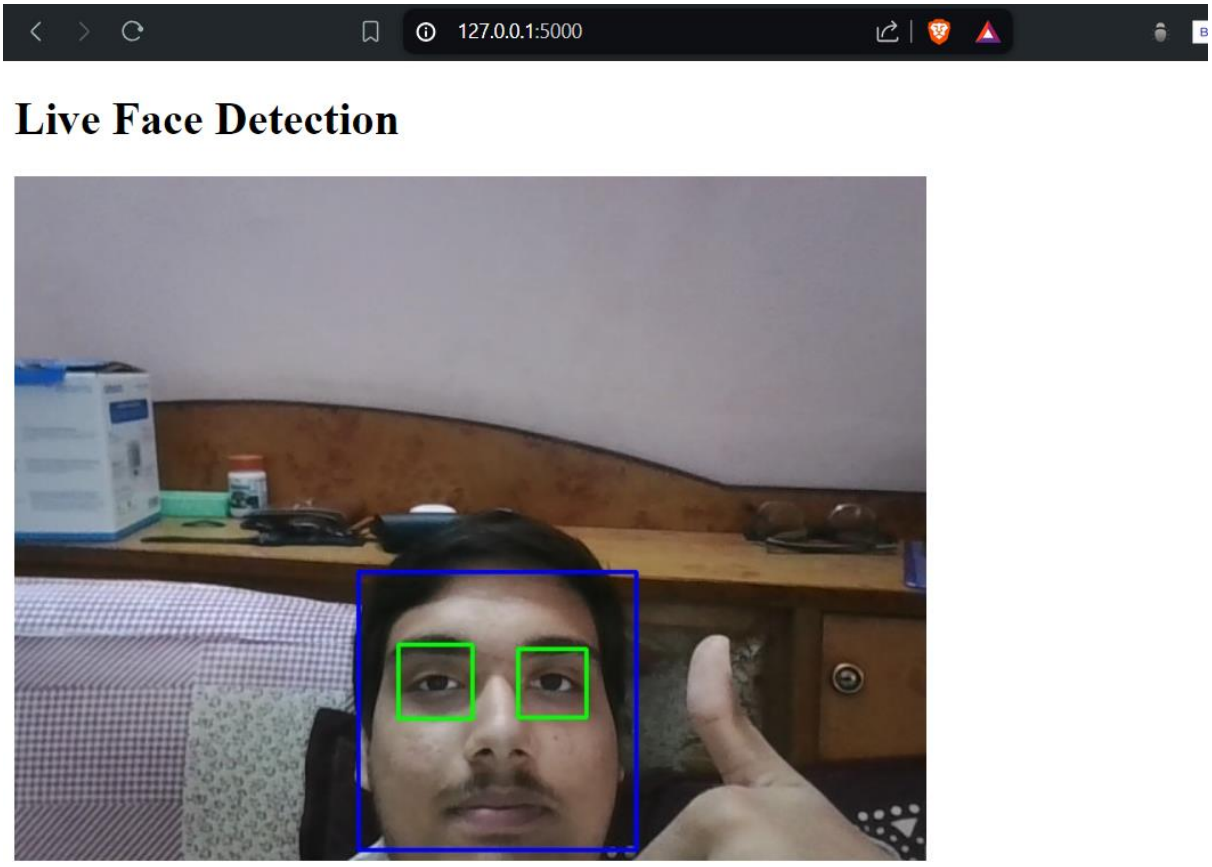
7.) Video Streaming using Webcam

app.py:

```python
import os
from flask import Flask, render_template, Response
import cv2

app = Flask(__name__)
camera = cv2.VideoCapture(0)

# Get the absolute path to the Haarcascades directory
haarcascades_dir = os.path.join(os.path.dirname(cv2.__file__), 'data')

# Initialize detectors
face_cascade_path = os.path.join(haarcascades_dir, 'haarcascade_frontalface_default.xml')
eye_cascade_path = os.path.join(haarcascades_dir, 'haarcascade_eye.xml')

face_detector = cv2.CascadeClassifier(face_cascade_path)
eye_detector = cv2.CascadeClassifier(eye_cascade_path)

def gen_frames():
    if face_detector.empty():
        raise ValueError("Failed to load face cascade classifier")
    if eye_detector.empty():
        raise ValueError("Failed to load eye cascade classifier")

    while True:
        success, frame = camera.read()  # read the camera frame
        if not success:
            break
        else:
            try:
                gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
                faces = face_detector.detectMultiScale(gray, 1.1, 7)

                # Draw the rectangle around each face
                for (x, y, w, h) in faces:
                    cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
                    roi_gray = gray[y:y+h, x:x+w]
                    roi_color = frame[y:y+h, x:x+w]
                    eyes = eye_detector.detectMultiScale(roi_gray, 1.1, 3)
                    for (ex, ey, ew, eh) in eyes:
                        cv2.rectangle(roi_color, (ex, ey), (ex+ew, ey+eh), (0, 255, 0), 2)

                ret, buffer = cv2.imencode('.jpg', frame)
                frame = buffer.tobytes()
                yield (b'--frame\r\n'
                       b'Content-Type: image/jpeg\r\n\r\n' + frame + b'\r\n')
            except Exception as e:
                print(f"Error in face detection: {str(e)}")
                # You might want to yield a default frame or error message here

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/video_feed')
def video_feed():
    return Response(gen_frames(), mimetype='multipart/x-mixed-replace; boundary=frame')

if __name__=='__main__':
    app.run(debug=True)
```

index.html:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Face Detection</title>
</head>
<body>
    <h1>Live Face Detection</h1>
    <img src="{{ url_for('video_feed') }}" width="640" height="480">
</body>
</html>
```



AI/ML

TOPIC:	Statistics	START DATE:	13/07/2024	↓

1.) Basics

example 1:

+ Code

```
[2]  import numpy as np
     import matplotlib.pyplot as plt
     %matplotlib inline
```

0s

```
[5]  import statistics
     import seaborn as sns
```

1s

```
[6]  # mean, median, mode
     df = sns.load_dataset('tips')
```

0s

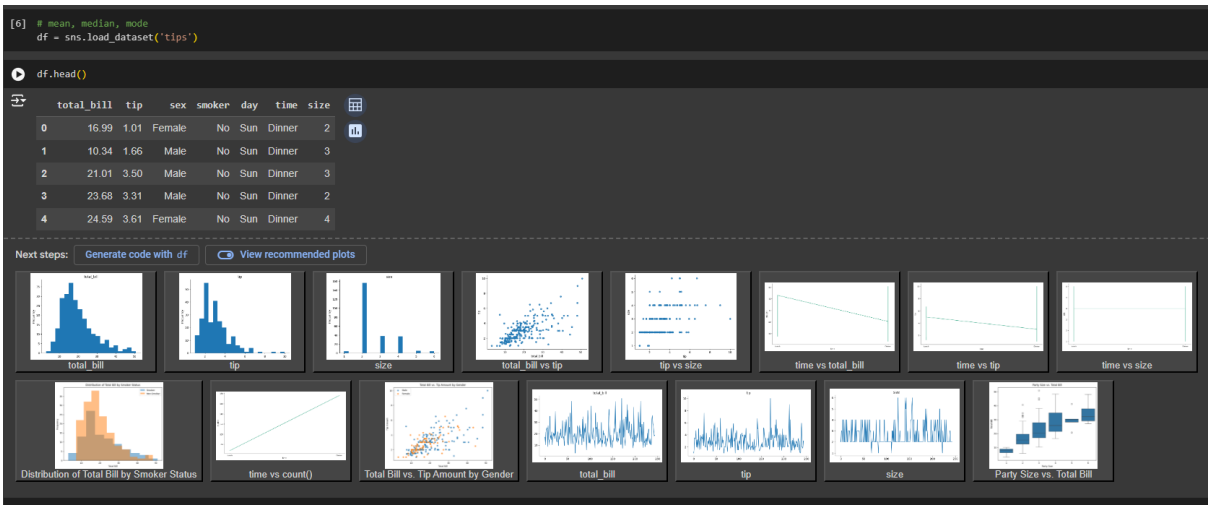
```
[20] df.head()
```

0s

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Next steps: Generate code with df | View recommended plots

0s completed at 12:06



```
[8]  np.mean(df['total_bill'])
```

```
19.78594262295082
```

```
[9]  np.median(df['total_bill'])
```

```
17.795
```

```
[13] np.std(df['total_bill'])
```

```
8.884150577771132
```

```
[14] statistics.mode(df['total_bill'])
```

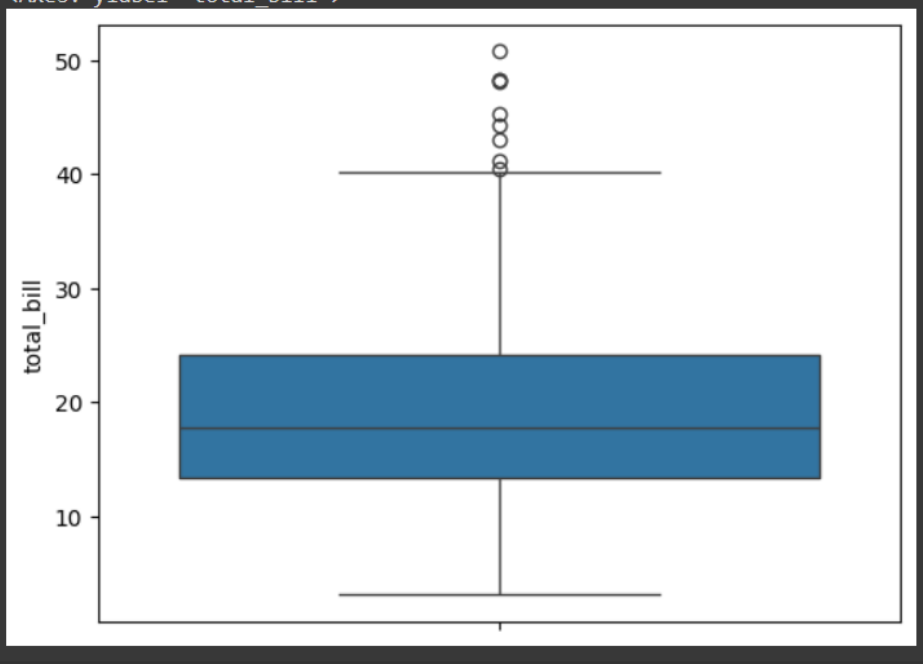
```
13.42
```

