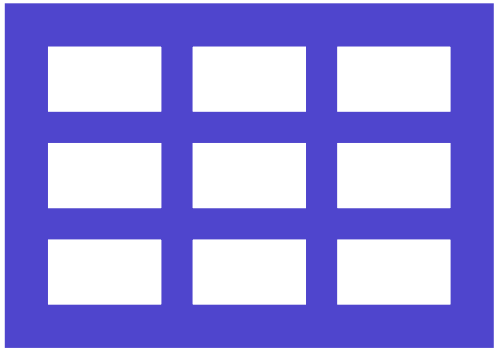
DATA EXPLORATION AND CLEANING USING PYTHON

Eng. Fatma Gamal

NumPy Arrays

- NumPy implements a data structure called the **N-dimensional** array or **ndarray**.
- ndarrays are **like lists** in that they contain a collection of items that **can be accessed via indexes.**
- ndarrays are **homogeneous**, meaning they can only contain **objects of the same type**.
- ndarrays can be **multi-dimensional**, making it easy to store **2-dimensional tables** or **matrices**.

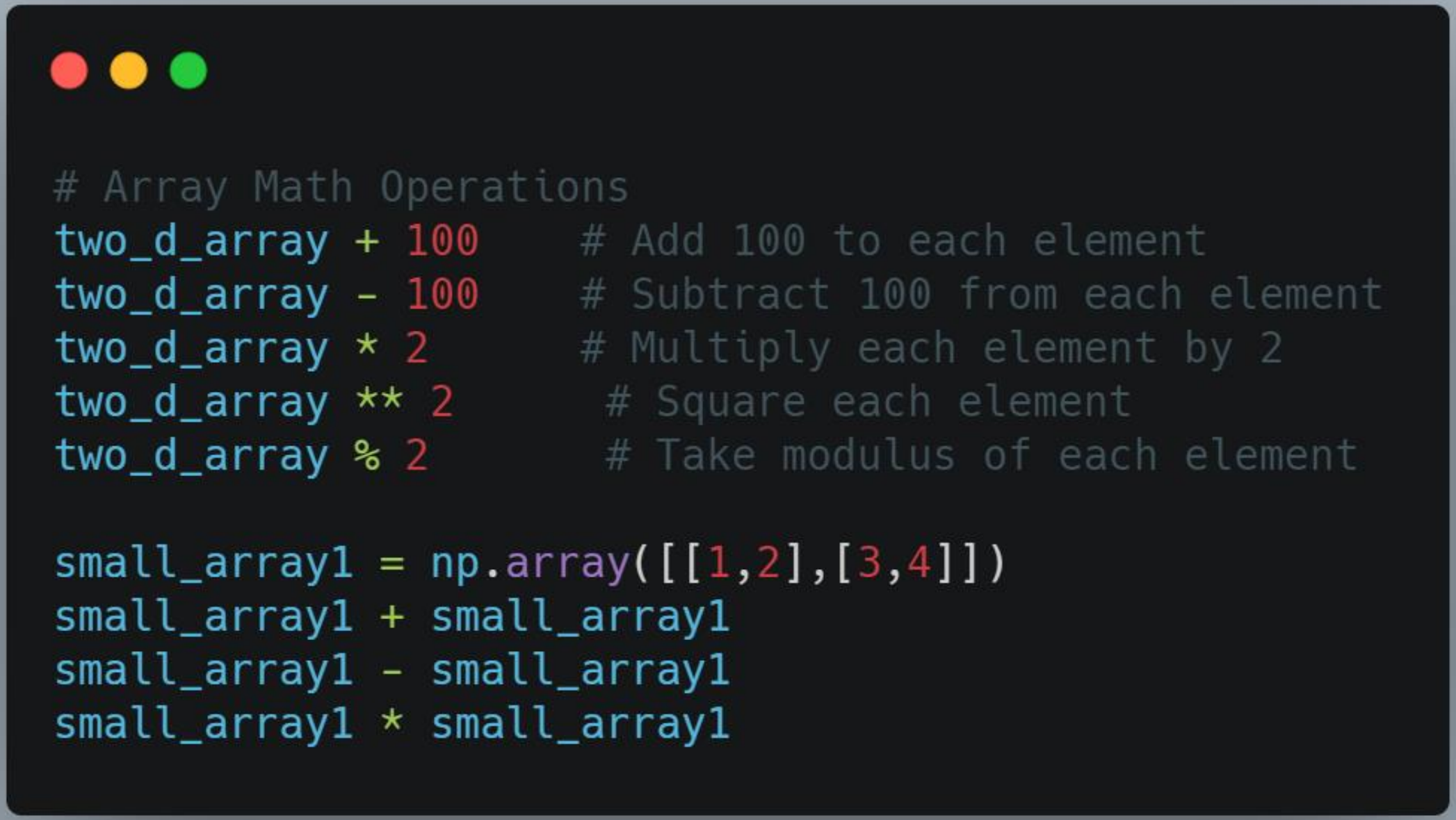


```python
import numpy as np

my_list = [1, 2, 3, 4]

my_array = np.array(my_list)

second_list = [5, 6, 7, 8]

two_d_array = np.array([my_list, second_list])

print(two_d_array)
```

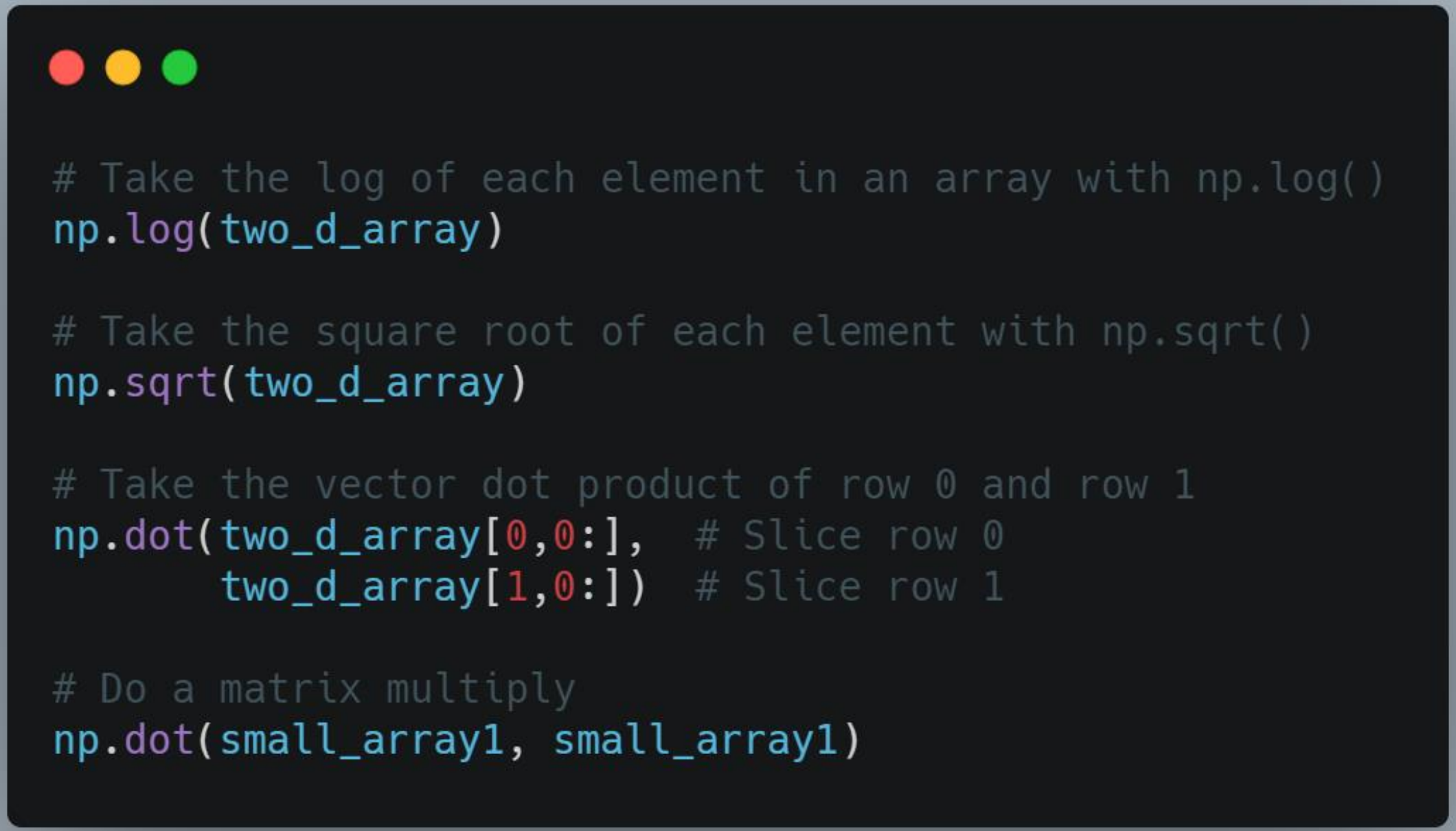
```python
two_d_array.shape

two_d_array.size

two_d_array.dtype

np.identity(n = 5)

np.eye(N = 3,  # Number of rows
       M = 5,  # Number of columns
       k = 1)  # Index of the diagonal (main diagonal (0) is default)

np.ones(shape= [2,4])

np.zeros(shape= [4,6])
```

```python
one_d_array = np.array([1,2,3,4,5,6])

one_d_array[3]          # Get the item at index 3

one_d_array[3:]         # Get a slice from index 3 to the end

one_d_array[::-1]       # Slice backwards to reverse the array

# Get the element at row index 1, column index 4
two_d_array[1, 4]

# Slice elements starting at row 2, and column 5
two_d_array[1:, 4:]

# Reverse both dimensions (180 degree rotation)
two_d_array[::-1, ::-1]
```

```python
#Reshaping Arrays
np.reshape(a=two_d_array,          # Array to reshape
           newshape=(6,3))         # Dimensions of the new array

np.ravel(a=two_d_array,
         order='C')           # Use C-style unraveling (by rows)

two_d_array.flatten()

two_d_array.T # Get the transpose of an array

# Flip an array vertically or horizontally
np.flipud(two_d_array)

np.fliplr(two_d_array)

array_to_join = np.array([[10,20,30],[40,50,60],[70,80,90]])

np.concatenate( (two_d_array,array_to_join),  # Arrays to join
                axis=1)                       # Axis to join upon
```


Array Math Operations
two_d_array + 100 # Add 100 to each element
two_d_array - 100 # Subtract 100 from each element
two_d_array * 2 # Multiply each element by 2
two_d_array ** 2 # Square each element
two_d_array % 2 # Take modulus of each element
small_array1 = np.array([[1,2],[3,4]])
small_array1 + small_array1
small_array1 - small_array1
small_array1 * small_array1

```python
# Get the mean of all the elements in an array with np.mean()
np.mean(two_d_array)

# Provide an axis argument to get means across a dimension
np.mean(two_d_array,
        axis = 1)     # Get means of each row

# Get the standard deviation all the elements in an array with np.std()
np.std(two_d_array)

# Provide an axis argument to get standard deviations across a dimension
np.std(two_d_array,
        axis = 0)     # Get stdev for each column

# Sum the elements of an array across an axis with np.sum()
np.sum(two_d_array,
       axis=1)        # Get the row sums
np.sum(two_d_array,
       axis=0)        # Get the column sums
```



