Sprints

Overview

Ask Sprinto

Sprints

Course Highlights

NumPy:
- Efficient numerical computations with multi-dimensional arrays and matrices.
- High-level mathematical functions for handling large datasets.

Pandas:
- Advanced data manipulation and analysis with Series and Data Frames.
- Importing, cleaning, and preprocessing data from various sources.
- Exploratory data analysis and data visualization techniques.

Sprinto

Sprints

What You'll Learn

- **Utilize NumPy for efficient array manipulations and mathematical operations.**
- **Employ Pandas for data handling, cleaning, and complex analyses.**
- **Integrate NumPy and Pandas in data science workflows.**
- **Visualize data to effectively communicate insights.**
- **Apply techniques to real-world datasets confidently.**

Sprinto

Sprints

Sprinto

Introduction to NumPy

Sprints

Sprinto

NumPy (Numerical Python)

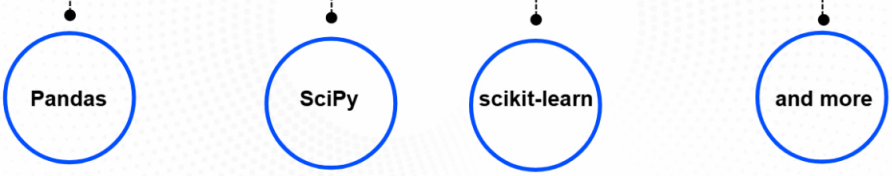
- Is a fundamental library for scientific computing in Python.
- It can deal with large data sets.
- It can deal with multi-dimensional arrays.
- It can deal with different types of data.

Sprints

Sprinto

NumPy serves as the backbone for many other Python data science libraries

such as



Sprinto

Sprints

Importance of NumPy

- crucial for performing efficient numerical computations
- essential for data analysis, machine learning, and scientific research.

Sprinto

Sprints

Key Features NumPy

- **Multi-dimensional array objects (ndarray)**
  - Efficient storage and manipulation of large datasets.
- **Mathematical functions**
  - Comprehensive collection of routines for fast operations on arrays even doe large size arrays.
- **Integration with C/C++**
  - Enables high-performance computations.

Sprinto

Sprints

Installing and Importing NumPy

Sprinto

Sprints

Installing and Importing NumPy

Installation:
NumPy can be installed using package managers like pip or conda.
Using pip: !pip install numpy
Using conda: conda install numpy

```
# Install NumPy (if not already installed)
!pip install numpy
```

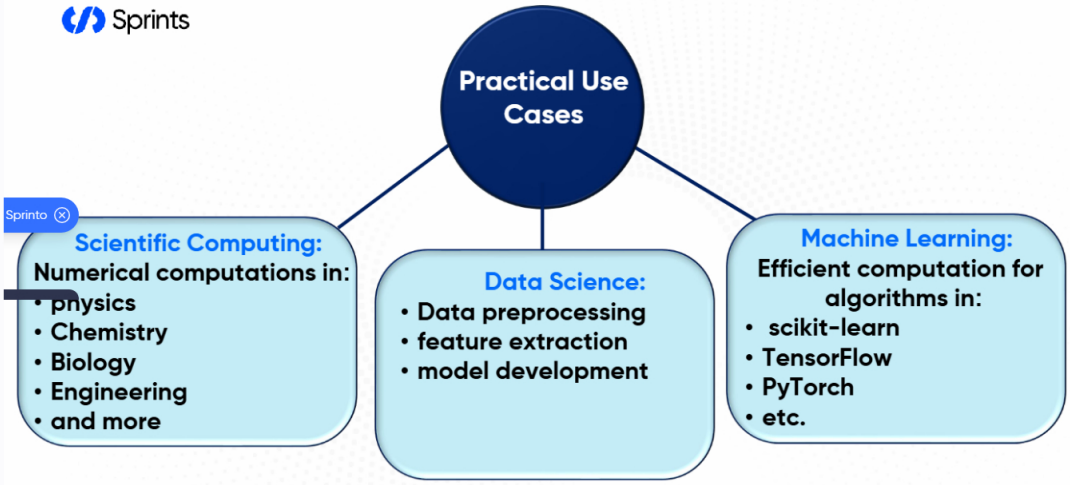
Importing NumPy:
It's a common convention to import NumPy using the alias np.

```
# Import NumPy
import numpy as np
```

Version Check:
You can check the installed version of NumPy using np.____version___.

```
# Check NumPy version
Print ("NumPy Version:", np.___version___)
```

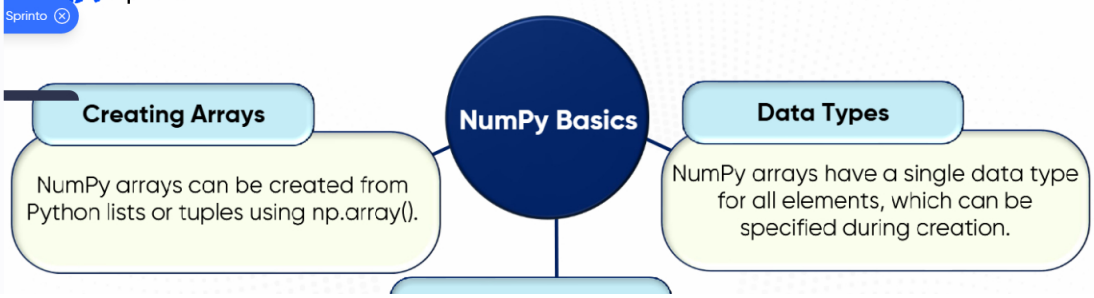
Sprints



Sprints

NumPy Basics

Sprints



Array Attributes

Understand key attributes like ndim (number of dimensions), shape (size of each dimension), and size (total number of elements).

Sprinto

Sprints

Practical Exercise:

Task: Create a one-dimensional and a two-dimensional array, explore their attributes, and perform basic operations.

```
# Creating a one-dimensional NumPy
array from a list
data_1d = [1, 2, 3, 4, 5]
array_1d = np.array(data_1d)
print("1D Array:", array_1d)
print("Dimensions:", array_1d.ndim)
print("Shape:", array_1d.shape)
print("Size:", array_1d.size)
print("Data Type:", array_1d.dtype)
```

Sprinto

Sprints

Practical Exercise:

Task: Create a one-dimensional and a two-dimensional array, explore their attributes, and perform basic operations.

```
# Creating a one-dimensional NumPy
array from a list
data_1d = [1, 2, 3, 4, 5]
array_1d = np.array(data_1d)
print("1D Array:", array_1d)
print("Dimensions:", array_1d.ndim)
print("Shape:", array_1d.shape)
print("Size:", array_1d.size)
print("Data Type:", array_1d.dtype)
```

