

EXPERIMENT-1

AIM:- Creating a NumPy Array

- a) Basic ndarray
- b) Array of zeros
- c) Array of ones
- d) Random numbers in ndarray
- e) An array of your choice
- f) Imatrix in NumPy
- g) Evenly spaced ndarray

PROCEDURE:-

Creating a NumPy Array

a) Basic ndarray :

To create a very basic ndarray, we use the [np.array\(\)](#) method. We have to pass the values of the array as a list.

```
np.array([1,2,3,4],dtype=np.float32)
```

Output:

```
array([1., 2., 3., 4.], dtype=float32)
```

Since NumPy arrays can contain only homogeneous datatypes, values will be upcast if the types do not match.

```
np.array([1,2.0,3,4])
```

Output:

```
array([1., 2., 3., 4.])
```

b) Array of zeros :

NumPy creates an array of all zeros using the [np.zeros\(\)](#) method. All we have to do is pass the shape of the desired array:

```
np.zeros(5)
```

Output:

```
array([0., 0., 0., 0., 0.])
```

The one above is a 1-D array while the one below is a 2-D array:

```
np.zeros((2,3))
```

Output:

```
array([[0., 0., 0.],  
       [0., 0., 0.]])
```

b) Array of ones :

We could also create an array of all **1s** using the [np.ones\(\)](#) method:

```
np.ones(5,dtype=np.int32)
```

Output:

```
array([1, 1, 1, 1, 1])
```

c) Random numbers in ndarrays :

Another very commonly used method to create ndarrays is [np.random.rand\(\)](#) method. It creates an array of a given shape with random values from [0,1):

```
# random
np.random.rand(2,3)
```

Output:

```
array([[0.95580785, 0.98378873, 0.65133872],
       [0.38330437, 0.16033608, 0.13826526]])
```

d) An array of your choice :

It can create an array filled with any given value using the [np.full\(\)](#) method.

```
np.full((2,2),7)
```

Output:

```
array([[7, 7],
       [7, 7]])
```

e) Imatrix in NumPy :

Another great method is [np.eye\(\)](#) that returns an array with **1s** along its diagonal and **0s** everywhere else.

```
# identity matrix
np.eye(3)
```

Output:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Main diagonal:

```
np.eye(3,k=-2)
```

Output:

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [1., 0., 0.]])
```

f) Evenly spaced ndarray :

We can quickly get an evenly spaced array of numbers using the `np.arange()` method:

```
np.arange(5)
```

Output:

```
array([0, 1, 2, 3, 4])
```

The start, end and step size of the interval of values :

```
np.arange(2,10,2)
```

Output:

```
array([2, 4, 6, 8])
```

We use another similar function

i.e ; `np.linspace()`,

```
np.linspace(0,1,5)
```

Output:

```
array([0. , 0.25, 0.5 , 0.75, 1.  ])
```

EXPERIMENT-2

AIM:- The Shape and Reshaping of NumPy Array

- a) Dimensions of NumPy array
- b) Shape of NumPy Array
- c) Size of NumPy Array
- d) Reshaping a NumPy Array
- e) Flattening a Numpy Array
- f) Transpose of a Numpy Array

PROCEDURE:-

a) Dimensions of NumPy arrays :

We can easily determine the number of dimensions or axes of a NumPy array using the **ndims** attribute:

```
# number of axis
a = np.array([[5,10,15],[20,25,20]])
print('Array :','\n',a)
print('Dimensions :','\n',a.ndim)
```

Output:

```
Array :
[[ 5 10 15]
 [20 25 20]]
Dimensions :
2
```

b) Shape of NumPy array :

It shows how many rows of elements are there along each dimension.

```
a = np.array([[1,2,3],[4,5,6]])
print('Array :','\n',a)
print('Shape :','\n',a.shape)
print('Rows = ',a.shape[0])
print('Columns = ',a.shape[1])
```

Output:

```
Array :
[[1 2 3]
 [4 5 6]]
Shape :
(2, 3)
Rows = 2
Columns = 3
```

c) Size of NumPy array :

This attribute shows how many values are there in the array using and it just multiplies the number of rows by the number of columns in the ndarray:

```
# size of array
a = np.array([[5,10,15],[20,25,20]])
print('Size of array :',a.size)
print('Manual determination of size of array :',a.shape[0]*a.shape[1])
```

Output:

```
Size of array : 6
Manual determination of size of array : 6
```

5	10	15	Shape : (2,3) Size : 6 N-dim : 2
20	25	30	

d) Reshaping a NumPy array :

It changes the shape of the ndarray without changing the data within the ndarray, it uses `np.reshape()` method :

```
# reshape
a = np.array([3,6,9,12])
np.reshape(a,(2,2))
```

Output:

```
array([[ 3,  6],
       [ 9, 12]])
```

While reshaping, if we are unsure about the shape of any of the axis, just input -1. NumPy automatically calculates the shape when it sees a -1:

```
a = np.array([3,6,9,12,18,24])
print('Three rows :','\n',np.reshape(a,(3,-1)))
print('Three columns :','\n',np.reshape(a,(-1,3)))
```

Output:

```
Three rows :
[[ 3  6]
 [ 9 12]
 [18 24]]
Three columns :
[[ 3  6  9]
 [12 18 24]]
```

e) Flattening a NumPy array :

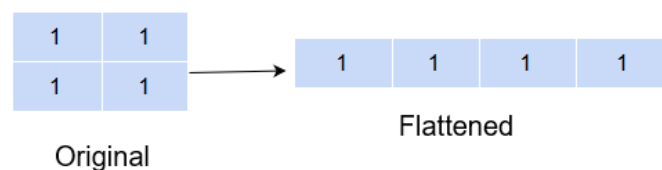
Sometimes when we have a multidimensional array and want to collapse it to a single-dimensional array, we can either use the **flatten()** method or the **ravel()** method:

```
a = np.ones((2,2))
b = a.flatten()
c = a.ravel()
print('Original shape :', a.shape)
print('Array :','\n', a)
print('Shape after flatten :',b.shape)
print('Array :','\n', b)
```

```
print('Shape after ravel :',c.shape)
print('Array :','\n', c)
Original shape : (2, 2)
```

Output:

```
Array :
[[1. 1.]
 [1. 1.]]
Shape after flatten : (4,)
Array :
[1. 1. 1. 1.]
Shape after ravel : (4,)
Array :
[1. 1. 1. 1.]
```



f) Transpose of a NumPy array :

It takes the input array and swaps the rows with the column values, and the column values with the values of the rows, it uses **transpose()** method:

```
a = np.array([[1,2,3],
 [4,5,6]])
b = np.transpose(a)
print('Original','\n','Shape',a.shape,'\n',a)
print('Expand along columns:','\n','Shape',b.shape,'\n',b)
Original
Shape (2, 3)
[[1 2 3]
 [4 5 6]]
Expand along columns:
Shape (3, 2)
[[1 4]
 [2 5]
 [3 6]]
```


EXPERIMENT-3

AIM:- Indexing And Slicing of NumPy Array

- a) Slicing 1-D NumPy Arrays
- b) Slicing 2-D NumPy Arrays
- c) Slicing 3-D NumPy Arrays
- d) Negative slicing of NumPy Arrays

PROCEDURE:-

We have seen how to create a NumPy array and how to play around with its shape. To extract specific values from the array we use indexing and slicing.

a) Slicing 1-D NumPy arrays :

Slicing means retrieving elements from one index to another index. All we have to do is to pass the starting and ending point in the index like this ([start: end]) and we can incorporate all this into a single index would look something like this: [start:end:step-size].

```
a = np.array([1,2,3,4,5,6])  
print(a[1:5:2])
```

Output:

```
[2 4]
```

If we don't specify the start or end index, it is taken as 0 or array size, respectively, as default. And the step-size by default is 1.

```
a = np.array([1,2,3,4,5,6])  
print(a[:6:2])
```

```
print(a[1::2])
print(a[1:6:])
```

Output:

```
[1 3 5]
[2 4 6]
[2 3 4 5 6]
```

b) Slicing 2-D NumPy arrays :

```
a = np.array([[1,2,3],
               [4,5,6]])
print(a[0,0])
print(a[1,2])
print(a[1,0])
```

Output:

```
1
6
4
```

So, to slice a 2-D array, we need to mention the slices for both, the row and the column:

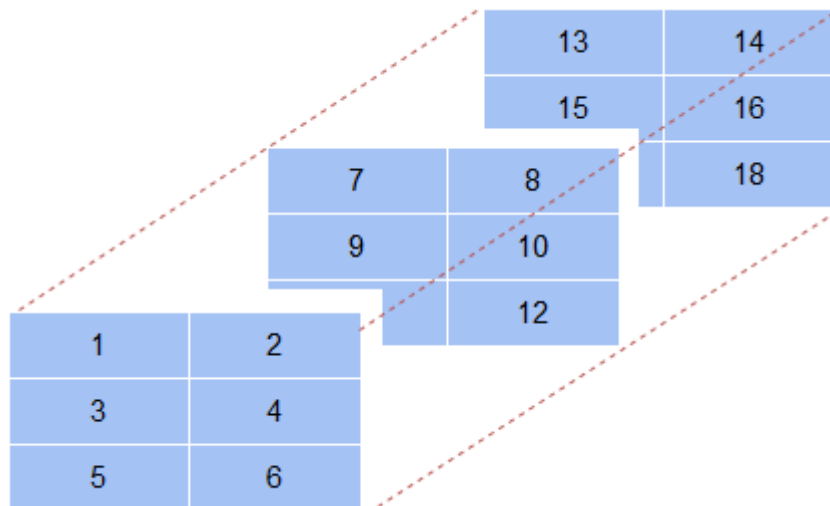
```
a = np.array([[1,2,3],[4,5,6]])
# print first row values
print('First row values :','\n',a[0:1,:])
# with step-size for columns
print('Alternate values from first row:','\n',a[0:1,::2])
#
print('Second column values :','\n',a[:,1::2])
print('Arbitrary values :','\n',a[0:1,1:3])
```

Output:

```
First row values :
[[1 2 3]]
Alternate values from first row:
[[1 3]]
Second column values :
[[2]
 [5]]
```

Arbitrary values :
[[2 3]]

c) Slicing 3-D NumPy arrays :



```
a = np.array([[[1,2],[3,4],[5,6]],# first axis array  
              [[7,8],[9,10],[11,12]],# second axis array  
              [[13,14],[15,16],[17,18]])# third axis array  
# 3-D array  
print(a)
```

Output:[[[1 2]

```
 [ 3  4]  
 [ 5  6]]
```

```
[[ 7  8]  
 [ 9 10]  
 [11 12]]
```

```
[[13 14]  
 [15 16]  
 [17 18]]]
```

d) Negative slicing of NumPy arrays :

Negative slicing prints elements from the end rather than the beginning.