# Take the log of each element in an array with np.log()
np.log(two_d_array)

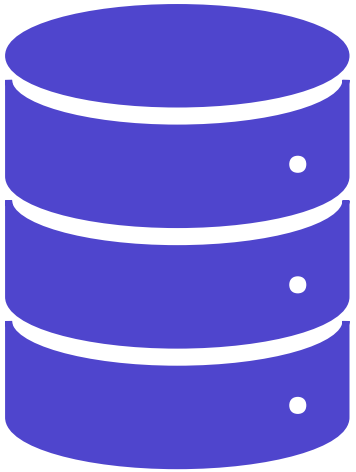
# Take the square root of each element with np.sqrt()
np.sqrt(two_d_array)

Take the vector dot product of row 0 and row 1
np.dot(two_d_array[0,0:], # Slice row 0
       two_d_array[1,0:]) # Slice row 1

# Do a matrix multiply
np.dot(small_array1, small_array1)

Pandas DataFrames

- To store data from an external source like an excel workbook or database, we need a data structure that can hold **different data types**.
- The **pandas** library offers data structures designed with this in mind: the **series** and the **DataFrame**. **Series** are **1-dimensional labeled arrays** similar to numpy's ndarrays, while **DataFrames** are labeled **2-dimensional structures**, that essentially function **as spreadsheet tables.**



```python
import numpy as np
import pandas as pd

my_series = pd.Series( data = [2,3,5,4],              # Data
                       index= ['a', 'b', 'c', 'd'])  # Indexes

my_dict = {"x": 2, "a": 5, "b": 4, "c": 8}
my_series2 = pd.Series(my_dict)
my_series["a"]
my_series[0]
my_series[1:3]
my_series + my_series
np.mean(my_series)         # numpy array functions generally work on series
```

```python
# Create a dictionary with some different data types as values

my_dict = {"name" : ["Joe","Bob","Frans"],
           "age" : np.array([10,15,20]),
           "weight" : (75,123,239),
           "height" : pd.Series([4.5, 5, 6.1],
                                index=my_dict["name"]),
           "siblings" : 1,
           "gender" : "M"}

df = pd.DataFrame(my_dict)   # Convert the dict to DataFrame

df                           # Show the DataFrame
```

	name	age	weight	height	siblings	gender
Joe	Joe	10	75	4.5	1	M
Bob	Bob	15	123	5.0	1	M
Frans	Frans	20	239	6.1	1	M

```python
# Get a column by name
df2["weight"]
df2.weight

# Delete a column
del df2['name']

# Add a new column
df2["IQ"] = [130, 105, 115]

# Inserting a single value into a DataFrame
df2["Married"] = False

#rows are matched by index. Unmatched rows will be filled with NaN
df2["College"] = pd.Series(["Harvard"],
                           index=["Frans"])
```

```python
df2.loc["Joe"]          # Select row "Joe"

df2.loc["Joe","IQ"]     # Select row "Joe" and column "IQ"

df2.loc["Joe":"Bob" , "IQ":"College"]   # Slice by label

#Select rows or columns by numeric index
df2.iloc[0]           # Get row 0
df2.iloc[0, 5]        # Get row 0, column 5
df2.iloc[0:2, 5:8]    # Slice by numeric row and column index

# select rows by passing in a sequence boolean(True/False) values
boolean_index = [False, True, True]
df2[boolean_index]

# Create a boolean sequence with a logical comparison
boolean_index = df2["age"] > 12
df2[boolean_index]
```

```
# load in data set
titanic_train = pd.read_csv("../input/train.csv")

titanic_train.head(6)   # Check the first 6 rows

titanic_train.tail(6)   # Check the last 6 rows

titanic_train.index = titanic_train["Name"]  # Set index to name
del titanic_train["Name"]                    # Delete name column
print(titanic_train.index[0:10])             # Print new indexes

# access the column labels
titanic_train.columns

titanic_train.describe()    # Summarize the first 6 columns

np.mean(titanic_train,
        axis=0)          # Get the mean of each numeric column

#get an overview of the overall structure of a DataFram
titanic_train.info()
```

```
# Load data from an Excel file
draft = pd.read_excel('input/draft2015/draft2015.xlsx', # Path to Excel file
                      sheet_name = 'draft2015') # Name of sheet to read from

draft.head(6)                    # Check the first 6 rows

# write data
draft.to_csv("draft_saved.csv")
```

Data Exploration and Cleaning

- The first part of any data analysis or predictive modeling task is an initial **exploration** of the data. it is important to explore the data before doing any serious analysis, since oddities in the data can cause bugs and muddle your results.

```
matplotlib inline
import numpy as np
import pandas as pd

titanic_train = pd.read_csv("../input/train.csv")      # Read the data
titanic_train.shape      # Check dimensions

titanic_train.head(5)  # Check the first 5 rows

titanic_train.describe()

# get a summary of the categorical
categorical = titanic_train.dtypes[titanic_train.dtypes == "object"].index
print(categorical)
titanic_train[categorical].describe()
```

Data Exploration and Cleaning (Cont.)

After looking at the data for the first time, you should ask yourself a few questions:

- Do I need all the variables?
- Should I transform any variables?
- Are there NA values, outliers or other strange values?
- Should I create new variables?

```
# VARIABLE DESCRIPTIONS:
# survival          Survival
#                   (0 = No; 1 = Yes)
# pclass            Passenger Class
#                   (1 = 1st; 2 = 2nd; 3 = 3rd)
# name              Name
# sex               Sex
# age               Age
# sibsp             Number of Siblings/Spouses Aboard
# parch             Number of Parents/Children Aboard
# ticket            Ticket Number
# fare              Passenger Fare
# cabin             Cabin
# embarked          Port of Embarkation
#                   (C = Cherbourg; Q = Queenstown; S = Southampton)
```

Do I Need All of the Variables?

- the goal is to use the training data to predict whether passengers of the titanic listed in a second data set survived or not.
- "**PassengerId**" is just a number assigned to each passenger. It is nothing more than an arbitrary identifier.
- "**Survived**" indicates whether each passenger lived or died. Since predicting survival is our goal, we need to keep it.
- Features that describe passengers numerically or group them into a few broad categories could be useful for predicting survival. The variables **Pclass**, **Sex**, **Age**, **SibSp**, **Parch**, **Fare** and **Embarked** appear to fit this description, so let's keep all of them.

Do I Need All of the Variables? (Cont.)

- the **Name** variable has 889 unique values. Since there are 889 rows in the data set we know each name is **unique**.
- the Name variable **could be removed**. On the other hand, it can be nice to have some way to **uniquely identify cases** and names are interesting from a personal and historical perspective, so let's keep Name

Do I Need All of the Variables? (Cont.)

- **Ticket** has 680 unique values: almost as many as there are passengers. Categorical variables with almost as many levels as there are records are often **not very useful** for prediction.
- The ticket numbers don't appear to follow any logical pattern we could use for grouping. Let's **remove** it
- **Cabin** also has 145 unique values, which indicates it may not be particularly useful for prediction. On the other hand, the names of the levels for the cabin variable seem to have a **regular structure**: each starts with a **capital letter followed by a number**.

Should I Transform Any Variables?

- When you first load a data set, some of the variables may be **encoded** as data types that **don't fit well** with what the data really is or what it means.
- Variables that indicate a state or the presence or absence of something with the numbers **0** and **1** are sometimes called **indicator variables** or **dummy variables.**
- **Survived** is just an integer variable that takes on the value **0 or 1** depending on whether a passenger **died** or **survived**, respectively. However, the predictions need to be encoded as 0 or 1.
- **Pclass** is an integer that indicates a passenger's class, with **1** being first class, **2** being second class and **3** being third class.

```
new_Pclass = pd.Categorical(titanic_train["Pclass"],
                            ordered=True)

new_Pclass = new_Pclass.rename_categories(
  ["Class1","Class2","Class3"])


titanic_train["Pclass"] = new_Pclass
```

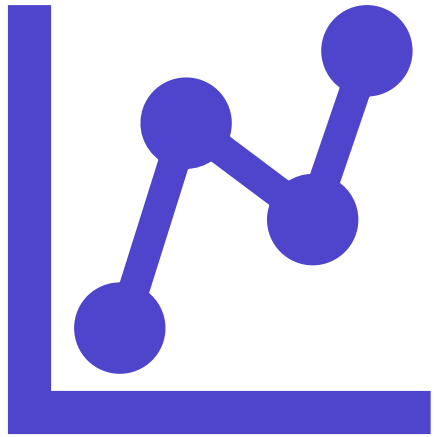
Should I Transform Any Variables? (Cont.)

- the **Cabin** variable appears that each Cabin is in a general section of the ship indicated by **the capital letter at the start** of each factor level.

```
# Convert data to str
char_cabin = titanic_train["Cabin"].astype(str)

# Take first letter
new_Cabin = np.array([cabin[0] for cabin in char_cabin])

new_Cabin = pd.Categorical(new_Cabin)

titanic_train["Cabin"] = new_Cabin
```

Are there NA Values, Outliers or Other Strange Values?

- Data sets are often littered with **missing data**, **extreme data points called outliers** and other **strange values**. Missing values, outliers and strange values can negatively **affect statistical tests and models** and may even cause certain functions to fail.

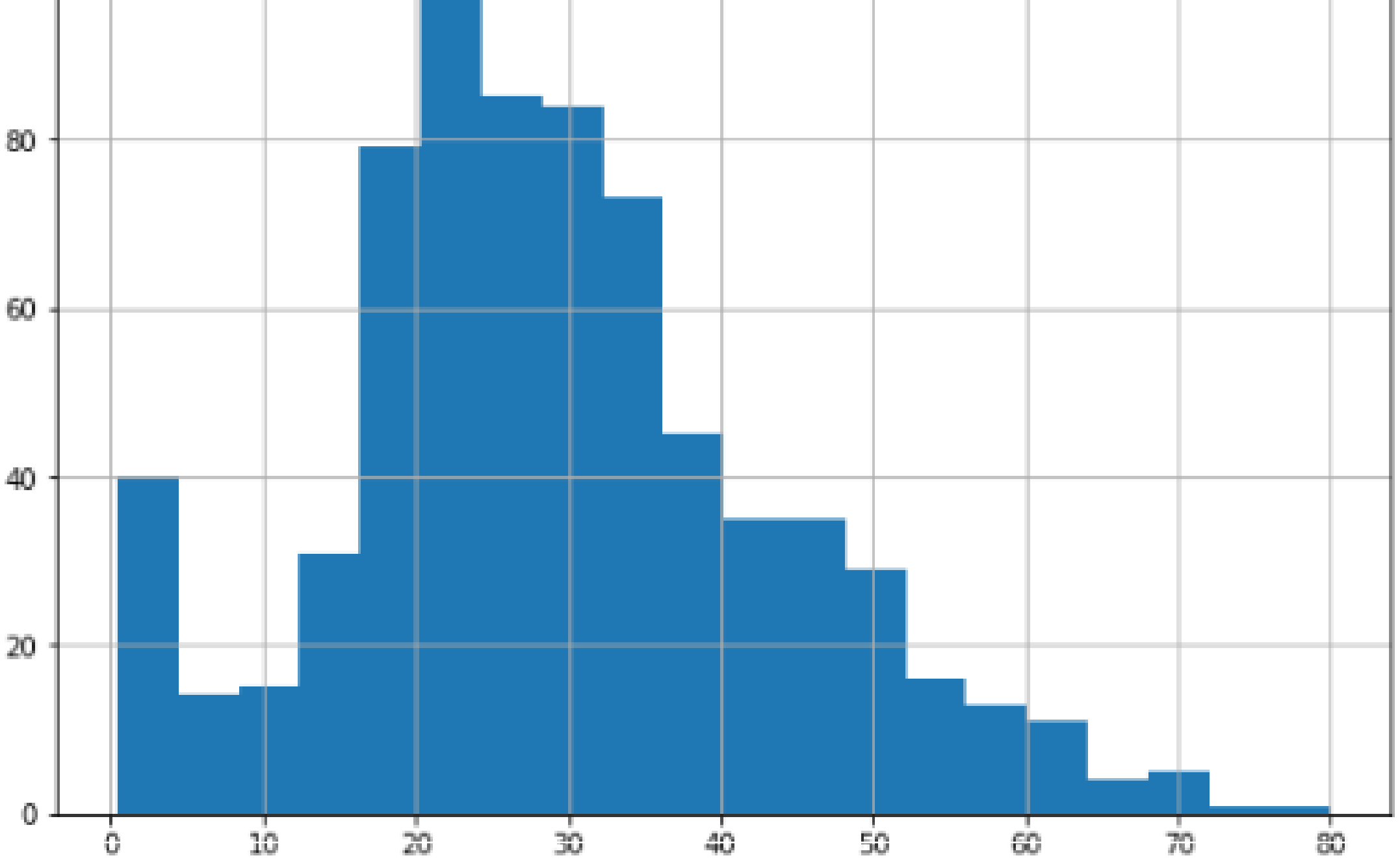


```python
# detect missing values
dummy_vector = pd.Series([1,None,3,None,7,8])
dummy_vector.isnull()

titanic_train["Age"].describe()
missing = np.where(titanic_train["Age"].isnull() == True)
len(missing[0])

#output is 177
```