```
np.percentile(df['total_bill'], [25, 50, 75])
```

```
array([23.56143318, 30.94096119, 37.37077353])
```

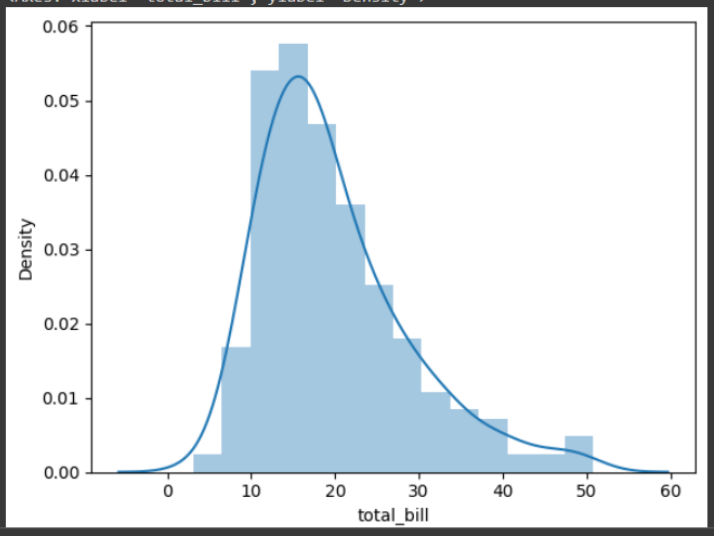
```
sns.boxplot(df['total_bill'])
```

```
<Axes: ylabel='total_bill'>
```



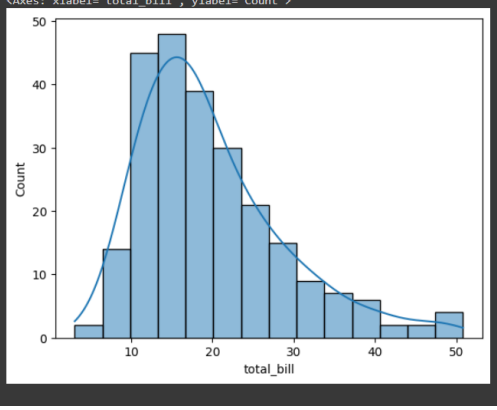

```
sns.distplot(df['total_bill'])
```

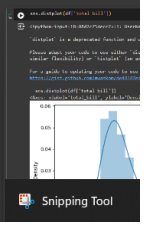
```
<ipython-input-18-08d2c75decc2>:1: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(df['total_bill'])
<Axes: xlabel='total_bill', ylabel='Density'>
```



```
sns.histplot(df['total_bill'], kde=True)

# The histogram is a visual representation of the distribution of the total_bill values for the entire dataset, which includes more than the five values shown in the table.
```

```
<Axes: xlabel='total_bill', ylabel='Count'>
```





example 2:

```
[23] # @title Default title text
     df1=sns.load_dataset('iris')
```

Start coding or generate with AI.

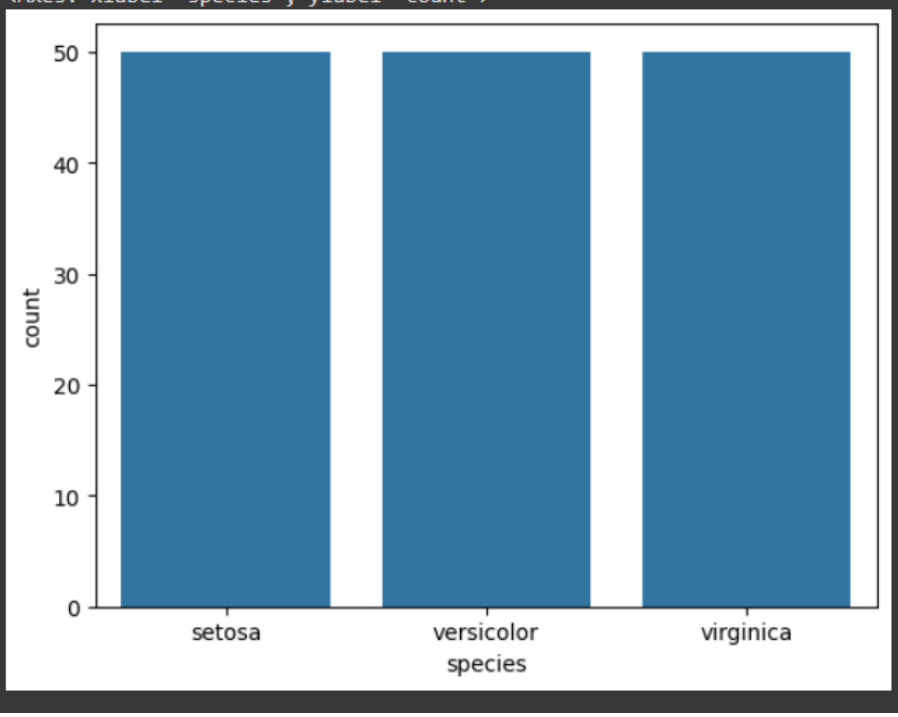
Double-click (or enter) to edit

```
[24] df1.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
sns.countplot(x='species', data=df1)
```

```
<Axes: xlabel='species', ylabel='count'>
```



2.) Finding Outliers using Z Score

```
[2] ## Define our dataset
    dataset= [11,10,12,14,12,15,14,13,15,102,12,14,17,19,107, 10,13,12,14,12,108,12,11,14,13,15,10,15,12,10,14,13,15,10]
```

We are considering that any data **outside the 3 Standard Deviations** will be considered as outliers.

```
[4] ## Z Score
outliers=[]

def detect_outliers(data):
  threshold=3 ## 3 std deviation
  mean=np.mean(data)
  std=np.std(data)

  for i in data:
    z_score=(i-mean)/std
    if np.abs(z_score)> threshold:
      outliers.append(i)

  return outliers
```

0s

```
detect_outliers(dataset)
```

```
[102, 107, 108]
```

3.) Hypothesis Testing

In hypothesis testing, a Type I error occurs when a null hypothesis (H0) is rejected when it is actually true. In other words, you conclude that there is a significant effect or relationship when, in fact, there isn't one.

A false positive is a specific type of Type I error that occurs when a test or experiment incorrectly detects a statistically significant effect or relationship when none exists. This can happen due to chance, sampling variability, or other factors.

For example, imagine conducting a study to test whether a new medication is effective in reducing blood pressure. If the study finds a statistically significant effect, but in reality, the effect is due to chance or other factors, then the result is a false positive (FP) and a Type I error.

On the other hand, Type II errors (β) occur when a null hypothesis is not rejected when it is actually false. In other words, you fail to detect a statistically significant effect or relationship when one exists.

So, to summarize: FP is a type of Type I error, which occurs when a null hypothesis is rejected when it is actually true.

In the context of hypothesis testing and statistical significance, True Positives (TP) and True Negatives (TN) are indeed always true... but only when the null hypothesis is correctly rejected or accepted, respectively.

Here's what I mean:

- True Positive (TP): When the null hypothesis is correctly rejected (i.e., the alternative hypothesis is accepted), and the effect or relationship is real.
- True Negative (TN): When the null hypothesis is correctly accepted (i.e., the alternative hypothesis is rejected), and there is no effect or relationship.

In other words, TP and TN are only "true" when the test or experiment accurately reflects the underlying reality. If the null hypothesis is incorrectly rejected (Type I error) or accepted (Type II error), then the result is not a true positive or true negative.

example:

suppose alpha=0.05 and consider two cases of p values,

The decision to accept or reject the null hypothesis depends on the p-value in relation to the significance level (alpha).

1. **Case 1:** $p = 0.11$
   - p is greater than α (0.11 > 0.05).
   - Therefore, we **fail to reject the null hypothesis**.
2. **Case 2:** $p = 0.002$
   - p is less than α (0.002 < 0.05).
   - Therefore, we **reject the null hypothesis**.

In summary:

- For $p = 0.11$: Fail to reject the null hypothesis.
- For $p = 0.002$: Reject the null hypothesis.

In other words, if the p-value is:

- Less than the significance level (e.g., α = 0.05), you reject the null hypothesis and conclude that the observed effect is statistically significant.
- Greater than the significance level (e.g., α = 0.05), you fail to reject the null hypothesis and conclude that the observed effect is not statistically significant.

4.) When to use which test?

```
Here's a breakdown of when to use each of these tests:

**Z-Test**

A Z-test is used to compare a sample mean to a known population mean, or to compare two sample means. It's commonly
used in situations where:

1. You want to determine if a sample mean is significantly different from a known population mean.
2. You want to compare the means of two groups to determine if there's a significant difference between them.
3. You have a large sample size and want to test a hypothesis about the population mean.

Conditions for using a Z-test:

* The sample size is large enough to assume normality (usually n > 30).
* The population standard deviation is known or can be estimated from a large sample.
* The data is continuous and normally distributed.

**T-Test**

A T-test is used to compare the means of two groups, but it's more flexible than the Z-test and can be used in
situations where:

1. The sample size is small or the population standard deviation is unknown.
2. The data is not normally distributed, but the sample size is large enough to assume normality.
3. You want to compare the means of two groups, but the data is not continuous (e.g., categorical data).

Conditions for using a T-test:

* The sample size is small or the population standard deviation is unknown.
* The data is not normally distributed, but the sample size is large enough to assume normality.
* The data is continuous or categorical.

**Chi-Square Test**

A Chi-Square test is used to determine whether there's a significant association between two categorical variables.
It's commonly used in situations where:

1. You want to determine if there's an association between two categorical variables.
2. You want to test the independence of two categorical variables.
3. You want to compare the distribution of a categorical variable across different groups.