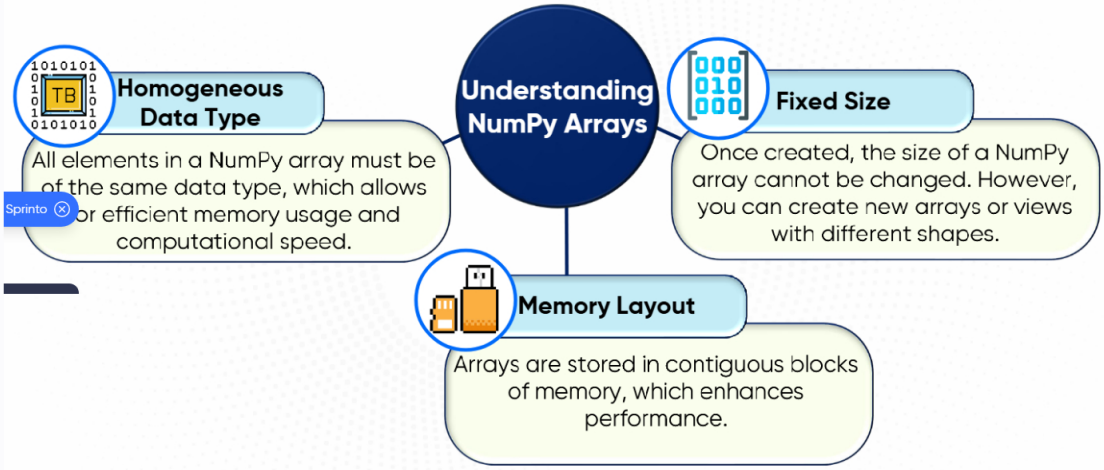
Sprinto

Sprints

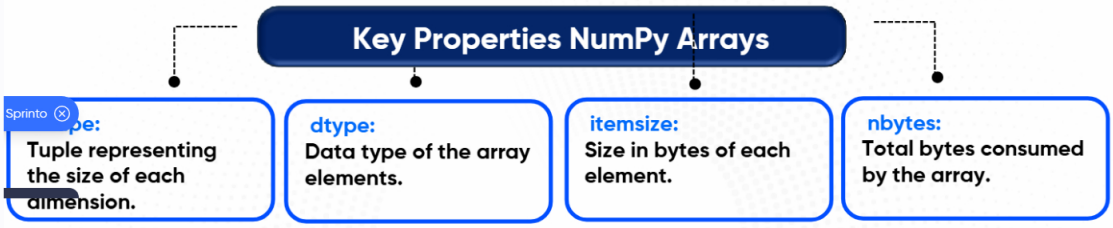
Understanding NumPy Arrays

Sprinto

Sprints



Sprints

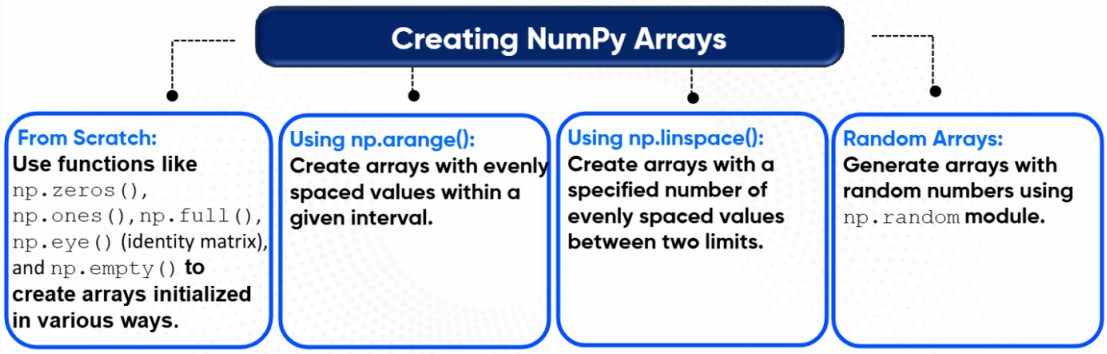



```
# Exploring array properties
print("Array Shape:", array_2d.shape)
print("Data Type:", array_2d.dtype)
print("Item Size:", array_2d.itemsize,
"bytes")
print("Total Size:", array_2d.nbytes,
"bytes")
```

Sprints

Creating NumPy Arrays

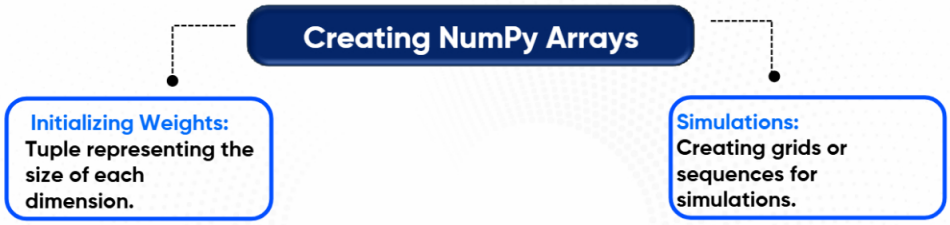
Sprints



Sprints


```
# Creating arrays with different functions
zeros_array = np.zeros((2, 3))
ones_array = np.ones((2, 3))
full_array = np.full((2, 3), 7)
identity_matrix = np.eye(3)
empty_array = np.empty((2, 3))

sequence_array = np.arange(0, 10, 2)  # Start, Stop, Step
linspace_array = np.linspace(0, 1, 5)  # Start, Stop, Number of samples
```


Sprints




```
print("Zeros Array:\n", zeros_array)
print("\nOnes Array:\n", ones_array)
print("\nFull Array (7s):\n", full_array)
print("\nIdentity Matrix:\n", identity_matrix)
print("\nEmpty Array:\n", empty_array)
print("\nSequence Array:", sequence_array)
print("\nLinspace Array:", linspace_array)
```


Sprints

Creating NumPy Arrays from Files

Sprinto

Sprints

Creating NumPy Arrays from Files

Loading Data:
Use functions like `np.loadtxt()`, or `np.load()` to read data from text files or binary files.

Data Formats:
NumPy can read and write arrays to disk in various formats, including text, CSV, and its own binary .npy format.

Sprints

Example Use Case

- Loading CSV Data:

```
# Assuming 'data.csv' contains numerical data
data = np.loadtxt('data.csv', delimiter=',')
print("Loaded Data:\n", data)
```

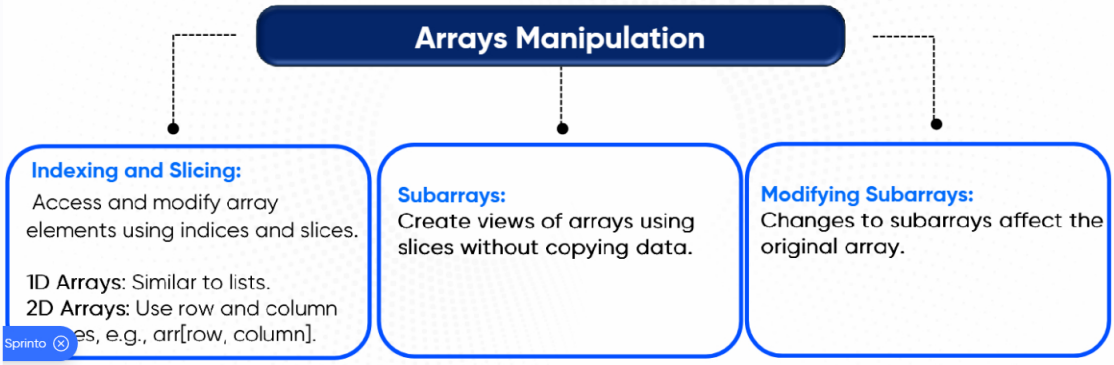
- Note: For demonstration purposes, you can generate sample data and save it to a file, then load it.

```
# Generating sample data and saving to a file
sample_data = np.random.rand(5, 3)
np.savetxt('sample_data.csv', sample_data, delimiter=',', fmt='%.2f')

# Loading the data back
loaded_data = np.loadtxt('sample_data.csv', delimiter=',')
print("Loaded Data from CSV:\n", loaded_data)
```

Sprinto Sprints

Arrays Manipulation

Sprints



Sprints

Practical Exercise

- Task: Slice and modify elements in arrays, and observe how it affects the original array.

```
arr = np.array([10, 20, 30, 40, 50])
print("Original Array:", arr)

# Indexing
print("Element at index 1:", arr[1])

# Slicing
("Slice from index 1 to 3:", arr[1:4])

# Modifying a slice
arr_slice = arr[1:4]
arr_slice[0] = 25
print("\nModified Slice:", arr_slice)
print("Array After Slice Modification:", arr)
```

Sprints

Practical Exercise

- Task: Slice and modify elements in arrays, and observe how it affects the original array.

```
arr_slice = arr[1:4]
arr_slice[0] = 25
print("\nModified Slice:", arr_slice)
print("Array After Slice Modification:", arr)
```