```
a = np.array([[1,2,3,4,5],  
[6,7,8,9,10]])  
print(a[:,-1])
```

Output:

```
[ 5 10]
```

Explicitly providing negative slicing:

```
print(a[:,-1:-3:-1])
```

Output:

```
[[ 5  4]  
[10  9]]
```

EXPERIMENT-4

AIM:- Performing following operations using pandas

- a) Creating dataframe
- b) Concat()
- c) Setting conditions
- d) Adding a new column

PROCEDURE:-

a) Creating dataframe :

Creating a small sample dataset to try out various operations with Pandas. We shall create a Football data frame that stores the record of 4 players each from Euro Cup 2020's finalists – England and Italy.

```
import pandas as pd
# Create team data
data_england = {'Name': ['Kane', 'Sterling', 'Saka', 'Maguire'],
                'Age': [27, 26, 19, 28]}
data_italy = {'Name': ['Immobile', 'Insigne', 'Chiellini',
                      'Chiesa'], 'Age': [31, 30, 36, 23]}

# Create Dataframe
df_england = pd.DataFrame(data_england)
df_italy = pd.DataFrame(data_italy)
```

The England data frame looks something like this

Output:

	Name	Age
0	Kane	27
1	Sterling	26
2	Saka	19
3	Maguire	28

b) Concat() :

Concatenating of two data frames. The word “concatenate” means to “link together in series”. We do this by implementing the concat() function.

```
frames = [df_england, df_italy]  
both_teams = pd.concat(frames)  
both_teams
```

The result looks something like this:

Output:

	Name	Age
0	Kane	27
1	Sterling	26
2	Saka	19
3	Maguire	28
0	Immobile	31
1	Insigne	30
2	Chiellini	36
3	Chiesa	23

g) Setting conditions :

Conditional statements basically define conditions for data frame columns. There may be situations where we have to filter out various data by applying certain column conditions (numeric or non-numeric). For eg: In an Employee data frame, we might have to list out a bunch of people whose salary is more than Rs. 50000. Also, we might want to filter the people who live in New Delhi, or whose name starts with "A". Let's see a hands-on example.

Imagine we want to filter experienced players from our squad. Let's say, we want to filter those players whose age is greater than or equal to 30. In such case, try doing:

```
both_teams[both_teams["Age"] >= 30]
```

Output:

	Name	Age
0	Immobile	31
1	Insigne	30
2	Chiellini	36

h) Adding a new column :

Adding more data to our df_england data frame.

```
club = ['Tottenham', 'Man City', 'Arsenal', 'Man Utd']  
# 'Associated Club' is our new column name  
df_england['Associated Clubs'] = club  
df_england
```

This will add a new column 'Associated Club' to England's data frame.

Output:

	Name	Age	Associated Clubs
0	Kane	27	Tottenham
1	Sterling	26	Man City
2	Saka	19	Arsenal
3	Maguire	28	Man Utd

Repeat implementing the concat function after updating the data for England.

```
frames = [df_england, df_italy]  
both_teams = pd.concat(frames)  
both_teams
```

Output:

	Name	Age	Associated Clubs
0	Kane	27	Tottenham
1	Sterling	26	Man City
2	Saka	19	Arsenal
3	Maguire	28	Man Utd
0	Immobile	31	NaN
1	Insigne	30	NaN
2	Chiellini	36	NaN
3	Chiesa	23	NaN

EXPERIMENT-5

AIM:- Read the following file formats using pandas

- a) Text files
- b) CSV() files
- c) Excel files
- d) JSON files

PROCEDURE:-

a) Text Files :

Text files are one of the most common file formats to store data. Python makes it very easy to read data from text files.

Python provides the **open()** function to read files that take in the file path and the file access mode as its parameters. For reading a text file, the file access mode is 'r'. We have mentioned the other access modes below:

- 'w' – writing to a file
- 'r+' or 'w+' – read and write to a file
- 'a' – appending to an already existing file
- 'a+' – append to a file after reading

Python provides us with three functions to read data from a text file:

1. **read(n)** – This function reads n bytes from the text files or reads the complete information from the file if no number is specified. It is smart enough to handle the delimiters when it encounters one and separates the sentences.
2. **readline(n)** – This function allows we to read n bytes from the file but not more than one line of information.
3. **readlines()** – This function reads the complete information in the file but unlike **read()**, it doesn't bother about the delimiting character and prints them as well in a list format

Let us see how these functions differ in reading a text file:

```
# read text file

with open(r'./Importing files/Analytics Vidhya.txt','r') as f:

    print(f.read())
```

Output:

```
Welcome to an article on how to import files in Python. We'll work with the following types of files:
1. Text
2. CSV
3. Excel
4. SQL
5. Web data
6. Image
Enjoy the read!
```

The **read()** function imported all the data in the file in the correct structured form.

By providing a number in the **read()** function, we were able to extract the specified amount of bytes from the file.

