1. import os

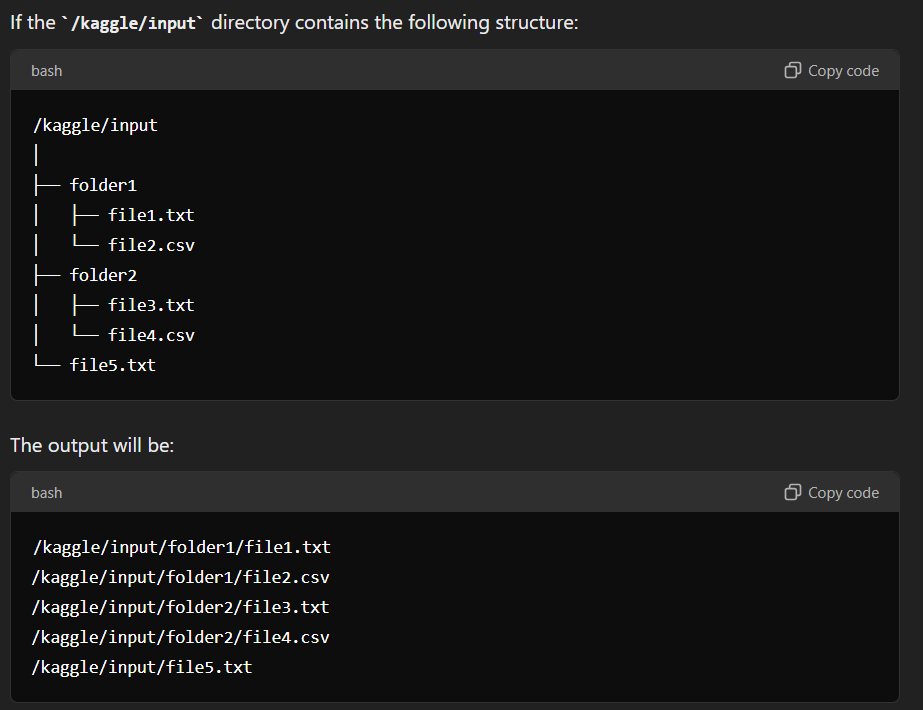
for dirname, \_, filenames in os.walk('/kaggle/input'):

for filename in filenames:

print(os.path.join(dirname, filename))

This code snippet is designed to recursively traverse through all the directories and files starting from the root directory '/kaggle/input', printing out the full path of each file it encounters. Here's a breakdown of how it works:

* os.walk('/kaggle/input'): This generates the file names in a directory tree by walking either top-down or bottom-up through the directory. It yields a 3-tuple (dirpath, dirnames, filenames):
  + dirpath: The path to the current directory.
  + dirnames: A list of the names of the subdirectories in dirpath.
  + filenames: A list of the names of the non-directory files in dirpath.
* for dirname, \_, filenames in os.walk('/kaggle/input'):: Iterates over each directory (dirname) and its files (filenames) in the directory tree.
* for filename in filenames:: Iterates over each file in the list of filenames.
* print(os.path.join(dirname, filename)): Joins the directory path (dirname) with the file name (filename) to create the full path, which is then printed.



1. import plotly.offline as py

import plotly.graph\_objects as go

from plotly.offline import init\_notebook\_mode, iplot

from plotly import tools

import plotly.figure\_factory as ff

import plotly.express as px

import matplotlib.pyplot as plt # Import pyplot for Matplotlib styling

plt.style.use('fivethirtyeight') # Apply the 'fivethirtyeight' style

# Initialize Plotly in offline mode within a notebook

init\_notebook\_mode(connected=True)

The code snippet you provided includes the importation and initialization of Plotly, along with an attempt to set a specific plotting style using Matplotlib. Here’s a breakdown of the code:

Plotly Imports and Initialization

* import plotly.offline as py: This imports Plotly's offline mode under the alias py, allowing you to work with Plotly in environments without requiring an internet connection (e.g., Jupyter notebooks).
* import plotly.graph\_objects as go: This imports the graph\_objects module, which provides the building blocks for creating plots, such as traces, layout, and figures.
* from plotly.offline import init\_notebook\_mode, iplot:
  + init\_notebook\_mode(connected=True): Initializes Plotly to run in offline mode within a Jupyter notebook.
  + iplot: A function for rendering plots within the notebook.
* import plotly.figure\_factory as ff: Imports the figure\_factory module, which provides specialized plot types (e.g., dendrograms, 2D density plots).
* import plotly.express as px: Imports Plotly Express, a high-level interface for creating simple, expressive figures.

Matplotlib Styling

* plt.style.use('fivethirtyeight'):
  + Purpose: This line attempts to apply a Matplotlib style, specifically the 'fivethirtyeight' style, to your plots.
  + Issue: The plt object isn't imported in this snippet. To use this line, you need to import Matplotlib's pyplot module as plt.

Explanation of plt.style.use('fivethirtyeight')

* fivethirtyeight style: This is a popular style that emulates the visual aesthetic of the graphics seen on the FiveThirtyEight website. It’s known for its clean and modern look, with muted color tones and larger font sizes.

This setup allows you to create visualizations using Plotly with an overall styling influence from Matplotlib, although the styling would mainly affect any plots created using Matplotlib, not directly influencing Plotly visualizations.

1. df = pd.read\_csv('Delhi Accident Data.csv',low\_memory = False)

df.head()

By setting low\_memory=False when reading the CSV file, Pandas reads the file in a single pass rather than in chunks. This can help resolve the DtypeWarning by allowing Pandas to infer the correct data types for each column more accurately.

* df = pd.read\_csv('Delhi Accident Data.csv', low\_memory=False): Reads the 'Delhi Accident Data.csv' file into a DataFrame named df. By setting low\_memory=False, Pandas processes the file more slowly but more accurately in terms of data type inference.
* df.head(): Displays the first 5 rows of the DataFrame df, allowing you to quickly inspect the data.

df = df.drop(["Unnamed: 7","Unnamed: 8","Unnamed: 9"], axis=1)

* df.drop(): The drop() function is used to remove specified labels from rows or columns.
* ["Unnamed: 7", "Unnamed: 8", "Unnamed: 9"]: This is a list of column names that you want to remove from the DataFrame. The columns appear to be unnamed and are given default names by Pandas when the CSV file is read.
* axis=1: The axis parameter determines whether to drop rows or columns. axis=1 specifies that columns should be dropped. If you wanted to drop rows instead, you would use axis=0.
* df = ...: This line reassigns the DataFrame df after the specified columns have been removed.

Why This Might Be Necessary

The columns named "Unnamed: 7", "Unnamed: 8", and "Unnamed: 9" might exist due to extra commas or separators in the CSV file, leading to empty or unnecessary columns being created. Dropping these columns cleans up your DataFrame by removing unwanted data.

k= pd.DataFrame()

k['df']= df.isnull().sum()

k.T

1. **df.isnull().sum()**:

* df.isnull() creates a DataFrame of the same shape as df, but with True where the values are NaN and False elsewhere.
* .sum() applied to this DataFrame counts the number of True values (i.e., the number of NaN values) in each column.

