**Python Code Snippets: -**

Decorators

OOP Encap/Inher/Abstrac/Classes/func

**DSA**

API

Pyspark

Testing

Git Help

Docker Help:- <https://github.com/docker/awesome-compose/blob/master/flask/README.md>

Docker Help:- <https://github.com/docker/awesome-compose/tree/master/official-documentation-samples/django/>

EC2 and AWS intro :- <https://medium.com/@shivambhadani_/aws-part-1-introduction-to-cloud-and-ec2-f06cdc80a1fc>

Important Notes:-

Login and Logout - [LoginLogout](https://author-ide.skills.network/render?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJtZF9pbnN0cnVjdGlvbnNfdXJsIjoiaHR0cHM6Ly9jZi1jb3Vyc2VzLWRhdGEuczMudXMuY2xvdWQtb2JqZWN0LXN0b3JhZ2UuYXBwZG9tYWluLmNsb3VkL0lCTS1DRDAyNTFFTi1Ta2lsbHNOZXR3b3JrL2xhYnMvbTVfZGphbmdvX2FkdmFuY2VkL2xhYjJfYXV0aGVudGljYXRpb24ubWQiLCJ0b29sX3R5cGUiOiJ0aGVpYSIsImF0bGFzX2ZpbGVfaWQiOjI1MjQwLCJhZG1pbiI6ZmFsc2UsImlhdCI6MTczMDE1MjgwOX0.0cjGdQQIha0NV9AunWskPN3sd4kezKwRRSvST6ewoPA)

Crud on Django Model Objets:- [CRUD on Django Model Objets](https://author-ide.skills.network/render?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJtZF9pbnN0cnVjdGlvbnNfdXJsIjoiaHR0cHM6Ly9jZi1jb3Vyc2VzLWRhdGEuczMudXMuY2xvdWQtb2JqZWN0LXN0b3JhZ2UuYXBwZG9tYWluLmNsb3VkL0lCTS1DRDAyNTFFTi1Ta2lsbHNOZXR3b3JrL2xhYnMvbTNfZGphbmdvX29ybS9sYWIyX2NydWQubWQiLCJ0b29sX3R5cGUiOiJ0aGVpYWRvY2tlciIsImF0bGFzX2ZpbGVfaWQiOjI1MjIwLCJhZG1pbiI6ZmFsc2UsImlhdCI6MTczMDE0NzAxOH0.LcadRxp326kkNafiJlCMygHt2EQY9XIL1vfFwS-UKo0)

Django Authentication:- [DjangoAuthentication](https://author-ide.skills.network/render?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJtZF9pbnN0cnVjdGlvbnNfdXJsIjoiaHR0cHM6Ly9jZi1jb3Vyc2VzLWRhdGEuczMudXMuY2xvdWQtb2JqZWN0LXN0b3JhZ2UuYXBwZG9tYWluLmNsb3VkL0lCTS1DRDAyNTFFTi1Ta2lsbHNOZXR3b3JrL2xhYnMvbTVfZGphbmdvX2FkdmFuY2VkL2xhYjJfYXV0aGVudGljYXRpb24ubWQiLCJ0b29sX3R5cGUiOiJ0aGVpYSIsImF0bGFzX2ZpbGVfaWQiOjI1MjQwLCJhZG1pbiI6ZmFsc2UsImlhdCI6MTczMDE1MjgwOX0.0cjGdQQIha0NV9AunWskPN3sd4kezKwRRSvST6ewoPA)

**Pyspark:-**

**Create a SparkSessionin:-**

from pyspark.sql import SparkSession

spark = SparkSession.builder \

        .appName("MySparkApp") \

        .master("local[\*]") \

        .getOrCreate()

**Describe different ways to read data into Pyspark:-**

df\_read\_csv = spark.read.csv(“my\_file.csv”,header=True)

df\_read\_parquet = spark.read.parquet(“my\_file.parquet”,header=True)

df\_read\_json = spark.read.json(“my\_file.json”,header=True)

**Explain the use of StructType and StructField classes in PySpark with examples**

# Explain the use of StructType and StructField classes in PySpark with examples.

# StructType is a class in PySpark that represents a structured type, which is a collection of

import pyspark

from pyspark.sql import SparkSession

from pyspark.sql.types import StructType, StructField, StringType, IntegerType

spark = SparkSession.builder.master("local[1]") \

        .appName('ProjectPro') \

        .getOrCreate()

data = [("James","","William","36636","M",3000),

    ("Michael","Smith","","40288","M",4000),

    ("Robert","","Dawson","42114","M",4000),

    ("Maria","Jones","39192","F",4000) ]

schema = StructType([ \

    StructField("firstname",StringType(),True), \

    StructField("middlename",StringType(),True), \

    StructField("lastname",StringType(),True), \

    StructField("id", StringType(), True), \

    StructField("gender", StringType(), True), \

    StructField("salary", IntegerType(), True) \

  ])

df = spark.createDataFrame(data=data, schema=schema)

df.printSchema()

df.show(truncate=False

Example of building Structype from raw fields

from pyspark.sql.types import StructType, StructField, StringType, IntegerType

# Example schema\_fields list (mocked for demonstration)

schema\_fields = [

    {'pubname': 'first\_name', 'type': 'string'},

    {'pubname': 'age', 'type': 'int'},

    {'pubname': 'salary', 'type': 'float'}

]

# Mocked config and match\_field functions

def get\_new\_column\_name(pubname, table):

    return f"{table}\_{pubname}"

def match\_field(field\_type):

    if field\_type == 'string':

        return StringType()

    elif field\_type == 'int':

        return IntegerType()

    # Add other type mappings as needed

table = "employee"

# Creating StructType using the list comprehension

schema = StructType(

    [

        StructField(get\_new\_column\_name(field['pubname'], table), match\_field(field['type']), True)

        for field in schema\_fields

    ]

)

print(schema)

**Expected Output:-**

StructType(

    [

        StructField('employee\_first\_name', StringType(), True),

        StructField('employee\_age', IntegerType(), True),

        StructField('employee\_salary', FloatType(), True)

    ]

)

**Example of nested struct:-**

# Create sample data

data = [

    ("John Doe", 30, {"street": "123 Main St", "city": "Anytown", "zip": "12345"}),

    ("Jane Smith", 25, {"street": "456 Maple Ave", "city": "Othertown", "zip": "67890"})

]

# Create a DataFrame with the specified schema

df = spark.createDataFrame(data, schema)

# Show the DataFrame

df.show(truncate=False)

df.printSchema()

# Stop the Spark session

spark.stop()

**Without explode using select splitting columns in pyspark:- splitting columns with “,” in pyspark**

from pyspark.sql import SparkSession

from pyspark.sql.types import StructType, StructField, StringType, IntegerType

from pyspark.sql.functions import col

# Create a Spark session

spark = SparkSession.builder.appName("NestedStructExample").getOrCreate()

# Define the schema using StructType and StructField

schema = StructType([

    StructField("name", StringType(), True),

    StructField("age", IntegerType(), True),

    StructField("address", StructType([

        StructField("street", StringType(), True),

        StructField("city", StringType(), True),

        StructField("zip", StringType(), True)

    ]), True)

])

# Create sample data

data = [

    ("John Doe", 30, {"street": "123 Main St", "city": "Anytown", "zip": "12345"}),

    ("Jane Smith", 25, {"street": "456 Maple Ave", "city": "Othertown", "zip": "67890"})

]

# Create a DataFrame with the specified schema

df = spark.createDataFrame(data, schema)

# Split the nested structure into separate columns

df\_split = df.select(

    col("name"),

    col("age"),

    col("address.street").alias("street"),

    col("address.city").alias("city"),

    col("address.zip").alias("zip")

)

# Show the new DataFrame

df\_split.show(truncate=False)

df\_split.printSchema()

# Stop the Spark session

spark.stop()

**With explode function:-**

from pyspark.sql import SparkSession

from pyspark.sql.functions import explode

from pyspark.sql.types import StructType, StructField, StringType, ArrayType

# Create a Spark session

spark = SparkSession.builder.appName("ExplodeExample").getOrCreate()

# Define the schema using StructType and StructField

schema = StructType([

    StructField("name", StringType(), True),

    StructField("hobbies", ArrayType(StringType()), True)

])

# Create sample data

data = [

    ("John Doe", ["reading", "traveling", "swimming"]),

    ("Jane Smith", ["cooking", "running"])

]

# Create a DataFrame with the specified schema

df = spark.createDataFrame(data, schema)

# Use the explode function to flatten the array column

df\_exploded = df.select(

    col("name"),

    explode(col("hobbies")).alias("hobby")

)

# Show the new DataFrame

df\_exploded.show(truncate=False)

df\_exploded.printSchema()

# Stop the Spark session

spark.stop()

**Handle missing data in pyspark:-**

df\_from\_csv.dropna(how=any)

df\_from\_csv.drop\_duplicates()

**Drop duplicates:-**

>>> df.drop\_duplicates(subset=['brand', 'style'], keep='last')

brand style rating

1 Yum Yum cup 4.0

2 Indomie cup 3.5

4 Indomie pack 5.0

**Fill the missing values with constant**

df\_from\_parquet.fillna(value=2)

df\_from\_csv.cache()

df\_from\_csv.persist(storageLevel=StorageLevel.DISK\_ONLY)

**Dataframe joins in pyspark:-**

df\_from\_csv.join(df\_from\_json, on="id", how="inner")

df\_from\_json.join(df\_from\_parquet, on="product\_id", how="outer")

**Lazy Evaluation:-**

No actual transformations happen on data unless actions are performed on them. Lazy evaluation in PySpark means that transformations on RDDs or DataFrames are not immediately executed when they are called. Instead, these transformations build up a logical plan of operations that will be executed when an action is called. This approach optimizes the execution plan for better performance and resource utilization.

**Examples of Transformations:-**

As name suggests theses are operations performed on datasets. These operations are lazy, meaning the actual computation will not happen until an action is called.

**Select():-**

df = spark.read.csv(“path/to/file.csv”,header=True, inferSchema= True)

selected\_df = df.selected(“col1”,”col2”)

**filter() and Where():-**

filtered\_data = df.filter(df[“column”] > 10)

or

filtered\_data = df.where(df[“column”]>10)

**groupBy():-**

grouped\_data = df.groupby(“column”).count()

**withColumn():-**

df.withcolumn(“new\_column”,df[column1]\*2)

**join():-**

df1 = spark.read.csv("path/to/file1.csv", header=True, inferSchema=True)

df2 = spark.read.csv("path/to/file2.csv", header=True, inferSchema=True)

joined\_df = df1.join(df2, df1["column1"] == df2["column1"])

**Sorted or OrderBy:-**

sorted\_df = df.orderBy(df[“col1”].desc())

sort\_df = df.sort(df[“col1”].desc())

**Distinct():-**

Distinct\_df = df.distinct()

**Examples of Actions:-**

Action triggers the execution of transformations and return a result. Some common actions are:-

df.show()

df.collect()

df.count()

df.first()

df.take(5)

df.write.csv()

**Custom Transformation on Pyspark:-**

Define a python function which operates on py data frame and then use .transform()

**# Define a python function that operates on pySpark DataFrames**

def get\_discounted\_price(df):

return df.withColumn("discounted\_price", \

df.price - (df.price \* df.discount) / 100)

**# Evoke the transformation**

df\_discounted = df\_from\_csv.transfrom(get\_discounted\_price)

**Window Function in Pyspark:-**

from pyspark.sql.window import window

from pyspark.sql.functions import row\_number

#Define the window function:-

window = window.orderBy("discounted\_price")

#Apply the window function:-

df = df\_from\_csv.withcolumn("row\_number",row\_number().over(window))

**Exmple with PartitionBy:-**

from pyspark.sql import SparkSession

from pyspark.sql.window import Window

from pyspark.sql.functions import row\_number

# Initialize SparkSession

spark = SparkSession.builder.appName("window\_function\_example").getOrCreate()

# Read a CSV file into a DataFrame

df\_from\_csv = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)

# Define the window function with partitioning

window = Window.partitionBy("category").orderBy("discounted\_price")

# Apply the window function

df\_with\_row\_number = df\_from\_csv.withColumn("row\_number", row\_number().over(window))

# Show the result

df\_with\_row\_number.show(

**Exceptional Handling**

from pyspark.sql import SparkSession

from pyspark.sql.utils import AnalysisException

# Initialize SparkSession

spark = SparkSession.builder.appName("error\_handling\_example").getOrCreate()

try:

    # Read a CSV file into a DataFrame

    df = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)

    # Perform some transformation

    df\_filtered = df.filter(df["age"] > 30)

    # Show the result

    df\_filtered.show()

except AnalysisException as e:

    print(f"An error occurred: {e}")

except Exception as e:

    print(f"An unexpected error occurred: {e}")

|  |  |  |
| --- | --- | --- |
| **Feature** | **Narrow Transformations** | **Wide Transformations** |
| **Definition** | Operations where each input partition contributes to only one output partition. | Operations where each input partition contributes to multiple output partitions. |
| **Examples** | map(), filter(), flatMap(), mapPartitions() | groupByKey(), reduceByKey(), join(), distinct(), coalesce() |
| **Data Shuffling** | No shuffling of data across the network. | Requires shuffling of data across the network. |
| **Execution** | Performed in a single stage. | Involves multiple stages of execution. |
| **Performance** | Generally faster due to no shuffling. | Slower due to the overhead of data shuffling. |
| **Use Case** | Used for transformations that do not require data from other partitions. | Used for transformations that require combining data across partitions. |
| **Fault Tolerance** | Easier to recover as no data shuffling is involved. | More complex to recover due to data shuffling. |

**Abstract Method :-**

from abc import ABC, abstractmethod

class Animal(ABC):

    @abstractmethod

    def make\_sound(self):

        pass

    @abstractmethod

    def move(self):

        pass

class Dog(Animal):

    def make\_sound(self):

        return "Bark"

    def move(self):

        return "Run"

class Cat(Animal):

    def make\_sound(self):

        return "Meow"

    def move(self):

        return "Jump"

# This will raise an error because you cannot instantiate an abstract class

# animal = Animal()

# This will work because Dog and Cat provide implementations for all abstract methods

dog = Dog()

cat = Cat()

print(dog.make\_sound())  # Output: Bark

print(cat.move())        # Output: Jump

from abc import ABC, abstractmethod

class Payment(ABC):

    @abstractmethod

    def authorize(self, amount):

        pass

    @abstractmethod

    def capture(self, amount):

        pass

class CreditcardPayment(Payment):

    def authorize(self, amount):

        return "Null"

    def capture(self, amount):

        return "Null"

class PaypalPayment(Payment):

    def authorize(self,amount):

        return "Null"

    def capture(self, amount):

        return super().capture(amount)

## Objects and amounts being passed

def process\_payments(payment:Payment, amount):

    auth = payment.authorize(amount)

    capture = payment.capture(amount)

## Objects being declared

Credit\_card\_Payment = CreditcardPayment()

Paypal\_Payment = PaypalPayment()

print(process\_payments(Credit\_card\_Payment(500)))

print(process\_payments(Paypal\_Payment(700)))

 **Defining a Template for Subclasses**:

* Abstract methods allow you to define a common interface for a group of subclasses.
* This ensures that all subclasses implement the same methods, providing a consistent API.

 **Enforcing Implementation in Subclasses**:

* By declaring methods as abstract, you force subclasses to provide concrete implementations.
* This is useful for defining core methods that every subclass must have.

**Encapsulation:-**

**Key Points of Encapsulation**

1. **Data Hiding**:
   * Encapsulation hides the internal state of an object and requires all interaction to be performed through an object's methods.
   * This helps prevent accidental interference and misuse of the data.
2. **Public and Private Members**:
   * Public members can be accessed from outside the class.
   * Private members (typically prefixed with an underscore \_ or double underscore \_\_ in Python) cannot be accessed directly from outside the class.
3. **Controlled Access**:
   * Methods in the class provide controlled access to the attributes, often including validation or other logic to ensure proper use.

class BankAccount:

    def \_\_init\_\_(self, owner, balance=0):

        self.owner = owner

        self.\_\_balance = balance  # Private attribute

    def deposit(self, amount):

        if amount > 0:

            self.\_\_balance += amount

            print(f"Deposited {amount}, new balance is {self.\_\_balance}")

        else:

            print("Deposit amount must be positive")

    def withdraw(self, amount):

        if 0 < amount <= self.\_\_balance:

            self.\_\_balance -= amount

            print(f"Withdrew {amount}, new balance is {self.\_\_balance}")

        else:

            print("Invalid withdrawal amount")

    def get\_balance(self):

        return self.\_\_balance

# Example usage

account = BankAccount("Alice", 100)

account.deposit(50)

account.withdraw(30)

print(f"Balance: {account.get\_balance()}")

# Trying to access private attribute directly will raise an AttributeError

try:

    print(account.\_\_balance)

except AttributeError as e:

    print(e)

### Inheritance

Inheritance allows a class to inherit attributes and methods from another class. This promotes code reuse and establishes a relationship between the parent (superclass) and child (subclass) classes.

class Animal:

    def \_\_init\_\_(self, name):

        self.name = name

    def move(self):

        return f"{self.name} moves"

class Dog(Animal):

    def bark(self):

        return f"{self.name} says woof!"

my\_dog = Dog("Buddy")

print(my\_dog.move())  # Output: Buddy moves

print(my\_dog.bark())  # Output: Buddy says woof!

### Polymorphism

Polymorphism allows methods to do different things based on the object it is acting upon. It is the ability of different classes to respond to the same method call in different ways.

class Animal:

    def make\_sound(self):

        pass

class Dog(Animal):

    def make\_sound(self):

        return "Woof!"

class Cat(Animal):

    def make\_sound(self):

        return "Meow!"

def animal\_sound(animal):

    print(animal.make\_sound())

dog = Dog()

cat = Cat()

dog.make\_sound()

cat.make\_sound()

animal\_sound(dog)  # Output: Woof!

animal\_sound(cat)  # Output: Meow!

**Static and classMethod can be called using class name:-**

class Utility:

    version = "1.0"

    @staticmethod

    def greet(name):

        return f"Hello, {name}!"

    @classmethod

    def get\_version(cls):

        return cls.version

    @classmethod

    def set\_version(cls, new\_version):

        cls.version = new\_version

# Using static methods

print(Utility.greet("Alice"))  # Output: Hello, Alice!

# Using class methods

print(Utility.get\_version())  # Output: 1.0

Utility.set\_version("2.0")

print(Utility.get\_version())  # Output: 2.0

**Python API Notes:-**

1. Installing requests Library

        pip install requests

2. Making a GET Request

        import requests

        response = requests.get('https://api.example.com/data')

        if response.status\_code == 200:

        data = response.json()  # Parse JSON response

        print(data)

        else:

        print(f"Error: {response.status\_code}")

3. Making a POST Request

        A POST request is used to send data to a server.

        import requests

        payload = {'key1': 'value1', 'key2': 'value2'}

        response = requests.post('https://api.example.com/submit', data=payload)

        if response.status\_code == 200:

        data = response.json()

        print(data)

        else:

        print(f"Error: {response.status\_code}")

4. Sending JSON Data in a POST Request

        When sending JSON data, set the Content-Type header to application/json.

        import requests

        import json

        payload = {'key1': 'value1', 'key2': 'value2'}

        response = requests.post('https://api.example.com/submit', json=payload)

        if response.status\_code == 200:

        data = response.json()

        print(data)

        else:

        print(f"Error: {response.status\_code}")

5. Handling Query Parameters

        Add query parameters to a GET request by passing them as a dictionary to the params argument.

        import requests

        params = {'param1': 'value1', 'param2': 'value2'}

        response = requests.get('https://api.example.com/search', params=params)

        if response.status\_code == 200:

        data = response.json()

        print(data)

        else:

        print(f"Error: {response.status\_code}")

6. Adding Headers to a Request

        Headers can be added to a request to provide additional information to the server.

        import requests

        headers = {'Authorization': 'Bearer YOUR\_ACCESS\_TOKEN', 'Content-Type': 'application/json'}

        response = requests.get('https://api.example.com/secure-data', headers=headers)

        if response.status\_code == 200:

        data = response.json()

        print(data)

        else:

        print(f"Error: {response.status\_code}")

7. Handling Timeouts

        You can specify a timeout to prevent your request from hanging indefinitely.

        import requests

        try:

        response = requests.get('https://api.example.com/data', timeout=5)  # Timeout in seconds

        response.raise\_for\_status()  # Raise an exception for HTTP errors

        data = response.json()

        print(data)

        except requests.exceptions.Timeout:

        print("Request timed out")

        except requests.exceptions.RequestException as e:

        print(f"Request failed: {e}")

8. Handling HTTP Errors

You can check for HTTP errors using raise\_for\_status() which raises an exception for HTTP error responses.

        import requests

        response = requests.get('https://api.example.com/data')

        try:

        response.raise\_for\_status()  # Raise an HTTPError for bad responses

        data = response.json()

        print(data)

        except requests.exceptions.HTTPError as http\_err:

        print(f"HTTP error occurred: {http\_err}")

        except Exception as err:

        print(f"Other error occurred: {err}")

9. Uploading Files

You can upload files by using the files parameter.

        import requests

        files = {'file': ('filename.txt', open('filename.txt', 'rb'))}

        response = requests.post('https://api.example.com/upload', files=files)

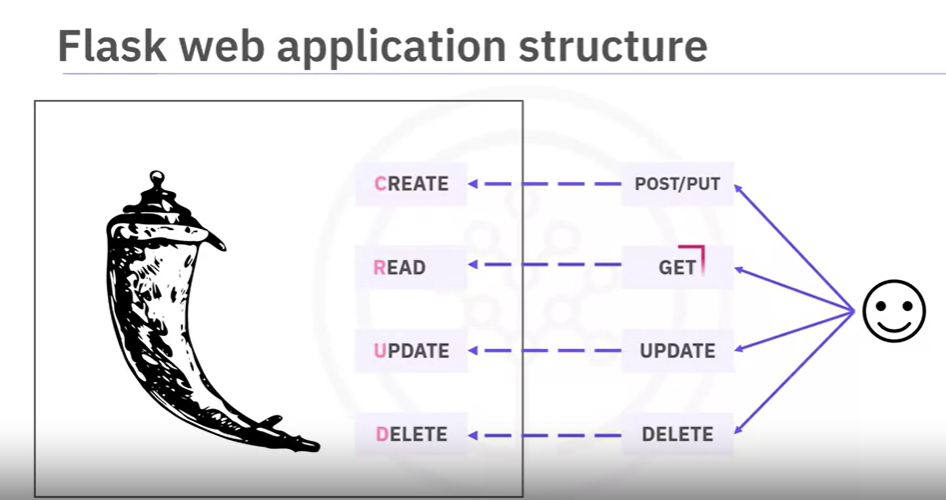
        if response.status\_code == 200:

        print("File uploaded successfully")

        else:

        print(f"Error: {response.status\_code}")

**API Creation:-**



API Creation Using Flask:-

from flask import Flask, jsonify, request

app = Flask(\_\_name\_\_)

# Example data

data = {

    '1': {'name': 'Item 1', 'price': 10.99},

    '2': {'name': 'Item 2', 'price': 20.99}

}

@app.route('/items', methods=['GET'])

def get\_items():

    return jsonify(data)

@app.route('/items/<item\_id>', methods=['GET'])

def get\_item(item\_id):

    item = data.get(item\_id)

    if item:

        return jsonify(item)

    else:

        return jsonify({'error': 'Item not found'}), 404

@app.route('/items', methods=['POST'])

def create\_item():

    if not request.json or 'name' not in request.json or 'price' not in request.json:

        return jsonify({'error': 'Bad request'}), 400

    item\_id = str(len(data) + 1)

    item = {

        'name': request.json['name'],

        'price': request.json['price']

    }

    data[item\_id] = item

    return jsonify(item), 201

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

API Creation using FastAPI:-

from fastapi import FastAPI, HTTPException

from typing import Optional

app = FastAPI()

# Example data

data = {

    '1': {'name': 'Item 1', 'price': 10.99},

    '2': {'name': 'Item 2', 'price': 20.99}

}

@app.get('/items')

def read\_items():

    return data

@app.get('/items/{item\_id}')

def read\_item(item\_id: str):

    item = data.get(item\_id)

    if item:

        return item

    else:

        raise HTTPException(status\_code=404, detail='Item not found')

@app.post('/items')

def create\_item(name: str, price: float):

    item\_id = str(len(data) + 1)

    item = {'name': name, 'price': price}

    data[item\_id] = item

    return item

**Create Flask Application with SQLAlchemy:-**

from flask import Flask, jsonify, request, abort

from flask\_sqlalchemy import SQLAlchemy

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///data.db'  # SQLite database file