```
# read text file

with open(r'./Importing files/Analytics Vidhya.txt','r') as f:
```

Output:

```
Welcome to an article on how to import files in Python. We'll work with the following types of files:
```

Using **readline()**, only a single line from the text file was extracted.

```
# read text file

with open(r'./Importing files/Analytics Vidhya.txt','r') as f:

    print(f.readline())
```

Output:

```
["Welcome to an article on how to import files in Python. We'll work with the following types of files:\n", '1. Text\n', '2. CSV\n', '3. Excel\n', '4. SQL\n', '5. Web data\n', '6. Image\n', 'Enjoy the read!']
```

Here, the **readline()** function extracted all the text file data in a list format.

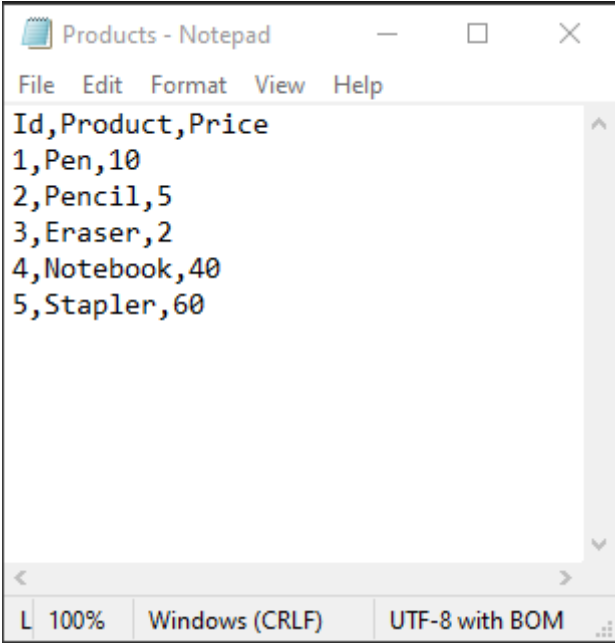
b) CSV Files :

Ah, the good old CSV format. A CSV (or Comma Separated Value) file is the most common type of file that a data scientist will ever work with. These files use a “,” as a delimiter to separate the values and each row in a CSV file is a data record.

These are useful to transfer data from one application to another and is probably the reason why they are so commonplace in the world of data science.

If we look at them in the Notepad, you will notice that the values are separated by commas:

Output:



```
Products - Notepad
File Edit Format View Help
Id,Product,Price
1,Pen,10
2,Pencil,5
3,Eraser,2
4,Notebook,40
5,Stapler,60
L 100% Windows (CRLF) UTF-8 with BOM
```

The screenshot shows a Notepad window with a menu bar (File, Edit, Format, View, Help) and a text area containing a CSV file. The file has 6 rows: a header row 'Id,Product,Price' and five data rows. The data rows are: '1,Pen,10', '2,Pencil,5', '3,Eraser,2', '4,Notebook,40', and '5,Stapler,60'. The status bar at the bottom shows 'L 100% Windows (CRLF) UTF-8 with BOM'.

The [Pandas](#) library makes it very easy to read CSV files using the `read_csv()` function:

But CSV can run into problems if the values contain commas. This can be overcome by using different delimiters to separate information in the file, like '\t' or ';', etc. These can also be imported with the `read_csv()` function by specifying the delimiter in the parameter value as shown below while reading a TSV (Tab Separated Values) file:

```
import pandas as pd
```

```
df = pd.read_csv(r'./Importing files/Employee.txt', delimiter='\t')
```

```
Df
```

Output:

	Id	Name	Job
0	1	Raju	Full Stack
1	2	Shyam	Frontend
2	3	Ghanshyam	Backend
3	4	Radheshyam	Data Science

c) Excel Files :

We will be quite familiar with Excel files and why they are so widely used to store tabular data. So I'm going to jump right to the code and import an Excel file in Python using Pandas.

Pandas has a very handy function called `read_excel()` to read Excel files:

```
# read Excel file into a DataFrame
```

```
df = pd.read_excel(r'./Importing files/World_city.xlsx')

# print values

Df
```

Output:

	Id	City	Country
0	1	Delhi	India
1	2	Tokyo	Japan
2	3	Thimpu	Bhutan
3	4	Kathmandu	Nepal

But an Excel file can contain multiple sheets, right? So how can we access them?

For this, we can use the Pandas' **ExcelFile()** function to print the names of all the sheets in the file:

```
# read Excel sheets in pandas

xl = pd.ExcelFile(r'./Importing files/World_city.xlsx')

# print sheet name

xl.sheet_names
```

Output:

```
['Asia', 'Europe', 'Australia']
```

After doing that, we can easily read data from any sheet we wish by providing its name in the **sheet_name** parameter in the **read_excel()** function:

```
# read Europe sheet

df = pd.read_excel(r'./Importing files/World_city.xlsx', sheet_name='Europe')

df
```

Output:

	Id	City	Country
0	1	Vienna	Austria
1	2	Stockholm	Sweden
2	3	Copenhagen	Denmark

d) JSON Files :

JSON (JavaScript Object Notation) files are lightweight and human-readable to store and exchange data. It is easy for machines to parse and generate these files and are based on the JavaScript programming language.

JSON files store data within {} similar to how a dictionary stores it in Python. But their major benefit is that they are language-independent, meaning they can be used with any programming language – be it Python, C or even Java!

This is how a JSON file looks:

```
{"Name": {"0": "Aniruddha", "1": "Jill"}, "Company": {"0": "Analytics Vidhya", "1": "Google"}, "Job": {"0": "Intern", "1": "Full time"}}
```

Python provides a **json** module to read JSON files. We can read JSON files just like simple text files. However, the read function, in this case, is replaced by **json.load()** function that returns a JSON dictionary.

Once we have done that, we can easily convert it into a Pandas dataframe using the **pandas.DataFrame()** function:

```
import json

# open json file

with open('./Importing files/sample_json.json','r') as file:

    data = json.load(file)

# json dictionary

print(type(data))

# loading into a DataFrame

df_json = pd.DataFrame(data)

df_json
```


Output:

```
<class 'dict'>
```

	Name	Company	Job
0	Aniruddha	Analytics Vidhya	Intern
1	Jill	Google	Full time

But we can even load the JSON file directly into a dataframe using the `pandas.read_json()` function as shown below:

```
# reading directly into a DataFrame using pd.read_json()

path = './Importing files/sample_json.json'

df = pd.read_json(path)

df
```

Output:

	Name	Company	Job
0	Aniruddha	Analytics Vidhya	Intern
1	Jill	Google	Full time

EXPERIMENT-6

AIM:- Performing following visualizations using matplotlib

- a) Bar Graph
- b) Pie Chart
- c) Box Plot
- d) Histogram
- e) Line Chart and Subplots
- f) Scatter Plot

PROCEDURE:-

a) Bar Graph :

First, we want to find the most popular food item that customers have bought from the company.

I will be using the Pandas ***pivot_table*** function to find the total number of orders for each category of the food item:

```
table = pd.pivot_table(data=df, index='category', values='num_orders', aggfunc=np.sum)
```

```
Table
```

Output:

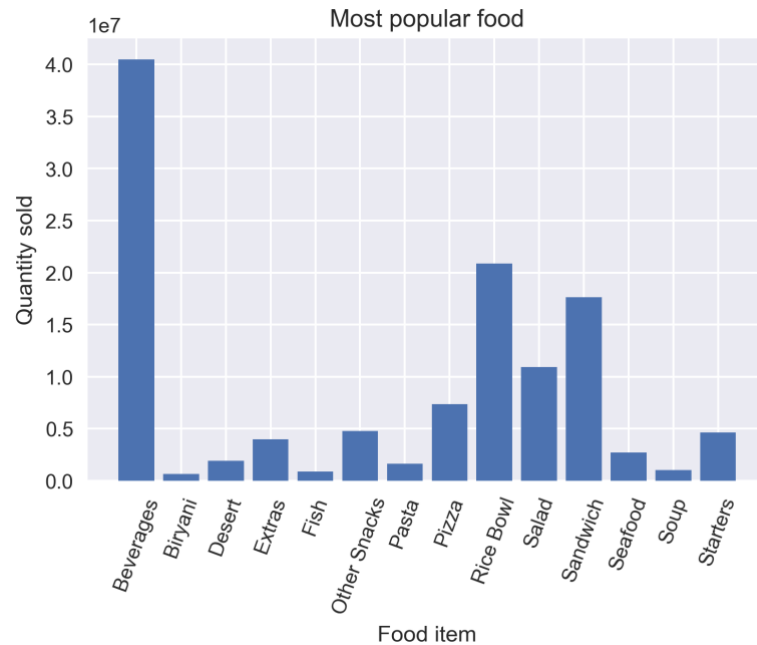
	num_orders
category	
Beverages	40480525
Biryani	631848
Desert	1940754
Extras	3984979
Fish	871959
Other Snacks	4766293
Pasta	1637744
Pizza	7383720
Rice Bowl	20874063
Salad	10944336
Sandwich	17636782
Seafood	2715714
Soup	1039646
Starters	4649122

Next, I will try to visualize this using a bar graph.

Bar graphs are best used when we need to compare the quantity of categorical values within the same category.

Bar graph is generated using ***plt.bar()*** in matplotlib:

Python Code:



It is always important to label our axis. We can do this by employing the `plt.xlabel()` and `plt.ylabel()` functions. We can use `plt.title()` for naming the title of the plot. If your xticks are overlapping, rotate them using the `rotate` parameter in `plt.xticks()` so that they are easy to view for the audience.

We can save your plot using the `plt.savefig()` function by providing the file path as a parameter. Finally, always display your plot using `plt.show()`.

While analyzing the plot, we can see that **Beverages** were the most popular food item sold by the company.

Let's divide the total food item order by the number of unique meals it is present in.