1. **k = pd.DataFrame()**:

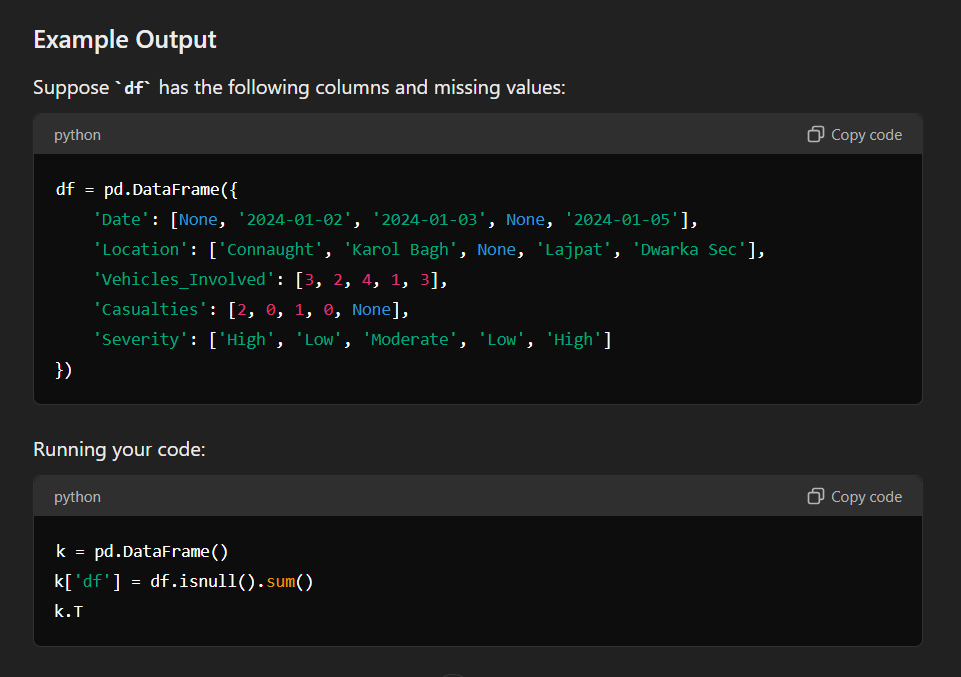
* Initializes an empty DataFrame k.

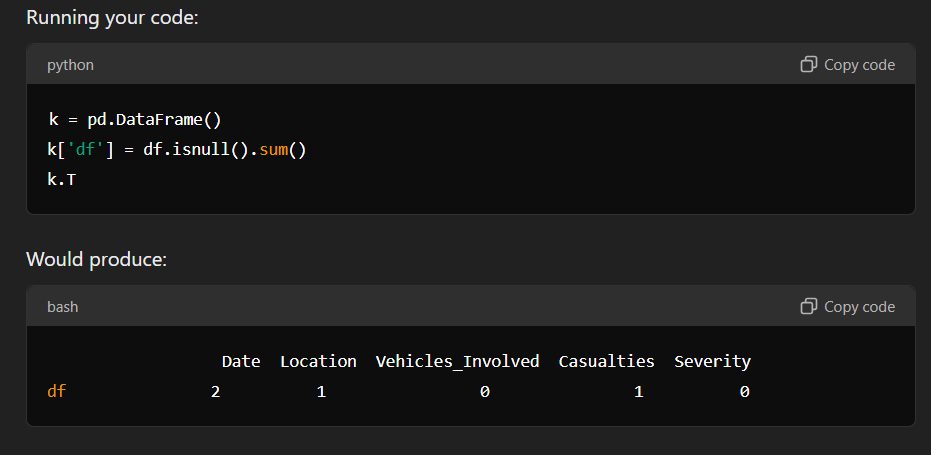
1. **k['df'] = df.isnull().sum()**:

* Stores the count of missing values from each column of df in a new column called 'df' in the DataFrame k.

1. **k.T**:

* Transposes the DataFrame k, swapping its rows and columns. This step is usually done for better readability, making the column names appear as rows and vice versa.





This table shows the number of missing values in each column, with the column names listed horizontally for easier interpretation.

**Use Cases**

This kind of summary is useful for quickly assessing data quality, helping you identify columns with missing data that might require cleaning or imputation.

1. df.shape

output: (75748, 7)

75748 : The number of rows in the DataFrame.

**7** : The number of columns in the DataFrame.

1. victim=['PEDESTRIAN', 'S/C & M/C', 'CAR']

year=[2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017]

districts=["SOUTH EAST DELHI",

"NEW DELHI",

"NORTH DELHI(ROHINI)",

"CENTRAL DELHI",

"SOUTH WEST DELHI",

"WEST DELHI",

"NORTH WEST DELHI",

"EAST DELHI",

"SHAHDARA",

"SOUTH DELHI",

"NORTH EAST DELHI",

"UNK",

"OUTER"]

vehichle\_at\_fault=['PVT CAR', 'UNKNOWN', 'S/C&M/C','HTV/GDS']

Defining a list. These lists can be useful for filtering, grouping, or analyzing accident data by different categories. For example, you could use them to categorize data, create visualizations, or perform statistical analysis based on victims, years, districts, or vehicles at fault.

1. Probability\_of\_fatalities=[0.2780847145,0.2798742138,0.3129689175,0.3182332956,0.283847981,0.2616136919,0.1974584555,0.1856710394,0.2284730195,0.2301480485]

fig = go.Figure()

fig.add\_trace(go.Scatter(x=year, y=Probability\_of\_fatalities, name="NEW DELHI", line=dict(color='firebrick', width=1,dash='dash')))

fig.update\_layout(title='Fatality rate across the different Districts across the Given Year Given district',

xaxis\_title='Years',

yaxis\_title='Probability of Fality')

fig.show()

1. **Probability\_of\_fatalities**: This list contains the probabilities of fatalities for each year.

Probability\_of\_fatalities = [ 0.2780847145, 0.2798742138, 0.3129689175, 0.3182332956, 0.283847981,0.2616136919, 0.1974584555, 0.1856710394, 0.2284730195, 0.2301480485]

1. **fig = go.Figure()**: Initializes an empty figure object.
2. **Adding a Trace**:
   * **fig.add\_trace(go.Scatter(...))**: Adds a trace (line) to the figure.
   * **x=year**: The x values represent the years.
   * **y=Probability\_of\_fatalities**: The y values represent the probability of fatalities.
   * **name="NEW DELHI"**: The name for this line, which will appear in the legend.
   * **line=dict(...)**: Defines the style of the line (color, width, and dash pattern).
3. **Updating the Layout**:
   * **fig.update\_layout(...)**: Sets the title and axis labels for the plot.
   * **title='Fatality rate across the different Districts across the Given Year Given district'**: Title of the plot.
   * **xaxis\_title='Years'**: Label for the x-axis.
   * **yaxis\_title='Probability of Fality'**: Label for the y-axis (typo: "Fality" should be "Fatality").
4. **Displaying the Figure**:
   * **fig.show()**: Renders the plot.

**Visualization Output**

The output will be a line graph with:

* **X-Axis**: Years from 2008 to 2017.
* **Y-Axis**: Probability of fatalities.
* **Line**: A dashed red line representing the probability of fatalities over the years in "NEW DELHI".

**Notes**

* You might want to correct the typo in the y-axis title from "Probability of Fality" to "Probability of Fatality".
* The graph can be extended by adding more traces if you want to compare probabilities across multiple districts.

The values for the Probability\_of\_fatalities that you provided seem to represent the probability of fatal accidents occurring in a specific district ("NEW DELHI") over a range of years (2008–2017). These probabilities could have been calculated or derived from the accident data you are working with.

**To Clarify:**

* If these values were provided to you (e.g., as part of a dataset or report), they might have been pre-calculated using one of the above methods.
* If you derived them yourself, they would likely have come from performing some analysis or calculations on accident data, such as counting fatal accidents relative to the total accidents for each year.

plt.ylim(0.05, 0.5)

The plt.ylim() function in Matplotlib is used to set the limits for the y-axis of a plot. By specifying plt.ylim(0.05, 0.5), you're instructing Matplotlib to display the y-axis values only between 0.05 and 0.5.