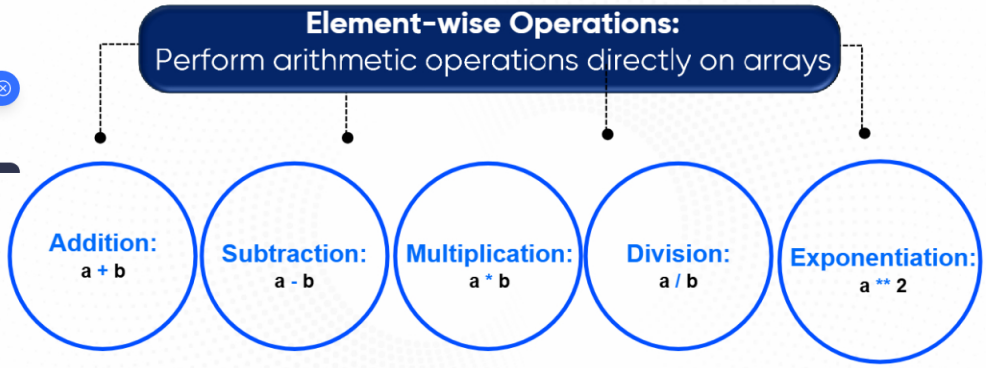
```
# Concatenation
arr2 = np.array([60, 70])
concatenated_arr = np.concatenate((arr, arr2))
print("\nConcatenated Array:", concatenated_arr)
```

• **Note:** Modifying `arr_slice` affects `arr` because slices are views, not copies.

Sprints

Arrays Math Operations

Sprints



Broadcasting: NumPy can perform operations on arrays of different shapes (e.g., scalar and array).

Sprints

Practical Use Case

• **Data Normalization:** Apply mathematical transformations across datasets.

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

# Arithmetic operations
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
```

Sprinto

Sprints

Practical Use Case

- **Data Normalization:** Apply mathematical transformations across datasets.

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

# Scalar operations
print("\nScalar Addition:", a + 10)
print("Scalar Multiplication:", a * 3)

# Broadcasting Example
c = np.array([[1], [2], [3]])
d = np.array([4, 5, 6])
print("\nBroadcasting Multiplication:\n", c * d)
```

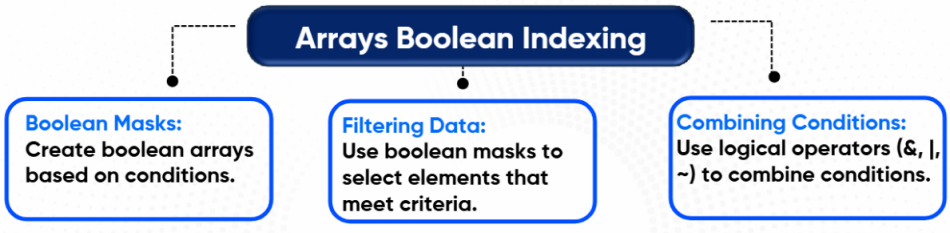
Sprinto

Sprints

Arrays Boolean Indexing

Sprinto

Sprints



Sprinto

Sprints

Practical Use Case

- Task: Filter an array to extract elements based on multiple conditions.

```
# Boolean indexing example
a = np.array([1, 2, 3, 4, 5])
mask = (a > 2) & (a < 5)
filtered_array = a[mask]
print("Filtered Array (Elements >2 and <5):", filtered_array)

# Modifying elements based on condition
a[a > 3] = 10
print("Array After Modification:", a)
```

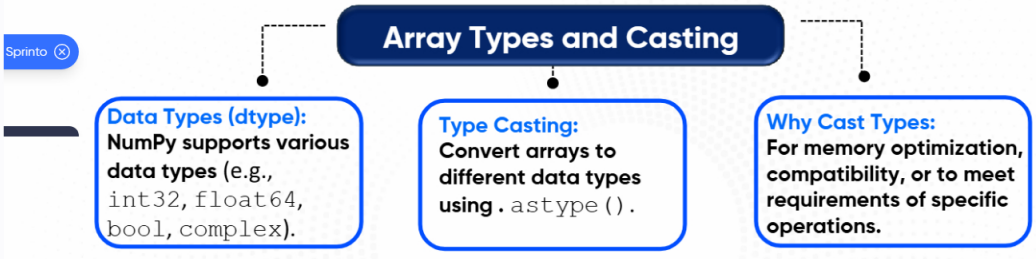
Sprints

Sprinto

Array Types and Casting

Sprints

Sprinto



Sprinto Sprints

Practical Use Case

Practical Use Case

• Task: Explore type casting and observe the effects on data.

```
float_array = np.array([1.1, 2.2, 3.3], dtype='float64')
print("Float Array:", float_array)
print("Data Type:", float_array.dtype)

# Casting to integer
int_array = float_array.astype('int32')
print("Casted to Integer:", int_array)
print("Data Type:", int_array.dtype)

# Casting to boolean
bool_array = np.array([0, 1, 2, 0], dtype='int32')
bool_array = bool_array.astype('bool')
print("Casted to Boolean:", bool_array)
print("Data Type:", bool_array.dtype)
```


Random Number Generation

Random Number Generation

Random Arrays:
Generate random numbers with various distributions.

- **Uniform distribution:** `np.random.rand()`
- **Normal distribution:** `np.random.randn()`
- **Integers:** `np.random.randint(low, high, size)`

Seed Setting:
For reproducibility, set the seed using `np.random.seed()`.

Practical Use Case

• Simulation: Generate random datasets for testing algorithms.

```
# Set seed for reproducibility
```

```
np.random.seed(42)

# Uniform distribution [0, 1)
random_array_uniform = np.random.rand(5)
print("Uniform Random Array:", random_array_uniform)

# Normal distribution (mean=0, std=1)
random_array_normal = np.random.randn(5)
print("Normal Random Array:", random_array_normal)

# Random integers between low (inclusive) and high (exclusive)
random_ints = np.random.randint(1, 100, size=5)
print("Random Integers:", random_ints)
```

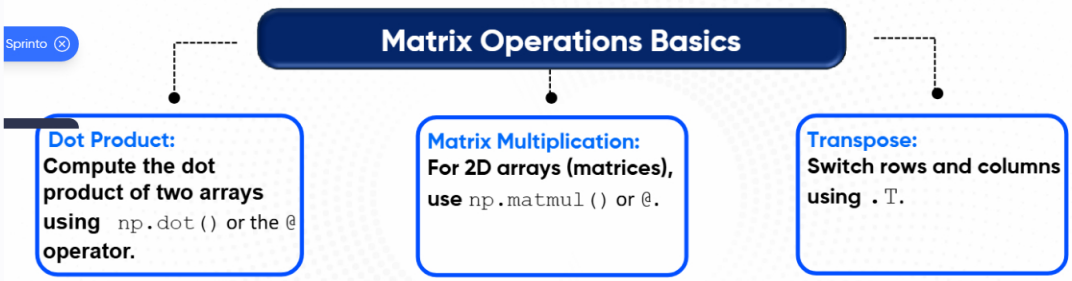
Sprints

Sprinto

Matrix Operations Basics

Sprinto

Sprints



Sprints

Sprinto

Practical Use Case

Linear Algebra: Essential for solving equations, transformations, and more.

```
# Set seed for reproducibility
np.random.seed(42)

# Uniform distribution [0, 1)
random_array_uniform = np.random.rand(5)
```

```
print("Uniform Random Array:", random_array_uniform)

# Normal distribution (mean=0, std=1)
random_array_normal = np.random.randn(5)
print("Normal Random Array:", random_array_normal)

# Random integers between low (inclusive) and high (exclusive)
random_ints = np.random.randint(1, 100, size=5)
print("Random Integers:", random_ints)
```