Are there NA Values, Outliers or Other Strange Values?

1. **Replace** the null values with **0s**
2. **Replace** the null values with some central value like the **mean** or **median**
3. Impute some **other value** (estimating age based on other variables)
4. **Split** the data set into **two parts**: one set with where records have an Age value and another set where age is null.

```
# get a sense of the distribution of ages
#by creating a histogram of the age variable
titanic_train.hist(column='Age',     # Column to plot
                   figsize=(9,6),    # Plot size
                   bins=20)          # Number of histogram bins
```


0
20
40
60
80
0
10
20
30
40
50
60
70
80

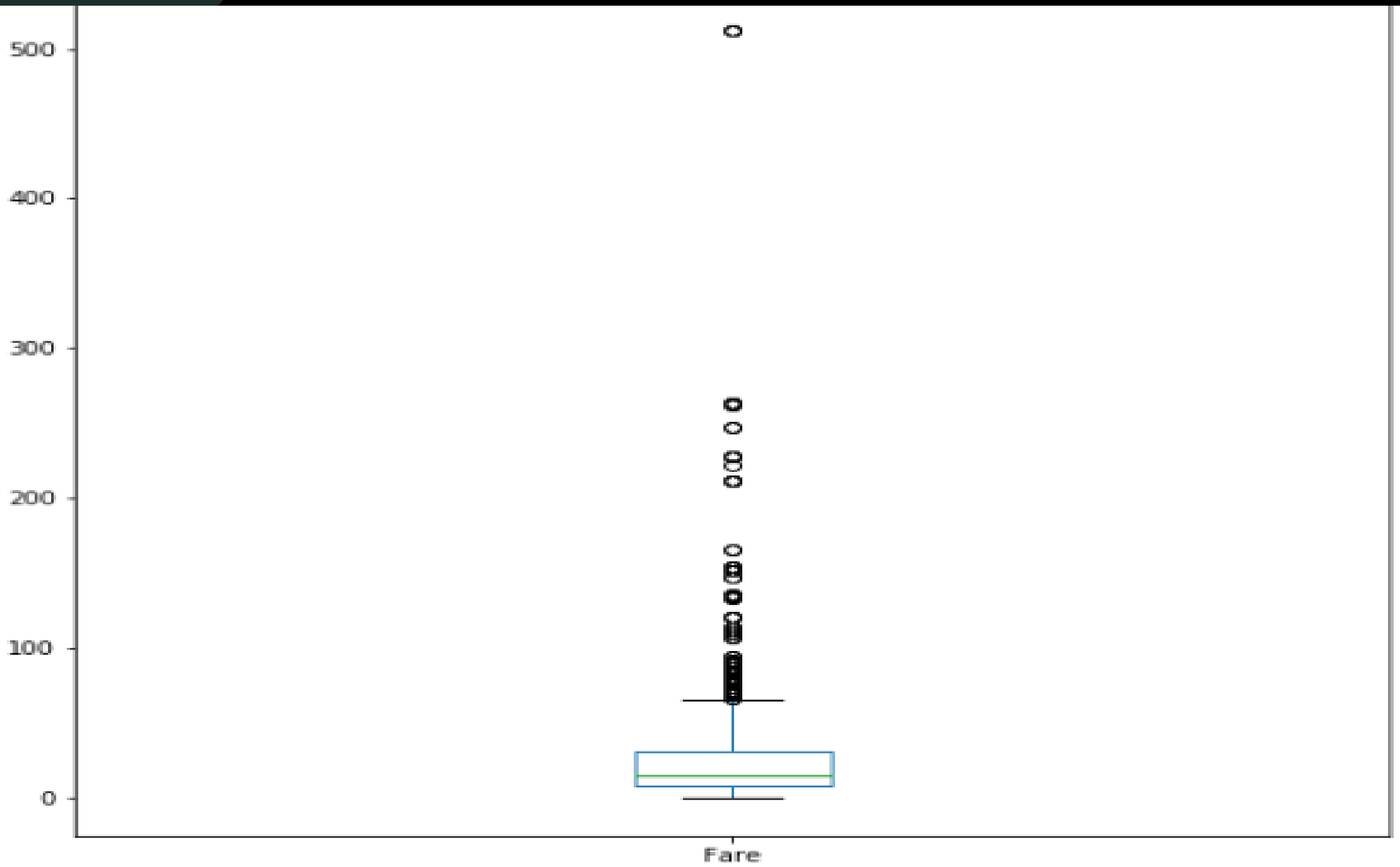
```
new_age_var = np.where(titanic_train["Age"].isnull(),
                       28,  # Value if check is true
        titanic_train["Age"]) # Value if check is false

titanic_train["Age"] = new_age_var
```

```
from sklearn.preprocessing import Imputer

# The following line sets a few mpg values to None
mtcars["mpg"] = np.where(mtcars["mpg"]>22, None, mtcars["mpg"])

mtcars["mpg"]    # Confirm that missing values were added

imp = Imputer(missing_values='NaN',  # Create imputation model
              strategy='mean',       # Use mean imputation
              axis=0)                # Impute by column
# Use imputation model to get values
imputed_cars = imp.fit_transform(mtcars)
# Remake DataFrame with new values
imputed_cars = pd.DataFrame(imputed_cars,
                            index=mtcars.index,
                            columns = mtcars.columns)

imputed_cars.head(10)
```

Are there NA Values, Outliers or Other Strange Values? (Cont.)

- **Outliers** are **extreme** numerical values: values that lie **far away** from the typical values a variable takes on.
- Creating **plots** is one of the quickest ways to **detect outliers**.
- Use **boxplot** of the "**Fare**" variable, since boxplots are designed to show **the spread of the data** and help identify outliers.
- You can **keep** them, **delete** them or **transform** them in some way to try to reduce their impact.

```
titanic_train["Fare"].plot(kind="box",
                           figsize=(9,9))
```


500
400
300
200
100
0
Fare

Should I Create New Variables?

- Creating **new** variables that are derivations or combinations existing ones is a **common step** to take before jumping into an analysis or modeling task.
- **Family**, that combines **SibSp** and **Parch** to indicate the total number of family members (siblings, spouses, parents and children) a passenger has on board.

```
titanic_train["Family"] = titanic_train["SibSp"] +
titanic_train["Parch"]
```

Working With Text Data

- **Text data** can be extremely **messy** and **difficult to work** with because it can contain all sorts of characters and symbols that may have little meaning for your analysis.

```
# This code is used for loading in data
#from the Reddit comment database
import sqlite3
import pandas as pd

sql_conn = sqlite3.connect('../input/reddit-comments-may-2015/database.sqlite')

comments = pd.read_sql("SELECT body FROM May2015 WHERE subreddit =
                       'timberwolves'", sql_conn)

comments = comments["body"]   # Convert from df to series

print(comments.shape)
comments.head(8)
```

```
comments[0].lower()       # Convert the first comment to lowercase
comments.str.lower().head(8)  # Convert all comments to lowercase
comments.str.upper().head(8)  # Convert all comments to uppercase
comments.str.len().head(8)  # Get the length of all comments
comments.str.split(" ").head(8)  # Split comments on spaces
comments.str.strip("[]").head(8)  # Strip leading and trailing brackets
comments.str.cat()[0:500]   # Check the first 500 characters
comments.str.slice(0, 10).head(8)  # Slice the first 10 characters
comments.str[0:10].head(8)  # Slice the first 10 characters
comments.str.slice_replace(5, 10, " Wolves Rule! " ).head(8)
comments.str.replace("Wolves", "Pups").head(8)
```

```python
my_series = pd.Series(["will","bill","Till","still","gull"])

my_series.str.contains(".ill")     # Match any substring ending in ill
my_series.str.contains("[Tt]ill")   # Matches T or t followed by "ill"


ex_str1 = pd.Series(["Where did he go", "He went to the mall", "he is good"])

ex_str1.str.contains("^(He|he)") # Matches He or he at the start of a string

ex_str1.str.contains("(go)$") # Matches go at the end of a string

# find posts with web links
web_links = comments.str.contains(r"https?:")

posts_with_links = comments[web_links]

print( len(posts_with_links))

posts_with_links.head(5)
```

Preparing Numeric Data

- **Numeric data** tends to be better-behaved than text data. There's only so many symbols that appear in numbers and they have well-defined values.
- Numeric variables are often on **different scales** and cover **different ranges**, so they **can't** be easily **compared**.
- variables with **large values** can **dominate** those with **smaller values** when using certain modeling techniques.

Preparing Numeric Data (Cont.)

- **Centering** and **scaling** is a common **preprocessing** task that puts numeric variables on a **common scale** so **no** single **variable** will **dominate** the others.
- The simplest way to center data is to **subtract the mean** value from each data point. It **centers the data around zero** and sets the **new mean to zero**.
- One way to put data on a **common scale** is to **divide** by the **standard deviation**.

