Pre-Intern Notes

By Mehul Jaiswal

Contents

1 NumPy

To use it:

```
import numpy as np
```

Arrays

- `a = np.array([1, 2, 3, 4])`

Attributes of np.array (different from normal Python list):

- `.shape`: gives dimension of the array
- `.dtype`: returns the data type of the array's elements
- `.size`: returns the total number of elements in the array
- `.ndim`: returns the number of dimensions (axes) of the array
- `.itemsize`: returns the size (in bytes) of each element in the array
- `.nbytes`: returns the total number of bytes consumed by the elements of the array

Creating Arrays

- To set the array to a certain data type:
  - `np.array([1, 2, 3, 4], dtype=np.int32)`
  - `np.array([1, 2, 3, 4], dtype=np.float32)`
  - `np.array([1, 2, 3, 4], dtype=str)`
- To fill an empty initialized array with a specific value:

  ```
  a = np.full((2, 3, 4), 8)  # (2, 3, 4) is the dimension of the array, 8 is the value
  ```

- To initialize an array with all zeros:

  ```
  np.zeros((2, 3, 4))
  ```

- To initialize an array with all ones:

  ```
  np.ones((2, 3, 4))
  ```

- To initialize an array with empty memory:

```
np.empty((2, 3, 4))  # Contains whatever values are already present in that memory loc
```

- Array with a range of values:

```
a = np.arange(0, 10, 1)  # Values from 0 to 9 with step 1
```

- Array with evenly spaced values:

```
a = np.linspace(0, 1, 5)  # 5 values evenly spaced between 0 and 1
```

Special Values

- `np.nan`: Not a number, used to fill data with NaN if the item is empty

```
np.isnan(a)  # Check if a is NaN
```

- `np.inf`: Infinite, used when the result is divided by zero

```
np.isinf(a)  # Check if a is infinity
```

Mathematical Operations

- `a = np.array([1, 2]), b = np.array([2, 4])`
- `a + b = np.array([3, 6])`
- `a - b = np.array([-1, -2])`
- `a * 2 = np.array([2, 4])`
- `b / 2 = np.array([1, 2])`
- `a + 5 = np.array([6, 7])`
- `a1 = np.array([1, 2, 3]), a2 = np.array([[1], [2]]), a + b = np.array([[2, 3, 4], [3, 4, 5`
- Other operations:

```
np.sqrt(a)
np.sin(a)
np.cos(a)
np.log(a)
np.exp(a)
np.log10(a)
```

Array Methods

- `a = np.array([1, 2, 3])`
- `np.append(a, [4, 5, 6])`
- `np.insert(a, 3, [1, 2])  # Inserts [1, 2] in a at index 3`
- `np.delete(a, 1, 0)  # Delete at index 1 with axis 0 (0: row, 1: column)`

Reshaping Arrays

- `a.reshape((2, 2))  # Reshapes the array to 2x2 (must be same multiple)`
- `a.resize((2, 2))  # Modifies the array itself, if the new shape is larger, array is fille`
- `a.ravel()  # Flattens the array to a single dimension (view)`
- `a.flatten()  # Flattens the array to a single dimension (copy)`
- `a.T or np.transpose(a)  # Transposes the array (flips dimensions)`

Merging, Stacking & Splitting

- `np.concatenate((a1, a1), axis=0)  # axis=0: merge in rows, axis=1: merge in columns`
- `np.stack((a1, a2))  # Stack a1 on a2`
- `np.vstack((a1, a2)) = np.concatenate((a1, a2), axis=0)`
- `np.hstack((a1, a2)) = np.concatenate((a1, a2), axis=1)`
- `np.split(a, 2, axis=0)  # Split array a into 2 arrays by axis=0`

Statistical Operations

- `a.sum()  # Sum of all elements`
- `a.mean()  # Mean of all elements`
- `a.std()  # Standard deviation`
- `a.min(), a.max()  # Minimum and maximum values`
- `a.argmin(), a.argmax()  # Indices of the minimum and maximum values`

Linear Algebra

- Dot product: `np.dot(a, b)` or `a.dot(b)`
- Matrix multiplication: `np.matmul(a, b)`
- Determinant: `np.linalg.det(a)`
- Inverse: `np.linalg.inv(a)`
- Eigenvalues and Eigenvectors: `np.linalg.eig(a)`

Random Numbers

- `np.random.rand(3, 2)  # 3x2 array of random values between 0 and 1`
- `np.random.randn(3, 2)  # 3x2 array of random values from a standard normal distribution`
- `np.random.randint(0, 10, (3, 2))  # 3x2 array of random integers between 0 and 9`

2 Seaborn

Seaborn is a powerful Python library for making statistical graphics. It builds on top of Matplotlib and integrates closely with Pandas data structures.

Key Features

- High-level interface: Provides a high-level interface for drawing attractive and informative statistical graphics.
- Statistical plots: Easily create complex statistical plots, such as regression plots, box plots, and heatmaps.
- Integration with Pandas: Works seamlessly with Pandas data structures, allowing for easy manipulation and visualization of data.

Installation

To install Seaborn, use the following command:

```
pip install seaborn
```

Importing Seaborn

To use Seaborn, you typically import it as follows:

```
import seaborn as sns
import matplotlib.pyplot as plt
```

Basic Plot Types

Distribution Plots

- Histogram:

  ```
  sns.histplot(data, bins=30)
  plt.show()
  ```

- KDE Plot (Kernel Density Estimate):

```
sns.kdeplot(data)
plt.show()
```

- Distribution Plot:

```
sns.displot(data, kde=True)
plt.show()
```

Categorical Plots

- Bar Plot:

```
sns.barplot(x='category', y='value', data=df)
plt.show()
```

- Box Plot:

```
sns.boxplot(x='category', y='value', data=df)
plt.show()
```

- Violin Plot:

```
sns.violinplot(x='category', y='value', data=df)
plt.show()
```

Relational Plots

- Scatter Plot:

```
sns.scatterplot(x='x', y='y', data=df)
plt.show()
```

- Line Plot:

```
sns.lineplot(x='x', y='y', data=df)
plt.show()
```

Matrix Plots

- Heatmap:

  ```
  sns.heatmap(data, annot=True, cmap='viridis')
  plt.show()
  ```

Pair Plots

- Pair Plot:

  ```
  sns.pairplot(df)
  plt.show()
  ```

Customization

Titles and Labels

```
sns.scatterplot(x='x', y='y', data=df)
plt.title('Title')
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.show()
```

Figure Size

```
plt.figure(figsize=(10, 6))
sns.heatmap(data, annot=True, cmap='viridis')
plt.show()
```

Style and Context

```
sns.set_style('whitegrid')
sns.set_context('talk')
sns.scatterplot(x='x', y='y', data=df)
plt.show()
```

Example

Here's an example of using Seaborn to create a regression plot with customization:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load example dataset
tips = sns.load_dataset('tips')

# Create a regression plot
sns.set_style('whitegrid')
plt.figure(figsize=(10, 6))
sns.regplot(x='total_bill', y='tip', data=tips)
plt.title('Total Bill vs. Tip')
plt.xlabel('Total Bill')
plt.ylabel('Tip')
plt.show()
```

3 Git

3.1 Version Control System (VCS)

Definition 1. A version control system (VCS) is a tool that helps manage changes to source code over time.

3.2 Git Basics

- **git init**: Initializes a new Git repository.
- **git clone [url]**: Clones a repository from a remote source.
- **git status**: Displays the state of the working directory and the staging area.
- **git add [file]**: Adds a file to the staging area.
- **git commit -m "[message]"**: Commits the staged changes with a message.
- **git push**: Pushes the committed changes to the remote repository.
- **git pull**: Fetches and integrates changes from the remote repository.

3.3 Branching and Merging

- **git branch**: Lists all the branches in the repository.
- **git branch [branch-name]**: Creates a new branch.
- **git checkout [branch-name]**: Switches to the specified branch.
- **git merge [branch-name]**: Merges the specified branch into the current branch.
- **git branch -d [branch-name]**: Deletes the specified branch.

3.4 Remote Repositories

- **git remote**: Lists the remote connections.
- **git remote add [name] [url]**: Adds a new remote repository.
- **git fetch**: Fetches changes from the remote repository.
- **git push [remote] [branch]**: Pushes changes to the specified remote repository and branch.
- **git pull [remote] [branch]**: Pulls changes from the specified remote repository and branch.

3.5 Rebasing

- **git rebase [branch]**: Reapplies commits on top of another base tip.

3.6 Stashing Changes

- **git stash**: Temporarily saves changes that are not yet ready to be committed.
- **git stash pop**: Applies the stashed changes and removes them from the stash list.
- **git stash list**: Lists all stashed changes.
- **git stash apply [stash]**: Applies the specified stash without removing it from the stash list.

3.7 Tagging

- **git tag**: Lists all the tags in the repository.
- **git tag [tag-name]**: Creates a new tag.
- **git push [remote] [tag-name]**: Pushes the specified tag to the remote repository.

4 Apache

4.1 Hadoop

Apache Hadoop is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.

Key Components

- **HDFS (Hadoop Distributed File System)**: A distributed file system that provides high-throughput access to application data.
- **YARN (Yet Another Resource Negotiator)**: A cluster management technology.
- **MapReduce**: A YARN-based system for parallel processing of large data sets.

Installation

To install Hadoop, follow the steps in the official documentation:

`http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html`

Basic Commands

- **Starting HDFS**:

  ```
  start-dfs.sh
  ```

- **Starting YARN**:

  ```
  start-yarn.sh
  ```

- **Putting a file in HDFS**:

  ```
  hdfs dfs -put localfile.txt /hdfs/path/
  ```

- **Running a MapReduce job**:

  ```
  hadoop jar /path/to/hadoop-examples.jar wordcount /hdfs/input /hdfs/output
  ```

5 Apache Spark

Apache Spark is an open-source unified analytics engine for large-scale data processing, with built-in modules for streaming, SQL, machine learning, and graph processing.

Key Features

- **Speed**: Spark processes data in-memory, which makes it much faster than traditional disk-based processing.
- **Ease of Use**: Provides easy-to-use APIs in Python, Java, Scala, and R.
- **Generality**: Supports a wide range of workloads, such as batch applications, iterative algorithms, and streaming.

Installation

To install Spark, download the pre-built package from the official website:

```
https://spark.apache.org/downloads.html
```

Basic Usage

- **Starting Spark Shell**:

  ```
  ./bin/spark-shell
  ```

- **Reading a file**:

  ```
  val data = spark.read.textFile("path/to/file.txt")
  ```

- **Performing a simple transformation**:

  ```
  val words = data.flatMap(line => line.split(" "))
  ```

- **Performing an action**:

  ```
  words.count()
  ```

6 Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams.

Key Features

- **Event Time Processing**: Supports event time processing and late data handling.
- **Stateful Stream Processing**: Maintains state for streams, providing exactly-once consistency.
- **Scalability**: Designed to run on all common cluster environments, providing high throughput and low latency.

Installation

To install Flink, download the latest stable release from the official website:

`https://flink.apache.org/downloads.html`

Basic Usage

- **Starting Flink Cluster**:

  ```
  ./bin/start-cluster.sh
  ```

- **Submitting a Flink Job**:

  ```
  ./bin/flink run examples/streaming/WordCount.jar
  ```

- **Running a Flink Shell**:

  ```
  ./bin/start-scala-shell.sh local
  ```

- **Reading from a source**:

  ```
  val text = env.readTextFile("path/to/file.txt")
  ```

- **Performing a transformation**:

```
val counts = text.flatMap(_.toLowerCase.split("\\W+")).map((_, 1)).keyBy(0).sum(1)
```

- **Writing to a sink**:

```
counts.writeAsCsv("path/to/output.csv")
```

7 Software Engineering Principles

Software engineering principles are guidelines that help software engineers create high-quality software. These principles ensure that software is reliable, maintainable, and scalable.

7.1 Key Principles

- **Modularity**: Dividing a software system into separate modules that can be developed, tested, and debugged independently.
- **Encapsulation**: Bundling data and methods that operate on the data within one unit, such as a class in object-oriented programming.
- **Abstraction**: Hiding complex implementation details and showing only the necessary features of an object or a system.
- **Separation of Concerns**: Separating different aspects of a software system to reduce complexity and increase maintainability.
- **DRY (Don't Repeat Yourself)**: Reducing the repetition of code by abstracting out common functionality.
- **KISS (Keep It Simple, Stupid)**: Keeping the software design simple and avoiding unnecessary complexity.
- **YAGNI (You Aren't Gonna Need It)**: Avoiding the implementation of features that are not currently needed.
- **SOLID**: A set of five principles for object-oriented design: Single Responsibility, Open/-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion.

7.2 Testing

Testing is the process of evaluating a software application to ensure that it meets the specified requirements and works as expected. It helps identify defects and verify that the software is fit for use.

Types of Testing

- **Unit Testing**: Testing individual components or modules of a software to ensure they work correctly.

```
def test_add():
    assert add(2, 3) == 5
```

- **Integration Testing**: Testing the interaction between different components or modules to ensure they work together as expected.

```
def test_integration():
    result = component_a() + component_b()
    assert result == expected_result
```

- **System Testing**: Testing the entire system as a whole to ensure it meets the requirements and performs correctly in various environments.
- **Acceptance Testing**: Testing conducted to determine if the system meets the acceptance criteria and is ready for deployment.

Testing Frameworks

- **JUnit**: A widely used testing framework for Java.

```
@Test
public void testAdd() {
    assertEquals(5, add(2, 3));
}
```

- **PyTest**: A testing framework for Python.

```
def test_add():
    assert add(2, 3) == 5
```

- **Selenium**: A testing framework for web applications.

7.3 Debugging

Debugging is the process of identifying, analyzing, and removing errors or bugs from a software application to ensure it functions correctly.

Debugging Techniques

- **Print Statements**: Using print statements to display the values of variables at different points in the code.

```
print("Value of x:", x)
```

- **Logging**: Using logging libraries to record information about the execution of a program.

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.debug("Value of x: %s", x)
```

- **Interactive Debuggers**: Using tools like GDB, PDB, or IDE debuggers to step through the code and inspect the state of the program.

```
import pdb
pdb.set_trace()
```

- **Code Reviews**: Conducting code reviews with peers to identify and fix errors.
- **Automated Tools**: Using static analysis tools and linters to detect potential issues in the code.

Debugging in Data Engineering

Given the responsibilities at Harness Inc., debugging data pipelines and integration solutions is crucial. Common debugging techniques include:

- **Data Validation**: Ensuring that the data conforms to the expected format and values.

```
assert isinstance(data, pd.DataFrame), "Data should be a DataFrame"
assert not data.isnull().values.any(), "Data contains null values"
```

- **Pipeline Logging**: Adding logging to different stages of the data pipeline to trace data flow and transformations.

```
logging.info("Started data ingestion")
logging.info("Finished data transformation")
```

- **Error Handling**: Implementing error handling to catch and log exceptions.

```
try:
    result = process_data(data)
except Exception as e:
    logging.error("Error processing data: %s", e)
```

- **Performance Monitoring**: Monitoring the performance of data pipelines to identify bottlenecks.

```
start_time = time.time()
result = process_data(data)
logging.info("Processing time: %s seconds", time.time() - start_time)
```

8 Pandas

Pandas is a powerful Python library for data manipulation and analysis, providing data structures like DataFrames and Series that are ideal for handling structured data.

8.1 Installation

To install Pandas, use the following command:

```
pip install pandas
```

8.2 Importing Pandas

To use Pandas, you typically import it as follows:

```
import pandas as pd
```

8.3 Data Structures

- **Series**: A one-dimensional labeled array capable of holding any data type.

  ```
  s = pd.Series([1, 3, 5, np.nan, 6, 8])
  ```

- **DataFrame**: A two-dimensional labeled data structure with columns of potentially different types.

  ```
  df = pd.DataFrame({
      "A": [1, 2, 3, 4],
      "B": [5, 6, 7, 8],
      "C": [9, 10, 11, 12]
  })
  ```

8.4 Basic Operations

- **Viewing Data**: Use `head()`, `tail()`, `info()`, and `describe()` to inspect the data.

  ```
  print(df.head())
  print(df.tail())
  print(df.info())
  print(df.describe())
  ```

- **Selecting Data**: Use `loc[]` and `iloc[]` for label-based and integer-based selection, respectively.

```
df.loc[0:2, ["A", "B"]]
df.iloc[0:2, 0:2]
```

- **Filtering Data**: Use conditions to filter data.

```
df[df["A"] > 2]
```

- **Adding/Removing Columns**: Use assignment and `drop()`.

```
df["D"] = df["A"] + df["B"]
df = df.drop("D", axis=1)
```

8.5 Data Manipulation

- **Merging DataFrames**: Use `merge()` to combine DataFrames.

```
df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value': [1, 2, 3]})
df2 = pd.DataFrame({'key': ['A', 'B', 'D'], 'value': [4, 5, 6]})
merged_df = pd.merge(df1, df2, on="key")
```

- **Concatenating DataFrames**: Use `concat()` to concatenate DataFrames.

```
concatenated_df = pd.concat([df1, df2])
```

- **Grouping Data**: Use `groupby()` to group data and `aggregate()` or `apply()` to perform operations.

```
grouped = df.groupby("A")
summary = grouped["B"].sum()
```

- **Pivoting Data**: Use `pivot()` and `pivot_table()` for reshaping data.

```
pivoted = df.pivot(index="A", columns="B", values="C")
```

8.6 Handling Missing Data

- **Checking for Missing Data**: Use `isnull()` and `notnull()`.

```
df.isnull()
df.notnull()
```

- **Filling Missing Data**: Use `fillna()` to replace missing values.

```
df.fillna(0)
```

- **Dropping Missing Data**: Use `dropna()` to remove missing values.

```
df.dropna()
```

8.7 Reading and Writing Data

- **Reading from CSV**: Use `read_csv()` to load data from a CSV file.

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

- **Writing to CSV**: Use `to_csv()` to save data to a CSV file.

```
df.to_csv("output.csv", index=False)
```

- **Reading from Excel**: Use `read_excel()` to load data from an Excel file.

```
df = pd.read_excel("data.xlsx", sheet_name="Sheet1")
```

- **Writing to Excel**: Use `to_excel()` to save data to an Excel file.

```
df.to_excel("output.xlsx", sheet_name="Sheet1", index=False)
```

9 SQL

SQL (Structured Query Language) is a standardized language used to manage and manipulate relational databases. It is essential for data manipulation and querying.

9.1 Installation and Setup

To use SQL, you need to set up a database server. Popular choices include MySQL, PostgreSQL, and SQLite. For example, to install MySQL:

```
sudo apt-get install mysql-server
sudo mysql_secure_installation
```

9.2 Connecting to a Database

Connect to a MySQL database using Python with the **mysql-connector-python** library:

```
import mysql.connector

conn = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="yourdatabase"
)

cursor = conn.cursor()
```

9.3 Creating Tables

Create a table using the **CREATE TABLE** statement:

```
CREATE TABLE employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    position VARCHAR(255) NOT NULL,
    salary DECIMAL(10, 2),
    hire_date DATE
);
```

9.4 Inserting Data

Insert data into a table using the **INSERT INTO** statement:

```
INSERT INTO employees (name, position, salary, hire_date)
VALUES ('John Doe', 'Software Engineer', 70000, '2023-01-15');
```

9.5 Querying Data

Retrieve data from a table using the **SELECT** statement:

```
SELECT * FROM employees;
SELECT name, position FROM employees WHERE salary > 60000;
```

Common SQL Clauses

- **WHERE**: Filters records based on specified conditions.

  ```
  SELECT * FROM employees WHERE position = 'Software Engineer';
  ```

- **ORDER BY**: Sorts the result set in ascending or descending order.

  ```
  SELECT * FROM employees ORDER BY salary DESC;
  ```

- **GROUP BY**: Groups rows that have the same values in specified columns.

  ```
  SELECT position, COUNT(*) FROM employees GROUP BY position;
  ```

- **HAVING**: Filters groups based on specified conditions.

  ```
  SELECT position, AVG(salary) FROM employees GROUP BY position HAVING AVG(salary) > 60
  ```

- **JOIN**: Combines rows from two or more tables based on a related column.

  ```
  SELECT employees.name, departments.name
  FROM employees
  INNER JOIN departments ON employees.department_id = departments.id;
  ```

9.6 Updating Data

Update existing records in a table using the **UPDATE** statement:

```
UPDATE employees SET salary = 75000 WHERE id = 1;
```

9.7 Deleting Data

Delete records from a table using the **DELETE** statement:

```
DELETE FROM employees WHERE id = 1;
```

9.8 Creating and Using Indexes

Create an index to improve query performance:

```
CREATE INDEX idx_salary ON employees(salary);
```

9.9 Advanced SQL Functions

- **Aggregate Functions**: Perform calculations on a set of values and return a single value.

  ```
  SELECT AVG(salary) AS average_salary FROM employees;
  SELECT COUNT(*) AS total_employees FROM employees;
  SELECT MAX(salary) AS highest_salary FROM employees;
  ```

- **String Functions**: Manipulate string data.

  ```
  SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM employees;
  SELECT LENGTH(name) AS name_length FROM employees;
  SELECT UPPER(position) AS upper_position FROM employees;
  ```

- **Date Functions**: Manipulate date data.

  ```
  SELECT hire_date, YEAR(hire_date) AS hire_year FROM employees;
  SELECT hire_date, DATE_ADD(hire_date, INTERVAL 1 YEAR) AS next_year_anniversary FROM e
  ```

- **Subqueries**: Nested queries used to perform complex queries.

```
SELECT name FROM employees WHERE salary > (SELECT AVG(salary) FROM employees);
```

- **Case Statements**: Perform conditional logic in SQL queries.

```
SELECT name,
       CASE
           WHEN salary > 70000 THEN 'High'
           WHEN salary BETWEEN 50000 AND 70000 THEN 'Medium'
           ELSE 'Low'
       END AS salary_level
FROM employees;
```

9.10 Transactions

Ensure data integrity and consistency using transactions. Common commands include **START TRANSACTION**, **COMMIT**, and **ROLLBACK**.

```
START TRANSACTION;
UPDATE employees SET salary = 80000 WHERE id = 2;
DELETE FROM employees WHERE id = 3;
COMMIT;
```

9.11 Using SQL with Pandas

Leverage SQL with Pandas for advanced data manipulation and analysis:

```
import pandas as pd
import sqlite3

# Create a connection to a SQLite database
conn = sqlite3.connect("example.db")

# Load data into a DataFrame
df = pd.read_sql_query("SELECT * FROM employees", conn)

# Perform data analysis with Pandas
summary = df.groupby("position")["salary"].mean()

# Write DataFrame back to the database
summary.to_sql("salary_summary", conn, if_exists="replace", index=False)

conn.close()
```

10 Matplotlib

Matplotlib is a plotting library for creating static, interactive, and animated visualizations in Python.

10.1 Installation

To install Matplotlib, use the following command:

```
pip install matplotlib
```

10.2 Importing Matplotlib

To use Matplotlib, you typically import it as follows:

```
import matplotlib.pyplot as plt
```

10.3 Basic Plotting

- Line Plot:

  ```
  plt.plot([1, 2, 3, 4])
  plt.ylabel('some numbers')
  plt.show()
  ```

- Scatter Plot:

  ```
  plt.scatter(x, y)
  plt.show()
  ```

- Bar Plot:

  ```
  plt.bar(x, height)
  plt.show()
  ```

- Histogram:

  ```
  plt.hist(data, bins=10)
  plt.show()
  ```

- Pie Chart:

```
plt.pie(sizes, labels=labels)
plt.show()
```

10.4 Customization

- Titles and Labels:

```
plt.title('Title')
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
```

- Figure Size:

```
plt.figure(figsize=(10, 6))
```

- Grid:

```
plt.grid(True)
```

- Legends:

```
plt.legend(['Series 1', 'Series 2'])
```

11 Docker

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers.

11.1 Installation

To install Docker, follow the instructions for your operating system on the official Docker website:

`https://docs.docker.com/get-docker/`

11.2 Basic Commands

- **docker run**: Runs a command in a new container.

  ```
  docker run hello-world
  ```

- **docker ps**: Lists running containers.

  ```
  docker ps
  ```

- **docker build**: Builds an image from a Dockerfile.

  ```
  docker build -t my-image .
  ```

- **docker images**: Lists all images.

  ```
  docker images
  ```

- **docker stop**: Stops a running container.

  ```
  docker stop container_id
  ```

- **docker rm**: Removes a container.

  ```
  docker rm container_id
  ```

- **docker rmi**: Removes an image.

  ```
  docker rmi image_id
  ```

11.3 Dockerfile Basics

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim

# Set the working directory
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

12 Machine Learning with Scikit-Learn

Scikit-learn is a simple and efficient tool for predictive data analysis in Python.

12.1 Installation

To install Scikit-learn, use the following command:

```
pip install scikit-learn
```

12.2 Importing Scikit-learn

To use Scikit-learn, you typically import it as follows:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

12.3 Basic Workflow

- Load dataset:

  ```
  iris = datasets.load_iris()
  X = iris.data
  y = iris.target
  ```

- Split dataset:

  ```
  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
  ```

- Train model:

  ```
  model = LinearRegression()
  model.fit(X_train, y_train)
  ```

- Make predictions:

  ```
  predictions = model.predict(X_test)
  ```

- Evaluate model:

```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, predictions)
```

13 Data Cleaning

Data cleaning is the process of preparing data for analysis by removing or modifying data that is incorrect, incomplete, irrelevant, duplicated, or improperly formatted.

13.1 Common Techniques

- Handling Missing Values:

```
df.dropna()  # Drop rows with missing values
df.fillna(value=0)  # Fill missing values with a specified value
```

- Removing Duplicates:

```
df.drop_duplicates()
```

- Data Type Conversion:

```
df['column'] = df['column'].astype('int')
```

- Handling Outliers:

```
df = df[(df['column'] > lower_limit) & (df['column'] < upper_limit)]
```