Conditions for using a Chi-Square test:

* The data is categorical (nominal or ordinal).
* The sample size is large enough to assume normality (usually n > 30).
* The expected frequency of each cell is at least 5.

In summary:

* Use a Z-test when you have a large sample size, the population standard deviation is known, and the data is
normally distributed.
* Use a T-test when the sample size is small or the population standard deviation is unknown, or when the data is
not normally distributed but the sample size is large enough to assume normality.
* Use a Chi-Square test when you want to determine the association between two categorical variables, and the sample
size is large enough to assume normality.
```

5.) Example

https://github.com/krishnaik06/The-Grand-Complete-Data-Science-Materials/blob/main/Complete%20Statistics/T-test-an-Correlation-using-python-master.zip

example 1:

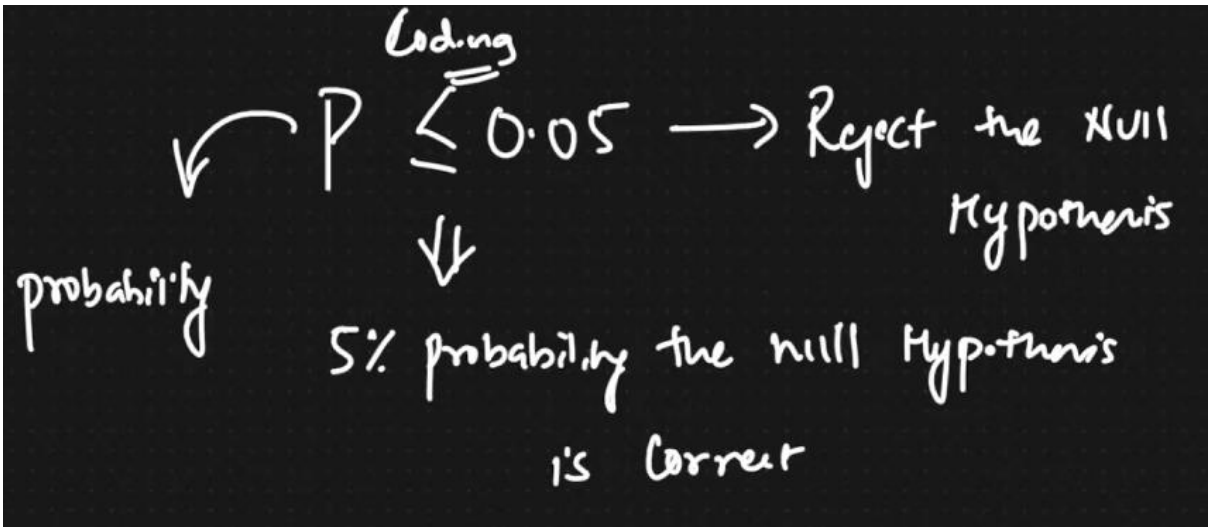
```
Suppose the IQ in a certain population is normally distributed with a mean of μ = 100 and standard
deviation of σ = 15.

A researcher wants to know if a new drug affects IQ levels, so he recruits 20 patients to try it and
records their IQ levels.

The following code shows how to perform a one sample z-test in Python to determine if the new drug
causes a significant difference in IQ levels:
```

the first value is the z-test value and the other is used for comparing (p-value),

```
from statsmodels.stats.weightstats import ztest as ztest

#enter IQ levels for 20 patients
data = [88, 92, 94, 94, 96, 97, 97, 97, 99, 99,
        105, 109, 109, 109, 110, 112, 112, 113, 114, 115]

ztest(data,value=100)
```

```
(1.5976240527147705, 0.1101266701438426)
```

0.11>0.05 so accept the H0 (null hypothesis),



5.) t-test

In [342]:
```
ages=[10,20,35,50,28,40,55,18,16,55,30,25,43,18,30,28,14,24,16,17,32,35,26,27,65,18,43,23,21,20,19,70]
```

In [343]:
```
len(ages)
```

Out[343]:
```
32
```

In [41]:
```
import numpy as np
ages_mean=np.mean(ages)
print(ages_mean)
```

```
30.34375
```

In [344]:
```
## Lets take sample

sample_size=10
age_sample=np.random.choice(ages,sample_size)
```

In [345]:
```
age_sample
```

Out[345]:
```
array([18, 10, 24, 10, 35, 55, 10, 70, 18, 28])
```

In [346]:
```
from scipy.stats import ttest_1samp
```

In [347]:
```
ttest,p_value=ttest_1samp(age_sample,30)
```

In [348]:
```
print(p_value)
```

```
0.7403808450296021
```

In [118]:
```
if p_value < 0.05:    # alpha value is 0.05 or 5%
    print(" we are rejecting null hypothesis")
else:
    print("we are accepting null hypothesis")
```

```
we are accepting null hypothesis
```

Two-sample T-test With Python

The Independent Samples t Test or 2-sample t-test compares the means of two independent groups in order to determine whether there is statistical evidence that the associated population means are significantly different. The Independent Samples t Test is a parametric test. This test is also known as: Independent t Test

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{s^2\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}}$$

$$s^2 = \frac{\sum_{i=1}^{n_1}(x_i - \bar{x}_1)^2 + \sum_{j=1}^{n_2}(x_j - \bar{x}_2)^2}{n_1 + n_2 - 2}$$

In [355]:
```
np.random.seed(12)
ClassB_ages=stats.poisson.rvs(loc=18,mu=33,size=60)
ClassB_ages.mean()
```

Out[355]:
```
50.63333333333333
```

In [356]:
```
_,p_value=stats.ttest_ind(a=classA_height,b=ClassB_ages,equal_var=False)
```

In [357]:
```
if p_value < 0.05:    # alpha value is 0.05 or 5%
    print(" we are rejecting null hypothesis")
else:
    print("we are accepting null hypothesis")
```

```
 we are rejecting null hypothesis
```

6.) Correlation

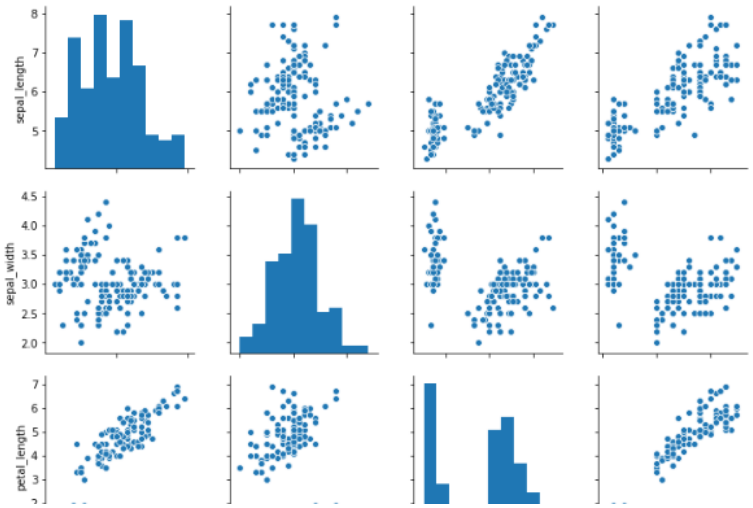
Correlation

```
In [323]: import seaborn as sns
          df=sns.load_dataset('iris')

In [326]: df.shape
Out[326]: (150, 5)

In [341]: df.corr()
```

Out[341]:

	sepal_length	sepal_width	petal_length	petal_width
sepal_length	1.000000	-0.117570	0.871754	0.817941
sepal_width	-0.117570	1.000000	-0.428440	-0.366126
petal_length	0.871754	-0.428440	1.000000	0.962865
petal_width	0.817941	-0.366126	0.962865	1.000000

```
In [328]: sns.pairplot(df)
Out[328]: <seaborn.axisgrid.PairGrid at 0x227f348ec88>
```



7.) PDF vs PMF (extra)

A **PMF** is a function that describes the probability distribution of a **discrete random variable**. It assigns a probability value to each possible value that the random variable can take. The PMF is typically denoted as P(X = x) or p(x), where X is the random variable and x is a possible value.

1. Discrete: PMFs are used for discrete random variables, which can only take on a finite or countable number of values.

2. Probability values: PMFs assign a probability value to each possible value of the random variable.

3. Sum to 1: The probabilities in a PMF sum to 1, ensuring that the total probability is 100%.

Examples of PMFs include:

* Bernoulli distribution (e.g., coin toss)

* Binomial distribution (e.g., number of heads in n coin tosses)

* Poisson distribution (e.g., number of defects in a manufacturing process)

A **PDF** is a function that describes the probability distribution of a **continuous random variable**. It assigns a probability value to each possible value within a continuous range. The PDF is typically denoted as f(x), where x is a value within the range of the random variable.

Key characteristics of a PDF:

1. Continuous: PDFs are used for continuous random variables, which can take on any value within a specified range.

2. Density: PDFs describe the density of the probability distribution, rather than assigning a probability value to each specific value.

3. Integrates to 1: The area under a PDF curve integrates to 1, ensuring that the total probability is 100%.

Examples of PDFs include:

* Normal distribution (e.g., height of a person)
* Uniform distribution (e.g., temperature in a room)
* Exponential distribution (e.g., time between events)

Key differences:

1. Discrete vs. Continuous: PMFs are used for discrete random variables, while PDFs are used for continuous random variables.
2. Probability values: PMFs assign a probability value to each specific value, while PDFs describe the density of the probability distribution.
3. Mathematical representation: PMFs are typically represented as a sum of probabilities, while PDFs are represented as an integral.

In summary, PMFs are used for discrete random variables, assigning a probability value to each specific value, while PDFs are used for continuous random variables, describing the density of the probability distribution.

AI/ML

TOPIC:	EDA and Feature Engineering	START DATE:	22/07/2024	↓

1.) Basics

https://github.com/krishnaik06/5-Days-Live-EDA-and-Feature-Engineering/blob/main/1-EDA%20And%20Feature%20Engineering.ipynb

Functions:

1. `pd.read_excel()`: to read an Excel file into a Pandas DataFrame
2. `pd.merge()`: to merge two DataFrames based on a common column
3. `groupby()`: to group data by one or more columns and perform aggregation operations
4. `size()`: to calculate the size of each group
5. `reset_index()`: to reset the index of a DataFrame
6. `value_counts()`: to count the unique values in a column
7. `head()`: to display the first few rows of a DataFrame
8. `tail()`: to display the last few rows of a DataFrame
9. `shape`: to get the shape of a DataFrame (number of rows and columns)
10. `isnull()`: to check for missing values in a DataFrame
11. `sum()`: to calculate the sum of missing values in a DataFrame
12. `pie()`: to create a pie chart
13. `barplot()`: to create a bar plot
14. `countplot()`: to create a count plot

Methods:

1. `df.columns`: to access the column names of a DataFrame
2. `df.index`: to access the index of a DataFrame
3. `df.head()`: to display the first few rows of a DataFrame
4. `df.tail()`: to display the last few rows of a DataFrame
5. `df.groupby()`: to group data by one or more columns and perform aggregation operations
6. `df.size()`: to calculate the size of each group
7. `df.reset_index()`: to reset the index of a DataFrame
8. `df.value_counts()`: to count the unique values in a column
9. `df.plot()`: to create a plot of a DataFrame
10. `plt.show()`: to display a plot

df.describe():

memory usage: 1.54 MB

```
In [8]: df.describe()
```

Out[8]:

	Restaurant ID	Country Code	Longitude	Latitude	Average Cost for two	Price range	Aggregate rating	Votes
count	9.551000e+03	9551.000000	9551.000000	9551.000000	9551.000000	9551.000000	9551.000000	9551.000000
mean	9.051128e+06	18.365616	64.126574	25.854381	1199.210763	1.804837	2.666370	156.909748
std	8.791521e+06	56.750546	41.467058	11.007935	16121.183073	0.905609	1.516378	430.169145
min	5.300000e+01	1.000000	-157.948486	-41.330428	0.000000	1.000000	0.000000	0.000000
25%	3.019625e+05	1.000000	77.081343	28.478713	250.000000	1.000000	2.500000	5.000000
50%	6.004089e+06	1.000000	77.191964	28.570469	400.000000	2.000000	3.200000	31.000000
75%	1.835229e+07	1.000000	77.282006	28.642758	700.000000	2.000000	3.700000	131.000000
max	1.850065e+07	216.000000	174.832089	55.976980	800000.000000	4.000000	4.900000	10934.000000

The output of `df.describe()` for your dataset is a summary of the central tendency, dispersion, and shape of the dataset's distribution. Here's a breakdown of what each column represents:

1. `count`: The number of non-null values in each column.
2. `mean`: The mean or average value of each column.
3. `std`: The standard deviation of each column, which measures the spread or dispersion of the values.
4. `min`: The minimum value in each column.
5. `25%`: The 25th percentile or first quartile of each column.
6. `50%`: The 50th percentile or median of each column.
7. `75%`: The 75th percentile or third quartile of each column.
8. `max`: The maximum value in each column.

example:

In [14]:
```
df.columns
```

Out[14]:
```
Index(['Restaurant ID', 'Restaurant Name', 'Country Code', 'City', 'Address',
       'Locality', 'Locality Verbose', 'Longitude', 'Latitude', 'Cuisines',
       'Average Cost for two', 'Currency', 'Has Table booking',
       'Has Online delivery', 'Is delivering now', 'Switch to order menu',
       'Price range', 'Aggregate rating', 'Rating color', 'Rating text',
       'Votes'],
      dtype='object')
```

In [16]:
```
final_df=pd.merge(df,df_country,on='Country Code', how='left')
```

In [17]:
```
final_df.head(2)
```

Out[17]:

	Restaurant ID	Restaurant Name	Country Code	City	Address	Locality	Locality Verbose	Longitude	Latitude	Cuisines	...	Has Table booking	Has Online delivery	del
0	6317637	Le Petit Souffle	162	Makati City	Third Floor, Century City Mall, Kalayaan Avenu...	Century City Mall, Poblacion, Makati City	Century City Mall, Poblacion, Makati City, Mak...	121.027535	14.565443	French, Japanese, Desserts	...	Yes	No	
1	6304287	Izakaya Kikufuji	162	Makati City	Little Tokyo, 2277 Chino Roces Avenue, Legaspi	Little Tokyo, Legaspi Village, Makati City	Little Tokyo, Legaspi Village, Makati City, Ma...	121.014101	14.553708	Japanese	...	Yes	No	

It looks like you've merged two DataFrames: `df` and `df_country` using the 'Country Code' as the key. This operation creates a new DataFrame called `final_df`. Let's break down what's happening:

1. The first code snippet shows the columns of the original `df` DataFrame. It has information about restaurants, including their ID, name, location details, cuisines, ratings, etc.
2. The second snippet performs a left join between `df` and `df_country` DataFrames:
   - `pd.merge()` is used to combine the DataFrames
   - `on='Country Code'` specifies that the 'Country Code' column is used as the key for merging
   - `how='left'` indicates it's a left join, meaning all rows from `df` will be kept, and matching rows from `df_country` will be added

This merge operation likely adds country-specific information (probably the country name and possibly other details) to your restaurant data.

The `merge` function is doing the following:

1. It's matching rows from `df` with rows from `df_country` based on the 'Country Code' column.
2. For each row in `df`, it's finding the corresponding row in `df_country` that has the same 'Country Code' value.
3. It's then combining the columns from both dataframes into a new dataframe, `final_df`.
4. If there's a match, it's combining the columns from both dataframes. If there's no match, it's adding a row to `final_df` with null values for the columns from `df_country`.

TS `how='left'` indicates it's a left join, meaning all rows from `df` will be kept, and matching rows from `df_country` will be added

and what if df has more values than df_country?

You're asking an excellent question about the behavior of left joins when the left DataFrame (in this case, `df`) has more values in the join column than the right DataFrame (`df_country`). Let's clarify what happens in this scenario:

1. All rows from `df` will be kept in the result, regardless of whether there's a match in `df_country`.
2. For rows in `df` where there's no matching 'Country Code' in `df_country`:
   - The columns from `df` will contain their original values.
   - The columns that would have come from `df_country` will be filled with NaN (Not a Number) or None values, depending on the data type.
3. This means that if `df` has 'Country Code' values that don't exist in `df_country`, those rows will still appear in `final_df`, but with NaN/None values for the columns that would have come from `df_country`.

python Copy

```python
df = pd.DataFrame({
    'Country Code': ['US', 'UK', 'FR', 'XY'],  # XY doesn't exist in df_country
    'Restaurant Name': ['A', 'B', 'C', 'D']
})

df_country = pd.DataFrame({
    'Country Code': ['US', 'UK', 'FR'],
    'Country Name': ['United States', 'United Kingdom', 'France']
})

final_df = pd.merge(df, df_country, on='Country Code', how='left')
```

The resulting `final_df` would look like this:

Copy

```
  Country Code Restaurant Name        Country Name
0           US               A       United States
1           UK               B      United Kingdom
2           FR               C              France
3           XY               D                 NaN
```

example: (Plotting pie chart)

```
In [17]: country_names=final_df.Country.value_counts().index
```

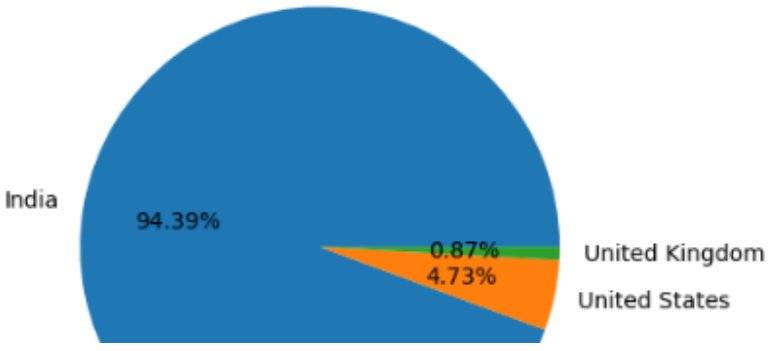
```
In [30]: country_val=final_df.Country.value_counts().values
```

```
In [32]: final_df.Country.value_counts()
```

```
Out[32]: India             8652
         United States      434
         United Kingdom      80
         Brazil              60
         UAE                 60
         South Africa        60
         New Zealand         40
         Turkey              34
         Australia           24
         Phillipines         22
         Indonesia           21
         Singapore           20
         Qatar               20
         Sri Lanka           20
         Canada               4
         Name: Country, dtype: int64
```

```
In [19]: ## Pie Chart- Top 3 countries that uses zomato
         plt.pie(country_val[:3],labels=country_names[:3],autopct='%1.2f%%')
```