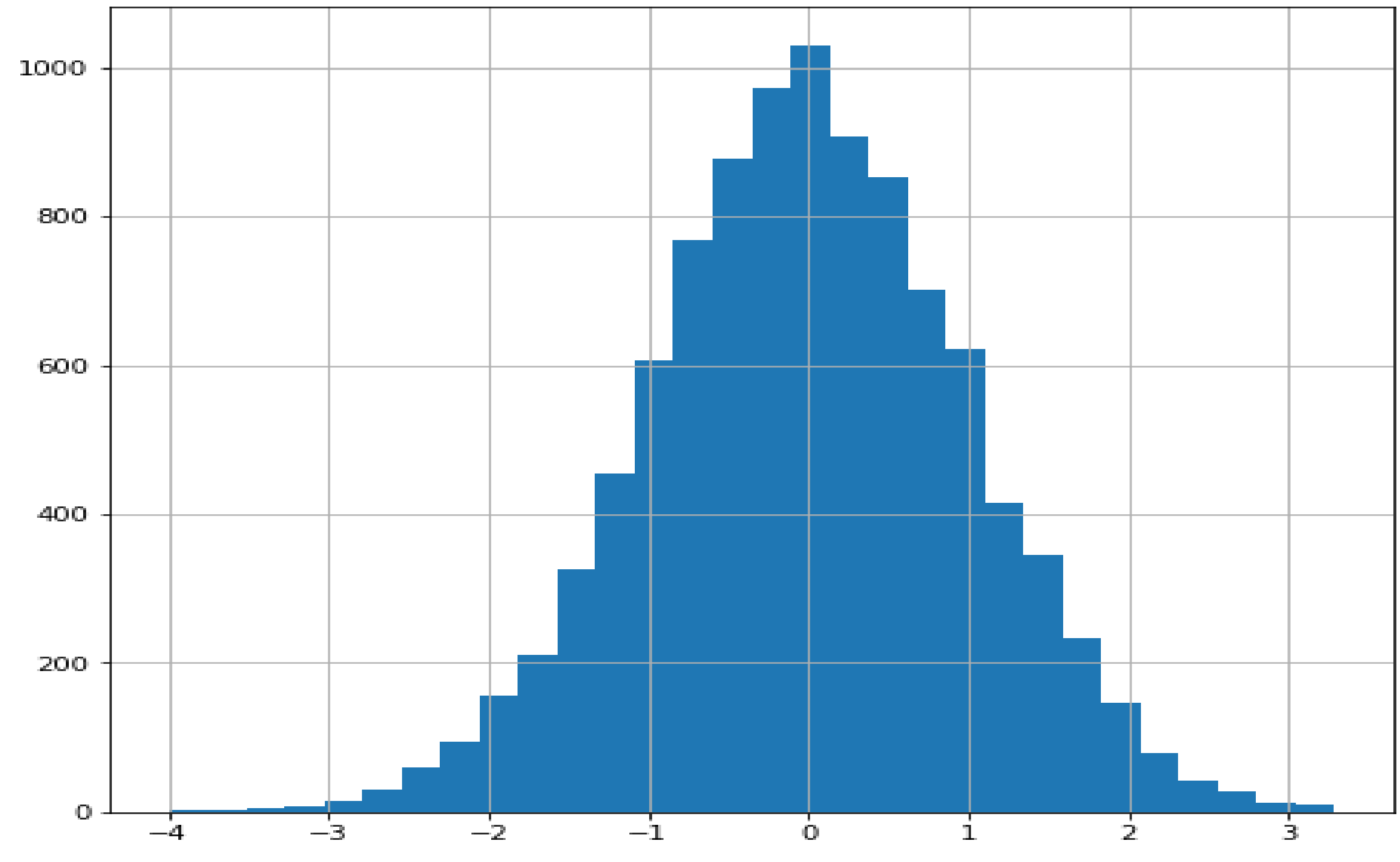
```
matplotlib inline
import numpy as np
import pandas as pd

mtcars = pd.read_csv("../input/mtcars/mtcars.csv")
print (mtcars.head() )
mtcars.index = mtcars.model          # Set row index to car model
del mtcars["model"]                  # Drop car name column
colmeans = mtcars.sum()/mtcars.shape[0]  # Get column means
colmeans
# zero-centered data
centered = mtcars-colmeans
centered.describe()
 # scale by Get column standard deviations
column_deviations = mtcars.std(axis=0)
centered_and_scaled = centered/column_deviations
centered_and_scaled.describe()
```

```
from sklearn import preprocessing

scaled_data = preprocessing.scale(mtcars)  # Scale the data*
scaled_cars = pd.DataFrame(scaled_data,    # Remake the DataFrame
                           index=mtcars.index,
                           columns=mtcars.columns)

scaled_cars.describe()
```

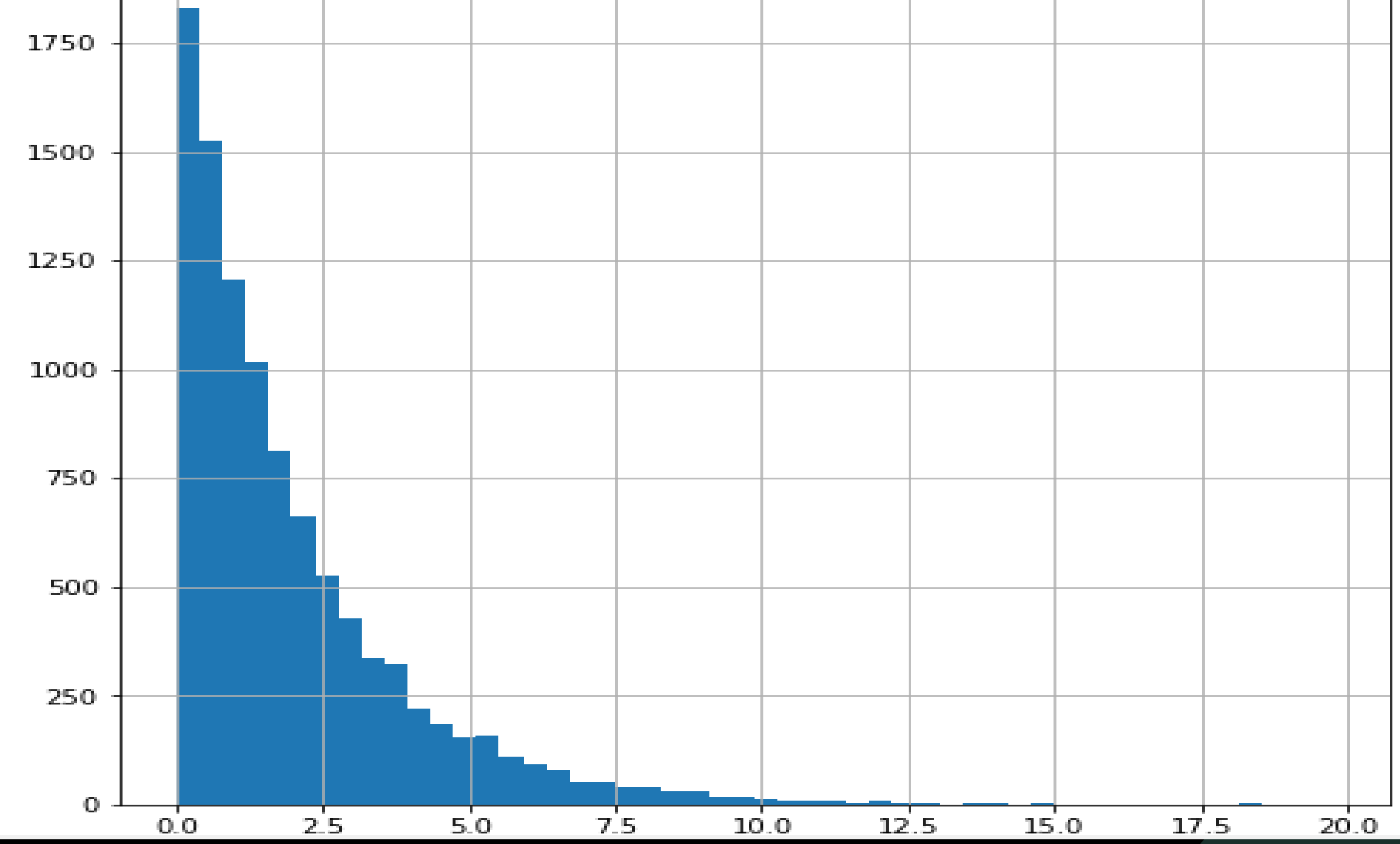
Preparing Numeric Data (Cont.)

- The **distribution** of data (**its overall shape and how it is spread out)** can have a significant **impact on analysis** and modeling.
- Data that is roughly evenly **spread around the mean** value known as **normally distributed** data tends to be well-behaved.
- On the other hand, **some data** sets exhibit significant **skewness** or **asymmetry**.
- Data with **a long tail that goes off to the right** is called **positively skewed** or **right skewed**.
- Taking the **square root** of each data point or taking the **natural logarithm** of each data point are two simple transformations that **can reduce skew**.

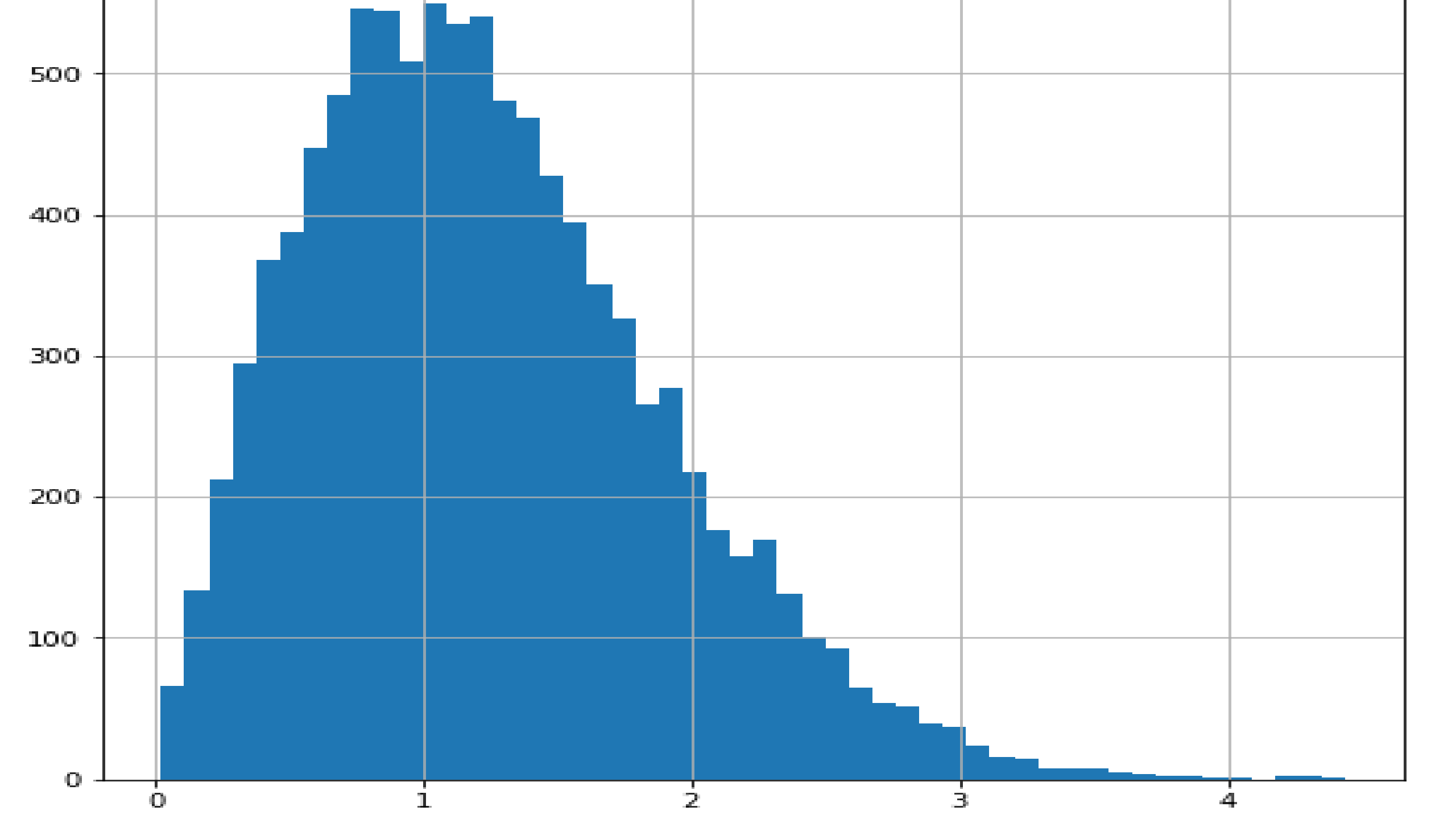
```
# Generate normal data*
normally_distributed = np.random.normal(size=10000)
# Convert to DF
normally_distributed = pd.DataFrame(normally_distributed)
# Plot histogram
normally_distributed.hist(figsize=(8,8),
                          bins=30);
```