```

#dictionary for meals per food item

item_count = {}

for i in range(table.index.nunique()):

    item_count[table.index[i]] =
table.num_orders[i]/df_meal[df_meal['category']==table.index[i]].shape[0]

#bar plot

plt.bar([x for x in item_count.keys()], [x for x in
item_count.values()],color='orange')

#adjust xticks

plt.xticks(rotation=70)

#label x-axis

plt.xlabel('Food item')

#label y-axis

plt.ylabel('No. of meals')

#label the plot

plt.title('Meals per food item')

#save plot

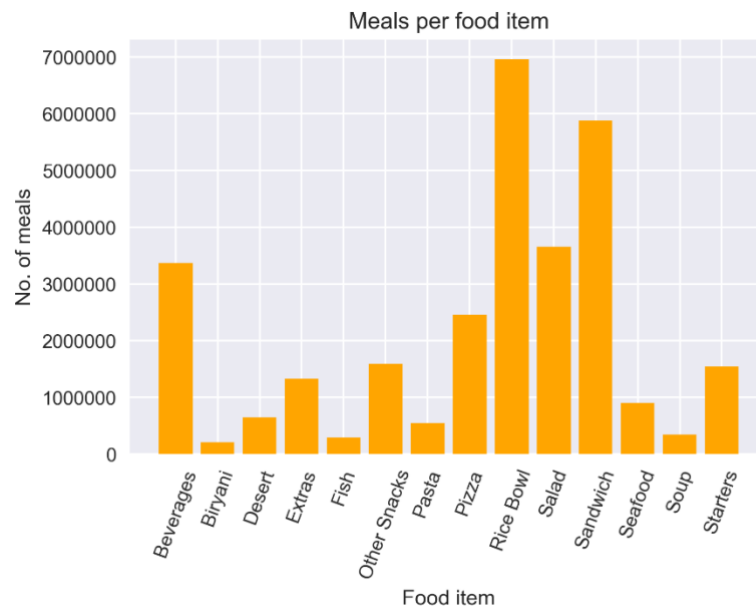
plt.savefig('C:\\Users\\Dell\\Desktop\\AV Plotting
images\\matplotlib_plotting_7.png',dpi=300,bbox_inches='tight')

#display plot

plt.show();

```

Output:



Yes, our hypothesis was correct! **Rice Bowl** was indeed the most popular food item sold by the company.

Bar graphs should not be used for continuous values.

b) Pie Chart :

Now see the ratio of orders from each cuisine.

A pie chart is suitable to show the proportional distribution of items within the same category.

```
#dictionary for cuisine and its total orders
```

```
d_cuisine = {}
```

```

#total number of order

total = df['num_orders'].sum()

#find ratio of orders per cuisine

for i in range(df['cuisine'].nunique()):

#cuisine

c = df['cuisine'].unique()[i]

#num of orders for the cuisine

c_order = df[df['cuisine']==c]['num_orders'].sum()

d_cuisine[c] = c_order/total

```

Let's plot the pie chart:

```

#pie plot

plt.pie([x*100 for x in d_cuisine.values()],labels=[x for x in
d_cuisine.keys()],autopct='%0.1f',explode=[0,0,0.1,0])

#label the plot

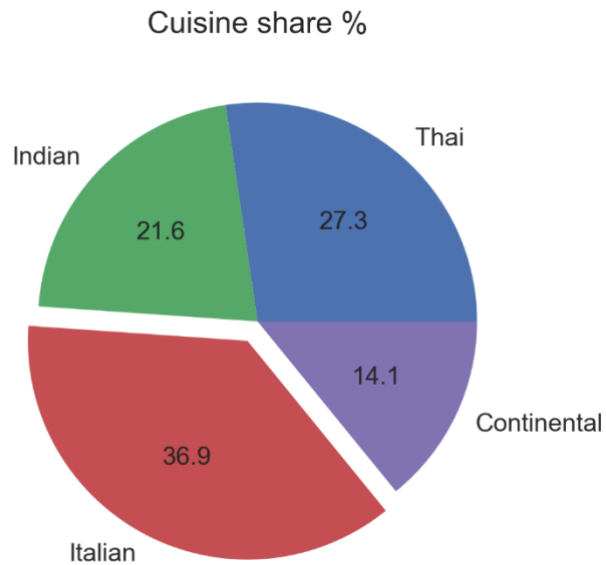
plt.title('Cuisine share %')

plt.savefig('C:\\Users\\Dell\\Desktop\\AV Plotting
images\\matplotlib_plotting_8.png',dpi=300,bbox_inches='tight')

plt.show();

```

Output:



- I used ***plt.pie()*** to draw the pie chart and adjust its parameters to make it more appealing
- The ***autopct*** parameter was used to print the values within the pie chart up to 1 decimal place
- The ***explode*** parameter was used to offset the Italian wedge to make it stand out from the rest. This makes it instantly clear to the viewer that people love Italian food!

A pie chart is rendered useless when there are a lot of items within a category. This will decrease the size of each slice and there will be no distinction between the items.

c) Box Plot :

Since we are discussing cuisine, let's check out which one is the most expensive cuisine! For this, I will be using a **Box Plot**.

Box plot gives statistical information about the distribution of numeric data divided into different groups. It is useful for detecting outliers within each group.

- The lower, middle and upper part of the box represents the **25th, 50th, and 75th percentile** values respectively
- The top whisker represents **$Q3 + 1.5 * IQR$**
- The bottom whisker represents **$Q1 - 1.5 * IQR$**
- Outliers are shown as scatter points
- Shows skewness in the data

```
#dictionary for base price per cuisine

c_price = {}

for i in df['cuisine'].unique():

    c_price[i] = df[df['cuisine']==i].base_price
```

Plotting the boxplot below:

```
#plotting boxplot

plt.boxplot([x for x in c_price.values()],labels=[x for x in c_price.keys()])

#x and y-axis labels

plt.xlabel('Cuisine')

plt.ylabel('Price')