Sprinto

Sprints

NumPy Universal Functions (ufuncs)

Sprinto

Sprints

NumPy Universal Functions (ufuncs)

- **Functions that operate element-wise on arrays.**
- **Examples of ufuncs:**
- **Mathematical:** `np.sqrt(), np.exp(), np.log().`
- **Benefits: Faster and more efficient than looping over array elements.**

Sprinto

Sprints

Practical Use Case

- Task: Apply various ufuncs to an array and analyze the results.

```
# Applying universal functions
a = np.array([1, 4, 9, 16, 25])

sqrt_array = np.sqrt(a)
exp_array = np.exp(a)
log_array = np.log(a)
```

Sprinto

```
print("Original Array:", a)
print("Square Root:", sqrt_array)
print("Exponential:", exp_array)
```

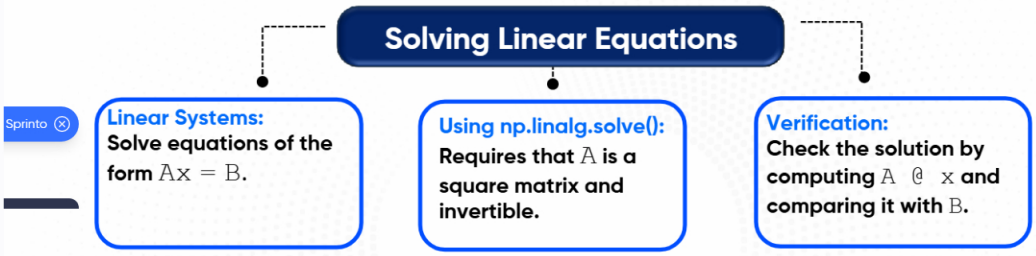
```
print("Logarithm:", log_array)
```

Sprints

Solving Linear Equations

Sprinto

Sprints



Sprinto

Sprints

Practical Use Case

Sprinto

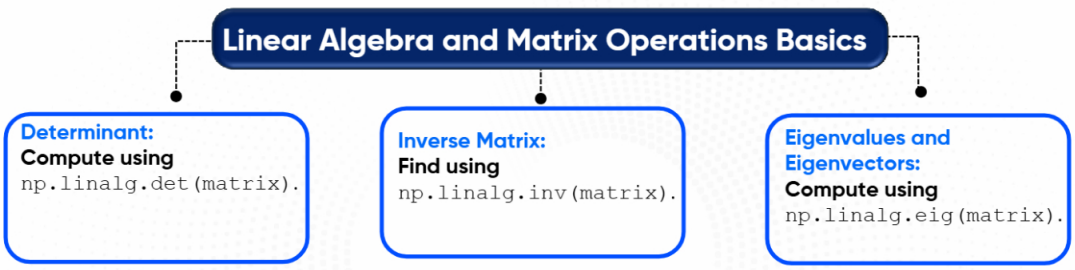
- **Engineering and Physics Problems:** Circuit analysis, mechanical structures, etc.

```
A = np.array([[3, 1], [1, 2]])
B = np.array([9, 8])

# Solving linear equations
solution = np.linalg.solve(A, B)
print("Solution x:", solution)

# Verification
verification = A @ solution
print("Verification (A @ x):", verification)
print("Original B:", B)
```

Sprints

Linear Algebra and Matrix Operations Basics

Sprints



Sprints

Practical Use Case

- Task: Compute determinant, inverse, eigenvalues, and eigenvectors.

```
matrix = np.array([[2, 1], [1, 2]])

# Determinant
determinant = np.linalg.det(matrix)
print("Determinant:", determinant)

verse
inverse_matrix = np.linalg.inv(matrix)
print("\nInverse Matrix:\n", inverse_matrix)

# Eigenvalues and Eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(matrix)
print("\nEigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

Sprints

Reshaping Arrays

Sprinto

Sprints

Reshaping Arrays

Reshape:
nge the dimensions of an array using `reshape()`.

Flatten:
Convert multi-dimensional arrays to 1D using `flatten()` **or** `.ravel()`.

Reshaping Rules:
The total number of elements must remain the same.

Sprints

Practical Use Case

Sprinto

k: Reshape arrays into different dimensions and understand their layout.

```
array = np.arange(12)
print("Original Array:", array)

# Reshape to 3x4
reshaped_array = array.reshape(3, 4)
print("\nReshaped to 3x4:\n", reshaped_array)

# Reshape to 2x2x3
reshaped_array_3d = array.reshape(2, 2, 3)
print("\nReshaped to 2x2x3:\n", reshaped_array_3d)

# Flatten the array
flattened_array = reshaped_array.flatten()
print("\nFlattened Array:", flattened_array)
```

Sprints

Sprinto

Stacking and Splitting Arrays

Sprinto

Sprints

Stacking and Splitting Arrays

Stacking Arrays:

Vertical Stack: `np.vstack((array1, array2))` **stacks arrays vertically (row-wise).**

Horizontal Stack: `np.hstack((array1, array2))` **stacks arrays horizontally (column-wise).**

Depth Stack: `np.dstack((array1, array2))` **stacks arrays along the third axis.**

Splitting Arrays:

Split: `np.split(array, indices_or_sections,axis=0)` **splits an array into multiple sub-arrays.**

Horizontal Split: `np.hsplit().`

Vertical Split: `np.vsplit().`

Sprinto

Sprints

Practical Use Case

- Task: Stack and split arrays in different ways and observe the results.

```
array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[5, 6], [7, 8]])

# Vertical Stack
v_stacked = np.vstack((array1, array2))
print("Vertical Stacked Arrays:\n", v_stacked)

# Horizontal Stack
h_stacked = np.hstack((array1, array2))
print("\nHorizontal Stacked Arrays:\n", h_stacked)

# Splitting arrays
split_array = np.hsplit(h_stacked, 2)
print("\nSplit Arrays:")
for i, arr in enumerate(split_array):
    print(f"Array {i}:\n", arr)
```

Sprinto

Sprints

Advanced Indexing

Sprints

Advanced Indexing

Fancy Indexing:
Use arrays of indices to
ess multiple array
ments at once.

Index Arrays:
Can be integer arrays or boolean arrays (masks).

Application:
Useful for reordering data or selecting arbitrary elements.

Sprints

Practical Use Case

- Task: Use fancy indexing to reorder elements and extract specific patterns.

```
array = np.arange(10)
indices = [2, 5, 3]
fancy_indexed_elements = array[indices]
print("Original Array:", array)
print("Indices:", indices)
print("Fancy Indexed Elements:", fancy_indexed_elements)

# Using boolean array for indexing
bool_indices = array % 2 == 0  # Select even numbers
even_elements = array[bool_indices]
print("\nBoolean Indices:", bool_indices)
print("Even Elements:", even_elements)
```

Sprints

Overiew on Pandas

Sprinto

Sprints

Pandas

An open-source library providing high-performance, easy-to-use tables and data analysis tools.

Sprinto

Sprints

Key Data Structures Pandas

Series:
One-dimensional labeled array capable of holding any data type.

DataFrame:
Two-dimensional labeled data structure with columns of potentially different types (series).

Sprinto

Sprints

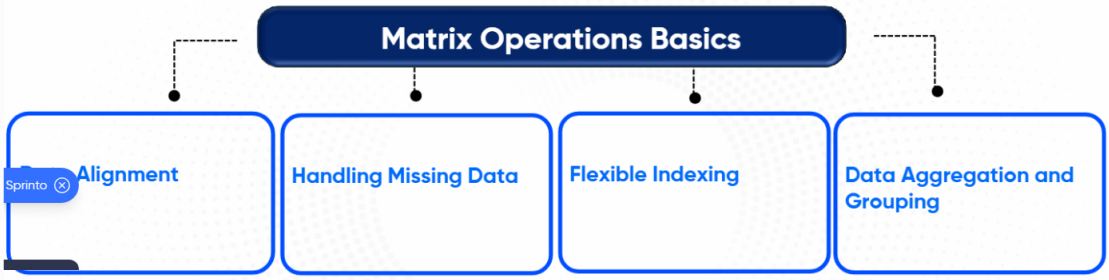
Why Pandas?

Simplifies data manipulation, analysis, and cleaning.

Ideal for working with table shape.

Sprinto

Sprints



Sprinto

Sprints

Sprinto

Series and DataFrames

Sprints

Sprinto

Series and DataFrames

Creating a Series:

- **From a list or array:** `pd.Series(data)`
- **With a custom index:** `pd.Series(data, index=index)`

Creating a DataFrame:

- **From a dictionary of Series or arrays.**
- **From a list of dictionaries.**
- **From a NumPy array with specified column names.**

Sprints

Sprinto

Practical Use Case

Practical Use Case

• Task: Create Series and DataFrames from various data sources.

```
# Creating a Pandas Series with custom index
data = [10, 20, 30, 40]
index = ['a', 'b', 'c', 'd']
series = pd.Series(data, index=index)
print("Series with Custom Index:\n", series)

# Creating a DataFrame from a dictionary
data_dict = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 70000]
}
data_frame = pd.DataFrame(data_dict)
print("\nDataFrame from Dictionary:\n", data_frame)
```


Creating Series and DataFrames

Creating Series and DataFrames

• From NumPy Arrays:

```
# Creating DataFrame from NumPy arrays
array_data = np.array([[1, 2], [3, 4], [5, 6]])
columns = ['Column1', 'Column2']
df_from_array = pd.DataFrame(array_data, columns=columns)
print("DataFrame from NumPy Array:\n", df_from_array)
```


Creating Series and DataFrames

• Specifying Index and Columns:

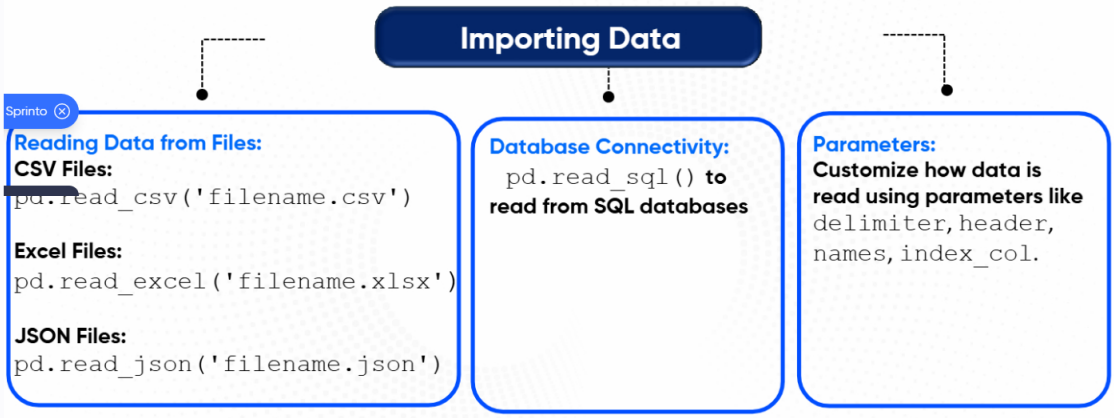
```
# Creating DataFrame with custom index and columns
data = {
```

```
    'Column1': pd.Series([1, 2, 3], index=['a', 'b', 'c']),
    'Column2': pd.Series([4, 5, 6], index=['a', 'b', 'c'])
}
custom_df = pd.DataFrame(data)
print("\nCustom DataFrame:\n", custom_df)
```

Sprints

Importing Data

Sprints



Sprints

Practical Use Case

- Load and Explore Data:

```
# Example of importing data from a CSV file
df = pd.read_csv('data.csv')
print("First 5 Rows:\n", df.head())
print("\nDataFrame Info:")
df.info()
```

Sprinto

Sprints

Practical Use Case

• **Note:** For practice, you can create a sample CSV file using Pandas.

```
# Create a sample DataFrame and save to CSV
sample_df = pd.DataFrame({
    'A': np.random.randint(1, 10, size=5),
    'B': np.random.rand(5),
    'C': ['foo', 'bar', 'baz', 'qux', 'quux']
})
sample_df.to_csv('sample_data.csv', index=False)
```

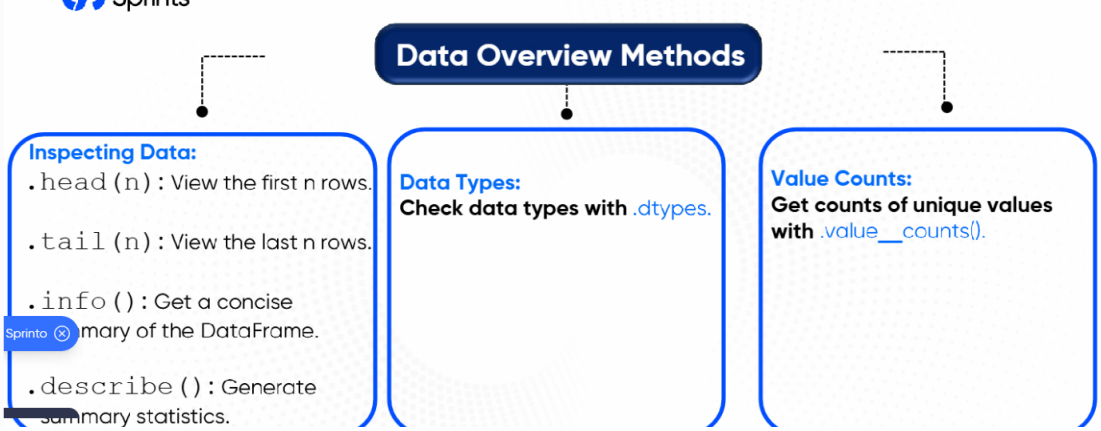
Sprinto

Sprints

Data Overview Methods

Sprinto

Sprints



Sprinto

Sprints

Practical Use Case

- Task: Use various methods to get an overview of the dataset.

```
print("First 5 Rows:\n", df.head())
print("\nDataFrame Info:")
df.info()
print("\nSummary Statistics:\n", df.describe())

# Checking data types
print("\nData Types:\n", df.dtypes)

# Value counts for column 'A'
print("\nValue Counts for Column 'A':\n", df['A'].value_counts())
```

Sprints

Indexing and Selection

Sprints

Indexing and Selection

Selection by Label:

Single Column:
`df['ColumnName']` or `df.ColumnName`

Multiple Columns:
`df[['Col1', 'Col2']]`

Selection by Position:

Rows: `df[0:3]` selects rows by position.

Using `.loc[]`: Access a group of rows and columns by labels.

Using `.iloc[]`: Access by integer position.

Sprinto Sprints

Practical Use Case

• Task: Practice selecting data using different methods.

```
# Selecting a single column
col_a = df['A']
print("Column A:\n", col_a)

# Selecting multiple columns
cols_ab = df[['A', 'B']]
print("\nColumns A and B:\n", cols_ab)

# Selecting rows by position
rows_1_to_3 = df[1:4]
print("\nRows 1 to 3:\n", rows_1_to_3)
```

Sprinto

Sprinto Sprints

Practical Use Case

• Task: Practice selecting data using different methods.

```
# Using .loc[] for label-based selection
# Assuming the DataFrame has a label-based index
df.set_index('C', inplace=True)
selected_data = df.loc['foo']
print("\nData for label 'foo':\n", selected_data)

# Using .iloc[] for position-based selection
selected_data_iloc = df.iloc[2]
print("\nData at position 2:\n", selected_data_iloc)
```

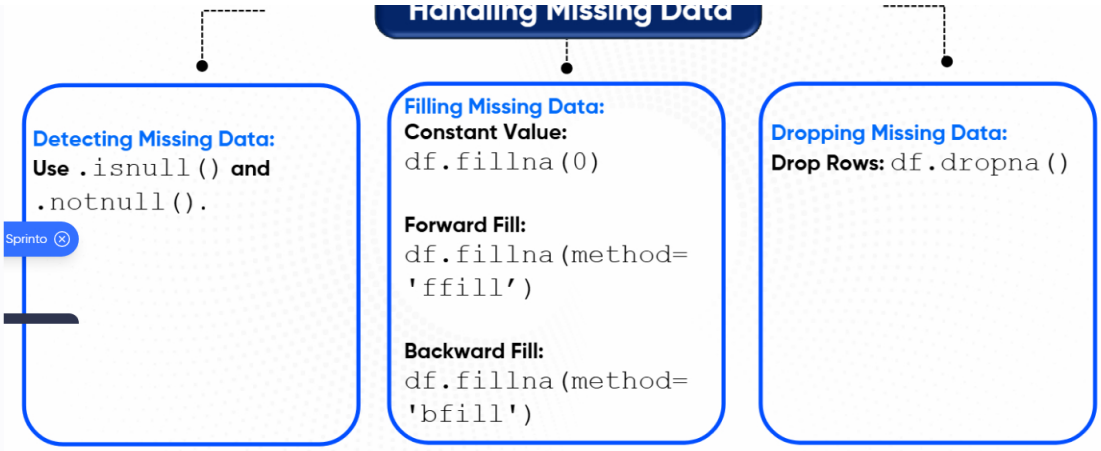
Sprinto

Sprints

Handling Missing Data

Sprinto

Sprints



Sprints

Practical Use Case

- Task: Handle missing data in a DataFrame.

```
# Create DataFrame with missing values
data_with_nan = {
    'A': [1, np.nan, 3],
    'B': [4, 5, np.nan],
    'C': [7, 8, 9]
}
df_nan = pd.DataFrame(data_with_nan)
print("DataFrame with Missing Values:\n", df_nan)
```

Sprints

Practical Use Case

- Task: Handle missing data in a DataFrame.

```
# Detect missing values
print("\nIs Null:\n", df_nan.isnull())

# Fill missing values with 0
df_filled = df_nan.fillna(0)
print("\nDataFrame After Filling Missing Values:\n", df_filled)

# Drop rows with missing values
df_dropped = df_nan.dropna()
print("\nDataFrame After Dropping Rows with Missing Values:\n",
df_dropped)
```

Sprints

Merging and Joining DataFrames

Sprinto

Sprints

Merging and Joining DataFrames

`pd.merge()`:
Inner Join: Only matching keys.

Outer Join: All keys, combining data.

Left Join: All keys from left DataFrame.

Right Join: All keys from right DataFrame.

`join()`:
Convenient for joining on index.

`pd.concat()`:
Append or stack DataFrames.

Sprinto

Sprints

Methods to Combine DataFrames (`pd.merge()`)

1- pd.merge()

```
df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'data1': [1, 2, 3]})
df2 = pd.DataFrame({'key': ['B', 'C', 'D'], 'data2': [4, 5, 6]})

merged_df = pd.merge(df1, df2, on='key', how='outer')
print("Merged DataFrame (Outer Join):\n", merged_df)
```

Sprinto

Sprints

Methods to Combine DataFrames (`.join()`)

2- `.join()`

```
df1.set_index('key', inplace=True)
df2.set_index('key', inplace=True)

joined_df = df1.join(df2, how='outer')
print("\nJoined DataFrame (Outer Join):\n", joined df)
```

Sprints

Methods to Combine DataFrames (pd.concat())

3- pd.concat()

```
df_concat = pd.concat([df1, df2], axis=0)
print("\nConcatenated DataFrame (Axis=0):\n", df_concat)
```

Sprints

Differences

pd.merge():

Versatile, merges on key columns or indexes, supports different types of joins.

join():

Primarily joins on index, suitable for DataFrames with similar indexes.

pd.concat():

Concatenates along a particular axis, useful for stacking DataFrames.

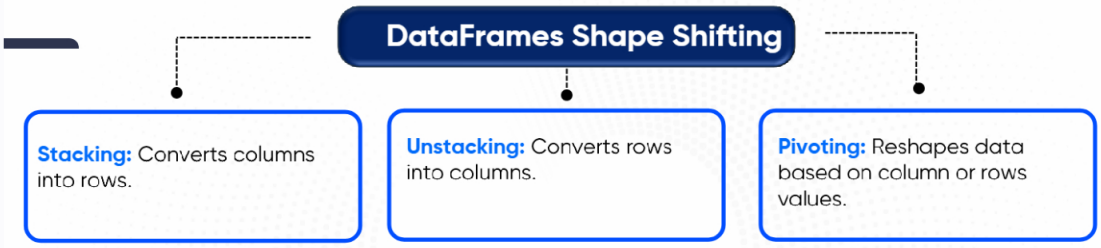
Sprints

Sprints

DataFrames Shape Shifting

Sprinto

Sprints



Sprinto

Sprints

Practical Use Case

- Task: Reshape a DataFrame using `stack()`, `unstack()`, and `pivot()`.

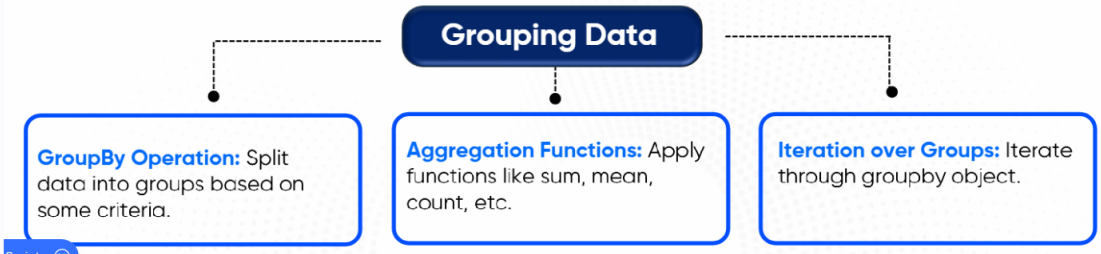
```
# Create a sample DataFrame
df_sample = pd.DataFrame({
    'A': ['foo', 'bar', 'foo', 'bar'],
    'B': ['one', 'one', 'two', 'three'],
    'C': np.random.randn(4),
    'D': np.random.randn(4)
})
print("Original DataFrame:\n", df_sample)

Sprinto t multi-index
df_multi = df_sample.set_index(['A', 'B'])
print("\nDataFrame with MultiIndex:\n", df_multi)
```

Sprints

Grouping Data

Sprinto

Sprints



Sprinto

Sprints

Practical Use Case

- Task: Group data and perform aggregate operations.

Sprinto

```
# Group data by column 'A'
grouped = df_sample.groupby('A')

# Calculate sum of 'C' and 'D' for each group
grouped_sum = grouped[['C', 'D']].sum()
print("Grouped Sum:\n", grouped_sum)

# Iterate over groups
for name, group in grouped:
    print(f"\nGroup: {name}")
    print(group)
```

Sprints

Sprinto

Aggregation Functions

Sprints

Sprinto

Aggregation Functions

Common Aggregations:

Sum: `df.sum()`

Mean: `df.mean()`

Median: `df.median()`

Max/Min: `df.max()`, `df.min()`

Standard Deviation: `df.std()`

Aggregation over Axis:

Specify axis with `axis` **parameter**

Sprints

Practical Use Case

- Task: Apply aggregation functions to the entire DataFrame and specific columns.

```
# Sum of all numeric columns
total_sum = df_sample[['C', 'D']].sum()
print("Total Sum:\n", total_sum)

# Mean of each group
group_mean = df_sample.groupby('A').mean()
print("\nGroup Mean:\n", group_mean)
```


Sprints

Advanced Aggregation

Sprints

Advanced Aggregation

Custom Aggregations with `.agg()`**:**

- Apply multiple functions to columns.
- Use dictionaries to specify functions per column.

Sprinto

Sprints

Practical Use Case

- Task: Perform advanced aggregations using `.agg()`.

```
# Apply multiple aggregations
agg_results = df_sample.groupby('A').agg({
    'C': ['sum', 'mean', 'max'],
    'D': ['min', 'std']
})
print("Advanced Aggregation Results:\n", agg_results)
```

Sprinto

Sprints

Sprinto

DataFrame Operations

Sprints

Sprinto

DataFrame Operations

Transposing DataFrames:

- Use `.T` to transpose, swapping rows and columns.

Sorting Data:

- Use `.sort_index()` or `.sort_values()` to sort data.

Adding and Dropping Columns:

- Add columns by assignment.
- Drop columns using `.drop()`.

Sprinto

Sprints

Practical Use Case

- Task: Perform various operations on a DataFrame.

```
# Transpose
transposed_df = time_series_df.T
print("Transposed DataFrame:\n", transposed_df)

# Sorting by index
sorted_df = time_series_df.sort_index(ascending=False)
print("\nDataFrame Sorted by Index (Descending):\n", sorted_df)
```

Sprinto

Sprints

Practical Use Case

- Task: Perform various operations on a DataFrame.

```
# Adding a new column
time_series_df['Profit'] = time_series_df['Sales'] -
time_series_df['Expenses']
print("\nDataFrame with Profit Column:\n", time_series_df)

# Dropping a column
time_series_df = time_series_df.drop('Expenses', axis=1)
print("\nDataFrame After Dropping 'Expenses' Column:\n",
time_series_df)
```

Sprinto

Sprints

Basic Visualization with Pandas

Sprinto

Sprints

Basic Visualization with Pandas

Plotting Methods:

Line Plot: `df.plot.line()`

Bar Plot: `df.plot.bar()`

Histogram: `df.plot.hist()`

Scatter Plot: `df.max(),df.min()`

Customization:

Use parameters like `title, xlabel, ylabel, figsize.`

Sprints

Practical Use Case

- Task: Create various plots to visualize data.

```
rt matplotlib.pyplot as plt

# Line Plot
time_series_df.plot(y='Sales', kind='line', title='Sales Over Time')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.show()

# Bar Plot
time_series_df.plot(y='Profit', kind='bar', title='Profit Over Time')
plt.xlabel('Date')
plt.ylabel('Profit')
plt.show()
```

Sprints

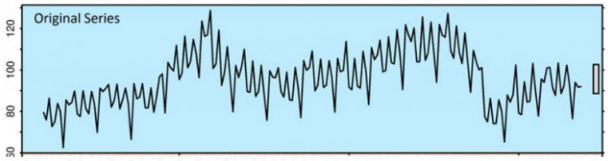
Time Series Basics

Sprints

Time Series Analysis

Is a statistical technique that deals with time-ordered data points.

It involves methods for analyzing time series data to extract meaningful statistics and characteristics.



Stock Exchange | Temperatures | Sales | Call Center

Sprints

Why is it Important?

- Forecasting future values
- Understanding past behavior
- Identifying trends and seasonality
- Anomaly detection

Sprints

Example Application

Finance
Stock prices, exchange rates

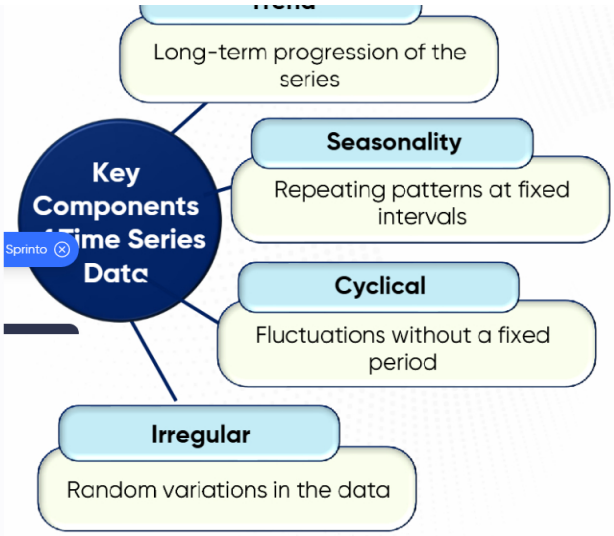
Economics
GDP growth, unemployment rates

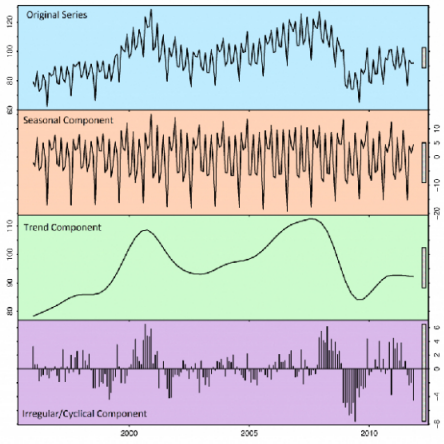
Weather
Temperature, rainfall predictions

Business
Sales forecasting, demand prediction

Sprints

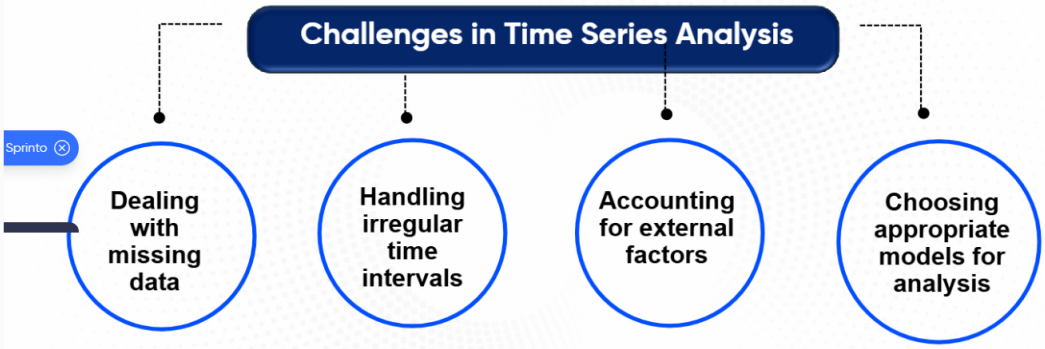
Trend





Sprinto

Sprints



Sprinto

Sprints

Practical Use Case

- Task: Create and manipulate time series data.

Sprinto

```
# Create date range
dates = pd.date_range('2024-01-01', periods=10, freq='D')
time_series_df = pd.DataFrame({
    'Sales': np.random.randint(100, 200, size=10),
    'Expenses': np.random.randint(50, 100, size=10)
}, index=dates)
print("Time Series DataFrame:\n", time_series_df)

# Access data by date
print("\nData on 2024-01-05:\n", time_series_df.loc['2024-01-05'])
```

Sprints

Sprinto

Time Series Analysis

Sprinto

Sprints

Time Series Analysis

Resampling Data:
Aggregate data over a new time frequency using `.resample()`.

Rolling Statistics:
Compute rolling averages or other statistics using `.rolling(window)`.

Sprinto

Sprints

Practical Use Case

- Task: Resample and compute rolling averages.

```
# Resample to weekly frequency and sum
weekly_sales = time_series_df['Sales'].resample('W').sum()
print("Weekly Sales:\n", weekly_sales)
# Plot weekly sales
weekly_sales.plot(kind='bar', title='Weekly Sales')
plt.xlabel('Week')
plt.ylabel('Sales')
plt.show()
```

Sprinto

Sprints

Practical Use Case

- Task: Resample and compute rolling averages.