```
Out[19]: ([<matplotlib.patches.Wedge at 0x2978e6cbc50>,
           <matplotlib.patches.Wedge at 0x2978e65d850>,
           <matplotlib.patches.Wedge at 0x2978e5f7610>],
          [Text(-1.0829742700952103, 0.19278674827836725, 'India'),
           Text(1.077281715838356, -0.22240527134123297, 'United States'),
           Text(1.0995865153823035, -0.03015783794312073, 'United Kingdom')],
          [Text(-0.590713238233751, 0.10515640815183668, '94.39%'),
           Text(0.5876082086391032, -0.12131196618612707, '4.73%'),
           Text(0.5997744629358018, -0.01644972978715676, '0.87%')])
```



example:

In [21]:
```
ratings=final_df.groupby(['Aggregate rating','Rating color','Rating text']).size().reset_index().rename(columns={0:'Rating Count'
```

In [41]:
```
ratings
```

Out[41]:

	Aggregate rating	Rating color	Rating text	Rating Count
0	0.0	White	Not rated	2148
1	1.8	Red	Poor	1
2	1.9	Red	Poor	2
3	2.0	Red	Poor	7
4	2.1	Red	Poor	15
5	2.2	Red	Poor	27
6	2.3	Red	Poor	47
7	2.4	Red	Poor	87
8	2.5	Orange	Average	110
9	2.6	Orange	Average	191
10	2.7	Orange	Average	250

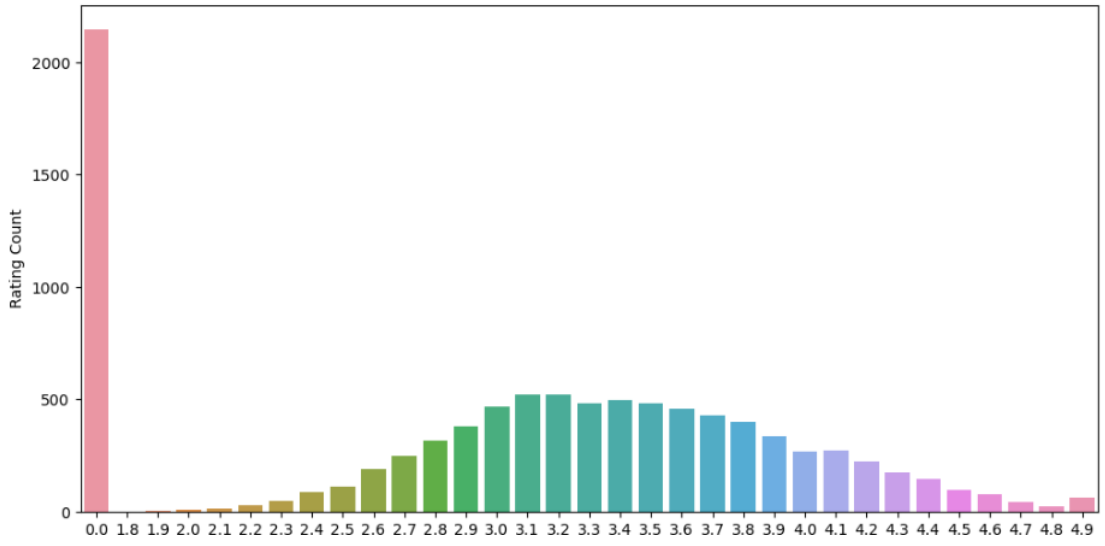
In [43]:
```
ratings.head()
```

Out[43]:

	Aggregate rating	Rating color	Rating text	Rating Count
0	0.0	White	Not rated	2148
1	1.8	Red	Poor	1
2	1.9	Red	Poor	2
3	2.0	Red	Poor	7
4	2.1	Red	Poor	15

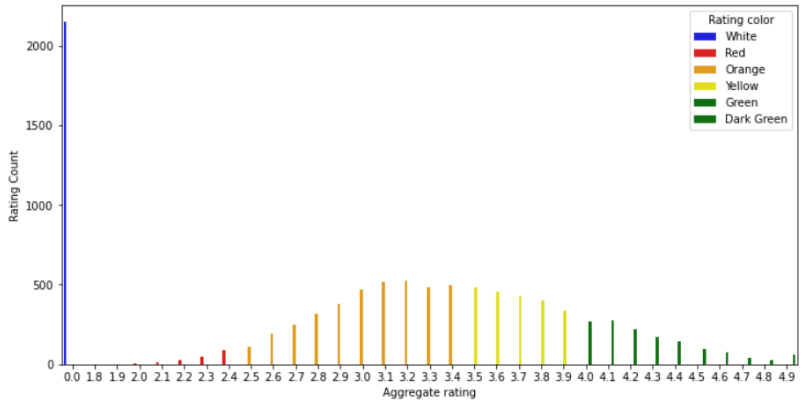
In [39]:
```
import matplotlib
matplotlib.rcParams['figure.figsize'] = (12, 6)
sns.barplot(x="Aggregate rating",y="Rating Count",data=ratings)
```

Out[39]:
```
<Axes: xlabel='Aggregate rating', ylabel='Rating Count'>
```



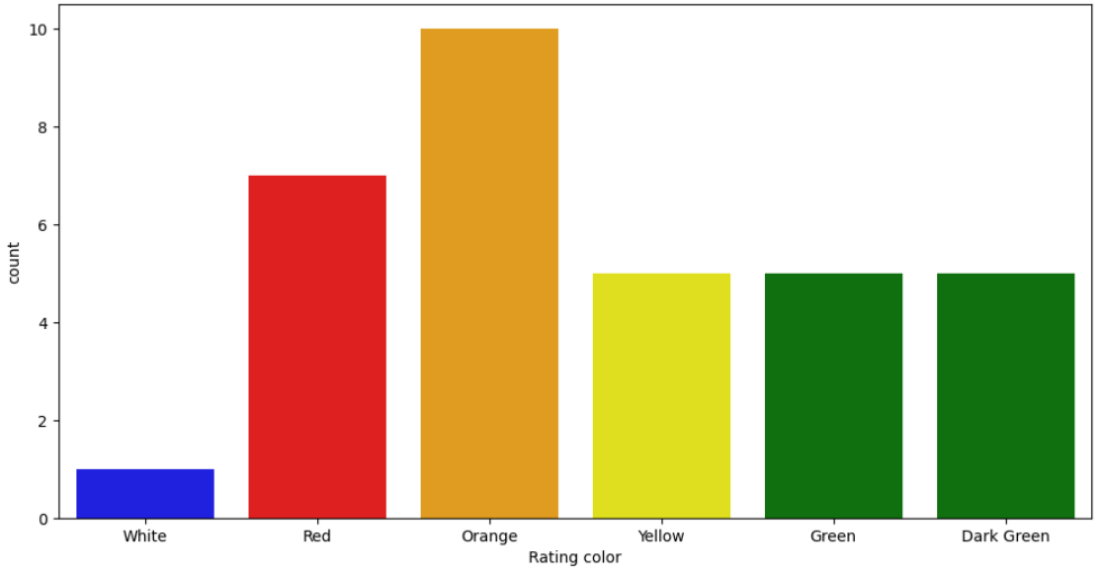
```
In [51]: sns.barplot(x="Aggregate rating",y="Rating Count",hue='Rating color',data=ratings,palette=['blue','red','orange','yellow','green'
```

```
Out[51]: <AxesSubplot:xlabel='Aggregate rating', ylabel='Rating Count'>
```




```
In [41]: ## Count plot
sns.countplot(x="Rating color",data=ratings,palette=['blue','red','orange','yellow','green','green'])
```

```
Out[41]: <Axes: xlabel='Rating color', ylabel='count'>
```



- find the country's name that with 0 ratings

```
In [60]: ### Find the countries name that has given 0 rating
final_df[final_df['Rating color']=='White'].groupby('Country').size().reset_index()
```

```
Out[60]:
```

	Country	0
0	Brazil	5
1	India	2139
2	United Kingdom	1
3	United States	3

- find the countries and ratings with 0 ratings

In [61]:
```
final_df.groupby(['Aggregate rating','Country']).size().reset_index().head(5)
```

Out[61]:

	Aggregate rating	Country	0
0	0.0	Brazil	5
1	0.0	India	2139
2	0.0	United Kingdom	1
3	0.0	United States	3
4	1.8	India	1

Observations Maximum number of 0 ratings are from Indian customers

- find which countries have what currency

In [63]:
```
##find out which currency is used by which country?
final_df.columns
```

Out[63]:
```
Index(['Restaurant ID', 'Restaurant Name', 'Country Code', 'City', 'Address',
       'Locality', 'Locality Verbose', 'Longitude', 'Latitude', 'Cuisines',
       'Average Cost for two', 'Currency', 'Has Table booking',
       'Has Online delivery', 'Is delivering now', 'Switch to order menu',
       'Price range', 'Aggregate rating', 'Rating color', 'Rating text',
       'Votes', 'Country'],
      dtype='object')
```

In [64]:
```
final_df[['Country','Currency']].groupby(['Country','Currency']).size().reset_index()
```

Out[64]:

	Country	Currency	0
0	Australia	Dollar($)	24
1	Brazil	Brazilian Real(R$)	60
2	Canada	Dollar($)	4
3	India	Indian Rupees(Rs.)	8652
4	Indonesia	Indonesian Rupiah(IDR)	21
5	New Zealand	NewZealand($)	40
6	Phillipines	Botswana Pula(P)	22
7	Qatar	Qatari Rial(QR)	20
8	Singapore	Dollar($)	20
9	South Africa	Rand(R)	60
10	Sri Lanka	Sri Lankan Rupee(LKR)	20
11	Turkey	Turkish Lira(TL)	34
12	UAE	Emirati Diram(AED)	60
13	United Kingdom	Pounds(□£)	80
14	United States	Dollar($)	434

- which countries have online delivery available?

In [ ]:
```
## Which Countries do have online deliveries option
```

In [68]:
```
final_df[final_df['Has Online delivery'] =="Yes"].Country.value_counts()
```

Out[68]:
```
India    2423
UAE        28
Name: Country, dtype: int64
```

In [69]:
```
final_df[['Has Online delivery','Country']].groupby(['Has Online delivery','Country']).size().reset_index()
```

Out[69]:

	Has Online delivery	Country	0
0	No	Australia	24
1	No	Brazil	60
2	No	Canada	4
3	No	India	6229
4	No	Indonesia	21
5	No	New Zealand	40
6	No	Phillipines	22
7	No	Qatar	20
8	No	Singapore	20
9	No	South Africa	60
10	No	Sri Lanka	20
11	No	Turkey	34
12	No	UAE	32
13	No	United Kingdom	80
14	No	United States	434
15	Yes	India	2423
16	Yes	UAE	28

- Top 5 cities ka distribution

In [74]:
```
## Create a pie chart for top 5 cities distribution
```

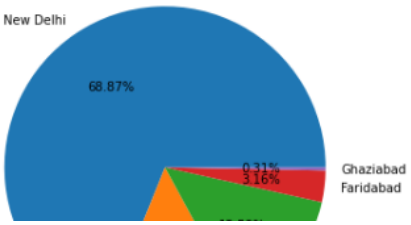
In [72]:
```
final_df.City.value_counts().index
```

Out[72]:
```
Index(['New Delhi', 'Gurgaon', 'Noida', 'Faridabad', 'Ghaziabad',
       'Bhubaneshwar', 'Amritsar', 'Ahmedabad', 'Lucknow', 'Guwahati',
       ...
       'Ojo Caliente', 'Montville', 'Monroe', 'Miller', 'Middleton Beach',
       'Panchkula', 'Mc Millan', 'Mayfield', 'Macedon', 'Vineland Station'],
      dtype='object', length=141)
```

In [76]:
```
city_values=final_df.City.value_counts().values
city_labels=final_df.City.value_counts().index
```

In [78]:
```
plt.pie(city_values[:5],labels=city_labels[:5],autopct='%1.2f%%')
```

Out[78]:
```
([<matplotlib.patches.Wedge at 0x1efa7165280>,
  <matplotlib.patches.Wedge at 0x1efa7165940>,
  <matplotlib.patches.Wedge at 0x1efa7165fd0>,
  <matplotlib.patches.Wedge at 0x1efa7458760>,
  <matplotlib.patches.Wedge at 0x1efa7458e80>],
 [Text(-0.6145352824185932, 0.9123301960708633, 'New Delhi'),
  Text(0.0623675251198054, -1.0982305276263407, 'Gurgaon'),
  Text(0.8789045225625368, -0.6614581167535246, 'Noida'),
  Text(1.0922218418223437, -0.13058119407559224, 'Faridabad'),
  Text(1.099946280005612, -0.010871113182029924, 'Ghaziabad')],
 [Text(-0.3352010631374145, 0.497634652402289, '68.87%'),
  Text(0.0340186500653484, -0.5990348332507311, '14.07%'),
  Text(0.47940246685229276, -0.36079533641101336, '13.59%'),
  Text(0.5957573682667329, -0.07122610585941394, '3.16%'),
  Text(0.5999706981848791, -0.005929698099289049, '0.31%')])
```



- top 5 cuisines

Assignment

Find the top 10 cuisines