#plot title

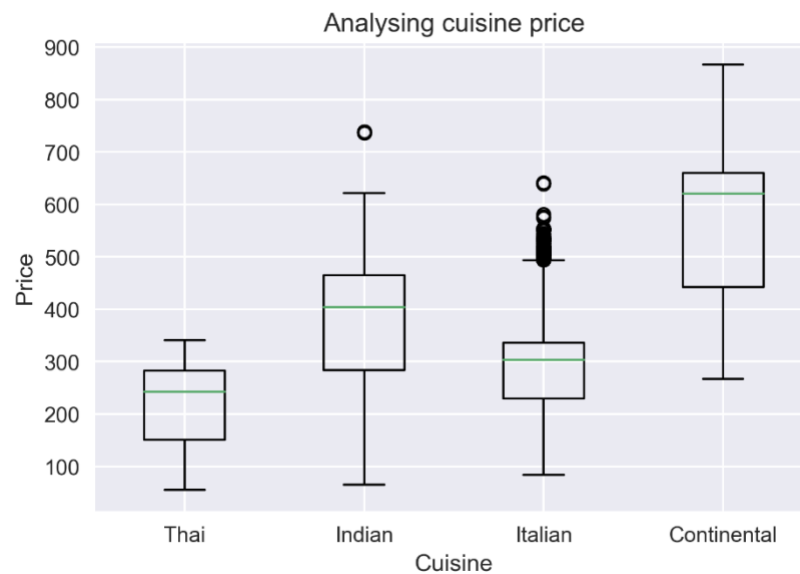
plt.title('Analysing cuisine price')

#save and display
```

```
plt.savefig('C:\\Users\\Dell\\Desktop\\AV Plotting
images\\matplotlib_plotting_9.png',dpi=300,bbox_inches='tight')

plt.show();
```

Output:



Continental cuisine was the most expensive cuisine served by the company! Even its median price is higher than the maximum price of all the cuisines.

Box plot does not show the distribution of data points within each group.

d) Histogram :

A histogram shows the distribution of numeric data through a continuous interval by segmenting data into different bins. Useful for inspecting skewness in the data.

Since **base_price** is a continuous variable, we will inspect its range in different distinct orders using a histogram. We can do this using **plt.hist()**.

But the confusing part is what should be the number of bins? By default, it is 10. However, there is no correct answer and we can vary it according to your dataset to best visualize it.

```
#plotting histogram

plt.hist(df['base_price'],rwidth=0.9,alpha=0.3,color='blue',bins=15,edgecolor='red'
)

#x and y-axis labels

plt.xlabel('Base price range')

plt.ylabel('Distinct order')

#plot title

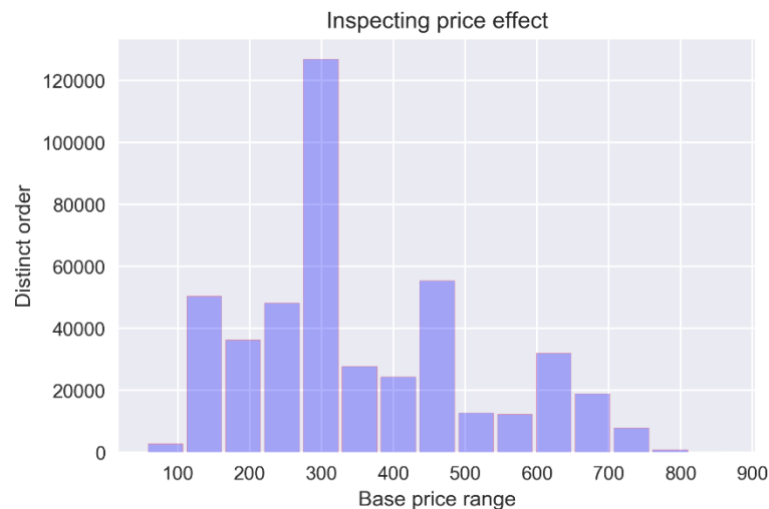
plt.title('Inspecting price effect')

#save and display the plot

plt.savefig('C:\\Users\\Dell\\Desktop\\AV Plotting
images\\matplotlib_plotting_10.png',dpi=300,bbox_inches='tight')

plt.show();
```

Output:



We have chosen the number of bins as 15 and it is evident that most of the orders had a base price of ~300.

It is easy to confuse histograms with bar plots. But remember, histograms are used with continuous data whereas bar plots are used with categorical data.

e) Line Plot and Subplots :

A line plot is useful for visualizing the trend in a numerical value over a continuous time interval.

How are the weekly and monthly sales of the company varying? This is a critical business question that makes or breaks the marketing strategy.

Before exploring that, we will create two lists for storing the week-wise and month-wise revenue of the company:

```
#new revenue column
```

```
df['revenue'] = df.apply(lambda x: x.checkout_price*x.num_orders,axis=1)
```

```
#new month column
```

```
df['month'] = df['week'].apply(lambda x: x//4)
```

```
#list to store month-wise revenue
```

```
month=[]
```

```
month_order=[]
```

```
for i in range(max(df['month'])):
```

```
    month.append(i)
```

```
    month_order.append(df[df['month']==i].revenue.sum())
```

```
#list to store week-wise revenue
```

```
week=[]
```

```
week_order=[]
```

```
for i in range(max(df['week'])):
```

```
    week.append(i)
```

```
    week_order.append(df[df['week']==i].revenue.sum())
```

We will compare the revenue of the company in every week as well as in every month using two line-plots drawn side by side. We will be using the ***plt.subplots()*** function.

Matplotlib subplots makes it easy to view and compare different plots in the same figure.

To understand how this function works, we need to know what ***Figure***, ***Axes***, and ***Axis*** are in a matplotlib plot.

Figure is the outermost container for the Matplotlib plot(s). There can a single or multiple plots, called **Axes**, within a **Figure**. Each of these Axes contains the x and y-axis known as the **Axis**.

The `plt.subplots()` figure returns the figure and axes. We can provide as an input to the function how we want to display the axes within the figure. These will be adjusted using the **nrows** and **ncols** parameters. We can even adjust the size of the figure using the **figsize** parameter.

Axes are returned as a list. To plot for specific axes, we can access them as a list object. The rest of the plotting is done the same way as simple plots:

```
#subplots returns a Figure and an Axes object

fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))

#manipulating the first Axes

ax[0].plot(week,week_order)

ax[0].set_xlabel('Week')

ax[0].set_ylabel('Revenue')

ax[0].set_title('Weekly income')

#manipulating the second Axes

ax[1].plot(month,month_order)

ax[1].set_xlabel('Month')

ax[1].set_ylabel('Revenue')

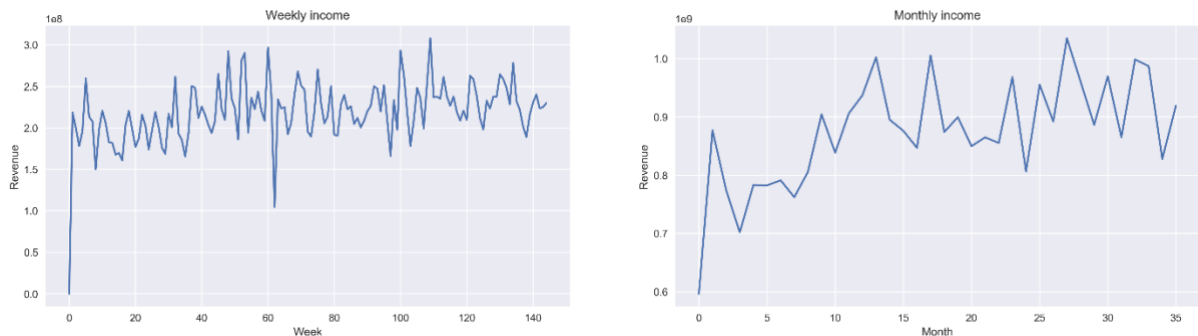
ax[1].set_title('Monthly income')
```

```
#save and display the plot

plt.savefig('C:\\Users\\Dell\\Desktop\\AV Plotting
images\\matplotlib_plotting_11.png',dpi=300,bbox_inches='tight')

plt.show();
```

Output:



We can see an increasing trend in the number of food orders with the number of weeks and months, though the trend is not very strong.

f) Scatter Plot :

Finally, we will try to analyze whether the center type had any effect on the number of orders from different center types. We will do this by comparing a scatter plot, a boxplot and a bar graph in the same figure.

We have already seen the use of boxplots and bar graphs, but scatter plots have their own advantages.

Scatter plots are useful for showing the relationship between two variables.

Any correlation between variables or outliers in the data can be easily spotted using scatter plots.

```
center_type_name = ['TYPE_A', 'TYPE_B', 'TYPE_C']
```

```

#relation between op area and number of orders

op_table=pd.pivot_table(df,index='op_area',values='num_orders',aggfunc=np.sum)


#relation between center type and op area

c_type = {}

for i in center_type_name:

    c_type[i] = df[df['center_type']==i].op_area


#relation between center type and num of orders

center_table=pd.pivot_table(df,index='center_type',values='num_orders',aggfunc=np.sum)


#subplots

fig,ax = plt.subplots(nrows=3,ncols=1,figsize=(8,12))


#scatter plots

ax[0].scatter(op_table.index,op_table['num_orders'],color='pink')

ax[0].set_xlabel('Operation area')

ax[0].set_ylabel('Number of orders')

ax[0].set_title('Does operation area affect num of orders?')

ax[0].annotate('optimum operation area of 4
km^2',xy=(4.2,1.1*10**7),xytext=(7,1.1*10**7),arrowprops=dict(facecolor='black',
shrink=0.05),fontsize=12)


#boxplot

ax[1].boxplot([x for x in c_type.values()], labels=[x for x in c_type.keys()])

ax[1].set_xlabel('Center type')

ax[1].set_ylabel('Operation area')

```



```
ax[1].set_title('Which center type had the optimum operation area?')

#bar graph

ax[2].bar(center_table.index,center_table['num_orders'],alpha=0.7,color='orange',width=0.5)

ax[2].set_xlabel('Center type')

ax[2].set_ylabel('Number of orders')

ax[2].set_title('Orders per center type')

#show figure

plt.tight_layout()

plt.savefig('C:\\Users\\Dell\\Desktop\\AV Plotting
images\\matplotlib_plotting_12.png',dpi=300,bbox_inches='tight')

plt.show();
```

Output:

