

Python Data Science-1

Python Data Science- Pandas

What is Pandas?

Pandas is an open-source Python Library used for high-performance data manipulation and data analysis using its powerful data structures. Python with pandas is in use in a variety of academic and commercial domains, including Finance, Economics, Statistics, Advertising, Web Analytics, and more. Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data load, organize, manipulate, model, and analyse the data.

Below are the some of the important features of Pandas which is used specifically for Data processing and Data analysis work.

Key Features of Pandas

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Pandas deals with the following three data structures –

- Series
- DataFrame

These data structures are built on top of Numpy array, making them fast and efficient.

Dimension & Description

The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, DataFrame is a container of Series, Panel is a container of DataFrame.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, size-immutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.

DataFrame is widely used and it is the most important data structures.

Series

Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56,

10	23	56	17	52	61	73	90	26	72
----	----	----	----	----	----	----	----	----	----

Key Points of Series

- Homogeneous data
- Size Immutable
- Values of Data Mutable

DataFrame

DataFrame is a two-dimensional array with heterogeneous data. For example,

Name	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Vin	45	Male	3.9

Katie	38	Female	2.78
-------	----	--------	------

The table represents the data of a sales team of an organization with their overall performance rating. The data is represented in rows and columns. Each column represents an attribute and each row represents a person.

Data Type of Columns

The data types of the four columns are as follows –

Column	Type
Name	String
Age	Integer
Gender	String
Rating	Float

Key Points of Data Frame

- Heterogeneous data
- Size Mutable
- Data Mutable

Python Data Science- NumPy

What is NumPy?

NumPy is a Python package which stands for 'Numerical Python'. It is a library consisting of multidimensional array objects and a collection of routines for processing of array.

Operations using NumPy

Using NumPy, a developer can perform the following operations –

- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

NumPy A Replacement for MatLab

NumPy is often used along with packages like **SciPy** (Scientific Python) and **Matplotlib** (plotting library). This combination is widely used as a replacement for MatLab, a popular platform for technical computing. However, Python alternative to MatLab is now seen as a more modern and complete programming language.

It is open source, which is an added advantage of NumPy.

ndarray Object

The most important object defined in NumPy is an N-dimensional array type called **ndarray**. It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index. Every item in an ndarray takes the same size of block in the memory. Each element in ndarray is an object of data-type object (called **dtype**). Any item extracted from ndarray object (by slicing) is represented by a Python object of one of array scalar types.

Python Data Science – SciPy

What is SciPy?

The SciPy library of Python is built to work with NumPy arrays and provides many user-friendly and efficient numerical practices such as routines for numerical integration and optimization. Together, they run on all popular operating systems, are quick to install and are free of charge. NumPy and SciPy are easy to use, but powerful enough to depend on by some of the world's leading scientists and engineers.

SciPy Sub-packages

SciPy is organized into sub-packages covering different scientific computing domains. These are summarized in the following table –

scipy.constants	Physical and mathematical constants
-----------------	-------------------------------------

scipy.fftpack	Fourier transform
---------------	-------------------

scipy.integrate	Integration routines
-----------------	----------------------

scipy.interpolate	Interpolation
-------------------	---------------

scipy.io	Data input and output
----------	-----------------------

scipy.linalg	Linear algebra routines
--------------	-------------------------

scipy.optimize	Optimization
----------------	--------------

scipy.signal	Signal processing
--------------	-------------------

scipy.sparse	Sparse matrices
--------------	-----------------

scipy.spatial	Spatial data structures and algorithms
---------------	--

scipy.special	Any special mathematical functions
---------------	------------------------------------

scipy.stats	Statistics
-------------	------------

Data Structure

The basic data structure used by SciPy is a multidimensional array provided by the NumPy module. NumPy provides some functions for Linear Algebra, Fourier Transforms and Random Number Generation, but not with the generality of the equivalent functions in SciPy.

Python Data Science – Matplotlib

What is Matplotlib?

Matplotlib is a python library used to create 2D graphs and plots by using python scripts. It has a module named pyplot which makes things easy for plotting by providing feature to control line styles, font properties, formatting axes etc. It supports a very wide variety of graphs and plots namely - histogram, bar charts, power spectra, error charts etc. It is used along with NumPy to provide an environment that is an effective open source alternative for MatLab. It can also be used with graphics toolkits like PyQt and wxPython.

Conventionally, the package is imported into the Python script by adding the following statement –

```
from matplotlib import pyplot as plt
```

Matplotlib Example

The following script produces the **sine wave plot** using matplotlib.

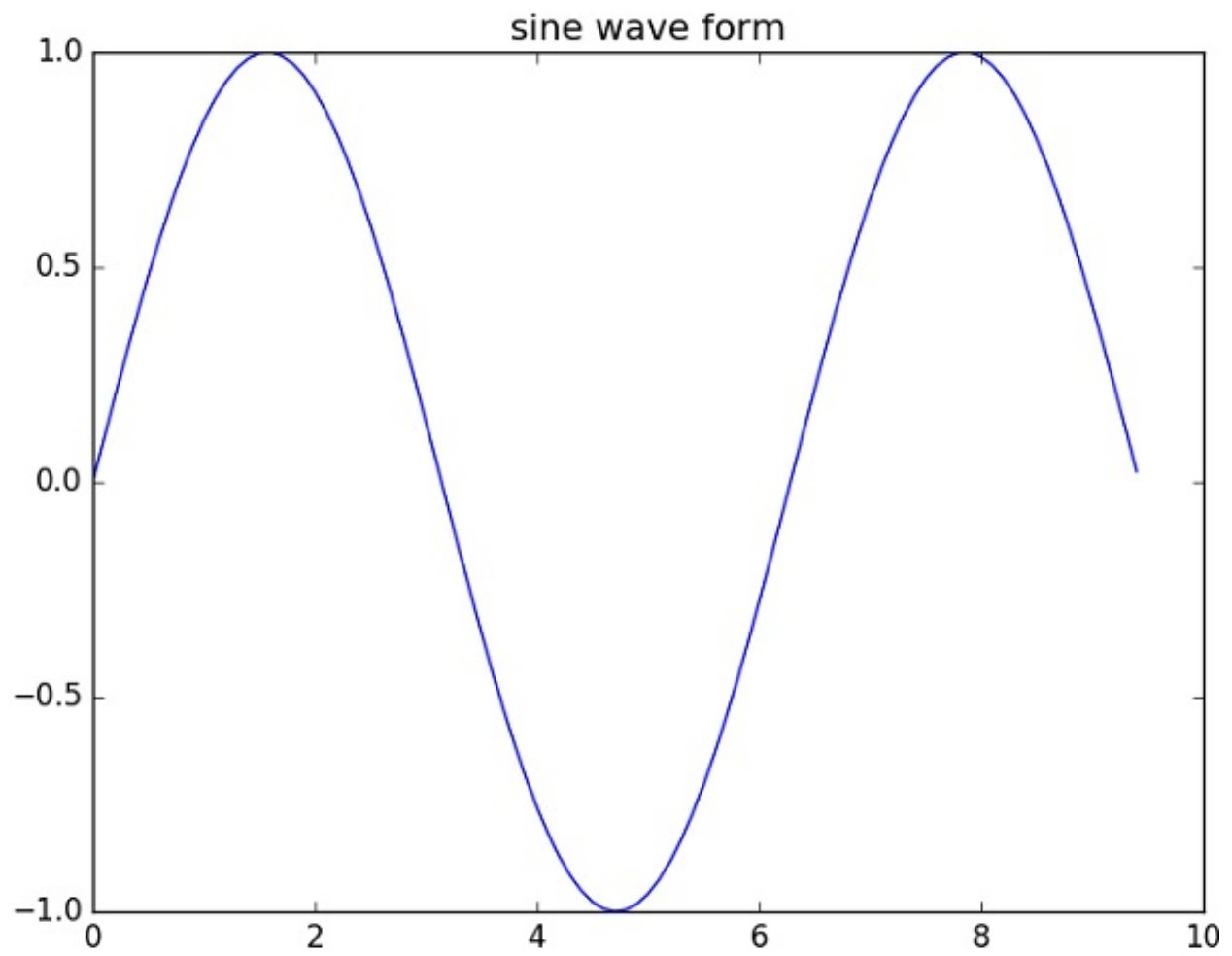
Example

```
import numpy as np
import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)
plt.title("sine wave form")

# Plot the points using matplotlib
plt.plot(x, y)
plt.show()
```

Its **output** is as follows –



Python- Data Operations

Python handles data of various formats mainly through the two libraries, Pandas and Numpy. We have already seen the important features of these two libraries in the previous chapters. Now we will see some basic examples from each of the libraries on how to operate on data.

Data Operations in Numpy

The most important object defined in NumPy is an N-dimensional array type called ndarray. It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index. An instance of ndarray class can be constructed by different array creation routines described later in the tutorial. The basic ndarray is created using an array function in NumPy as follows –

`numpy.array`

Following are some examples on Numpy Data handling.

Example 1

```
# more than one dimensions
import numpy as np
a = np.array([[1, 2], [3, 4]])
print a
```

The output is as follows –

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

Example 2

```
# minimum dimensions
import numpy as np
a = np.array([1, 2, 3,4,5], ndmin = 2)
print a
```

The output is as follows –

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

Example 3

```
# dtype parameter
import numpy as np
```

```
a = np.array([1, 2, 3], dtype = complex)
print a
```

The output is as follows –

```
[ 1.+0.j,  2.+0.j,  3.+0.j]
```

Data Operations in Pandas

Pandas handles data through **Series**, **Data Frame**, and **Panel**. We will see some examples from each of these.

Pandas Series

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index. A pandas Series can be created using the following constructor –

```
pandas.Series( data, index, dtype, copy)
```

Example 4

Here we create a series from a Numpy Array.

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print s
```

Its **output** is as follows –

```
0    a
1    b
2    c
3    d
dtype: object
```

Pandas DataFrame

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. A pandas DataFrame can be created using the following constructor –

`pandas.DataFrame( data, index, columns, dtype, copy)`

Example 5

Let us now create an indexed DataFrame using arrays.

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve',
'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print df
```

Its **output** is as follows –

	Age	Name
rank1	28	Tom
rank2	34	Jack
rank3	29	Steve
rank4	42	Ricky

Pandas Read CSV in Python

CSV files are the Comma Separated Files. It allows users to load tabular data into a **DataFrame**, which is a powerful structure for data manipulation and analysis. To access data from the CSV file, we require a function `read_csv()` from Pandas that retrieves data in the form of the data frame. Here's a quick example to get you started.

Suppose you have a file named `people.csv`. First, we must import the [Pandas](#) library. then using Pandas load this data into a DataFrame as follows:

```
import pandas as pd
```

```
# reading csv file
```

```
df = pd.read_csv("people.csv")
```

```
df
```

Output:

	First Name	Last Name	Sex	Email	Date of birth	Job Title
0	Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
1	Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
2	Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath
3	Yesenia	Martinez	Male	kaitlinkaiser@example.com	2017-08-03	Market researcher
4	Lori	Todd	Male	buchananmanuel@example.net	1938-12-01	Veterinary surgeon

`read_csv()` function - Syntax & Parameters

`read_csv()` function in Pandas is used to read data from CSV files into a Pandas **DataFrame**. A DataFrame is a powerful data structure that allows you to manipulate and analyze tabular data efficiently. CSV files are plain-text files where each row represents a record, and columns are separated by commas (or other delimiters).

Here is the **Pandas read CSV** syntax with its parameters.

Syntax: `pd.read_csv(filepath_or_buffer, sep=',', header='infer', index_col=None, usecols=None, engine=None, skiprows=None, nrows=None)`

Parameters:

- **`filepath_or_buffer`:** Location of the csv file. It accepts any string path or URL of the file.
- **`sep`:** It stands for separator, default is `,`.
- **`header`:** It accepts int, a list of int, row numbers to use as the column names, and the start of the data. If no names are passed, i.e., `header=None`, then, it will display the first column as 0, the second as 1, and so on.
- **`usecols`:** Retrieves only selected columns from the CSV file.
- **`nrows`:** Number of rows to be displayed from the dataset.
- **`index_col`:** If `None`, there are no index numbers displayed along with records.
- **`skiprows`:** Skips passed rows in the new data frame.

Features in Pandas read_csv

1. Read specific columns using read_csv

The **usecols** parameter allows to load only specific columns from a CSV file. This reduces memory usage and processing time by importing only the required data.

```
df = pd.read_csv("people.csv", usecols=["First Name", "Email"])
print(df)
```

Output:

	First Name	Email
0	Shelby	elijah57@example.net
1	Phillip	bethany14@example.com
2	Kristine	bthompson@example.com
3	Yesenia	kaitlinkaiser@example.com
4	Lori	buchananmanuel@example.net

2. Setting an Index Column (index_col)

The **index_col** parameter **sets one or more columns as the DataFrame index**, making the specified column(s) act as row labels for easier data referencing.

```
df = pd.read_csv("people.csv", index_col="First Name")
print(df)
```

Output:

	Last Name	Sex	Email	Date of birth	Job Title
First Name					
Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath
Yesenia	Martinez	Male	kaitlinkaiser@example.com	2017-08-03	Market researcher
Lori	Todd	Male	buchananmanuel@example.net	1938-12-01	Veterinary surgeon

Using pd.read_csv, setting columns as the DataFrame index

3. Handling Missing Values Using read_csv

The **na_values** parameter replaces specified strings (e.g., "N/A", "Unknown") with NaN, enabling consistent handling of missing or incomplete data during analysis.

```
df = pd.read_csv("people.csv", na_values=["N/A", "Unknown"])
```

We won't get nan values as there is no missing value in our dataset.

4. Reading CSV Files with Different Delimiters

In this example, we will take a CSV file and then add some special characters to see how the **sep** parameter works.

```
import pandas as pd
```

```
# Sample data stored in a multi-line string
```

```
data = """totalbill_tip, sex:smoker, day_time, size
16.99, 1.01:Female|No, Sun, Dinner, 2
10.34, 1.66, Male, No|Sun:Dinner, 3
21.01:3.5_Male, No:Sun, Dinner, 3
23.68, 3.31, Male|No, Sun_Dinner, 2
24.59:3.61, Female_No, Sun, Dinner, 4
25.29, 4.71|Male, No:Sun, Dinner, 4"""
```

```
# Save the data to a CSV file
```

```
with open("sample.csv", "w") as file:
    file.write(data)
print(data)
```

Output:

```
totalbill_tip, sex:smoker, day_time, size
16.99, 1.01:Female|No, Sun, Dinner, 2
10.34, 1.66, Male, No|Sun:Dinner, 3
21.01:3.5_Male, No:Sun, Dinner, 3
23.68, 3.31, Male|No, Sun_Dinner, 2
24.59:3.61, Female_No, Sun, Dinner, 4
25.29, 4.71|Male, No:Sun, Dinner, 4
```

The sample data is stored in a multi-line string for demonstration purposes.

- **Separator (sep):** The sep='[:, |_] ' argument allows Pandas to handle multiple delimiters (:, |, _, ,) using a regular expression.
- **Engine:** The engine='python' argument is used because the default C engine does not support regular expressions for delimiters.

```
# Load the CSV file using pandas with multiple delimiters
```

```
df = pd.read_csv('sample.csv',
                 sep='[:, |_] ', # Define the delimiters
                 engine='python') # Use Python engine for regex separators
```

df

Output:

	totalbill	tip	Unnamed: 2	sex	smoker	Unnamed: 5	day
	time	Unnamed: 8	size				
16.99	NaN	1.01	Female	No	NaN	Sun	NaN
	2.0						
10.34	NaN	1.66	NaN	Male	NaN	No	Sun
	3.0						
21.01	3.50	Male	NaN	No	Sun	NaN	Dinner
	NaN						3.0
23.68	NaN	3.31	NaN	Male	No	NaN	Sun
	2.0						

24.59	3.61 4.0	NaN	Female	No	NaN	Sun	NaN	Dinner	NaN
25.29	NaN 4.0	4.71	Male	NaN	No	Sun	NaN	Dinner	NaN

5. Using nrows in read_csv()

The `nrows` parameter limits the number of rows read from a file, enabling quick previews or partial data loading for large datasets. Here, we just display only 5 rows using **`nrows` parameter**.

```
df = pd.read_csv('people.csv', nrows=3)
df
```

Output:

	First Name	Last Name	Sex	Email	Date of birth	Job Title
0	Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
1	Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
2	Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath

6. Using skiprows in read_csv()

The `skiprows` parameter skips unnecessary rows at the start of a file, which is useful for ignoring metadata or extra headers that are not part of the dataset.

```
df= pd.read_csv("people.csv")
print("Previous Dataset: ")
print(df)
# using skiprows
df = pd.read_csv("people.csv", skiprows = [4,5])
print("Dataset After skipping rows: ")
print(df)
```

Output:

Previous Dataset:

	First Name	Last Name	Sex	Email	Date of birth	Job Title
0	Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
1	Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
2	Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath
3	Yesenia	Martinez	Male	kaitlinkaiser@example.com	2017-08-03	Market researcher
4	Lori	Todd	Male	buchananmanuel@example.net	1938-12-01	Veterinary surgeon
5	Erin	Day	Male	tconner@example.org	2015-10-28	Management officer
6	Katherine	Buck	Female	conniecowan@example.com	1989-01-22	Analyst
7	Ricardo	Hinton	Male	wyattbishop@example.com	1924-03-26	Hydrogeologist

Dataset After skipping rows:

	First Name	Last Name	Sex	Email	Date of birth	Job Title
0	Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
1	Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist
2	Kristine	Travis	Male	bthompson@example.com	1992-07-02	Homeopath

Pandas Read CSV

7. Parsing Dates (parse_dates)

The `parse_dates` parameter converts date columns into datetime objects, simplifying operations like filtering, sorting, or time-based analysis.

```
df = pd.read_csv("people.csv", parse_dates=["Date of birth"])
print(df.info())
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 6 columns):
#  Column      Non-Null Count  Dtype
---  -
0  First Name    5 non-null     object
1  Last Name     5 non-null     object
2  Sex           5 non-null     object
3  Email         5 non-null     object
4  Date of birth 5 non-null     datetime64[ns]
5  Job Title     5 non-null     object
dtypes: datetime64[ns](1), object(5)
memory usage: 368.0+ bytes
```

Loading a CSV Data from a URL

Pandas allows you to directly read a CSV file hosted on the internet using the file's URL. This can be incredibly useful when working with datasets shared on websites, cloud storage, or public repositories like GitHub.

```
url = "https://media.geeksforgeeks.org/wp-content/uploads/20241121154629307916/people_data.csv"
df = pd.read_csv(url)
df
```

Output:

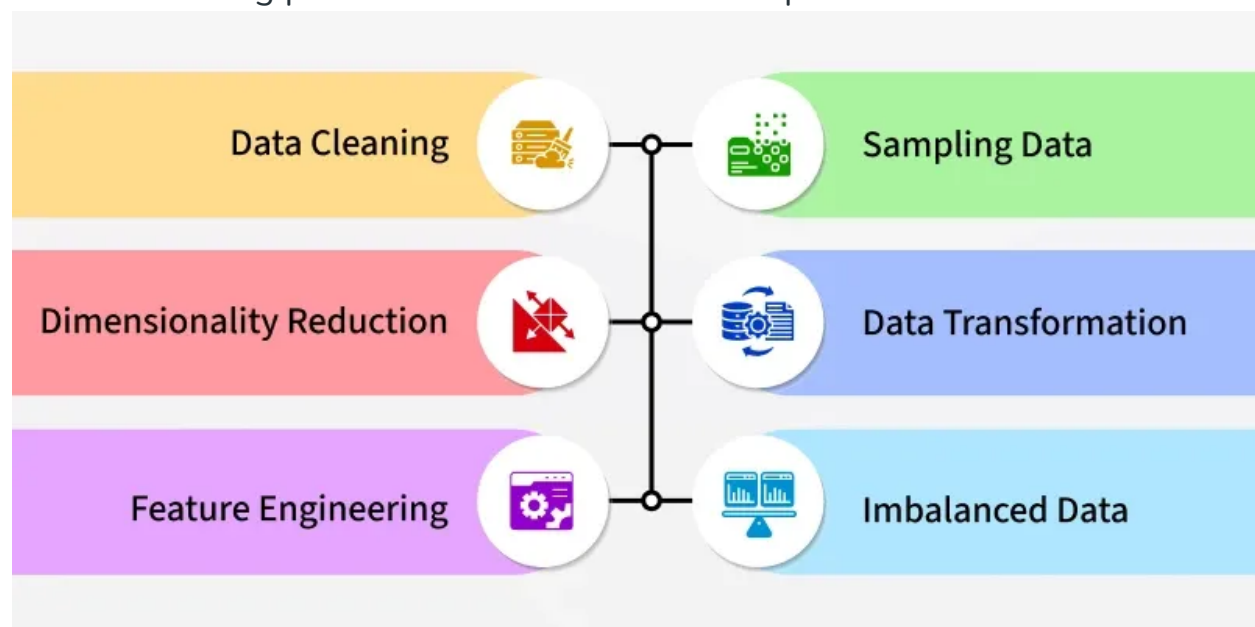
	First Name	Last Name	Sex	Email	Date of birth	Job Title
0	Shelby	Terrell	Male	elijah57@example.net	1945-10-26	Games developer
1	Phillip	Summers	Female	bethany14@example.com	1910-03-24	Phytotherapist

2	Kristine Travis	Male	bthompson@example.com	1992-07-02	
	Homeopath				
3	Yesenia Martinez	Male	kaitlinkaiser@example.com	2017-08-03	Market
	researcher				
4	Lori Todd	Male	buchananmanuel@example.net	1938-12-01	
	Veterinary surgeon				

Data Preprocessing in Python

Data preprocessing is the first step in any data analysis or machine learning pipeline. It involves cleaning, transforming and organizing raw data to ensure it is accurate, consistent and ready for modeling. It has a big impact on model building such as:

- Clean and well-structured data allows models to learn meaningful patterns rather than noise.
- Properly processed data prevents misleading inputs, leading to more reliable predictions.
- Organized data makes it simpler to create useful inputs for the model, enhancing model performance.
- Organized data supports better Exploratory Data Analysis (EDA), making patterns and trends more interpretable.



Steps of Data Preprocessing

Let's implement various preprocessing features,

Step 1: Import Libraries and Load Dataset

We prepare the environment with libraries like [pandas](#), [numpy](#), [scikit learn](#), [matplotlib](#) and [seaborn](#) for data manipulation, numerical operations, visualization and scaling. Load the dataset for preprocessing.

The sample dataset can be downloaded from [here](#).

```
import pandas as pd
```

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler
import seaborn as sns
import matplotlib.pyplot as plt
```

```
df = pd.read_csv('Geeksforgeeks/Data/diabetes.csv')
df.head()
```

Output:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Dataset

Step 2: Inspect Data Structure and Check Missing Values

We understand dataset size, data types and identify any incomplete (missing) data that needs handling.

- **df.info():** Prints concise summary including count of non-null entries and data type of each column.
- **df.isnull().sum():** Returns the number of missing values per column.

```
df.info()
print(df.isnull().sum())
```

Output:

RangeIndex: 768 entries, 0 to 767

Data columns (total 9 columns):

#	Column	Non-Null Count	Dtype
0	Pregnancies	768 non-null	int64
1	Glucose	768 non-null	int64
2	BloodPressure	768 non-null	int64
3	SkinThickness	768 non-null	int64
4	Insulin	768 non-null	int64
5	BMI	768 non-null	float64
6	DiabetesPedigreeFunction	768 non-null	float64
7	Age	768 non-null	int64
8	Outcome	768 non-null	int64

dtypes: float64(2), int64(7)

memory usage: 54.1 KB

RangeIndex: 768 entries, 0 to 767

Data columns (total 9 columns):

#	Column	Non-Null Count	Dtype
0	Pregnancies	768 non-null	int64
1	Glucose	768 non-null	int64
2	BloodPressure	768 non-null	int64
3	SkinThickness	768 non-null	int64
4	Insulin	768 non-null	int64
5	BMI	768 non-null	float64
6	DiabetesPedigreeFunction	768 non-null	float64
7	Age	768 non-null	int64
8	Outcome	768 non-null	int64

dtypes: float64(2), int64(7)

memory usage: 54.1 KB

Step 3: Statistical Summary and Visualizing Outliers

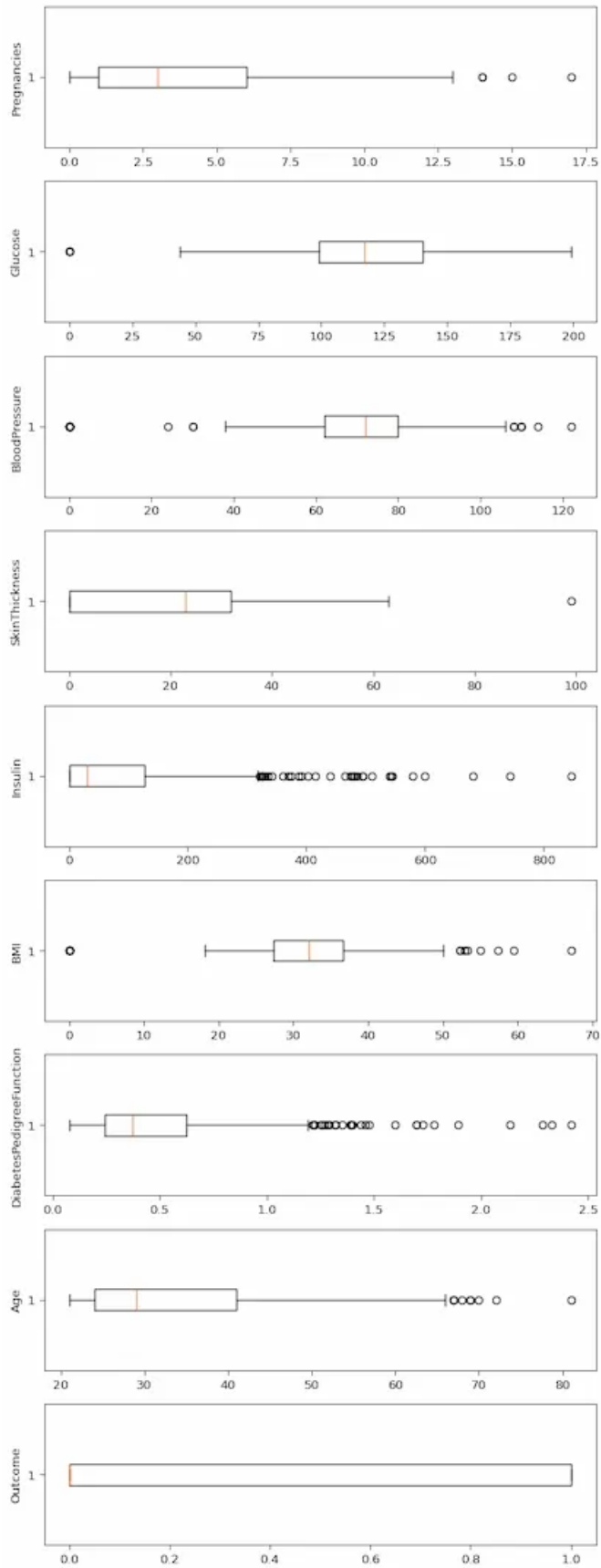
Get numeric summaries like mean, median, min/max and detect unusual points (outliers). Outliers can skew models if not handled.

- **df.describe():** Computes count, mean, std deviation, min/max and quartiles for numerical columns.
- **Boxplots:** Visualize spread and detect outliers using matplotlib's boxplot().

```
df.describe()
```

```
fig, axs = plt.subplots(len(df.columns), 1, figsize=(7, 18), dpi=95)
for i, col in enumerate(df.columns):
    axs[i].boxplot(df[col], vert=False)
    axs[i].set_ylabel(col)
plt.tight_layout()
plt.show()
```

Output:



Boxplot

Step 4: Remove Outliers Using the Interquartile Range (IQR)

Method

Remove extreme values beyond a reasonable range to improve model robustness.

- $IQR = Q3$ (75th percentile) – $Q1$ (25th percentile).
- Values below $Q1 - 1.5IQR$ or above $Q3 + 1.5IQR$ are outliers.
- Calculate lower and upper bounds for each column separately.
- Filter data points to keep only those within bounds.

```
q1, q3 = np.percentile(df['Insulin'], [25, 75])
iqr = q3 - q1
lower = q1 - 1.5 * iqr
upper = q3 + 1.5 * iqr
clean_df = df[(df['Insulin'] >= lower) & (df['Insulin'] <= upper)]
```

Step 5: Correlation Analysis

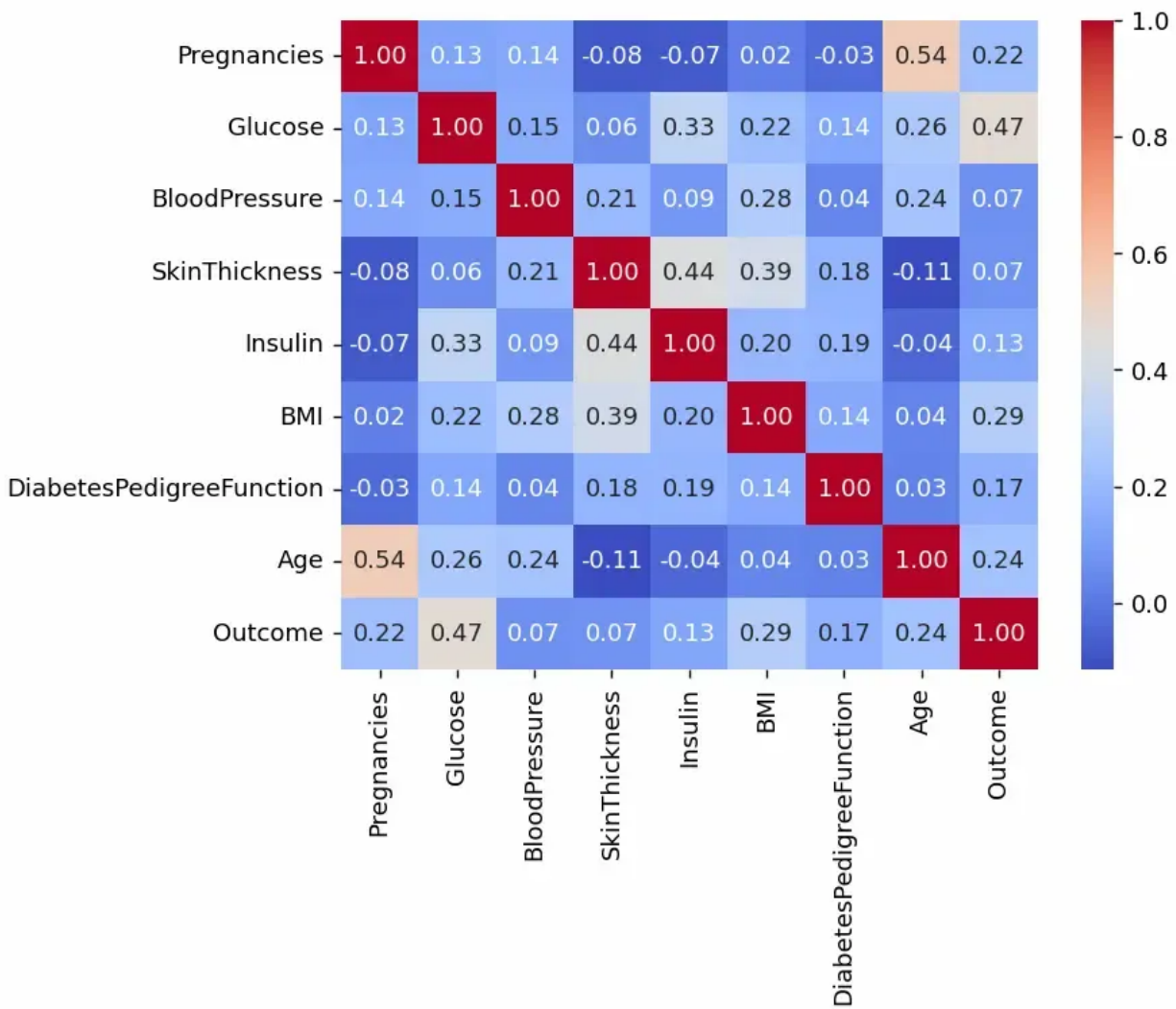
Understand relationships between features and the target variable (Outcome). Correlation helps gauge feature importance.

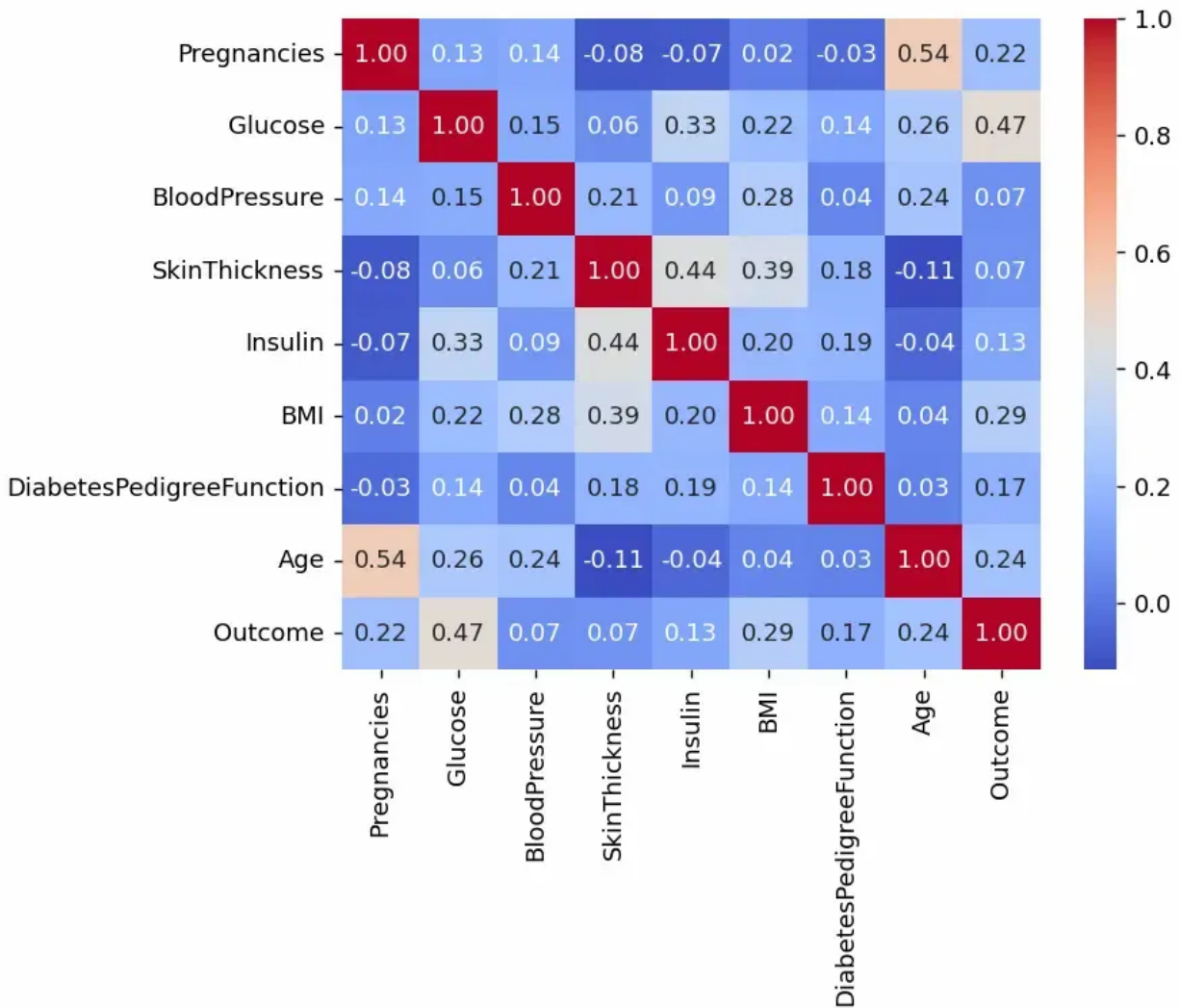
- **df.corr()**: Computes pairwise correlation coefficients between columns.
- Heatmap via seaborn visualizes correlation matrix clearly.
- Sorting correlations with `corr['Outcome'].sort_values()` highlights features most correlated with the target.

```
corr = df.corr()
plt.figure(dpi=130)
sns.heatmap(corr, annot=True, fmt='.2f', cmap='coolwarm')
plt.show()

print(corr['Outcome'].sort_values(ascending=False))
```

Output:





Step 6: Visualize Target Variable Distribution

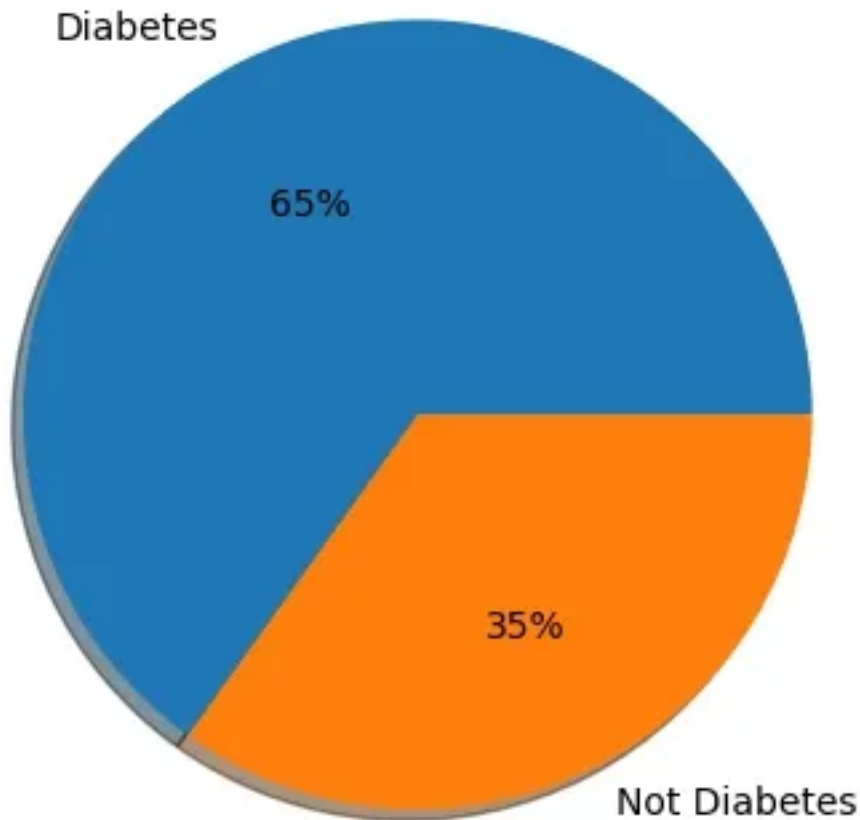
Check if target classes (Diabetes vs Not Diabetes) are balanced, affecting model training and evaluation.

- **plt.pie():** Pie chart to display proportion of each class in the target variable 'Outcome'.

```
plt.pie(df['Outcome'].value_counts(), labels=[
    'Diabetes', 'Not Diabetes'], autopct='%f%%', shadow=True)
plt.title('Outcome Proportionality')
plt.show()
```

Output:

Outcome Proportionality



Result

Step 7: Separate Features and Target Variable

Prepare independent variables (features) and dependent variable (target) separately for modeling.

- **df.drop(columns=[...]):** Drops the target column from features.
- Direct column selection df['Outcome'] selects target column.

```
X = df.drop(columns=['Outcome'])  
y = df['Outcome']
```

Step 8: Feature Scaling: Normalization and Standardization

Scale features to a common range or distribution, important for many ML algorithms sensitive to feature magnitudes.

1. Normalization (Min-Max Scaling): Rescales features between 0 and 1. Good for algorithms like k-NN and neural networks.

- **Class:** MinMaxScaler from sklearn.
- **.fit_transform():** Learns min/max from data and applies scaling.

```
scaler = MinMaxScaler()
X_normalized = scaler.fit_transform(X)
print(X_normalized[:5])
```

Output:

```
array([[0.35294118, 0.74371859, 0.59016393, 0.35353535, 0.
        0.50074516, 0.23441503, 0.48333333],
       [0.05882353, 0.42713568, 0.54098361, 0.29292929, 0.
        0.39642325, 0.11656704, 0.16666667],
       [0.47058824, 0.91959799, 0.52459016, 0.
        0.34724292, 0.25362938, 0.18333333],
       [0.05882353, 0.44723618, 0.54098361, 0.23232323, 0.11111111,
        0.41877794, 0.03800171, 0.
       ],
       [0.
        0.68844221, 0.32786885, 0.35353535, 0.19858156,
        0.64232489, 0.94363792, 0.2
       ]])
```

Normalization

2. Standardization: Transforms features to have mean = 0 and standard deviation = 1, useful for normally distributed features.

- **Class:** StandardScaler from sklearn.

```
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X)
print(X_standardized[:5])
```

Output:

```
array([[ 0.63994726,  0.84832379,  0.14964075,  0.90726993, -0.69289057,
         0.20401277,  0.46849198,  1.4259954 ],
       [-0.84488505, -1.12339636, -0.16054575,  0.53090156, -0.69289057,
        -0.68442195, -0.36506078, -0.19067191],
       [ 1.23388019,  1.94372388, -0.26394125, -1.28821221, -0.69289057,
        -1.10325546,  0.60439732, -0.10558415],
       [-0.84488505, -0.99820778, -0.16054575,  0.15453319,  0.12330164,
        -0.49404308, -0.92076261, -1.04154944],
       [-1.14185152,  0.5040552 , -1.50468724,  0.90726993,  0.76583594,
         1.4097456 ,  5.4849091 , -0.0204964 ]])
```

Standardization

Advantages

Let's see the advantages of data preprocessing,

- **Improves Data Quality:** Cleans and organizes raw data for better analysis.
- **Enhances Model Accuracy:** Removes noise and irrelevant data, leading to more precise predictions.
- **Reduces Overfitting:** Handles outliers and redundant features, improving model generalization.
- **Speeds Up Training:** Efficiently scaled data reduces computation time.
- **Ensures Algorithm Compatibility:** Converts data into formats suitable for machine learning models.