```
In [51]: final_df.columns
```

```
Out[51]: Index(['Restaurant ID', 'Restaurant Name', 'Country Code', 'City', 'Address',
       'Locality', 'Locality Verbose', 'Longitude', 'Latitude', 'Cuisines',
       'Average Cost for two', 'Currency', 'Has Table booking',
       'Has Online delivery', 'Is delivering now', 'Switch to order menu',
       'Price range', 'Aggregate rating', 'Rating color', 'Rating text',
       'Votes', 'Country'],
      dtype='object')
```

```
In [68]: final_df['Cuisines'].value_counts().head(10)
```

```
Out[68]: North Indian                       936
         North Indian, Chinese              511
         Chinese                            354
         Fast Food                          354
         North Indian, Mughlai              334
         Cafe                               299
         Bakery                             218
         North Indian, Mughlai, Chinese     197
         Bakery, Desserts                   170
         Street Food                        149
         Name: Cuisines, dtype: int64
```

2.) Train and Test

https://github.com/krishnaik06/5-Days-Live-EDA-and-Feature-Engineering/blob/main/2-%20BlackFriday%20EDA%20And%20Feature%20Engineering.ipynb

- Here, we are given two different datasets, **test** and **train**, first we need to merge these two datasets

```
In [10]: ##MErge both train and test data
         df = pd.concat([df_train, df_test])
         df.head()
```

Out[10]:

	User_ID	Product_ID	Gender	Age	Occupation	City_Category	Stay_In_Current_City_Years	Marital_Status	Product_Category_1	Product_Category_2	Product
0	1000001	P00069042	F	0-17	10	A	2	0	3	NaN	
1	1000001	P00248942	F	0-17	10	A	2	0	1	6.0	
2	1000001	P00087842	F	0-17	10	A	2	0	12	NaN	
3	1000001	P00085442	F	0-17	10	A	2	0	12	14.0	
4	1000002	P00285442	M	55+	16	C	4+	0	8	NaN	

- Remove the ‘User_ID’ column (axis=1)

```
In [8]: df.drop(['User_ID'],axis=1,inplace=True)
```

```
In [9]: df.head()
```

Out[9]:

	Product_ID	Gender	Age	Occupation	City_Category	Stay_In_Current_City_Years	Marital_Status	Product_Category_1	Product_Category_2	Product_Category
0	P00069042	F	0-17	10	A	2	0	3	NaN	N
1	P00248942	F	0-17	10	A	2	0	1	6.0	1
2	P00087842	F	0-17	10	A	2	0	12	NaN	N
3	P00085442	F	0-17	10	A	2	0	12	14.0	N
4	P00285442	M	55+	16	C	4+	0	8	NaN	N

- Handling categorical features

```
In [27]: df['Gender']=pd.get_dummies(df['Gender'],drop_first=1)
```

```
In [12]: ##HAndling categorical feature Gender
         df['Gender']=df['Gender'].map({'F':0,'M':1})
         df.head()
```

Out[12]:

	Product_ID	Gender	Age	Occupation	City_Category	Stay_In_Current_City_Years	Marital_Status	Product_Category_1	Product_Category_2	Product_Category
0	P00069042	0	0-17	10	A	2	0	3	NaN	N
1	P00248942	0	0-17	10	A	2	0	1	6.0	1
2	P00087842	0	0-17	10	A	2	0	12	NaN	N
3	P00085442	0	0-17	10	A	2	0	12	14.0	N
4	P00285442	1	55+	16	C	4+	0	8	NaN	N

In [13]:
```
## Handle categorical feature Age
df['Age'].unique()
```

Out[13]:
```
array(['0-17', '55+', '26-35', '46-50', '51-55', '36-45', '18-25'],
      dtype=object)
```

In [17]:
```
#pd.get_dummies(df['Age'],drop_first=True)
df['Age']=df['Age'].map({'0-17':1,'18-25':2,'26-35':3,'36-45':4,'46-50':5,'51-55':6,'55+':7})
```

In [ ]:
```
##second technqiue
from sklearn import preprocessing

# label_encoder object knows how to understand word labels.
label_encoder = preprocessing.LabelEncoder()

# Encode labels in column 'species'.
df['Age']= label_encoder.fit_transform(df['Age'])

df['Age'].unique()
```

In [18]:
```
df.head()
```

Out[18]:

	Product_ID	Gender	Age	Occupation	City_Category	Stay_In_Current_City_Years	Marital_Status	Product_Category_1	Product_Category_2	Product_Category
0	P00069042	0	1	10	A	2	0	3	NaN	N
1	P00248942	0	1	10	A	2	0	1	6.0	1
2	P00087842	0	1	10	A	2	0	12	NaN	N
3	P00085442	0	1	10	A	2	0	12	14.0	N
4	P00285442	1	7	16	C	4+	0	8	NaN	N

- replace the missing values with mode

	Product_ID	Gender	Age	Occupation	Stay_In_Current_City_Years	Marital_Status	Product_Category_1	Product_Category_2	Product_Category_3	Purchase
0	P00069042	0	1	10	2	0	3	NaN	NaN	8370.0
1	P00248942	0	1	10	2	0	1	6.0	14.0	15200.0
2	P00087842	0	1	10	2	0	12	NaN	NaN	1422.0
3	P00085442	0	1	10	2	0	12	14.0	NaN	1057.0
4	P00285442	1	7	16	4+	0	8	NaN	NaN	7969.0

In [34]:
```
df['Product_Category_2'].mode()[0]
```

Out[34]:
```
8.0
```

In [35]:
```
## Replace the missing values with mode
df['Product_Category_2']=df['Product_Category_2'].fillna(df['Product_Category_2'].mode()[0])
```

In [36]:
```
df['Product_Category_2'].isnull().sum()
```

Out[36]:
```
0
```

In [37]:
```
## Product_category 3 replace missing values
df['Product_Category_3'].unique()
```

Out[37]:
```
array([nan, 14., 17.,  5.,  4., 16., 15.,  8.,  9., 13.,  6., 12.,  3.,
       18., 11., 10.])
```

In [38]:
```
df['Product_Category_3'].value_counts()
```

Out[38]:
```
16.0    46469
15.0    39968
14.0    26283
17.0    23818
5.0     23799
8.0     17861
```

`df['Product_Category_2'].mode()[0]` is a way to get the most frequent value in the 'Product_Category_2' column and return it as a scalar value.

Here's what's happening:

- `df['Product_Category_2'].mode()` returns a pandas Series containing the most frequent value(s) in the 'Product_Category_2' column.
- `[0]` is used to index the Series and return the first (and in this case, only) value.

So, `df['Product_Category_2'].mode()[0]` is equivalent to `df['Product_Category_2'].mode().values[0]`.

- eg. replace '+' with ' '

```
In [70]: df['Stay_In_Current_City_Years'] = df['Stay_In_Current_City_Years'].str.replace('+', '', regex=False)

In [71]: df['Stay_In_Current_City_Years'].unique()
Out[71]: array(['2', '4', '3', '1', '0'], dtype=object)

In [72]: df.head()
```

Out[72]:

	Product_ID	Gender	Age	Occupation	City_Category	Stay_In_Current_City_Years	Marital_Status	Product_Category_1	Product_Category_2	Product_Category
0	P00069042	NaN	0	10	A	2	0	3	NaN	N
1	P00248942	NaN	0	10	A	2	0	1	6.0	1
2	P00087842	NaN	0	10	A	2	0	12	NaN	N
3	P00085442	NaN	0	10	A	2	0	12	14.0	N
4	P00285442	NaN	6	16	C	4	0	8	NaN	N

- typecasting

In [46]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 783667 entries, 0 to 233598
Data columns (total 12 columns):
 #   Column                      Non-Null Count   Dtype  
---  ------                      --------------   -----  
 0   Product_ID                  783667 non-null  object 
 1   Gender                      783667 non-null  int64  
 2   Age                         783667 non-null  int64  
 3   Occupation                  783667 non-null  int64  
 4   Stay_In_Current_City_Years  783667 non-null  object 
 5   Marital_Status              783667 non-null  int64  
 6   Product_Category_1          783667 non-null  int64  
 7   Product_Category_2          783667 non-null  float64
 8   Product_Category_3          783667 non-null  float64
 9   Purchase                    550068 non-null  float64
 10  B                           783667 non-null  uint8  
 11  C                           783667 non-null  uint8  
dtypes: float64(3), int64(5), object(2), uint8(2)
memory usage: 67.3+ MB
```

In [47]:
```
##convert object into integers
df['Stay_In_Current_City_Years']=df['Stay_In_Current_City_Years'].astype(int)
df.info()
```

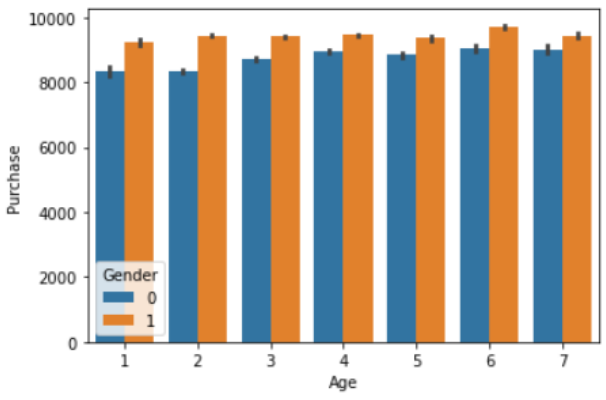
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 783667 entries, 0 to 233598
Data columns (total 12 columns):
 #   Column                      Non-Null Count   Dtype  
---  ------                      --------------   -----  
 0   Product_ID                  783667 non-null  object 
 1   Gender                      783667 non-null  int64  
 2   Age                         783667 non-null  int64  
 3   Occupation                  783667 non-null  int64  
```

- visualisation (here 0: females, 1: males, made upar using map)

In [53]:
```
##Visualisation Age vs Purchased
sns.barplot('Age','Purchase',hue='Gender',data=df)
```

```
C:\Users\win10\anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass
gs: x, y. From version 0.12, the only valid positional argument will be `data`, and passir
keyword will result in an error or misinterpretation.
  warnings.warn(
```

Out[53]: <AxesSubplot:xlabel='Age', ylabel='Purchase'>



- Feature Scaling

Feature scaling is a process in machine learning where you transform your features (or independent variables) to a common scale, usually between 0 and 1, to ensure that all features have the same importance and to prevent some features from dominating the model.

Here's an example:

Suppose you're building a model to predict the price of a house based on its size, number of bedrooms, and location. The features are:

- Size (in square feet): 1000-5000
- Number of bedrooms: 2-5
- Location (a categorical variable): Urban, Suburban, Rural

If you don't scale these features, the model will be heavily influenced by the size of the house, since it's a large range of values. The number of bedrooms and location will have less impact on the model.

To scale these features, you can use techniques such as:

1. Min-Max Scaling (also known as normalization):
    - Calculate the minimum and maximum values for each feature.
    - Subtract the minimum value from each feature and divide by the range (maximum - minimum).
    - This scales the features to a common range, usually between 0 and 1.

Here's an example in Python using Scikit-learn:

```
from sklearn.preprocessing import MinMaxScaler

# Create a sample dataset
data = {'Size': [1000, 2000, 3000, 4000, 5000],
        'Bedrooms': [2, 3, 4, 5, 5],
        'Location': ['Urban', 'Suburban', 'Rural', 'Urban', 'Suburban']}

# Scale the features using Min-Max Scaling
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)

print(scaled_data)
```

Output:

```
[[0.0, 0.25, 0.5],
 [0.25, 0.5, 0.75],
 [0.5, 0.75, 1.0],
 [0.75, 1.0, 0.5],
 [1.0, 1.0, 0.25]]
```

Now, we can see below that in the scaled data, mean=0 and sd=1,

2. Standardization (also known as Z-scoring):
   - Calculate the mean and standard deviation for each feature.
   - Subtract the mean from each feature and divide by the standard deviation.
   - This scales the features to have a mean of 0 and a standard deviation of 1.

Here's an example in Python using Scikit-learn:

```
from sklearn.preprocessing import StandardScaler

# Create a sample dataset
data = {'Size': [1000, 2000, 3000, 4000, 5000],
        'Bedrooms': [2, 3, 4, 5, 5],
        'Location': ['Urban', 'Suburban', 'Rural', 'Urban', 'Suburban']}

# Scale the features using Standardization
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)

print(scaled_data)
```

Output:

```
[[-1.0, -0.5, 0.0],
 [-0.5, 0.0, 0.5],
 [0.0, 0.5, 1.0],
 [0.5, 1.0, -0.5],
 [1.0, 1.0, -1.0]]
```

example:

python Copy

```
# Splitting the data into train and test sets
df_test = df[df['Purchase'].isnull()]
df_train = df[~df['Purchase'].isnull()]
```

This code separates the original dataframe `df` into two parts:

- `df_test`: Contains rows where 'Purchase' column is null (missing values)
- `df_train`: Contains rows where 'Purchase' column is not null

```python
# Preparing features (X) and target variable (y)
X = df_train.drop('Purchase', axis=1)
y = df_train['Purchase']
```

Here, we're creating:

- `X`: All columns from `df_train` except 'Purchase' (these are our features)
- `y`: Just the 'Purchase' column (this is our target variable)

```python
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=42)
```

This code:

1. Imports the `train_test_split` function from scikit-learn.
2. Splits the data (`X` and `y`) into training and testing sets:
   - 67% for training (`X_train`, `y_train`)
   - 33% for testing (`X_test`, `y_test`)
3. Uses `random_state=42` for reproducibility.

The split allows you to train your model on one set of data and validate its performance on unseen data, helping to assess how well the model generalizes.

1. Removing the 'Product_ID' column:

python Copy

```python
X_train.drop('Product_ID', axis=1, inplace=True)
X_test.drop('Product_ID', axis=1, inplace=True)
```

This step removes the 'Product_ID' column from both the training and testing datasets. Here's why this is often done:

- The 'Product_ID' is likely a unique identifier for each product. While it's useful for identification, it usually doesn't contribute to predicting the target variable (in this case, 'Purchase').
- Including 'Product_ID' might lead to overfitting, as the model could memorize specific product IDs instead of learning generalizable patterns.
- `axis=1` specifies that we're dropping a column (not a row).
- `inplace=True` means the operation is performed on the original dataframe, modifying it directly instead of returning a new dataframe.

2. Feature Scaling:

```python
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

This step standardizes the features, which is crucial for many machine learning algorithms. Here's a breakdown:

a. `from sklearn.preprocessing import StandardScaler` : This imports the StandardScaler class from scikit-learn.

b. `sc = StandardScaler()` : This creates an instance of the StandardScaler.

c. `X_train = sc.fit_transform(X_train)` : This does two things:

- `fit` : It computes the mean and standard deviation of each feature in `X_train` .
- `transform` : It standardizes `X_train` by subtracting the mean and dividing by the standard deviation for each feature.

d. `X_test = sc.transform(X_test)` : This standardizes `X_test` using the mean and standard deviation computed from `X_train` . We don't use `fit_transform` here because we want to use the same scaling parameters as the training data.

The purpose of standardization is to ensure all features are on a similar scale. This is important because:

- Many machine learning algorithms perform better or converge faster when features are on a similar scale.
- It prevents features with larger scales from dominating the model's learning process.

After these steps, your data is preprocessed and ready for model training. The features are scaled, unnecessary columns are removed, and you have separate sets for training and validation.

3.) Flight Prediction Example

- First we have test and train dataset, we will append,

```
In [52]: final_df=train_df.append(test_df)
         final_df.head()
```

```
C:\Users\Tarsh Swarnkar\AppData\Local\Temp\ipykernel_47712\2632932177.py:1: FutureWarning: The frame.append method is deprecate
d and will be removed from pandas in a future version. Use pandas.concat instead.
  final_df=train_df.append(test_df)
```

Out[52]:

	Airline	Date_of_Journey	Source	Destination	Route	Dep_Time	Arrival_Time	Duration	Total_Stops	Additional_Info	Price
0	IndiGo	24/03/2019	Banglore	New Delhi	BLR → DEL	22:20	01:10 22 Mar	2h 50m	non-stop	No info	3897.0
1	Air India	1/05/2019	Kolkata	Banglore	CCU → IXR → BBI → BLR	05:50	13:15	7h 25m	2 stops	No info	7662.0
2	Jet Airways	9/06/2019	Delhi	Cochin	DEL → LKO → BOM → COK	09:25	04:25 10 Jun	19h	2 stops	No info	13882.0
3	IndiGo	12/05/2019	Kolkata	Banglore	CCU → NAG → BLR	18:05	23:30	5h 25m	1 stop	No info	6218.0
4	IndiGo	01/03/2019	Banglore	New Delhi	BLR → NAG → DEL	16:50	21:35	4h 45m	1 stop	No info	13302.0

- Split the ‘Date of Journey’

```
In [10]: final_df['Date_of_Journey'].str.split('/').str[0]
```

```
Out[10]: 0        24
         1         1
         2         9
         3        12
         4        01
                  ..
         2666      6
         2667     27
         2668      6
         2669      6
         2670     15
         Name: Date_of_Journey, Length: 13354, dtype: object
```

```
##Fetaure Engineering Process
final_df['Date']=final_df['Date_of_Journey'].str.split('/').str[0]
final_df['Month']=final_df['Date_of_Journey'].str.split('/').str[1]
final_df['Year']=final_df['Date_of_Journey'].str.split('/').str[2]
```

```
final_df.head(2)
```

- Removing ‘h’ from Duration Hour

```
In [90]: final_df['duration_hour']=final_df['Duration'].str.split(' ').str[0].str.split('h').str[0]
```

```
In [136]: final_df
```

Out[136]:

Airline	Source	Destination	Arrival_Time	Duration	Total_Stops	Additional_Info	Price	Date	Month	Year	Arrival_hour	Arrival_min	Dept_hour	Dept_min
IndiGo	Banglore	New Delhi	01:10 22 Mar	2h 50m	NaN	No info	3897.0	24	3	2019	1	10 22 Mar	22	20
Air India	Kolkata	Banglore	13:15	7h 25m	NaN	No info	7662.0	1	5	2019	13	15	5	50
Jet Airways	Delhi	Cochin	04:25 10 Jun	19h	NaN	No info	13882.0	9	6	2019	4	25 10 Jun	9	25
IndiGo	Kolkata	Banglore	23:30	5h 25m	NaN	No info	6218.0	12	5	2019	23	30	18	5
IndiGo	Banglore	New Delhi	21:35	4h 45m	NaN	No info	13302.0	1	3	2019	21	35	16	50
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Air India	Kolkata	Banglore	20:25 07 Jun	23h 55m	NaN	No info	NaN	6	6	2019	20	25 07 Jun	20	30
IndiGo	Kolkata	Banglore	16:55	2h 35m	NaN	No info	NaN	27	3	2019	16	55	14	20
Jet Airways	Delhi	Cochin	04:25 07 Mar	6h 35m	NaN	No info	NaN	6	3	2019	4	25 07 Mar	21	50

- Label Encoding

```
In [96]: final_df['Airline'].unique()
```

Out[96]:
```
array(['IndiGo', 'Air India', 'Jet Airways', 'SpiceJet',
       'Multiple carriers', 'GoAir', 'Vistara', 'Air Asia',
       'Vistara Premium economy', 'Jet Airways Business',
       'Multiple carriers Premium economy', 'Trujet'], dtype=object)
```

```
In [97]: from sklearn.preprocessing import LabelEncoder
labelencoder=LabelEncoder()
```

```
In [98]: final_df['Airline']=labelencoder.fit_transform(final_df['Airline'])
final_df['Source']=labelencoder.fit_transform(final_df['Source'])
final_df['Destination']=labelencoder.fit_transform(final_df['Destination'])
final_df['Additional_Info']=labelencoder.fit_transform(final_df['Additional_Info'])
```

```
In [99]: final_df.shape
```

Out[99]:
```
(13351, 14)
```

```
In [144]: final_df.head(5)
```

Out[144]:

	Airline	Source	Destination	Arrival_Time	Duration	Total_Stops	Additional_Info	Price	Date	Month	Year	Arrival_hour	Arrival_min	Dept_hour	Dept_l
0	IndiGo	Banglore	New Delhi	01:10 22 Mar	2h 50m	NaN	No info	3897.0	24	3	2019	1	10 22 Mar	22	
1	Air India	Kolkata	Banglore	13:15	7h 25m	NaN	No info	7662.0	1	5	2019	13	15	5	
2	Jet Airways	Delhi	Cochin	04:25 10 Jun	19h	NaN	No info	13882.0	9	6	2019	4	25 10 Jun	9	
3	IndiGo	Kolkata	Banglore	23:30	5h 25m	NaN	No info	6218.0	12	5	2019	23	30	18	
4	IndiGo	Banglore	New Delhi	21:35	4h 45m	NaN	No info	13302.0	1	3	2019	21	35	16	

```
In [101]: final_df[['Airline']]
```

Out[101]:

	Airline
0	3
1	1
2	4
3	3

Here's a breakdown of what it does:

- `final_df['Airline']` is selecting the column 'Airline' from the DataFrame `final_df`.
- `labelencoder` is an object from scikit-learn's preprocessing module, which is used to transform categorical variables into numerical variables.
- `fit_transform` is a method that fits the label encoder to the data and then transforms the data. It returns a numpy array of the encoded labels.

- OneHotEncoder()

4.) EDA Automation

https://www.youtube.com/watch?v=zeCFHEJqAR4&list=PLZoTAELRMXVPzj1D0i_6ajJ6gyD22b3jh&index=5

- **Pandas Profiling**

https://www.analyticsvidhya.com/blog/2021/06/generate-reports-using-pandas-profiling-deploy-using-streamlit/

- **dtale**
- **lux**

AI/ML

TOPIC:	MongoDB with Python	START DATE:	26/07/2024	↓

1.) MongoDB installation and Basic CRUD Operations

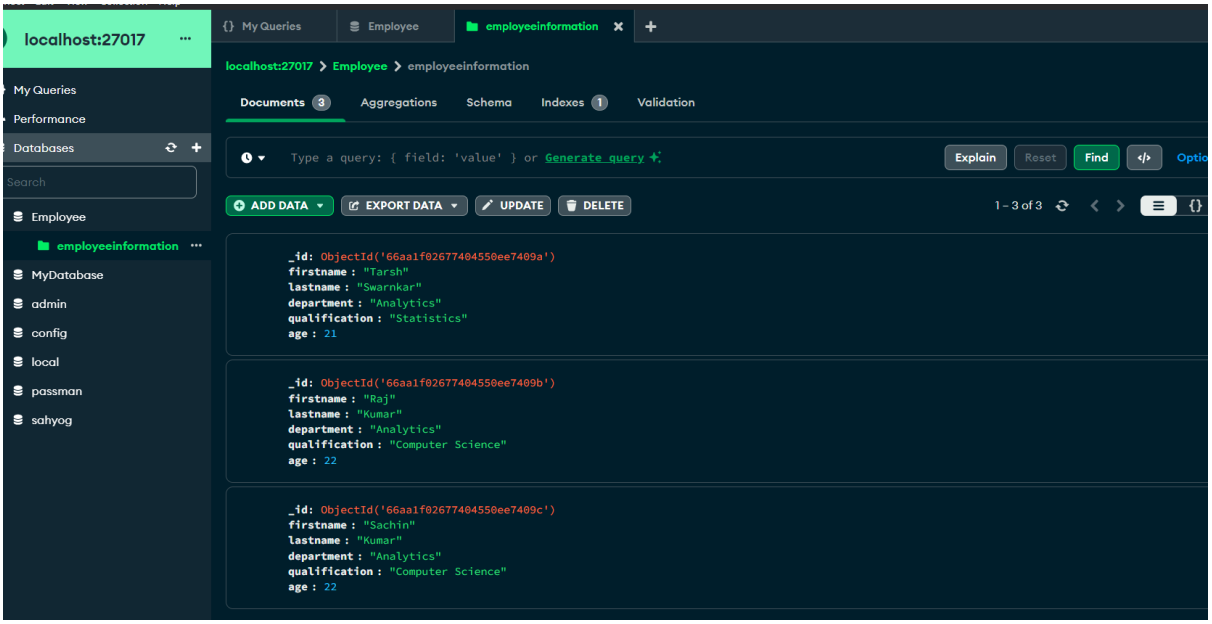
2.) PyMongo installation and Basics (Python with MongoDB)

In [1]:
```
import pymongo

client=pymongo.MongoClient('mongodb://127.0.0.1:27017/')

mydb=client['Employee']
empinfo=mydb.employeeinformation
```

In [2]:
```
record={
        'firstname':'Krish',
        'lastname':'Naik',
        'department':'Analytics',
        'qualification':'BE',
        'age':29
        }
empinfo.insert_one(record)
```

Out[2]:
```
InsertOneResult(ObjectId('66bb43c5731f569eac0c455f'), acknowledged=True)
```

In [3]:
```
records=[{
        'firstname':'John',
        'lastname':'Doe',
        'department':'Analytics',
        'qualification':'statistics',
        'age':35

        },
         {
        'firstname':'John ',
        'lastname':'Smith'
```



2.1) Find

In [8]:
```
## Simple way of querying
empinfo.find_one()
```

Out[8]:
```
{'_id': ObjectId('5e59ec2be0e386955e9489ba'),
 'firstname': 'Krish',
 'lastname': 'Naik',
 'department': 'Analytics',
 'qualification': 'BE',
 'age': 29}
```

In [11]:
```
## Select * from employeeinformation
for record in empinfo.find({}):
    print(record)
```

```
{'_id': ObjectId('5e59ec2be0e386955e9489ba'), 'firstname': 'Krish', 'lastname': 'Naik', 'department': 'Analytics', 'qualificati
on': 'BE', 'age': 29}
{'_id': ObjectId('5e59eca7e0e386955e9489bb'), 'firstname': 'John', 'lastname': 'Doe', 'department': 'Analytics', 'qualificatio
n': 'statistics', 'age': 35}
{'_id': ObjectId('5e59eca7e0e386955e9489bc'), 'firstname': 'John ', 'lastname': 'Smith', 'department': 'Analytics', 'qualificat
ion': 'masters', 'age': 30}
{'_id': ObjectId('5e59eca7e0e386955e9489bd'), 'firstname': 'Manish', 'lastname': 'Sen', 'department': 'Analytics', 'qualificati
on': 'phd', 'age': 34}
{'_id': ObjectId('5e59eca7e0e386955e9489be'), 'firstname': 'Ram', 'lastname': 'Singh', 'department': 'Analytics', 'qualificatio
n': 'master', 'age': 32}
```

In [12]:
```
## Query the json documents based on equality conditions
# Select * from employeeinformation where firstname=krish

for record in empinfo.find({'firstname':'Krish'}):
    print(record)
```

In your example:

```
empinfo.find({'qualification':'master','age':{'$lt':35}})
```

This query is equivalent to:

```
empinfo.find({'$and': [{'qualification': 'master'}, {'age': {'$lt': 35}}]})
```

Or ($or)

The $or operator is used to filter documents that match at least one of the conditions specified in the query. The syntax is:

```
{'$or': [condition1, condition2, ...]}
```

In your example:

```
empinfo.find({'$or':[{'firstname':'Krish'},{'qualification':'BE'}]})
```

In [13]:
```
## And and Query operators
for record in empinfo.find({'qualification':'master','age':{'$lt':35}}):
    print(record)
```

```
{'_id': ObjectId('66bb43df731f569eac0c4563'), 'firstname': 'Ram', 'lastname': 'Singh', 'department': 'Analytics', 'qualificatio
n': 'master', 'age': 32}
```

In [14]:
```
## OR operators
for record in empinfo.find({'$or':[{'firstname':'Krish'},{'qualification':'BE'}]}):
    print(record)
```

```
{'_id': ObjectId('66bb43c5731f569eac0c455f'), 'firstname': 'Krish', 'lastname': 'Naik', 'department': 'Analytics', 'qualificati
on': 'BE', 'age': 29}
```

2.2)