1000
800
600
400
200
0
−4
−3
−2
−1
0
1
2
3

```
 # Generate skewed data
skewed = np.random.exponential(scale=2,
                               size= 10000)
# Convert to DF
skewed = pd.DataFrame(skewed)

 # Plot histogram
skewed.hist(figsize=(8,8),
            bins=50);
```


1750
1500
1250
1000
750
500
250
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0


# Get the square root of data points*
sqrt_transformed = skewed.apply(np.sqrt)
# Plot histogram
sqrt_transformed.hist(figsize=(8,8),
bins=50);


500
400
300
200
100
0
0
1
2
3
4

Preparing Numeric Data (Cont.)

- In predictive modeling, **each variable** you use to construct a model would ideally represent some **unique feature** of the data. In other words, you want each variable to **tell you something different**. In reality, variables often exhibit **collinearity.**
- Variables with **strong correlations** can **interfere with one another** when performing modeling and muddy results.
- A **positive correlation** implies that when one **variable goes up the other tends to go up** as well. **Negative correlations indicate an inverse relationship.** A correlation **near zero indicates low correlation** while a correlation **near -1 or 1 indicates a large** negative or positive correlation.


Check the pairwise correlations
#of 6 variables
mtcars.iloc[:,0:6].corr()

	mpg	cyl	disp	hp	drat	wt
mpg	**1.000000**	**-0.852162**	**-0.847551**	**-0.776168**	**0.681172**	**-0.867659**
cyl	**-0.852162**	**1.000000**	**0.902033**	**0.832447**	**-0.699938**	**0.782496**
disp	**-0.847551**	**0.902033**	**1.000000**	**0.790949**	**-0.710214**	**0.887980**
hp	**-0.776168**	**0.832447**	**0.790949**	**1.000000**	**-0.448759**	**0.658748**
drat	**0.681172**	**-0.699938**	**-0.710214**	**-0.448759**	**1.000000**	**-0.712441**
wt	**-0.867659**	**0.782496**	**0.887980**	**0.658748**	**-0.712441**	**1.000000**

Preparing Numeric Data (Cont.)

- Inspecting the data table, we see that the number of **cylinders a car has (cyl)** and its **weight (wt)** have **fairly strong negative correlations to gas mileage (mpg.)**. This indicates that **heavier cars and cars with more cylinders tend to get lower gas mileage**.
- If you find highly correlated variables, there are a few things you can do including:
  1. **Leave** them be
  2. **Remove** one or more variables
  3. **Combine** them in some way

Dealing With Dates

- Common **date** formats contain **numbers** and sometimes **text** as well to specify months and days.
- Getting dates into a friendly format and **extracting features of dates** like **month** and **year** into new variables can be useful preprocessing steps.

```
import numpy as np
import pandas as pd

dates = pd.read_csv("../input/lesson-16-dates
                    /dates_lesson_16.csv")
dates # Check the dates

pd.to_datetime(odd_date,format= "%H:%M:%S %Y-%d-%m")
```

	month_day_year	day_month_year	date_time	year_month_day
0	04/22/96	22-Apr-96	Tue Aug 11 09:50:35 1996	2007-06-22
1	04/23/96	23-Apr-96	Tue May 12 19:50:35 2016	2017-01-09
2	05/14/96	14-May-96	Mon Oct 14 09:50:35 2017	1998-04-12
3	05/15/96	15-May-96	Tue Jan 11 09:50:35 2018	2027-07-22
4	05/16/01	16-May-01	Fri Mar 11 07:30:36 2019	1945-11-15
5	05/17/02	17-May-02	Tue Aug 11 09:50:35 2020	1942-06-22
6	05/18/03	18-May-03	Wed Dec 21 09:50:35 2021	1887-06-13
7	05/19/04	19-May-04	Tue Jan 11 09:50:35 2022	1912-01-25
8	05/20/05	20-May-05	Sun Jul 10 19:40:25 2023	2007-06-22

```
column_1 = dates.iloc[:,0]

pd.DataFrame({"year": column_1.dt.year,
              "month": column_1.dt.month,
              "day": column_1.dt.day,
              "hour": column_1.dt.hour,
              "dayofyear": column_1.dt.dayofyear,
              "week": column_1.dt.week,
              "weekofyear": column_1.dt.weekofyear,
              "dayofweek": column_1.dt.dayofweek,
              "weekday": column_1.dt.weekday,
              "quarter": column_1.dt.quarter,
             })
print(dates.iloc[1,0])
print(dates.iloc[3,0])
print(dates.iloc[3,0]-dates.iloc[1,0])
```

Merging Data

- Data you use for your projects **won't always be confined to a single table** in a CSV or excel file.
- Data is often **split across several tables** that you need to **combine in some way**. **DataFrames can be joined together** if they have **columns in common**.

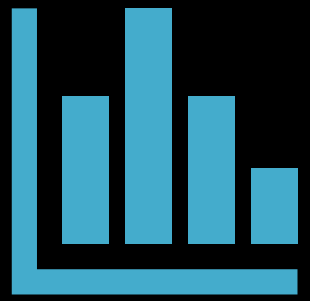
```
table1 = pd.DataFrame({"P_ID" : (1,2,3,4,5,6,7,8),
                       "gender" : ("male", "male", "female","female",
                                   "female", "male", "female", "male"),
                       "height" : (71,73,64,64,66,69,62,72),
                       "weight" : (175,225,130,125,165,160,115,250)})

table2 = pd.DataFrame({"P_ID" : (1, 2, 4, 5, 7, 8, 9, 10),
                       "sex" : ("male", "male", "female","female",
                                "female", "male", "male", "female"),
                       "visits" : (1,2,4,12,2,2,1,1),
                       "checkup" : (1,1,1,1,1,1,0,0),
                       "follow_up" : (0,0,1,2,0,0,0,0),
                       "illness" : (0,0,2,7,1,1,0,0),
                       "surgery" : (0,0,0,2,0,0,0,0),
                       "ER" : ( 0,1,0,0,0,0,1,1) } )

combined1 = pd.merge(table1,          # First table
                     table2,          # Second table
                     how="inner",     # Merge method
                     on="P_ID")       # Column(s) to join on
```

	P_ID	gender	height	weight	sex	visits	checkup	follow_up	illness	surgery	ER
0	1	male	71	175	male	1	1	0	0	0	0
1	2	male	73	225	male	2	1	0	0	0	1
2	4	female	64	125	female	4	1	1	2	0	0
3	5	female	66	165	female	12	1	2	7	2	0
4	7	female	62	115	female	2	1	0	1	0	0
5	8	male	72	250	male	2	1	0	1	0	0

```
# A left join keeps all key values in the first(left) data frame
left_join = pd.merge(table1,          # First table
                     table2,          # Second table
                     how="left",    # Merge method
                     on="P_ID")     # Column(s) to join on

# A right join keeps all key values in the second(right) data frame
right_join = pd.merge(table1,          # First table
                      table2,          # Second table
                      how="right",    # Merge method
                      on="P_ID")     # Column(s) to join on

# An outer join keeps all key values in both data frames
outer_join = pd.merge(table1,          # First table
                      table2,          # Second table
                      how="outer",    # Merge method
                      on="P_ID")     # Column(s) to join on
```

```python
# Rename "gender" column
table1.rename(columns={"gender":"sex"}, inplace=True)

combined2 = pd.merge(table1,                    # First data frame
                     table2,                    # Second data frame
                     how="outer",               # Merge method
                     on=["P_ID","sex"])     # Column(s) to join on
```

Frequency Tables



Frequency tables are a basic tool you can use to **explore data** and get an idea of the **relationships between variables**.

A frequency table is just a data table that shows **the counts of one or more categorical variables.**

```
my_tab = pd.crosstab(index=titanic_train["Survived"],# Make a crosstab
                     columns="count")      # Name the count column

pd.crosstab(index=titanic_train["Pclass"], # Make a crosstab
            columns="count")  # Name the count column

titanic_train.Sex.value_counts()

# Table of survival vs. sex
survived_sex = pd.crosstab(index=titanic_train["Survived"],
                           columns=titanic_train["Sex"])

survived_sex.index= ["died","survived"]
# create a 3-way table inspecting survival, sex and passenger class
surv_sex_class = pd.crosstab(index=titanic_train["Survived"],
                         columns=[titanic_train["Pclass"],
                           titanic_train["Sex"]],
                        margins=True)   # Include row and column totals
```

Pclass	1		2		3		All
Sex	female	male	female	male	female	male	
Survived							
0	3	77	6	91	72	300	549
1	91	45	70	17	72	47	342
All	94	122	76	108	144	347	891

Pclass	1		2		3		All
Sex	female	male	female	male	female	male	
Survived							
0	0.031915	0.631148	0.078947	0.842593	0.5	0.864553	0.616162
1	0.968085	0.368852	0.921053	0.157407	0.5	0.135447	0.383838
All	1.000000	1.000000	1.000000	1.000000	1.0	1.000000	1.000000

Plotting With Pandas

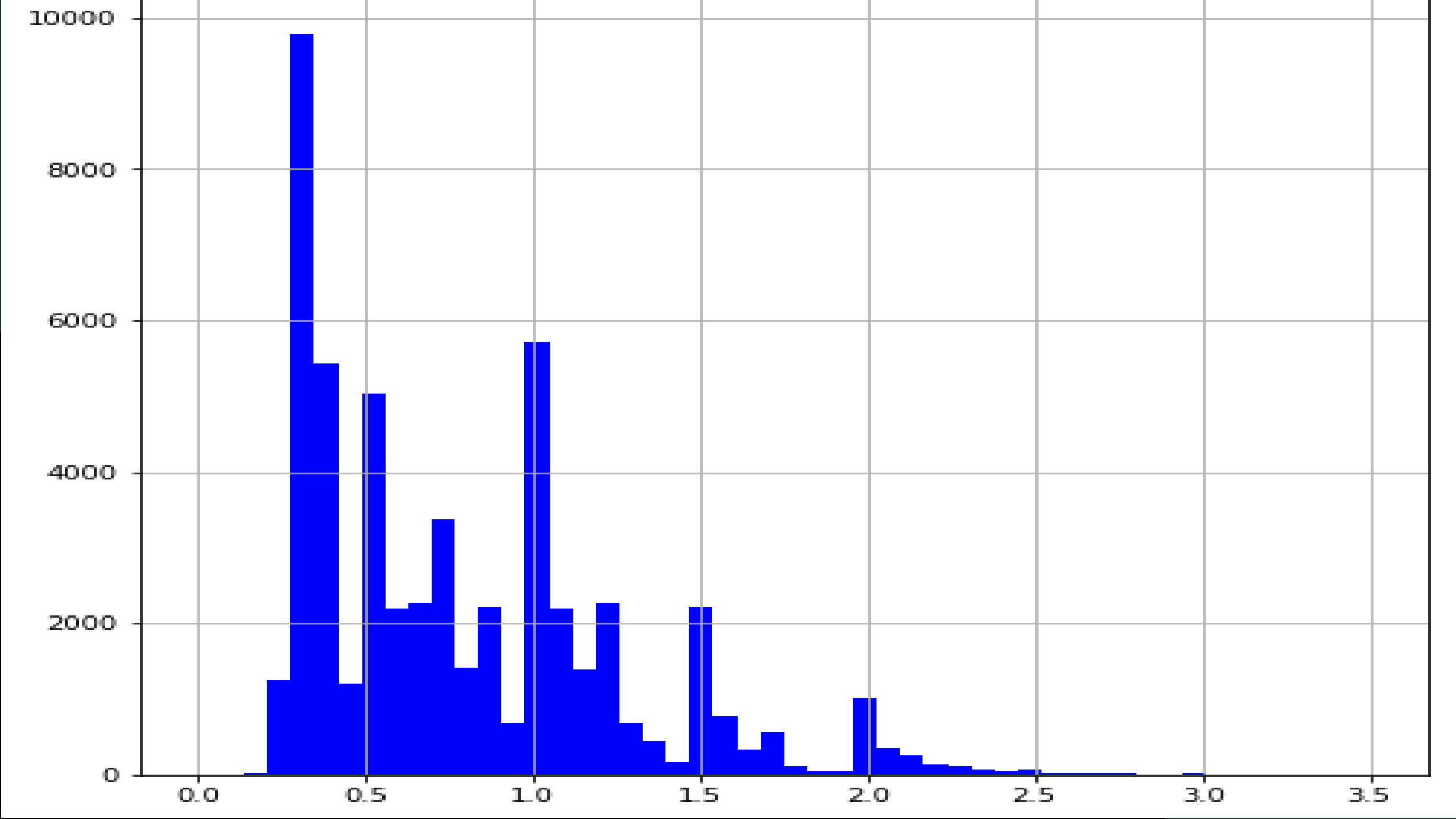
Visualizations are one of the most powerful tools at your disposal for **exploring data** and communicating data insights.

Plots in pandas are built on top of a popular Python plotting library called **matplotlib**

Histograms

- A **histogram** is a **univariate** plot (**a plot that displays one variable**) that **groups a numeric variable into bins** and **displays the number of observations that fall within each bin**. A histogram is a useful tool for getting a sense of the **distribution** of a numeric variable.

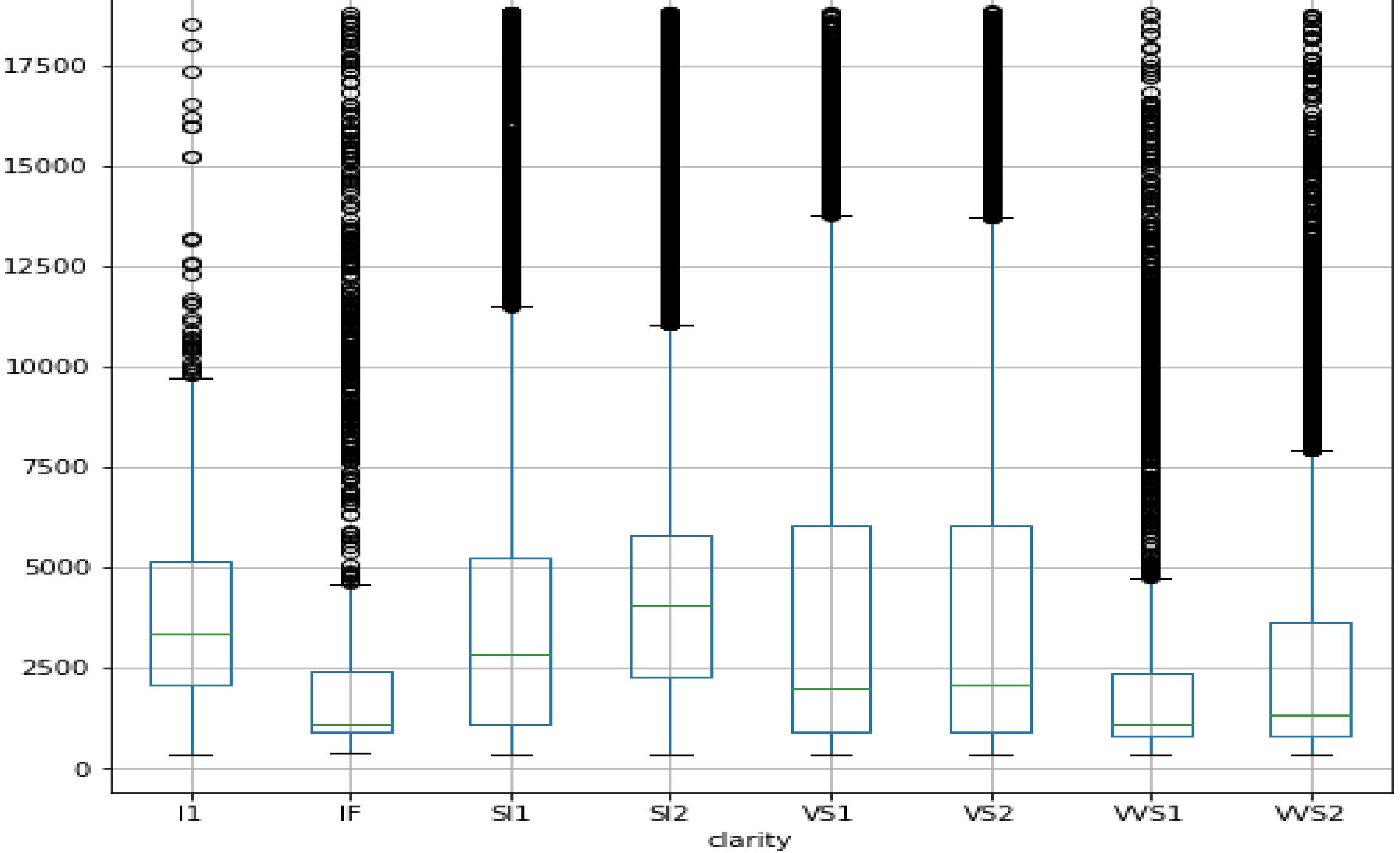
```
import numpy as np
import pandas as pd
import matplotlib

diamonds = pd.read_csv("../input/diamonds/diamonds.csv")

print(diamonds.shape)          # Check data shape

diamonds.head(5)

diamonds.hist(column="carat",          # Column to plot
              figsize=(8,8),           # Plot size
              color="blue",            # Plot color
              bins=50,                 # Use 50 bins
              range= (0,3.5));         # Limit x-axis range
```


10000
8000
6000
4000
2000
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

Boxplots

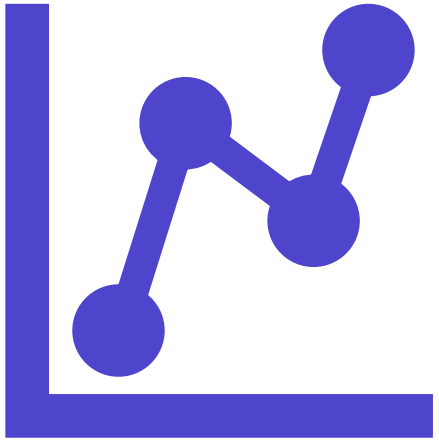
- **Boxplots** are another type of univariate plot for summarizing **distributions** of numeric data graphically.
- A **side-by-side boxplot** takes a numeric variable and **splits** it on **based on some categorical variable**, **drawing** a different boxplot **for each level of the categorical variable**.

```
diamonds.boxplot(column="carat");

diamonds.boxplot(column="price",    # Column to plot
                 by= "clarity",     # Column to split upon
                 figsize= (8,8));   # Figure size
```

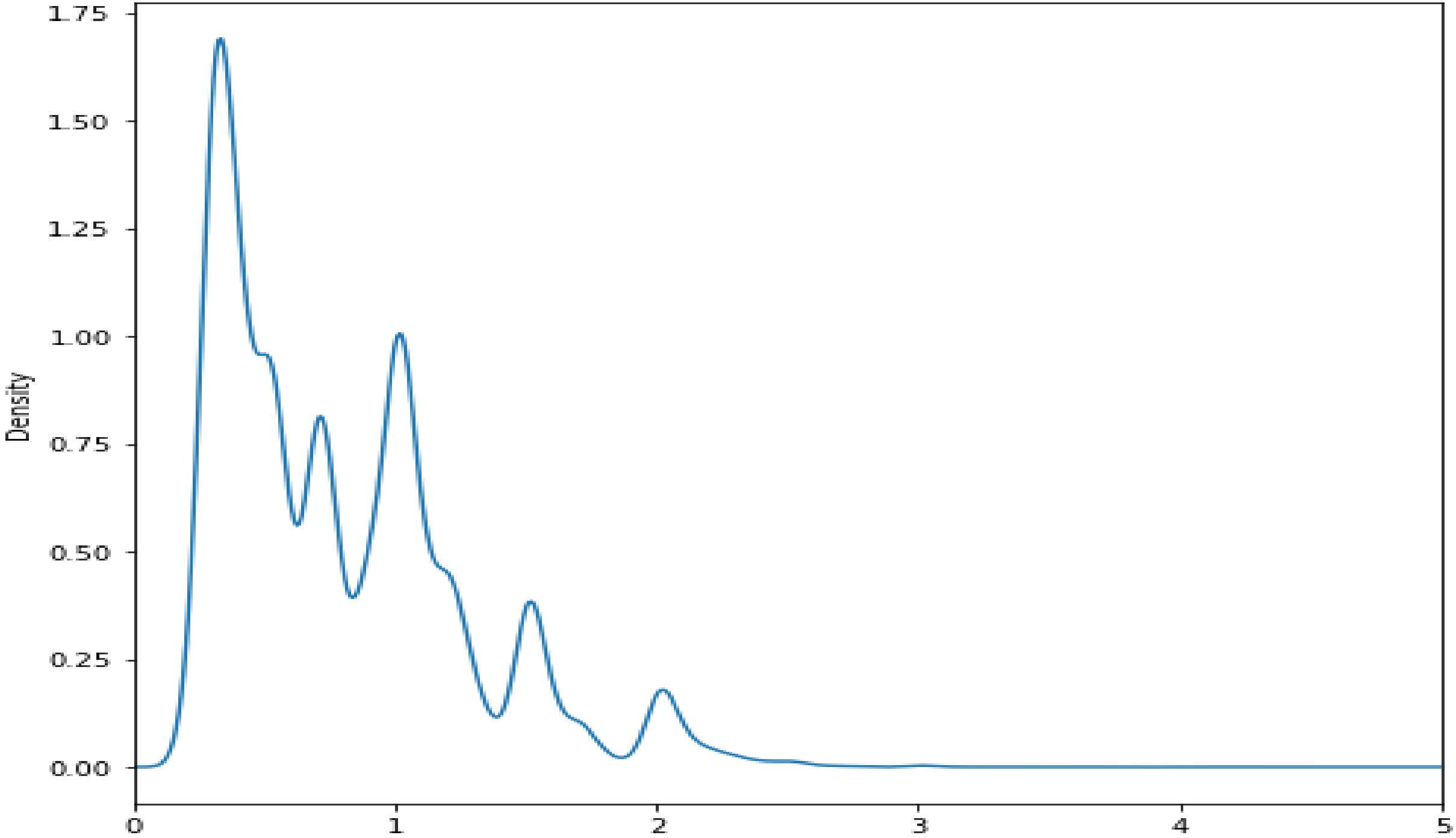

17500
15000
12500
10000
7500
5000
2500
0
I1
IF
SI1
SI2
VS1
VS2
VVS1
VVS2
clarity

Density Plots

- A **density plot** shows the **distribution** of a numeric variable with a **continuous** curve. It is like a histogram but without discrete bins.