```
# Compute rolling mean
time_series_df['Sales_Rolling_Mean'] =
time_series_df['Sales'].rolling(window=3).mean()
print("\nDataFrame with Rolling Mean:\n", time_series_df)
# Plot rolling mean
```

```
time_series_df[['Sales', 'Sales_Rolling_Mean']].plot(title='Sales with
Rolling Mean')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.show()
```

Sprints

Practical Use Case

- Task: Resample and compute rolling averages.

```
# Compute rolling mean
time_series_df['Sales_Rolling_Mean'] =
time_series_df['Sales'].rolling(window=3).mean()
print("\nDataFrame with Rolling Mean:\n", time_series_df)
# Plot rolling mean
time_series_df[['Sales', 'Sales_Rolling_Mean']].plot(title='Sales with
Rolling Mean')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.show()
```

Sprints

Vectorization

Sprints

Vectorization

Avoid Loops:
Use vectorized operations for efficiency.

Practical Use Case

- Task: Compare performance of vectorized operations vs loops.

```
# Using loop
import time
large_array = np.random.rand(1000000)
start_time = time.time()
result_loop = np.zeros_like(large_array)
for i in range(len(large_array)):
    result_loop[i] = large_array[i] * 2
print("Time with loop:", time.time() - start_time)

# Using vectorized operation
start_time = time.time()
result_vectorized = large_array * 2
print("Time with vectorization:", time.time() - start_time)
```

Observation:
- Vectorization is significantly faster.

NumPy and Pandas Together

Sprints

NumPy and Pandas Together

Integrating Libraries:
Use NumPy functions within Pandas operations.

Sprinto

Example:
• mathematical functions to DataFrame columns.

Sprints

Practical Use Case

• Task: Use NumPy functions in Pandas DataFrame manipulations.

```
# Create DataFrame
oyee_data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 28],
    'Salary': [50000, 60000, 70000, 65000]
}
employee_df = pd.DataFrame(employee_data)

# Use NumPy to calculate bonus
employee_df['Bonus'] = np.where(employee_df['Age'] > 28,
employee_df['Salary'] * 0.1, employee_df['Salary'] * 0.05)
print("Employee Data with Bonus:\n", employee_df)
```

Sprints

Sprinto

NumPy and Pandas for Feature Engineering

Sprints

Sprinto

Feature Engineering

- In machine learning, features are the inputs to our models.
- Features are individual measurable properties or characteristics of a phenomenon being observed.

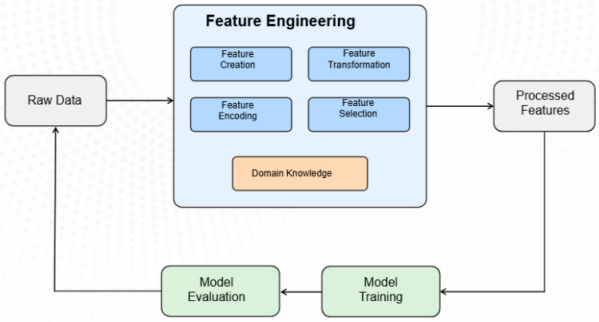
Example:

Sprinto

Sprints

Feature Engineering

Is the process of using domain knowledge to extract features from raw data via data mining techniques. It is a crucial step in the machine learning pipeline.



Sprinto

Sprints

Types of Feature Engineering

Feature Creation

- Combining existing features
- Extracting information from complex data types (e.g., text, datetime)

Feature Transformation

- Scaling (normalization, standardization)
- Mathematical transformations (log, square root)

Feature Encoding

- Converting categorical variables to numerical (one-hot encoding, label encoding)

Feature Selection

- Choosing the most relevant features for the model

Sprinto

Sprints

NumPy and Pandas for Feature Engineering

Feature Transformation:
Use functions like `.np.log()`, `np.sqrt()`, or custom functions.

Handling Categorical Data:
Use `pd.get_dummies()` for one-hot encoding.

Scaling Data:
Apply normalization or standardization using NumPy.

Sprinto

Sprints

Practical Use Case

- Task: Perform feature engineering on a dataset.

```
# Log-transform the Salary
employee_df['Salary_log'] = np.log(employee_df['Salary'])
t("Employee Data with Log-transformed Salary:\n", employee_df)

# One-hot encode 'Name' column
_dummies = pd.get_dummies(employee_df['Name'], prefix='Name')
employee_df = pd.concat([employee_df, name_dummies], axis=1)
print("\nEmployee Data with One-Hot Encoded Names:\n", employee_df)

# Normalize 'Salary' column
employee_df['Salary_norm'] = (employee_df['Salary'] -
employee_df['Salary'].mean()) / employee_df['Salary'].std()
print("\nEmployee Data with Normalized Salary:\n", employee_df)
```

Sprints

Sprinto

Comparing Pandas with Polars

Sprints

Sprinto

Comparing Pandas with Polars

Pandas

- Pros:
  Easy to use, integrates well with other Python libraries.

Polars

- Pros:
  Efficient with large data.

- Cons:
  Performance degrades with large data.

- Cons:
  Steeper learning curve.

Sprinto

Sprints

Example

```
import polars as pl

# Creating DataFrames in Polars and Pandas
data = {'column_1': [1, 2, 3], 'column_2': [4, 5, 6]}
df_polars = pl.DataFrame(data)
df_pandas = pd.DataFrame(data)

print("Polars DataFrame:\n", df_polars)
print("\nPandas DataFrame:\n", df_pandas)
```

Sprinto

Sprints

Pandas vs Polars Performance Comparison

- Performance Comparison:

```
# Generate large dataset
data_large = {'column_1': np.random.rand(1000000), 'column_2':
np.random.rand(1000000)}
df_pandas_large = pd.DataFrame(data_large)
df_polars_large = pl.DataFrame(data_large)

# Pandas operation
start_time = time.time()
df_pandas_large['sum'] = df_pandas_large['column_1'] +
andas_large['column_2']
print("Pandas operation time:", time.time() - start_time)
```

Sprinto

Sprints

Pandas vs Polars Performance Comparison

- Performance Comparison:

```
# Polars operation
start_time = time.time()
df_polars_large = df_polars_large.with_columns([
    (pl.col('column_1') + pl.col('column_2')).alias('sum')
])
print("Polars operation time:", time.time() - start_time)
```

Sprinto

Observation:

Polars can be faster with large datasets.

Sprints

Summary

Sprints

Summary

- NumPy
- Pandas
- Integration
- Key Takeaways
- Next Steps

Sprints

Summary

- NumPy:
  - Efficient array computations.
  - Multi-dimensional arrays, mathematical functions, linear algebra.
- Pandas
- Integration
- Key Takeaways
- Next Steps

Sprints

Sprinto

Summary

- **NumPy**
- **Pandas**
  - Data manipulation and analysis.
  - DataFrames and Series, handling missing data, group operations.
- **Integration**
- **Key Takeaways**
- **Next Steps**

Sprinto

Sprints

Summary

- **NumPy**
- **Pandas**
- **Integration**
  - Use both libraries together for powerful data processing.
- **Key Takeaways**
- **Next Steps**

Sprinto

Sprints

Summary

- **NumPy**
- **Pandas**
- **Integration**
- **Key Takeaways:**
  - Mastering these libraries enables efficient data analysis.
  - Practical experience with real datasets enhances understanding.
- **t Steps**

Sprinto

Sprints

Summary

Summary

- **NumPy**
- **Pandas**
- **Integration**
- **Key Takeaways**
- **Next Steps:**
  - Practice with datasets from sources like Kaggle.
  - Explore advanced features like Pandas' time series analysis, NumPy's broadcasting rules, and performance optimization techniques.
  - Learn about other libraries like Matplotlib and Seaborn for advanced visualization.