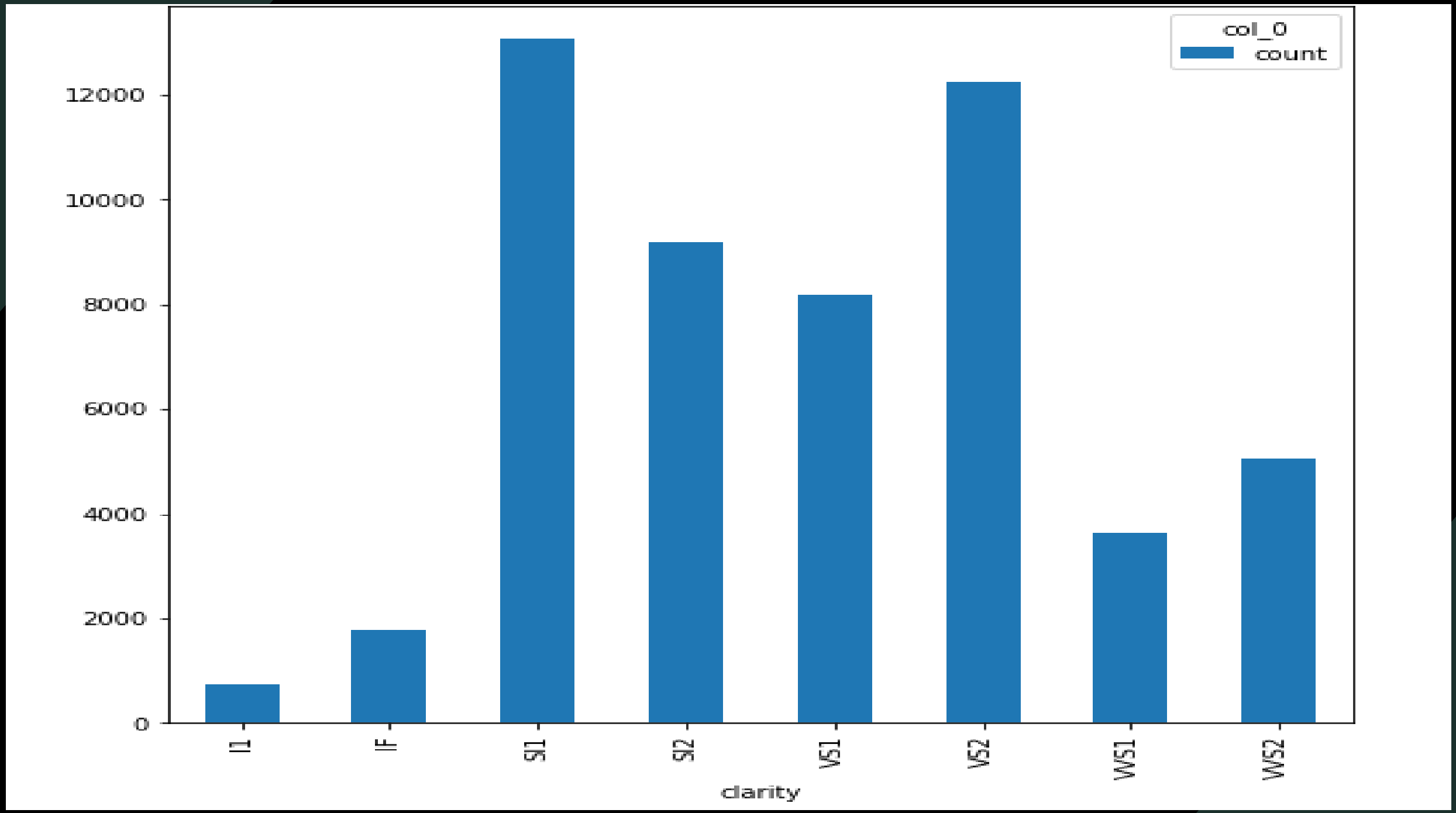
```
diamonds["carat"].plot(kind="density",# Create density plot
                       figsize=(8,8),  # Set figure size
                       xlim= (0,5));   # Limit x axis values
```

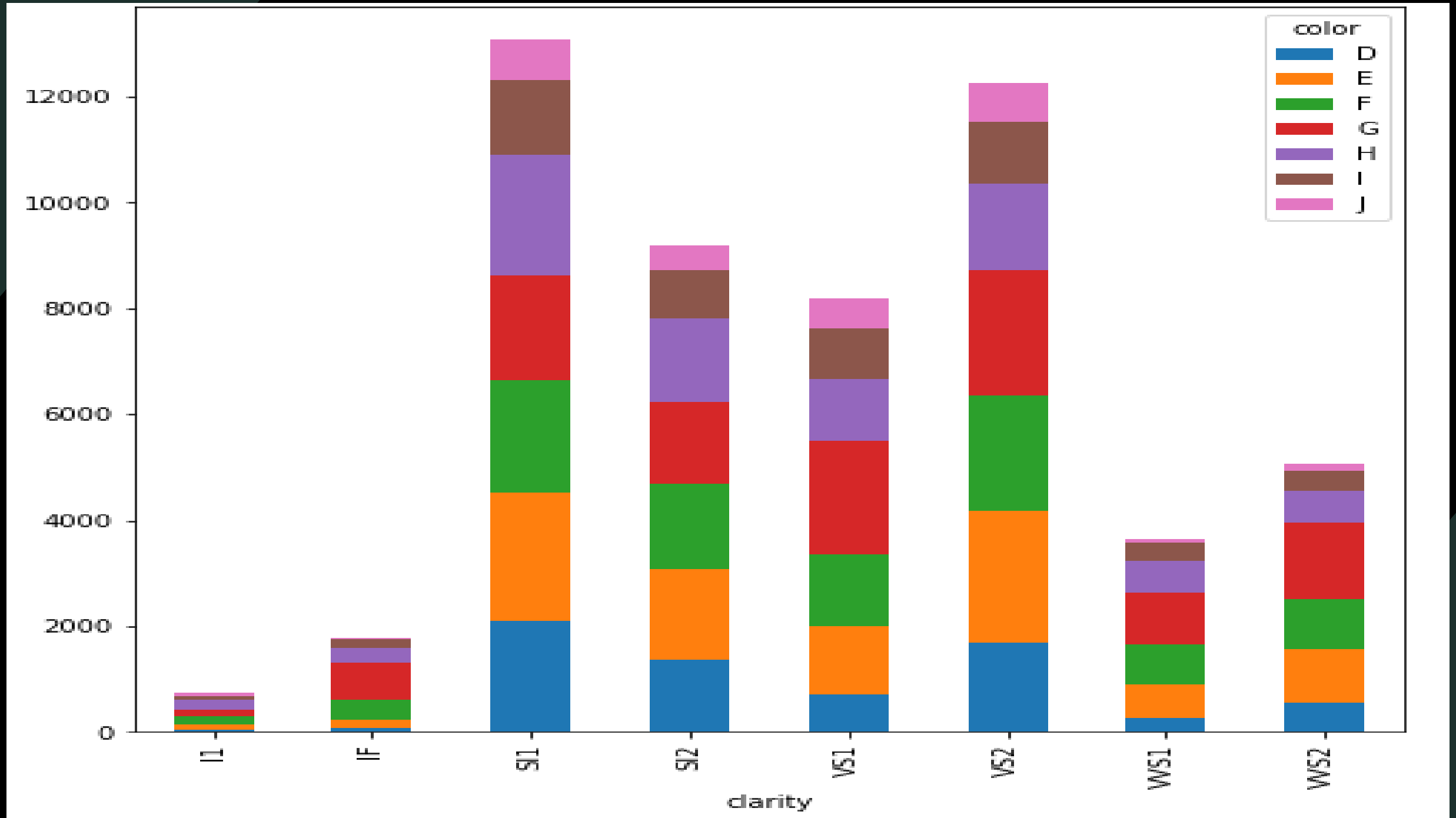

Density
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
1
2
3
4
5

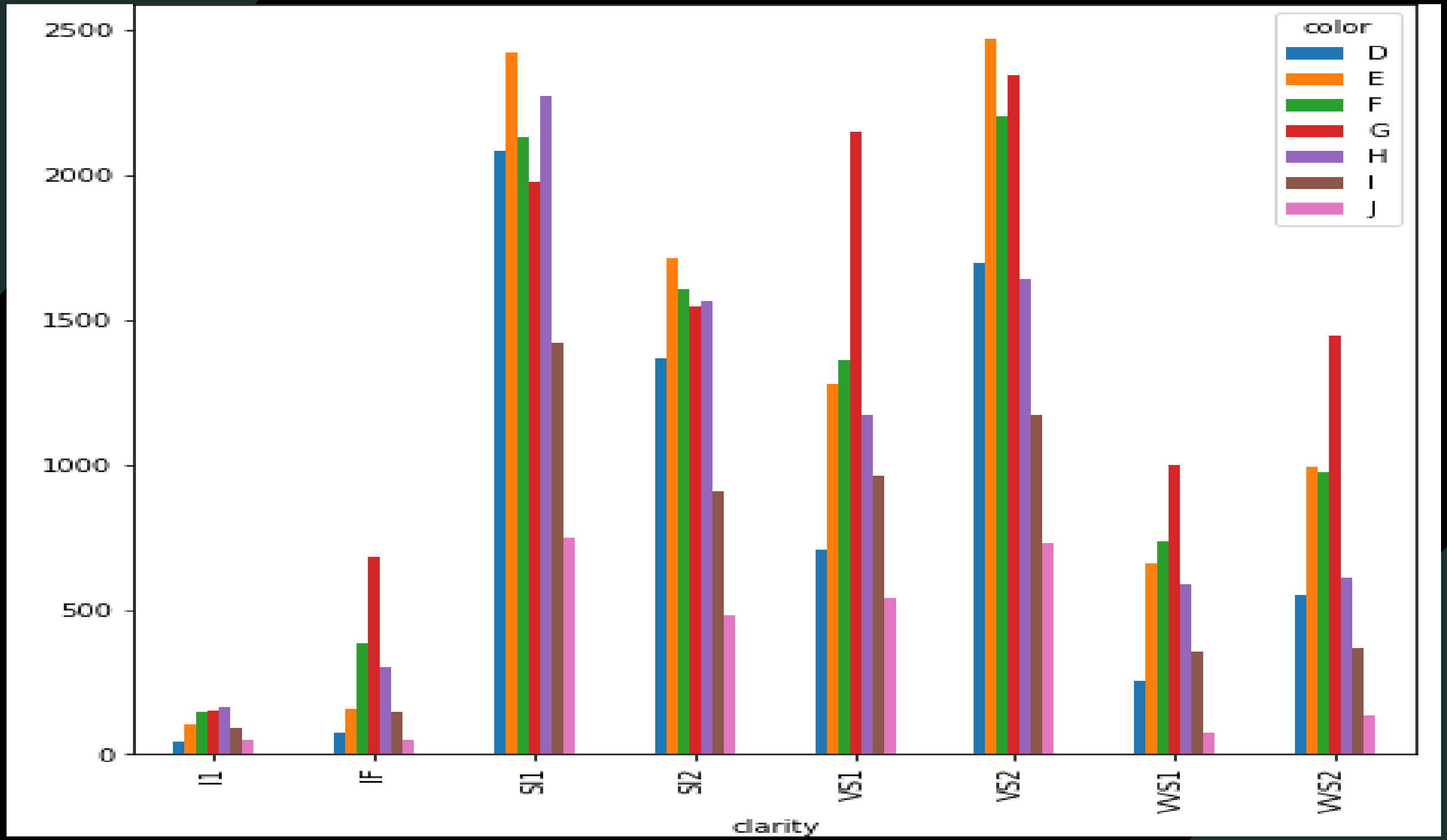
Barplots

- **Barplots** are graphs that visually **display counts** of **categorical** variables.

```
carat_table = pd.crosstab(index=diamonds["clarity"],
                          columns="count")

carat_table.plot(kind="bar",
                 figsize=(8,8));

#stacked bar
carat_table = pd.crosstab(index=diamonds["clarity"],
                          columns=diamonds["color"])
carat_table.plot(kind="bar",
                 figsize=(8,8),
                 stacked=True);

carat_table.plot(kind="bar",
                 figsize=(8,8),
                 stacked=False);
```


col_0
count
12000
10000
8000
6000
4000
2000
0
I1
IF
SI1
SI2
VS1
VS2
VVS1
VVS2
clarity

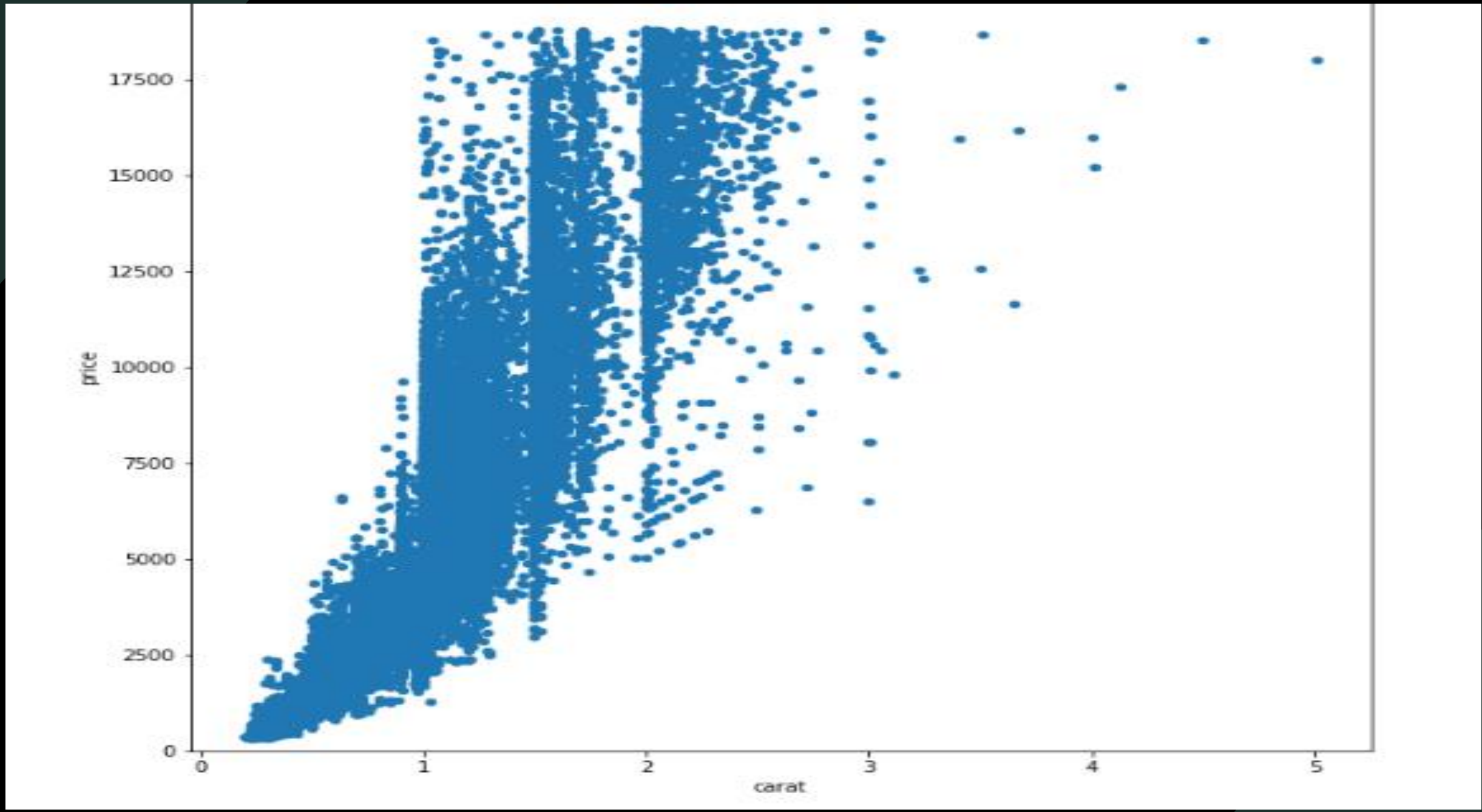

color
D
E
F
G
H
I
J
12000
10000
8000
6000
4000
2000
0
I1
IF
SI1
SI2
VS1
VS2
WS1
WS2
clarity


color
D
E
F
G
H
I
J
2500
2000
1500
1000
500
0
I1
IF
SI1
SI2
VS1
VS2
VVS1
VVS2
clarity

Scatterplots

- **Scatterplots** are **bivariate** (two variable) plots that take two numeric variables and plot data points on the **x/y plane**.

```
diamonds.plot(kind="scatter",   # Create a scatterplot
              x="carat",        # Put carat on the x axis
              y="price",        # Put price on the y axis
              figsize=(10,10),
              ylim=(0,20000));
```

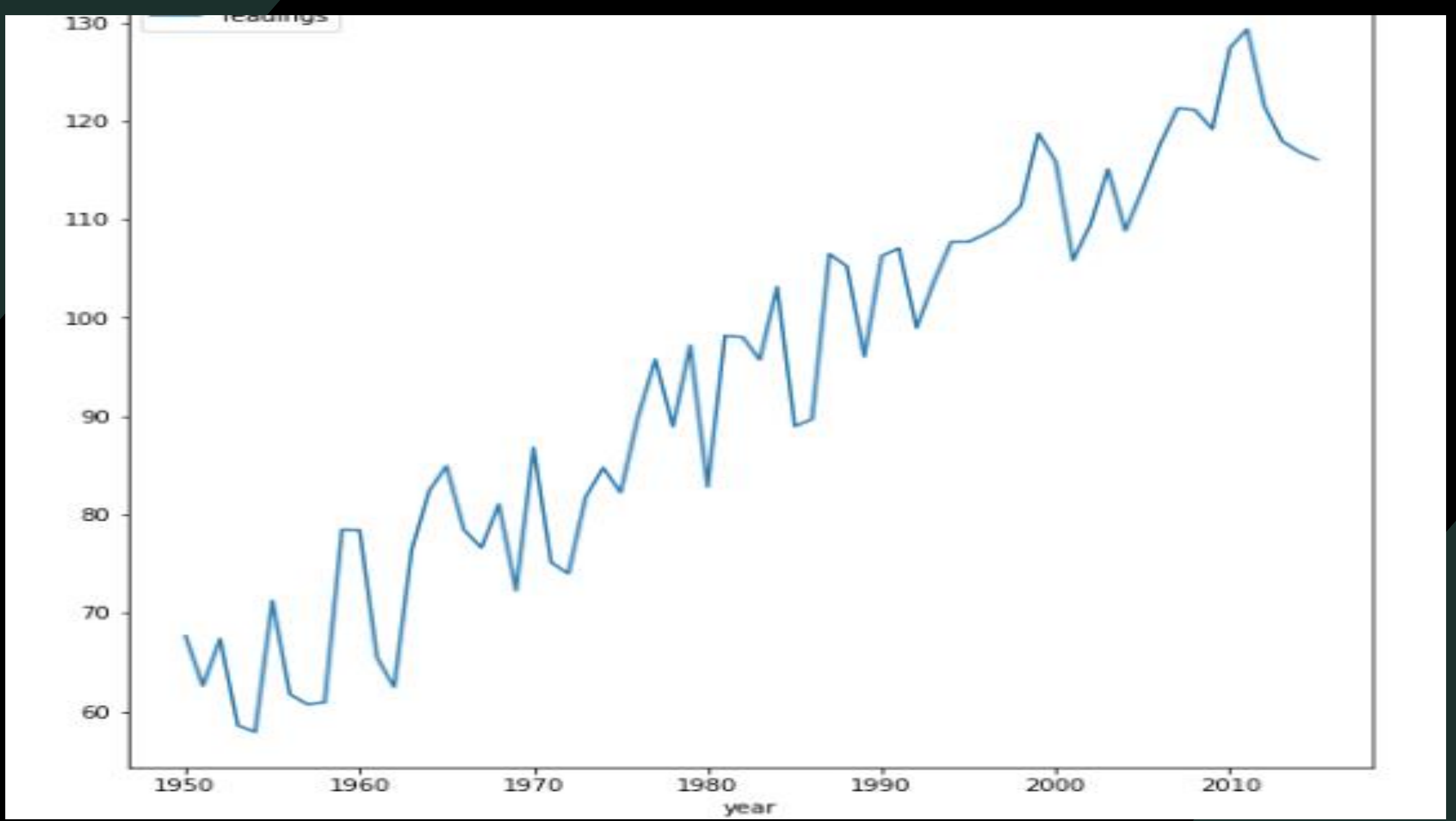

17500
15000
12500
10000
7500
5000
2500
0
price
0
1
2
3
4
5
carat

Line Plots

Line plots are charts used to show the change in a **numeric variable based on some other** ordered variable.

Line plots are often used to **plot time series data** to show the evolution of a variable over time.

```
# Create some data
years = [y for y in range(1950,2016)]

readings = [(y+np.random.uniform(0,20)-1900) for y in years]

time_df = pd.DataFrame({"year":years,
                        "readings":readings})

# Plot the data
time_df.plot(x="year",
             y="readings",
             figsize=(9,9));
```


130
120
110
100
90
80
70
60
1950
1960
1970
1980
1990
2000
2010
year

```
# Create the plot and save to a variable
my_plot = time_df.plot(x="year",
                       y="readings",
                       figsize=(9,9))

my_fig = my_plot.get_figure() # Get the figure

my_fig.savefig("line_plot_example.png")  # Save to file
```