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db = SQLAlchemy(app)

# Define the database model

class Item(db.Model):

    id = db.Column(db.Integer, primary\_key=True)

    name = db.Column(db.String(80), nullable=False)

    price = db.Column(db.Float, nullable=False)

    def \_\_repr\_\_(self):

        return f'<Item {self.name}>'

    def to\_dict(self):

        return {

            'id': self.id,

            'name': self.name,

            'price': self.price

        }

# Initialize the database

@app.before\_first\_request

def create\_tables():

    db.create\_all()

@app.route('/items', methods=['GET'])

def get\_items():

    items = Item.query.all()

    return jsonify([item.to\_dict() for item in items])

@app.route('/items/<int:item\_id>', methods=['GET'])

def get\_item(item\_id):

    item = Item.query.get(item\_id)

    if item:

        return jsonify(item.to\_dict())

    else:

        abort(404, description='Item not found')

@app.route('/items', methods=['POST'])

def create\_item():

    if not request.json or 'name' not in request.json or 'price' not in request.json:

        abort(400, description='Bad request')

    item = Item(

        name=request.json['name'],

        price=request.json['price']

    )

    db.session.add(item)

    db.session.commit()

    return jsonify(item.to\_dict()), 201

@app.route('/items/<int:item\_id>', methods=['PUT'])

def update\_item(item\_id):

    item = Item.query.get(item\_id)

    if not item:

        abort(404, description='Item not found')

    if not request.json:

        abort(400, description='Bad request')

    item.name = request.json.get('name', item.name)

    item.price = request.json.get('price', item.price)

    db.session.commit()

    return jsonify(item.to\_dict())

@app.route('/items/<int:item\_id>', methods=['DELETE'])

def delete\_item(item\_id):

    item = Item.query.get(item\_id)

    if not item:

        abort(404, description='Item not found')

    db.session.delete(item)

    db.session.commit()

    return jsonify({'result': True})

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

**Flask API examples:-**

from flask import Flask, jsonify, request

import json

from flask\_cors import CORS

app = Flask("Product Server")

CORS(app)

products = [

    {'id': 143, 'name': 'Notebook', 'price': 5.49},

    {'id': 144, 'name': 'Black Marker', 'price': 1.99}

]

#

#Add all the REST API end-points here

#

# Example request - http://localhost:5000/products

@app.route('/products', methods=['GET'])

def get\_products():

    return jsonify(products)

# Example request - http://localhost:5000/products/144 - with method GET

@app.route('/products/<id>', methods=['GET'])

def get\_product(id):

    id = int(id)

    product = [x for x in products if x["id"] == id][0]

    return jsonify(product)

# Example request - http://localhost:5000/products - with method POST

@app.route('/products', methods=['POST'])

def add\_product():

    products.append(request.get\_json())

    return '', 201

# Example request - http://localhost:5000/products/144 - with method PUT

@app.route('/products/<id>', methods=['PUT'])

def update\_product(id):

    id = int(id)

    updated\_product = json.loads(request.data)

    product = [x for x in products if x["id"] == id][0]

    for key, value in updated\_product.items():

        product[key] = value

    return '', 204

# Example request - http://localhost:5000/products/144 - with method DELETE

@app.route('/products/<id>', methods=['DELETE'])

def remove\_product(id):

    id = int(id)

    product = [x for x in products if x["id"] == id][0]

    products.remove(product)

    return '', 204

app.run(port=5000,debug=True)

**Unit Tests:-**

**unittest**

import unittest

# Code to be tested

def add(a, b):

    return a + b

def subtract(a, b):

    return a - b

# Test case class

class TestMathOperations(unittest.TestCase):

    def test\_add(self):

        self.assertEqual(add(2, 3), 5)

        self.assertEqual(add(-1, 1), 0)

    def test\_subtract(self):

        self.assertEqual(subtract(5, 3), 2)

        self.assertEqual(subtract(-1, -1), 0)

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

**pytest:-**

# Code to be tested

def add(a, b):

    return a + b

def subtract(a, b):

    return a - b

# Test functions

def test\_add():

    assert add(2, 3) == 5

    assert add(-1, 1) == 0

def test\_subtract():

    assert subtract(5, 3) == 2

    assert subtract(-1, -1) == 0

# Code to be tested

def add(a, b):

    return a + b

def subtract(a, b):

    return a - b

# Test functions

def test\_add():

    assert add(2, 3) == 5

    assert add(-1, 1) == 0

def test\_subtract():

    assert subtract(5, 3) == 2

    assert subtract(-1, -1) == 0

**Using Fixtures with Pytest:-**

import pytest

@pytest.fixture

def sample\_data():

    return [1, 2, 3, 4, 5]

def test\_sum(sample\_data):

    assert sum(sample\_data) == 15

def test\_max(sample\_data):

    assert max(sample\_data) == 5

**Using mark parameterize:-**

import pytest

@pytest.mark.parametrize("a, b, expected", [

    (2, 3, 5),

    (-1, 1, 0),

    (0, 0, 0),

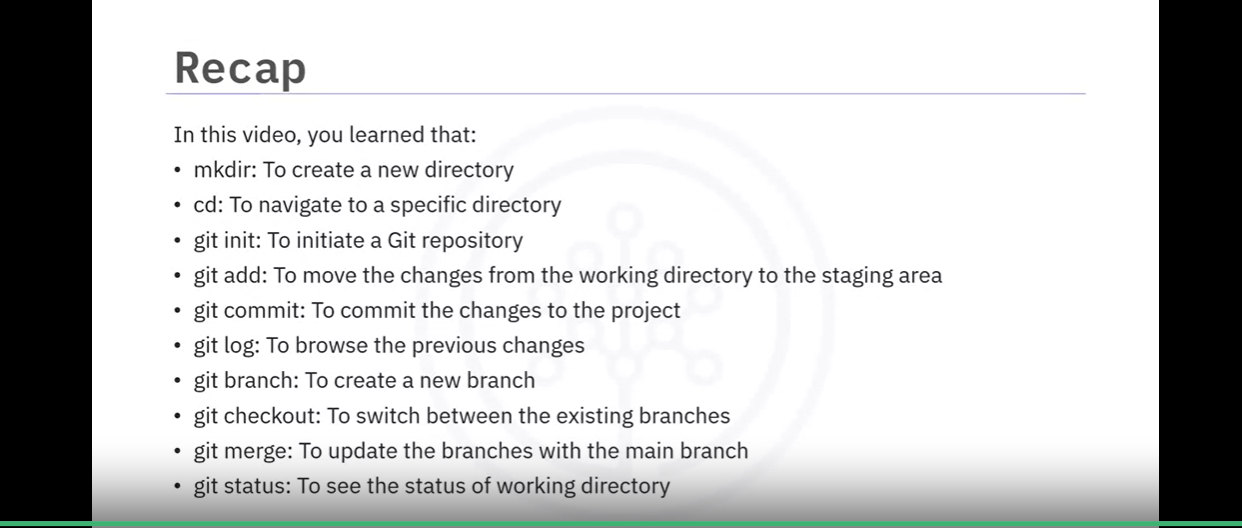
])

def test\_add(a, b, expected):

    assert add(a, b) == expected

[**https://www.deepnexus.tech/2024/07/git-commands-cheatsheet-essential.html**](https://www.deepnexus.tech/2024/07/git-commands-cheatsheet-essential.html)

**Git Help**



Let's look at some useful Git commands and understand them:

1. **git add**
   * *Description*: It adds changes to the staging area. This command stages the changes made to the files and prepares them for the next commit.
   * *Syntax*:
     + **git add <filename.txt>** (to add a specific file)
     + **git add .** (to add all the files that are new or changed in the current directory)
     + **git add -A** (to add all changes in the entire working tree, from the root of the repository, regardless of where you are in the directory structure)
2. **git reset**
   * *Description*: It resets changes in the working directory. When used with –hard HEAD, this command discards all changes made to the working directory and staging area and resets the repository to the last commit (HEAD).
   * *Syntax*:
     + **git reset**
     + **git reset –hard HEAD**
3. **git branch**
   * *Description*: It lists, creates, or deletes branches in a repository. To delete the branch, first check out the branch using **git checkout** and then run the command to delete the branch locally.
   * *Syntax*:
     + **git branch** (to list branches)
     + **git branch <new-branch>** (to create a new branch)
     + **git branch -d <branch-name>** (to delete a branch)
4. **git checkout main – Switching between branches.** 
   * *Description*: It switches to the "main" branch. This will switch your current branch to "main."
   * *Syntax*: **git checkout main**
5. **git clone**
   * *Description*: It copies a repository from a remote source to your local machine. This will create a copy of the repository in your current working directory.
   * *Syntax*: **git clone <repository URL>**
6. **git pull. Remote repo to your mentioned branch** 
   * *Description*: It fetches changes from a remote repository and merges them into your local branch. First, switch to the branch that you want to merge changes into by running the **git checkout(Your branch name)** command. Then, run the **git pull** command, which will fetch the changes from the main branch of the origin remote repository and merge them into your current branch.
   * *Syntax*: **git pull origin main**
7. **git push**
   * *Description*: It uploads local repository content to a remote repository. Make sure you are on the branch that you want to push by running the **git checkout** command first, then push the branch to the remote repository.
   * *Syntax*: **git push origin <branch-name>**
8. **git version**
   * *Description*: It displays the current Git version installed on your system.
   * *Syntax*: **git version**
9. **git diff**
   * *Description*: It shows changes between commits, commit and working tree, etc. It also compares the branches.
   * *Syntax*:
     + **git diff** (shows the difference between the working directory and the last commit)
     + **git diff HEAD~1 HEAD** (shows the difference between the last and second-last commits)
     + **git diff <branch-1> <branch-2>** (compares the specified branches)
10. **git revert**
    * *Description*: It reverts a commit by applying a new commit. This will create a new commit that undoes the changes made by the last commit.
    * *Syntax*: **git revert HEAD**
11. **git config –global user.email <Your GitHub Email>**
    * *Description*: It sets a global email configuration for Git. This needs to be executed before doing a commit to authenticate the user's email ID.
    * *Syntax*: **git config –global user.email <Your GitHub Email>**
12. **git config –global user.name <Your GitHub Username>**
    * *Description*: It sets a global username configuration for Git. This needs to be executed before doing a commit to authenticate users' username.
    * *Syntax*: **git config –global user.name <Your GitHub Username>**
13. **git remote**
    * *Description*: It lists the names of all remote repositories associated with your local repository.
    * *Syntax*: **git remote**
14. **git remote -v**
    * *Description*: It lists all remote repositories that your local Git repository is connected to, along with the URLs associated with those remote repositories.
    * *Syntax*: **git remote -v**
15. **git remote add origin <URL>**
    * *Description*: It adds a remote repository named "origin" with the specified URL.
    * *Syntax*: **git remote add origin <URL>**
16. **git remote rename**
    * *Description*: The git remote rename command is followed by the name of the remote repository (origin) you want to rename and the new name (upstream) you want to give it. This will rename the "origin" remote repository to "upstream."
    * *Syntax*: **git remote rename origin upstream**
17. **git remote rm <name>**
    * *Description*: It removes a remote repository with the specified name.
    * *Syntax*: **git remote** *rm origin*
18. **git format-patch**
    * *Description*: It generates patches for email submission. These patches can be used for submitting changes via email or for sharing them with others.
    * *Syntax*: **git format-patch HEAD~3** (creates patches for the last three commits)
19. **git request-pull**
    * *Description*: It generates a summary of pending changes for an email request. It helps communicate the changes made in a branch or fork to the upstream repository maintainer.
    * *Syntax*: **git request-pull origin/main <myfork or branch\_name>**
20. **git send-email**
    * *Description*: It sends a collection of patches as emails. It allows you to send multiple patch files to recipients via email. Please make sure to set the email address and name using the **git config** command so that the email client knows the sender's information when sending the emails.
    * *Syntax*: **git send-email \*.patch**
21. **git am**
    * *Description*: It applies patches to the repository. It takes a patch file as input and applies the changes specified in the patch file to the repository.
    * *Syntax*: **git am <patchfile.patch>**
22. **git daemon**
    * *Description*: It exposes repositories via the Git:// protocol. The Git protocol is a lightweight protocol designed for efficient communication between Git clients and servers.
    * *Syntax*: **git daemon –base-path=/path/to/repositories**
23. **git instaweb**
    * *Description*: It instantly launches a web server to browse repositories. It provides a simplified way to view repository contents through a web interface without the need for configuring a full web server.
    * *Syntax*: **git instaweb –httpd=webrick**
24. **git rerere**
    * *Description*: It reuses recorded resolution of previously resolved merge conflicts. Please note that rerere.enabled configuration option needs to be set to "true" (**git config –global rerere.enabled true**) for git rerere to work. Additionally, note that git rerere only applies to conflicts that have been resolved using the same branch and commit.

**Git Commands:-**

* mkdir myrepo
* cd myrepo
* git init
* ls -la .git
* touch newfile
* git add newfile
* git config --global user.email [you@example.com](mailto:you@example.com)
* git config --global user.name "Your Name"
* git commit -m "added newfile"
* git commit -m "added newfile"
* git branch
* git checkout my1stbranch
* git branch
* git checkout -b my1stbranch
* echo 'Here is some text in my newfile.' >> newfile
* cat newfile
* touch readme.md
* git add readme.md
* git status
* git add \*
* git status
* git commit -m "added readme.md modified newfile"
* git log
* git revert HEAD --no-edit
* touch goodfile
* git add goodfile
* git commit -m "added goodfile"
* git log
* git checkout master
* git merge my1stbranch
* git log
* git branch -d my1stbranch
* git add . 🡪 adds all untracked files in the current directory and subdirectories to the staging area
* git add style.css 🡪 you can add the single file you created by using th
* git add -A 🡪recursively add all files from the top level git folder.
* git branch -d feature-circle-500 🡪 Git delete branch(Once branch is merged to it’s original branch delete newly created branch)

**Git Workflow:-**

* The steps included in a GitHub workflow are:
  + Clone the remote repository or initialize a Git repository.
  + Move files to a staging area.
  + Perform an initial commit.
  + Create a branch and work on it.
  + Add files to the staging area and commit.
  + Push local commits to the remote repository.
  + Create a pull request for review and merging.
  + Use the pull operation to update the local repository.

**Fork**(Create a copy of the main project) 🡪 **Clone**(on your local repo) 🡪 **Branch**(create new branch) 🡪 Make the changes(Add, commit to this branch) 🡪 **git status**(Status check) -> change to original branch 🡪 **git merge** to original branch -> Check status 🡪 **git branch -d** (Delete new branch) 🡪 Git push(to the Forked repo) 🡪 to contribute (Open a pull requests )

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Code Example** |
| **git add** | Used to move changes from the working directory to the staging area | 1 |
| 1. git add sample.md |
| Copied! |
| **git add .** | Allows to move the changed files into the staging area on GitHub repositories | 1 |
| 1. git add . |
| Copied! |
| **git am** | Used to apply patches emailed to the repository | 1 |
| 1. git am < patchfile.patch |
| Copied! |
| **git branch** | Allows to create an isolated environment within the repository to make changes | 1 |
| 1. git branch <new-branch> |
| Copied! |
| **git checkout** | Allows to see and change existing branches | 1 |
| 1. git checkout <existing-branch> |
| Copied! |
| **git checkout main** | Allows to switch to the main branch | 1 |
| 1. git checkout main |
| Copied! |
| **git clone** | Allows to create a copy of the remote repository | 1 |
| 1. git clone <repository-url> |
| Copied! |
| **git commit** | Allows you to take staged snapshots if changes and commit them to the project | 1 |
| 1. git commit -m "Your commit message here" |
| Copied! |
| **git config --global user.email** | Example 1: Sets a global email configuration for Git | Example 1: |
|  | 1 |
| Example 2: Sets a global username configuration for Git | 1. git config --global user.email "your.email@example.com" |
|  | Copied! |
|  |  |
|  |  |
|  | Example 2: |
|  | 1 |
|  | 1. git config --global user.name "Your Name" |
|  | Copied! |
| **git daemon** | Used to allow anonymous download from the repository | 1 |
| 1. git daemon --reuseaddr --verbose |
| Copied! |
| **git diff** | Helps others to review your code to identify and compare the changes | 1 |
| 1. git diff example.txt |
| Copied! |
| **git fetch** | Used to transfer the changes from the remote repo to your local repo | 1 |
| 1. git fetch <options> <remote name> <branch name> |
| Copied! |
| **git fetch upstream/master** | Used to grab upstream branches | 1 |
| 1. git fetch upstream master:upstream-master |
| Copied! |
| **git format-patch** | Generates or prepares e-mail submission if you adopt Linux kernel-style public forum workflow | 1 |
| 1. git format-patch -n <number\_of\_commits> |
| Copied! |
| **git http-backend** | Provides a server-side implementation of Git-over-HTTP, allowing both fetch and push services | 1 |
| 2 |
| 3 |
| 1. git clone --bare /path/to/repos/myrepo.git |
| 2. cd myrepo.git |
| 3. git update-server-info |
| Copied! |
| **git init** | Used to clone an existing repository | 1 |
| 1. git init <directory> |
| Copied! |
| **git instaweb** | Allows to set up web front-end to Git repositories | 1 |
| 1. git instaweb -p 8080 |
| Copied! |
| **git log** | Enables to browse previous changes to a project | 1 |
| 1. git log -p filename |
| Copied! |
| **git merge** | Used to merge changes in the active branch into another branch | 1 |
| 1. git merge feature\_branch |
| Copied! |
| **git merge upstream/master** | Merges changes from the 'upstream/master' branch to the current branch | 1 |
| 1. git merge upstream/master |
| Copied! |
| **git pull** | Used to transfer the changes from the remote repo to your local repo, and merge them to a branch | 1 |
| 1. git pull origin main |
| Copied! |
| **git pull downstream** | Pulls changes from a downstream repository, specifically from the master branch of that repository | 1 |
| 1. git pull downstream main |
| Copied! |
| **git pull upstream** | Pulls changes from the "upstream" repository into the current branch | 1 |
| 1. git pull upstream main |
| Copied! |
| **git push** | Used to push all the committed changes into the repository | 1 |
| 1. git push origin your\_branch\_name |
| Copied! |
| **git remote** | A command to manage a set of tracked repositories | 1 |
| 1. git remote add upstream https://github.com/original/repo.git |
| Copied! |
| **git remote add origin <URL>** | Adds a remote repository named "origin" with the specified URL | 1 |
| 1. git remote add origin https://github.com/yourusername/your-repo.git |
| Copied! |
| **git remote add upstream** | Adds the original repository as a new remote repository labeled upstream | 1 |
| 1. git remote add upstream https://github.com/original/repo.git |
| Copied! |
| **git remote rename** | The git remote rename command is followed by the name of the remote repository(origin) you want to rename and the new name(upstream) you want to give it | 1 |
| 1. git remote rename origin new-origin |
| Copied! |
| **git remote -v** | Allows to view the remotes associated with the local repository | 1 |
| 1. git remote -v |
| Copied! |
| **git request-pull** | Example 1: Creates a summary of changes for your upstream to pull | Example 1: |
|  | 1 |
| Example 2: Generates a summary of pending changes for an email request | 1. git request-pull origin/main your-branch |
|  | Copied! |
|  |  |
|  |  |
|  | Example 2: |
|  | 1 |
|  | 1. git request-pull <base> <head> <repository> |
|  | Copied! |
| **git rerere** | Reuses recorded resolution of previously resolved merge conflicts | 1 |
| 2 |
| 1. git rerere |
| 2. git rerere diff |
| Copied! |
| **git reset** | Undoes changes that were made to the files in your working directory | 1 |
| 1. git reset HEAD~1 |
| Copied! |
| **git revert** | Used to undo botched commits | 1 |
| 1. git revert HEAD |
| Copied! |
| **git send-email** | Example 1: Sends your email submission without corruption by your MUA | Example 1: |
|  | 1 |
| Example 2: Sends a collection of patches as emails | 2 |
|  | 1. git send-email --to=recipient@example.com |
|  | 2. path/to/patchfile.patch |
|  | Copied! |
|  |  |
|  |  |
|  | Example 2: |
|  | 1 |
|  | 2 |
|  | 1. git send-email --to recipient@example.com |
|  | 2. patches/\*.patch |
|  | Copied! |
| **git-shell** | Used as a restricted login shell for shared central repository users | 1 |
| 1. sudo usermod -s /usr/bin/git-shell gituser |
| Copied! |
| **git status** | Allows to see the state of your working directory and the staged snapshot of the changes | 1 |
| 1. git status |
| Copied! |
| **git version** | Displays the current Git version installed on your system | 1 |
| 1. git --version |
| Copied! |
| **git web** | Provides a web front-end to Git repositories | 1 |
| 1. git instaweb --port=8080 |

**Git Terminologies:-**

|  |  |
| --- | --- |
| **Term** | **Definition** |
| **Cloning** | A process of creating a copy of the project's code and its complete version history from the remote repository on the local machine. |
| **Commit** | A snapshot of the project's current state at a specific point in time, along with a description of the changes made. |
| **Developer** | A computer programmer who is responsible for writing code. |
| **Distributed version control system (DVCS)** | A system that keeps track of changes to code, regardless of where it is stored. Multiple users work on the same codebase or repository, mirroring the codebase on their computers if needed, while the distributed version control software helps manage synchronization amongst the various codebase mirrors. |
| **Fork** | A copy of a repository into your GitHub account. |
| **Forking** | Forking creates a copy of a repository on which one can work without affecting the original repository. |
| **GitHub** | A web-hosted service for the Git repository. |
| **Git** | A free and open-source software distributed under the GNU General Public License. It is a distributed version control system that allows users to have a copy of their own project on their computer anywhere in the world. |
| **Integrator** | A role that is responsible for managing changes made by developers. |
| **Master branch** | A branch that stores the deployable version of your code. The master branch is created by default and is definitive. |
| **Merge** | A process to combine changes from one branch to another, typically merging a feature branch into the main branch. |
| **Origin** | A term that refers to the repository where a copy is cloned from. |
| **Pull request** | A process used to request that someone review and approve your changes before they become final. |
| **Remote repositories** | Repositories that are stored elsewhere. They can exist on the internet, on your network, or on your local computer. |
| **Repository administrator** | A role that is responsible for configuring and maintaining access to the repository. |
| **Repository** | A data structure for storing documents, including application source code. It contains the project folders that are set up for version control. |
| **Staging area** | An area where commits can be formatted and reviewed before completing the commit. |
| **Upstream** | A term used by developers to refer to the original source where the local copy was cloned from. |
| **Version control** | A system that allows you to keep track of changes to your documents. This process allows you to recover older versions of the documents if any mistakes are made. |

<https://author-ide.skills.network/render?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJtZF9pbnN0cnVjdGlvbnNfdXJsIjoiaHR0cHM6Ly9jZi1jb3Vyc2VzLWRhdGEuczMudXMuY2xvdWQtb2JqZWN0LXN0b3JhZ2UuYXBwZG9tYWluLmNsb3VkLzlZcXowOWxZYWdUSF9FbzVjOUJYcVEvR2V0JTIwZmFtaWxpYXIlMjB3aXRoJTIwR2l0JTIwQ29tbWFuZHMtdjEubWQiLCJ0b29sX3R5cGUiOiJpbnN0cnVjdGlvbmFsLWxhYiIsImF0bGFzX2ZpbGVfaWQiOjYzOTgyLCJhZG1pbiI6ZmFsc2UsImlhdCI6MTczMDIzNTA4MX0.7kNwCz6sxD4M2zLIdQg-31fMdh9Idqlp_2ulyL1XLg8>

**Set Operations:-**

# Correct way to create an empty set

empty\_set = set()

print(empty\_set)  # Output: set()

print(type(empty\_set))  # Output: <class 'set'>

# This creates an empty dictionary, not a set

empty\_dict = {}

#Get value of “OldRecord” if exists else define empty dictionary

update\_dict.get("OldRecord", {})

print(empty\_dict)  # Output: {}

print(type(empty\_dict))  # Output: <class 'dict'>

# Creating sets

set\_a = {1, 2, 3}

set\_b = {3, 4, 5}

subset = {1, 2}

disjoint\_set = {6, 7}

# 1. Add an element to set\_a

set\_a.add(6)

print("After add:", set\_a)  # Output: {1, 2, 3, 6}

# 2. Remove an element from set\_a

set\_a.remove(6)

print("After remove:", set\_a)  # Output: {1, 2, 3}

# 3. Discard an element from set\_a

set\_a.discard(3)

print("After discard:", set\_a)  # Output: {1, 2}

# 4. Pop an element from set\_a

popped\_element = set\_a.pop()

print("Popped element:", popped\_element)  # Output: 1 (or another element)

print("After pop:", set\_a)  # Output: {2} (or the remaining element)

# Reset set\_a for further operations

set\_a = {1, 2, 3}

# 5. Clear all elements from set\_a

set\_a.clear()

print("After clear:", set\_a)  # Output: set()

# Reset set\_a for further operations

set\_a = {1, 2, 3}

# 6. Copy set\_a

copy\_of\_set\_a = set\_a.copy()

print("Copy of set\_a:", copy\_of\_set\_a)  # Output: {1, 2, 3}

# 7. Union of set\_a and set\_b

union\_set = set\_a.union(set\_b)

print("Union:", union\_set)  # Output: {1, 2, 3, 4, 5}

# 8. Intersection of set\_a and set\_b

intersection\_set = set\_a.intersection(set\_b)

print("Intersection:", intersection\_set)  # Output: {3}

# 9. Difference between set\_a and set\_b

difference\_set = set\_a.difference(set\_b)

print("Difference (set\_a - set\_b):", difference\_set)  # Output: {1, 2}

# 10. Symmetric Difference between set\_a and set\_b

sym\_diff\_set = set\_a.symmetric\_difference(set\_b)

print("Symmetric Difference:", sym\_diff\_set)  # Output: {1, 2, 4, 5}

# 11. Check if subset is a subset of set\_a

is\_subset = subset.issubset(set\_a)

print("Is subset a subset of set\_a:", is\_subset)  # Output: True

# 12. Check if set\_a is a superset of subset

is\_superset = set\_a.issuperset(subset)

print("Is set\_a a superset of subset:", is\_superset)  # Output: True

# 13. Check if set\_a is disjoint with disjoint\_set

is\_disjoint = set\_a.isdisjoint(disjoint\_set)

print("Is set\_a disjoint with disjoint\_set:", is\_disjoint)  # Output: True

# Set Operations Using Operators

# Union

print("Union (using |):", set\_a | set\_b)  # Output: {1, 2, 3, 4, 5}

# Intersection

print("Intersection (using &):", set\_a & set\_b)  # Output: {3}

# Difference

print("Difference (using -):", set\_a - set\_b)  # Output: {1, 2}

# Symmetric Difference

print("Symmetric Difference (using ^):", set\_a ^ set\_b)  # Output: {1, 2, 4, 5}

############# 1. Summing a List of Numbers #############

from functools import reduce

def add(x, y):

    return x + y

numbers = [1, 2, 3, 4, 5]

result = reduce(add, numbers)

print(result)  # Output: 15

############## 2. Finding the Maximum in a List #############

from functools import reduce

def maximum(x, y):

    return x if x > y else y

numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]

result = reduce(maximum, numbers)

print(result)  # Output: 9

############## 3. Product of a List of Numbers Using a Lambda Function #############

from functools import reduce

numbers = [1, 2, 3, 4, 5]

result = reduce(lambda x, y: x \* y, numbers)

print(result)  # Output: 120

################### 4. Example of reduce in DF ###############################

from functools import reduce

from pyspark.sql import DataFrame

# Sample DataFrame union function (placeholder for DataFrame.union)

def union(df1, df2):

    # This is a simplified example; in reality, df1.union(df2) would be used

    return df1 + df2  # Assuming simple data for illustration

# Placeholder for the publish\_lms function

def publish\_lms(lms, rebuild\_version\_map, for\_legal):

    return lms  # Return lms as is, for this example

# Sample data to simulate DataFrames

lms\_list = ["DF1", "DF2", "DF3"]  # Pretending these are DataFrame objects

rebuild\_version\_map = {}

for\_legal = True

# Reducing the list of DataFrames by applying union

final\_output = reduce(

    union, [publish\_lms(lms, rebuild\_version\_map, for\_legal) for lms in lms\_list]

)

print(final\_output) # Output: DF1DF2DF3

### Usage of collect()

1. **Collect Data**: collect() gathers the data from the DataFrame into the driver program. This means that all the data in the DataFrame is brought into memory on the local machine, which is useful for inspecting small to moderately sized datasets.
2. **Print or Inspect Data**: By collecting the data, you can print or perform further operations on it that are not suited for distributed processing.

**Explanation**

1. **Filtering and Sorting**:
   * filtered\_data\_dictionary is a DataFrame filtered based on certain conditions and sorted by sequence.
2. **Collecting Data**:
   * filtered\_data\_dictionary.collect() converts the DataFrame into a list of Row objects. Each Row represents a record from the DataFrame.
   * This action brings all the filtered and sorted data into the driver’s memory.
3. **Printing Data**:
   * The collected data is printed to the console. This allows you to see the exact data that matches your filtering and sorting criteria.

**When to Use collect()**

* **Small Datasets**: When working with small datasets, collecting data to the driver is feasible and allows for easy inspection.
* **Debugging**: Useful for debugging to check intermediate results.
* **Local Operations**: If you need to perform operations on the data that aren’t supported in a distributed manner.

**When Not to Use collect()**

* **Large Datasets**: Avoid using collect() on very large datasets as it can lead to memory issues and performance degradation due to large amounts of data being moved to the driver.
* **Distributed Operations**: Prefer operations that can be performed in a distributed manner to leverage Spark’s parallel processing capabilities.

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

# Initialize Spark session

spark = SparkSession.builder.master("local[\*]").appName("FilterSortCollectExample").getOrCreate()

# Sample DataFrame creation

data = [

    ("Table1", "Yes", "No", 1),

    ("Table1", "No", "Yes", 2),

    ("Table2", "Yes", "No", 3),

    ("Table1", "Yes", "No", 4),

    ("Table2", "No", "Yes", 5)

]

columns = ["table\_name", "is\_visible", "is\_dataset\_primary\_key", "sequence"]

data\_dictionary = spark.createDataFrame(data, columns)

# Define the table name to filter

penstock\_table\_name = "Table1"

# Filter and sort the DataFrame

filtered\_data\_dictionary = data\_dictionary.where(

    (col("is\_visible") == 'Yes') | (col("is\_dataset\_primary\_key") == 'Yes') &

    (col("table\_name") == penstock\_table\_name)

).sort(col("sequence").asc())

# Show filtered and sorted DataFrame

filtered\_data\_dictionary.show()

# Output #

+----------+----------+------------------------+--------+

|table\_name|is\_visible|is\_dataset\_primary\_key |sequence|

+----------+----------+------------------------+--------+

|     Table1|       Yes|                     No |       1|

|     Table1|        No|                    Yes |       2|

|     Table1|       Yes|                     No |       4|

+----------+----------+------------------------+--------+

# End of Output #

**SQS sending messages in batches**

**Use of partial messages**:- Function double\_partial can be called multiple times by changing value of it’s parameters, this will call partial func multiply with value 2 as constant.

from functools import partial

# Example function with three arguments

def multiply(x, y, z):

    return x \* y \* z

# Create a new function with x fixed at 2

double\_partial = partial(multiply, 2)

result = double\_partial(3, 4)  # Equivalent to multiply(2, 3, 4)

print(result)  # Output: 24

Below code will be calling sqs.send\_message through send\_message\_partial call by keeping the value of Message.Body and QueueUrl as constant

import boto3

import json

from functools import partial

# SQS client from boto3

sqs = boto3.client('sqs')

# Constants and variables

INSERTION\_TYPE = "bulk\_insert"

SQS\_QUEUE\_URL = "https://sqs.<region>.amazonaws.com/<account-id>/<queue-name>"

emplid\_batch = [123, 456, 789]

# Partially apply arguments to sqs.send\_message

send\_message\_partial = partial(

    sqs.send\_message,

    MessageBody=json.dumps(

        {"insertion\_type": INSERTION\_TYPE, "amzn\_empl\_ids": [\*emplid\_batch]}

    ),

    QueueUrl=SQS\_QUEUE\_URL,

)

# Now you can call `send\_message\_partial()` without passing any arguments

response = send\_message\_partial()

print(response)

**functools.lru\_cache**

from functools import lru\_cache

# Example: Fibonacci with caching

@lru\_cache(maxsize=None)  # Cache all results

def fibonacci(n):

    if n < 2:

        return n

    return fibonacci(n-1) + fibonacci(n-2)

# Call the function and the result will be cached

print(fibonacci(30))  # Output: 832040

from functools import lru\_cache

@lru\_cache(maxsize=None)  # Cache unlimited results

def fibonacci(n):

    if n < 2:

        return n

    return fibonacci(n-1) + fibonacci(n-2)

# Without caching, this would take a long time for large n

print(fibonacci(50))  # Output: 12586269025

**Explanation**:

* The fibonacci function is recursive, and without caching, it would recompute the same values many times.
* By adding @lru\_cache, we store results, avoiding redundant calculations. For example, once fibonacci(10) is computed, it will not be recomputed for the remainder of the execution.

# Loop through numbers 1 to 5

for num in range(1, 6):

    if num == 3:

        print(f"Skipping {num}!")

        continue  # Skip the rest of this iteration when num is 3

    print(f"Processing {num}")

Processing 1

Processing 2

Skipping 3!

Processing 4

Processing 5

Example of continue and break:-

grades = [85, 92, None, 78, -1, 88]

# Iterate over a range of 10

for i in range(10):

    if i == 5:

        print("Encountered 5, breaking out of the loop.")

        break  # Exit the loop when i is 5

    grade = grades[i % len(grades)]  # Simulating grade assignment based on index (circular)

    if grade is None:

        print("Skipping missing grade")

        continue  # Skip this iteration when grade is None

    if grade < 0:

        print("Encountered invalid grade, stopping.")

        break  # Stop the loop when grade is negative

    print(f"Processing grade {i}: {grade}")

    Processing grade 0: 85

Processing grade 1: 92

Skipping missing grade

Processing grade 3: 78

Encountered invalid grade, stopping.

Creating a DS:-



from decimal import Decimal

record = {

    "edge": "user",

    "id": "12345",

    "score": Decimal('95.5'),

    "timestamp": "2024-09-06T12:00:00Z"

}

event = {

    "Keys": {"edge": {}, "id": {}},

    "NewRecord": {},

    "EventName": "INSERT",

    "InsertionType": insertion\_type

}

event["Keys"]["edge"]["S"] = record["edge"]

event["Keys"]["id"]["S"] = record["id"]

for key, val in record.items():

    if isinstance(val, Decimal):

        val = int(val)

    event["NewRecord"][key] = val

    {

        "Keys": {

            "edge": {"S": "user"},

            "id": {"S": "12345"}

        },

        "NewRecord": {

            "edge": "user",

            "id": "12345",

            "score": 95,  # Decimal('95.5') converted to int

            "timestamp": "2024-09-06T12:00:00Z"

        },

        "EventName": "INSERT",

        "InsertionType": "DEFAULT"  # Or any value provided to the function

    }

**Here's an example Python code that sends 50 messages to an SQS queue in batches of 10, using the AWS SDK (boto3):**

import boto3

from botocore.exceptions import ClientError

# Initialize the SQS client

sqs = boto3.client('sqs')

# Set the URL of the SQS queue

queue\_url = 'https://sqs.<your-region>.amazonaws.com/<your-account-id>/<your-queue-name>'

# Function to send messages in batches of 10

def send\_messages\_to\_sqs\_batch():

    total\_messages = 50

    batch\_size = 10

    # Generate messages and send in batches

    for i in range(0, total\_messages, batch\_size):

        # Create a batch of messages

        entries = [

            {

                'Id': str(j),  # Unique identifier for each message in the batch

                'MessageBody': f'Message {i + j + 1}'  # The message content

            }

            for j in range(batch\_size)

        ]

        # Send the batch to the SQS queue

        try:

            response = sqs.send\_message\_batch(

                QueueUrl=queue\_url,

                Entries=entries

            )

            print(f'Successfully sent batch {i // batch\_size + 1}: {response}')

        except ClientError as e:

            print(f'Error sending batch {i // batch\_size + 1}: {e}')

# Call the function to send 50 messages

send\_messages\_to\_sqs\_batch()

**Example :-**

First iteration (i = 0):

j goes from 0 to 9.

The batch will contain messages numbered 1 to 10 (i + j + 1 = 0 + j + 1).

Output for this batch:

python

Copy code

[

    {'Id': '0', 'MessageBody': 'Message 1'},

    {'Id': '1', 'MessageBody': 'Message 2'},

    {'Id': '2', 'MessageBody': 'Message 3'},

    {'Id': '3', 'MessageBody': 'Message 4'},

    {'Id': '4', 'MessageBody': 'Message 5'},

    {'Id': '5', 'MessageBody': 'Message 6'},

    {'Id': '6', 'MessageBody': 'Message 7'},

    {'Id': '7', 'MessageBody': 'Message 8'},

    {'Id': '8', 'MessageBody': 'Message 9'},

    {'Id': '9', 'MessageBody': 'Message 10'}

]

Second iteration (i = 10):

j goes from 0 to 9.

The batch will contain messages numbered 11 to 20 (i + j + 1 = 10 + j + 1).

Output for this batch:

python

Copy code

[

    {'Id': '0', 'MessageBody': 'Message 11'},

    {'Id': '1', 'MessageBody': 'Message 12'},

    {'Id': '2', 'MessageBody': 'Message 13'},

    {'Id': '3', 'MessageBody': 'Message 14'},

    {'Id': '4', 'MessageBody': 'Message 15'},

    {'Id': '5', 'MessageBody': 'Message 16'},

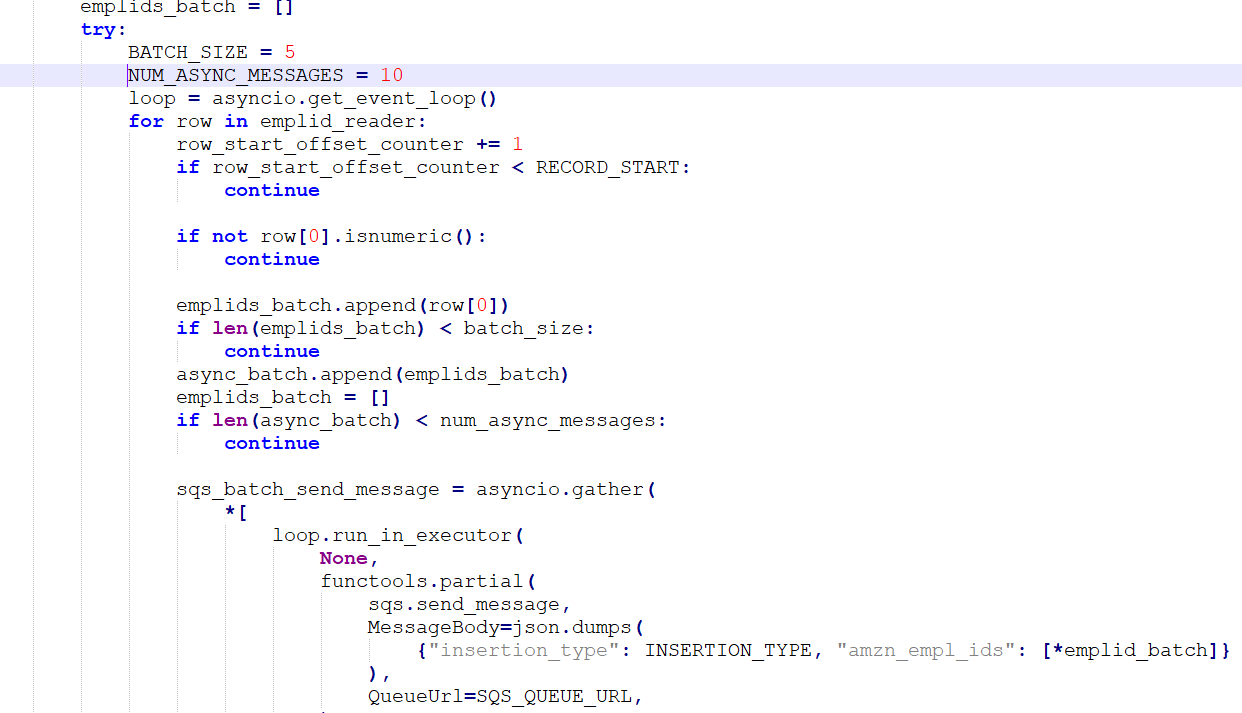
    {'Id': '6', 'MessageBody': 'Message 17'},

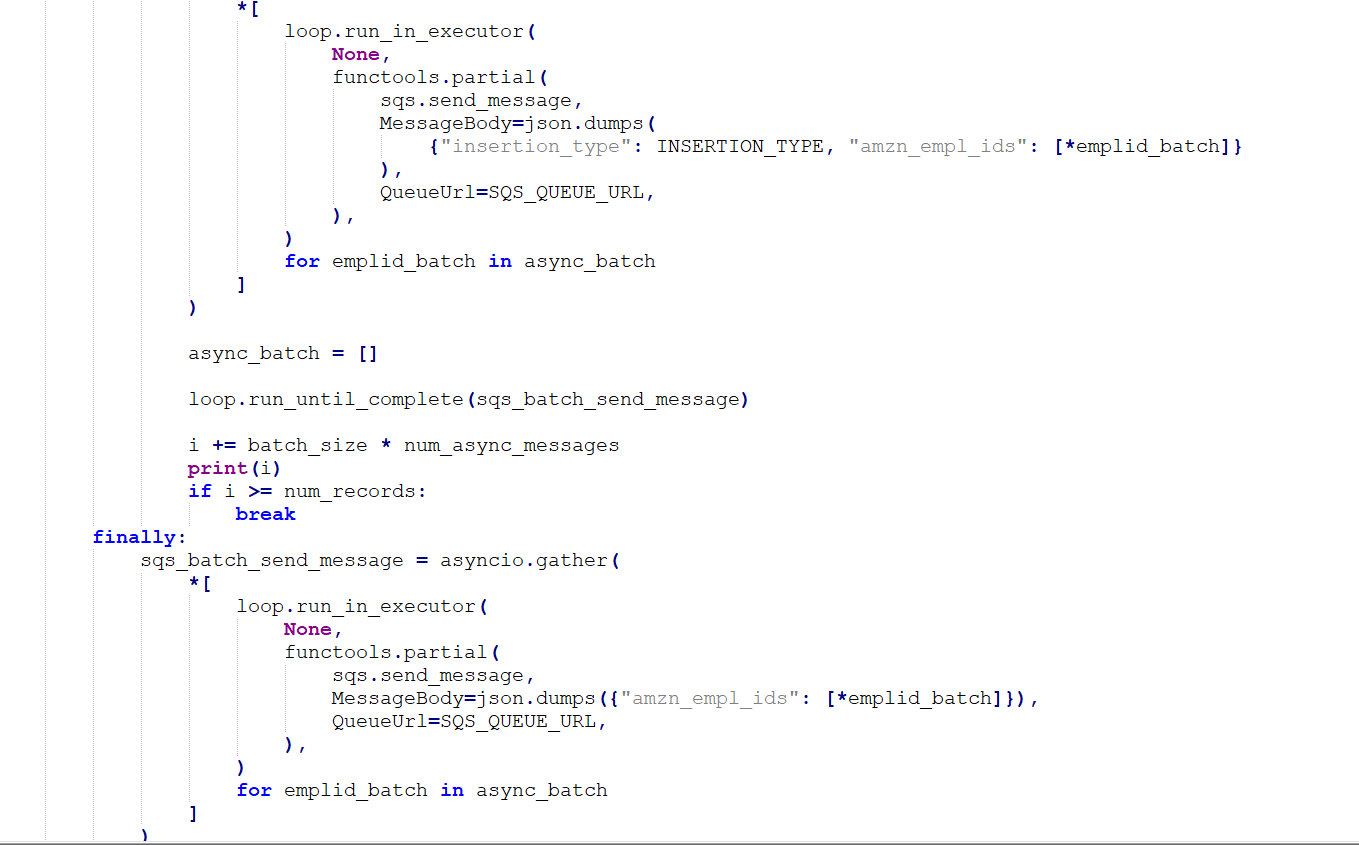
    {'Id': '7', 'MessageBody': 'Message 18'},

    {'Id': '8', 'MessageBody': 'Message 19'},

    {'Id': '9', 'MessageBody': 'Message 20'}

Here as long as if conditions are met we are appending data in the list once the condition is broken it would go to next if and repeat. Batch size is 5 unless batch size isn’t 5 it would keep appending rows to empls\_id\_batch once first batch is completed then it would go to next batch unless length of batch size is total asyn messages. For first call to lambda there were





**Medium** - <https://medium.com/@oguzhanhiziroglu/unleashing-the-power-of-aws-sqs-and-lambda-for-efficient-message-handling-9a1f3ff200ff>

# **AWS Lambda Functions with Batch Messages**

:- <https://qxf2.com/blog/testing-aws-lambda-functions-with-batch-messages/>

1. Install the requests library (if not already installed):

pip install requests

2. Python code to read xAPI statements:

import requests

import json

from requests.auth import HTTPBasicAuth

# LRS endpoint (replace with your LRS endpoint)

LRS\_URL = "https://your-lrs.com/xAPI/statements"

# Authentication (replace with your LRS credentials)

username = "your\_username"

password = "your\_password"

# Function to fetch xAPI statements from LRS

def fetch\_xapi\_statements():

    try:

        # Parameters for the request (e.g., limit the number of statements)

        params = {

            "limit": 10  # Fetch the latest 10 xAPI statements

        }

        # Send GET request to LRS with basic authentication

        response = requests.get(LRS\_URL, auth=HTTPBasicAuth(username, password), params=params)

        # Check if the request was successful

        if response.status\_code == 200:

            # Parse the JSON response

            statements = response.json()

            # Print each statement

            for statement in statements['statements']:

                print(json.dumps(statement, indent=4))

        else:

            print(f"Failed to fetch xAPI statements: {response.status\_code}")

    except Exception as e:

        print(f"An error occurred: {e}")

# Fetch and print xAPI statements

fetch\_xapi\_statements()

Explanation:

LRS URL: This is the endpoint provided by your LRS to access xAPI statements (typically /xAPI/statements).

Authentication: Many LRS systems use HTTP Basic Authentication, where you provide a username and password.

Query Parameters: You can modify the params dictionary to include other parameters like filtering by actor, verb, object, date range, etc. In this example, we are limiting the response to 10 statements.

Response: The response is a JSON object that contains an array of statements. Each statement captures learning experience data in the form of actor, verb, and object.

{

    "actor": {

        "name": "John Doe",

        "mbox": "mailto:john.doe@example.com"

    },

    "verb": {

        "id": "http://adlnet.gov/expapi/verbs/completed",

        "display": { "en-US": "completed" }

    },

    "object": {

        "id": "http://example.com/elearning/course/abc",

        "definition": {

            "name": { "en-US": "Introduction to Data Analysis" }

        }

    },

    "timestamp": "2024-09-11T12:00:00Z"

}

3. Python code to fetch xAPI statements and load them into Pandas DataFrame:-

import requests

import json

import pandas as pd

from requests.auth import HTTPBasicAuth

# LRS endpoint (replace with your LRS endpoint)

LRS\_URL = "https://your-lrs.com/xAPI/statements"

# Authentication (replace with your LRS credentials)

username = "your\_username"

password = "your\_password"

# Function to fetch xAPI statements from LRS and load into a DataFrame

def fetch\_xapi\_statements():

    try:

        # Parameters for the request (e.g., limit the number of statements)

        params = {

            "limit": 10  # Fetch the latest 10 xAPI statements

        }

        # Send GET request to LRS with basic authentication

        response = requests.get(LRS\_URL, auth=HTTPBasicAuth(username, password), params=params)

        # Check if the request was successful

        if response.status\_code == 200:

            # Parse the JSON response

            statements = response.json()['statements']

            # Prepare a list to hold rows for DataFrame

            data = []

            # Extract relevant data from each statement

            for statement in statements:

                name = statement.get('actor', {}).get('name', None)

                emailid = statement.get('actor', {}).get('mbox', None)

                verb\_display = statement.get('verb', {}).get('display', {}).get('en-US', None)

                object\_id = statement.get('object', {}).get('id', None)

                object\_definition = statement.get('object', {}).get('definition', {}).get('name', {}).get('en-US', None)

                timestamp = statement.get('timestamp', None)

                # Append row to the data list

                data.append([name, emailid, verb\_display, object\_id, object\_definition, timestamp])

            # Create DataFrame with specified columns

            df = pd.DataFrame(data, columns=['name', 'emailid', 'verb\_display', 'object\_id', 'object\_definition', 'timestamp'])

            # Display DataFrame

            print(df)

            # Optionally, save to CSV

            df.to\_csv('xapi\_statements.csv', index=False)

        else:

            print(f"Failed to fetch xAPI statements: {response.status\_code}")

    except Exception as e:

        print(f"An error occurred: {e}")

# Fetch and load xAPI statements into DataFrame

fetch\_xapi\_statements()

**Logging:-**

import logging

# Set up basic configuration for logging

logging.basicConfig(

    level=logging.DEBUG,  # Set the logging level (DEBUG, INFO, WARNING, ERROR, CRITICAL)

    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',  # Format of the log messages

    handlers=[

        logging.FileHandler("app.log"),  # Log messages will also be written to app.log file

        logging.StreamHandler()  # Log messages will also be shown in the console

    ]

)

# Create a logger object

logger = logging.getLogger(\_\_name\_\_)

# Log messages with different severity levels

logger.debug("This is a debug message")

logger.info("This is an info message")

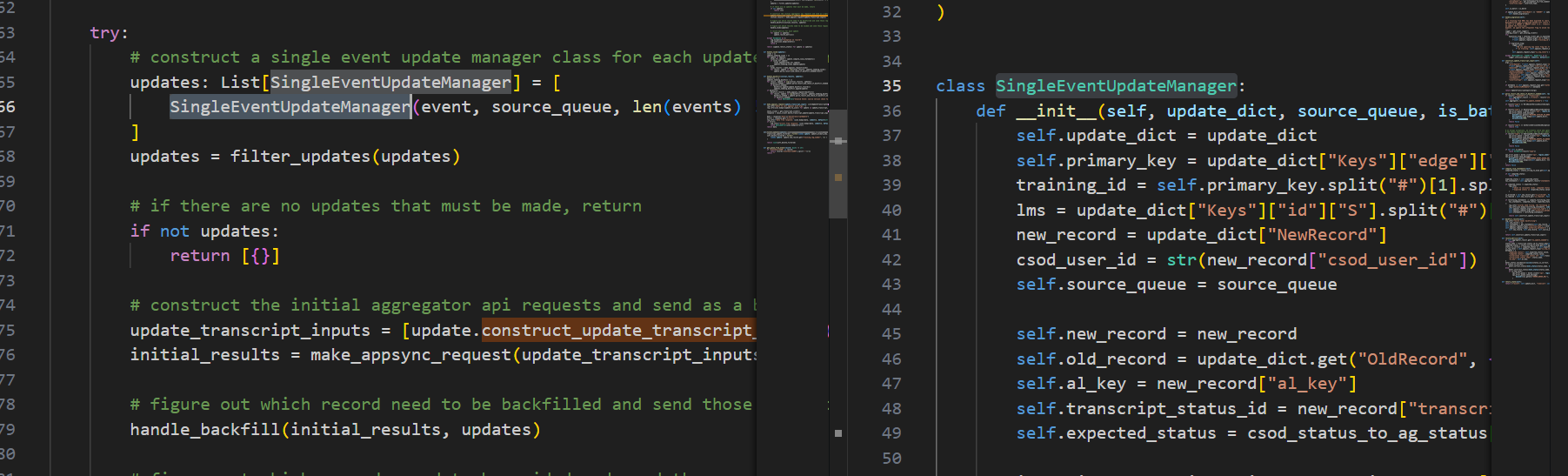
logger.warning("This is a warning message")

logger.error("This is an error message")

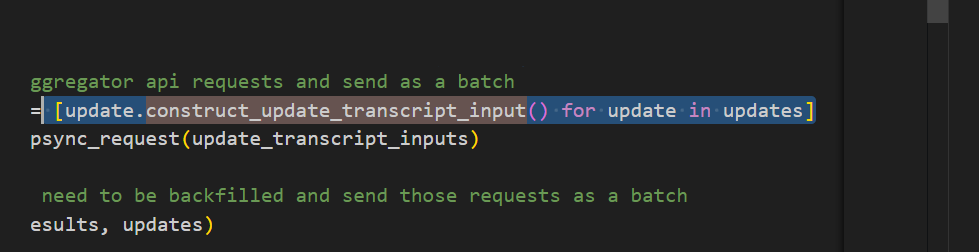
logger.critical("This is a critical message")

**Implementation from AZ calling class:-**

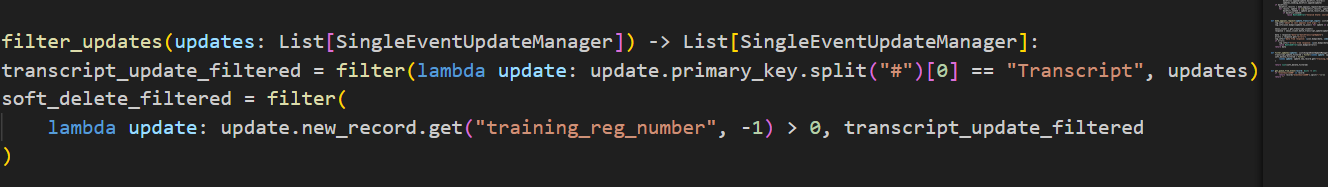
Calling a class for multiple SQS events in the list. So it would be a list of multiple classes with each event.



Method within the class defined in the list within class



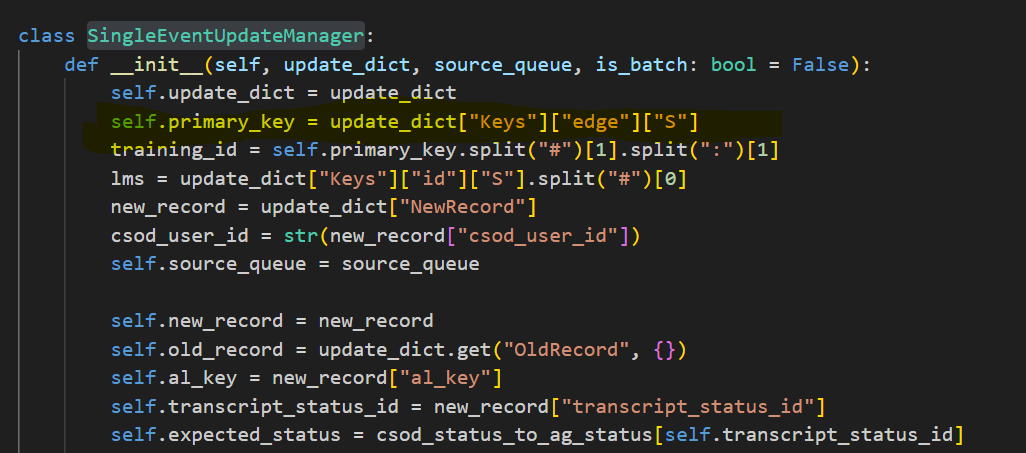
Variable of class defined in the list with an SQS event.



Methods within the class



Variable defined within the class:-



class person:

    def \_\_init\_\_(self, x, y) -> None:

       self.x = x

       self.y = y

    def \_\_add\_\_(self,other):

        return person(self.x + other.x , self.y + other.y)

    def \_\_repr\_\_(self):

        return f"x: {self.x}, y={self.y}"

v1 = person(10,20)

v2 = person(20,30)

v5 = v1 + v2

print(v5)

def logged(function):

    def wrapper(\*args, \*\*kwargs):

        value = function(\*args,\*\*kwargs)

        fname = function.\_\_name\_\_

        print(f"{fname} returned value {value}")

        return value

    return wrapper

@logged

def add(a,b):

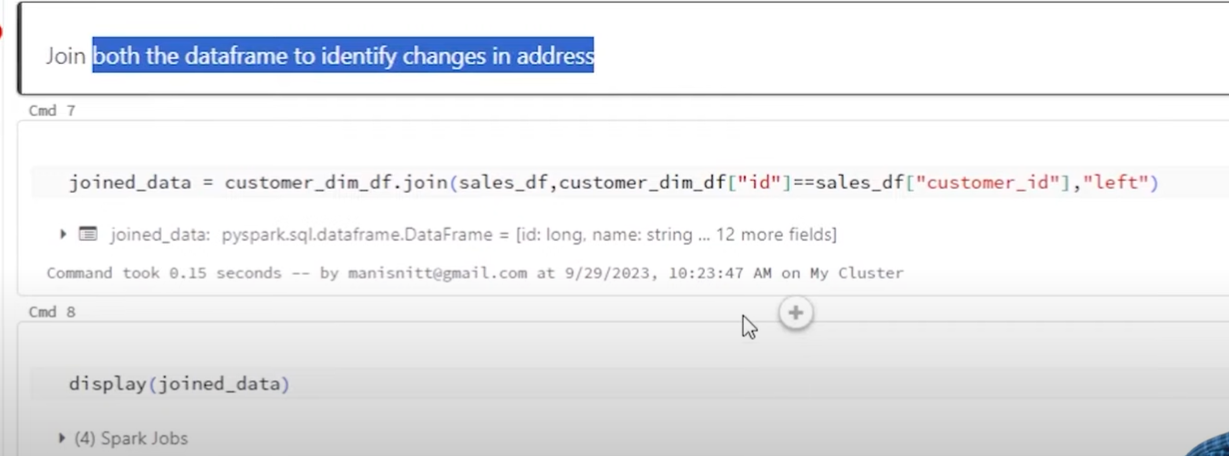
    return a + b

print(add(10,20))

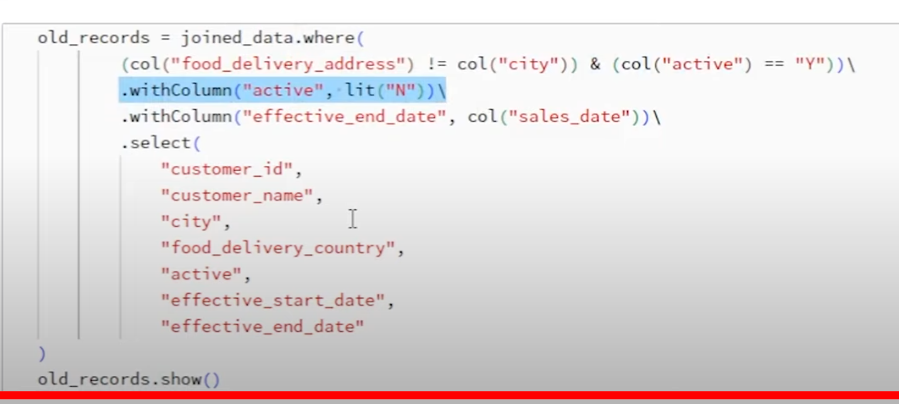
**Pyspark Videos: -**

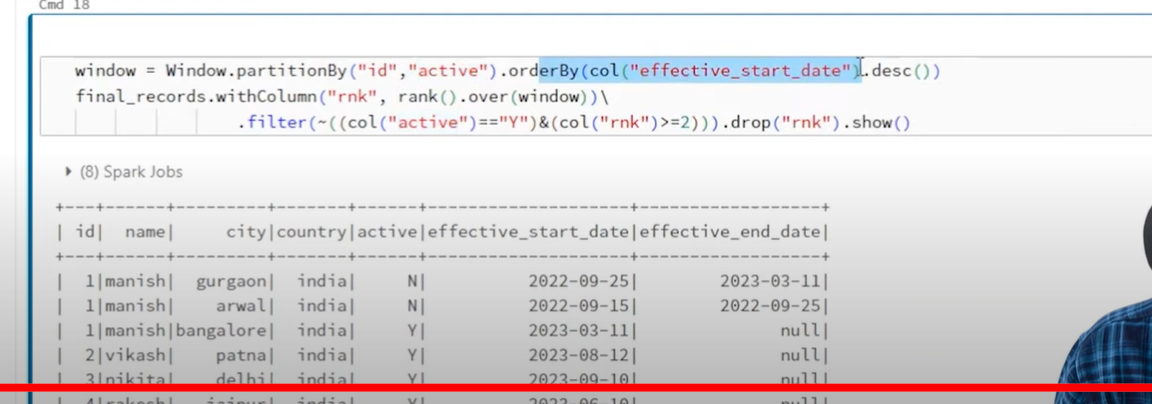
1. [Sagar Prajapati](https://www.linkedin.com/in/sagar-prajapati-86073b155/)'s playlist :  
<https://lnkd.in/gJaCvKpd>  
  
2. [Manish Kumar](https://www.linkedin.com/in/manish-kumar-data-engineer/)'s playlist  
<https://lnkd.in/gQJqD_mr>  
  
3. [Maheer Basha Shaik](https://www.linkedin.com/in/maheer-basha-shaik/)'s playlist:  
<https://lnkd.in/gMUYsYDn>  
  
4. Raja's Data engineering :  
<https://lnkd.in/gTkrYBGG>  
  
5. [Subham Khandelwal](https://www.linkedin.com/in/subhamkharwal/)'s playlist :  
<https://lnkd.in/gZdvSVb8>

SCD 2 in Python/Pyspark (There is some problem wih below snippets)









customer dim table with old and new customer info saved based on sales data and sales fact table. Write SCD2 for based on transactional file of sales fact in pyspark

from pyspark.sql import SparkSession

from pyspark.sql.functions import lit, col, current\_date

from pyspark.sql.window import Window

import pyspark.sql.functions as F

# Initialize Spark session

spark = SparkSession.builder \

    .appName("SCD Type 2") \

    .getOrCreate()

# Load existing customer dimension (dim\_customer)

dim\_customer\_df = spark.read.format("parquet").load("/path/to/dim\_customer")

# Load new sales transactional data (sales\_fact)

sales\_fact\_df = spark.read.format("parquet").load("/path/to/sales\_fact")

# Join sales\_fact with dim\_customer to identify new/changed customers

join\_cond = [dim\_customer\_df["customer\_id"] == sales\_fact\_df["customer\_id"]]

# Filter for active records in the dimension table

active\_dim\_customer\_df = dim\_customer\_df.filter(col("end\_date").isNull())

# Identifying new customers and changed customers

updated\_customers = sales\_fact\_df.join(active\_dim\_customer\_df, join\_cond, "left\_outer") \

    .filter(

        (active\_dim\_customer\_df["customer\_id"].isNull()) |  # New customer

        (sales\_fact\_df["customer\_name"] != active\_dim\_customer\_df["customer\_name"])  # Changed customer details

    )

# New records from sales and old records from Dim

# Marking the old records as expired for changed customers

expired\_customers = updated\_customers.select(

    active\_dim\_customer\_df["customer\_id"],

    active\_dim\_customer\_df["customer\_name"],

    active\_dim\_customer\_df["start\_date"],

    lit(current\_date()).alias("end\_date")  # Set current date as end date for expired records

)

# Insert new customer records into the dimension table with new details

new\_customer\_records = updated\_customers.select(

    sales\_fact\_df["customer\_id"],

    sales\_fact\_df["customer\_name"],

    current\_date().alias("start\_date"),

    lit(None).cast("date").alias("end\_date")  # New record without an end date

)

# Combine expired records with new records

final\_dim\_customer = active\_dim\_customer\_df \

    .union(expired\_customers) \

    .union(new\_customer\_records)

# Write final\_dim\_customer back to storage

final\_dim\_customer.write.mode("overwrite").parquet("/path/to/final\_dim\_customer")

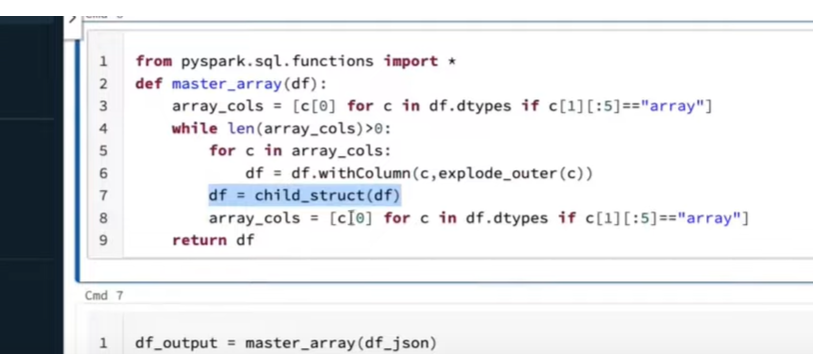
Handling a complex datatypes using python:-

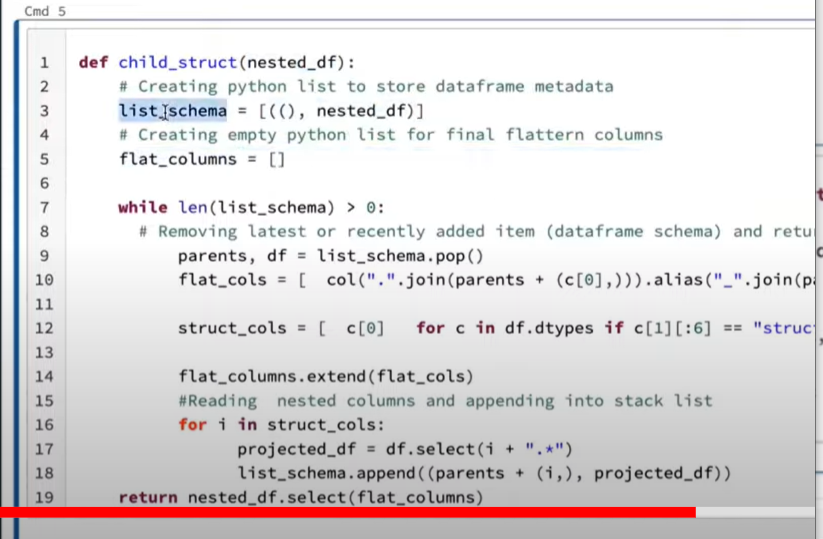
**Azure Help:-**

<https://github.com/raveendratal/ravi_azureadbadf/blob/main/azure_realtime_scenarios/dynamic_json_process.ipynb>

**TechLake:-** Complex Data types <https://www.youtube.com/watch?v=qfJb45SusMo&list=PL50mYnndduIF868zbDUPMBpJpwJwd4NZh&index=3>

YouTube Link:- <https://www.youtube.com/@TRRaveendra/videos>





**Pyspark:-**

**1. map()**

**Explanation**:

* The map() transformation applies a specified function to each element of the RDD and returns a new RDD containing the results.
* Each input element is transformed into exactly one output element.

**Use Case**:

* Use map() when you want to perform the same operation on each element of the dataset, such as mathematical transformations, string manipulations, etc.

from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "Map Example")

# Create an RDD from a collection of numbers

numbers\_rdd = sc.parallelize([1, 2, 3, 4, 5])

# Apply the map transformation to square each number

squared\_rdd = numbers\_rdd.map(lambda x: x \*\* 2)

# Collect and print the results

print(squared\_rdd.collect())

[1, 4, 9, 16, 25]

**2. flatMap()**

**Explanation**:

* The flatMap() transformation applies a specified function to each element of the RDD and flattens the results into a new RDD.
* Unlike map(), flatMap() can return multiple output elements for each input element or no output at all (resulting in a flattened structure).

**Use Case**:

* Use flatMap() when you want to process elements that can be split into multiple elements, such as tokenizing strings, or when you need to flatten nested lists.

**Example**: Suppose we have a list of sentences, and we want to split each sentence into words.

# Create an RDD of sentences

sentences\_rdd = sc.parallelize(["Hello world", "Welcome to Spark", "FlatMap is useful"])

# Use flatMap to split sentences into words

words\_rdd = sentences\_rdd.flatMap(lambda sentence: sentence.split(" "))

# Collect and print the results

print(words\_rdd.collect())

['Hello', 'world', 'Welcome', 'to', 'Spark', 'FlatMap', 'is', 'useful']

**Code Example:-**

from pyspark import SparkContext

# Create a SparkContext

sc = SparkContext("local", "List of Lists Example")

# Create an RDD of sentences

sentences\_rdd = sc.parallelize([

    "Hello world",

    "Welcome to Spark",

    "FlatMap is useful",

    "Map and FlatMap"

])

# Use map to split each sentence into a list of words (returns list of lists)

list\_of\_lists\_rdd = sentences\_rdd.map(lambda sentence: sentence.split(" "))

# Collect the RDD into a list of lists in the driver

list\_of\_lists = list\_of\_lists\_rdd.collect()

# Print the list of lists

print(list\_of\_lists)

[['Hello', 'world'],

['Welcome', 'to', 'Spark'],

['FlatMap', 'is', 'useful'],

['Map', 'and', 'FlatMap']]

### 1. Transformations in Spark

#### filter()

* **Explanation**: The filter() transformation filters the elements of an RDD or DataFrame based on a provided condition and returns a new RDD/DataFrame.

**Example with RDD**:

rdd = sc.parallelize([1, 2, 3, 4, 5])

filtered\_rdd = rdd.filter(lambda x: x % 2 == 0)

print(filtered\_rdd.collect())  # Output: [2, 4]

#### groupByKey()

* **Explanation**: The groupByKey() transformation groups values with the same key. It returns an RDD of key-value pairs where the key is associated with a collection of all values for that key. It does not aggregate, so it can be inefficient for large datasets.

rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3), ('b', 4)])

grouped\_rdd = rdd.groupByKey()

print([(x, list(y)) for x, y in grouped\_rdd.collect()])  # Output: [('a', [1, 3]), ('b', [2, 4])]

#### reduceByKey()

* **Explanation**: The reduceByKey() transformation is used for aggregation. It applies a reduction function (e.g., sum, max) to the values with the same key, reducing them to a single value for each key.

rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3), ('b', 4)])

reduced\_rdd = rdd.reduceByKey(lambda x, y: x + y)

print(reduced\_rdd.collect())  # Output: [('a', 4), ('b', 6)]

### 2. Actions in Spark

#### collect()

* **Explanation**: collect() brings all elements of the RDD or DataFrame into memory on the driver. It’s useful for small datasets but should be avoided for large datasets.

rdd = sc.parallelize([1, 2, 3])

result = rdd.collect()  # Output: [1, 2, 3]

#### count()

* **Explanation**: count() returns the number of elements in the RDD or DataFrame.

rdd = sc.parallelize([1, 2, 3])

count = rdd.count()  # Output: 3

#### take()

* **Explanation**: take() returns the first n elements of the RDD or DataFrame.

rdd = sc.parallelize([1, 2, 3, 4, 5])

result = rdd.take(3)  # Output: [1, 2, 3]

### 3. DataFrames and Datasets in Spark

#### **Creating DataFrames**

##### **From CSV**:

df = spark.read.csv("data.csv", header=True, inferSchema=True)

df.show()

**From JSON**:

df = spark.read.json("data.json")

df.show()

**From Parquet**:

df = spark.read.parquet("data.parquet")

df.show()

**From JDBC**:

df = spark.read \

    .format("jdbc") \

    .option("url", "jdbc:mysql://localhost:3306/mydb") \

    .option("dbtable", "mytable") \

    .option("user", "root") \

    .option("password", "password") \

    .load()

df.show()

### 4. DataFrame Operations

#### select()

* **Explanation**: select() allows you to select specific columns from a DataFrame.

**Example**:

df.select("name", "age").show()

#### filter()

* **Explanation**: filter() filters rows based on a condition.

**Example**:

df.filter(df.age > 30).show()

#### groupBy()

* **Explanation**: groupBy() groups rows by one or more columns.
* df.groupBy("age").count().show()

#### join()

* **Explanation**: join() allows you to join two DataFrames based on a common column.

**Example**:

df1.join(df2, df1.id == df2.id, "inner").show()

#### drop()

* **Explanation**: drop() removes one or more columns from a DataFrame.

**Example**:

df.drop("age").show()

#### withColumn()

* **Explanation**: withColumn() adds a new column or replaces an existing one.

**Example**:

df.withColumn("new\_col", df.age + 10).show()

### 7. Aggregation in DataFrames

#### agg()

* **Explanation**: agg() performs multiple aggregate functions on grouped data.

**Example**:

df.groupBy("age").agg({"salary": "avg", "bonus": "max"}).show()

#### groupBy().sum()

* **Explanation**: You can use groupBy() followed by sum() to compute the sum for grouped data.

**Example**:

df.groupBy("department").sum("salary").show()

### 8. Transformations in DataFrames

#### **Adding Columns**:

You can add a column using withColumn().

df.withColumn("new\_column", df["salary"] \* 2).show()

#### **Type Casting**:

To cast a column to a different type:

df.withColumn("salary\_int", df["salary"].cast("integer")).show()

### 9. Reading/Writing Data

#### .read() **for reading data**

* **CSV**:

df = spark.read.csv("data.csv", header=True)

* **Parquet**:

df = spark.read.parquet("data.parquet")

* **Avro:-**

df = spark.read.format("avro").load("data.avro")

squares = ['orange', 'orange', 'purple', 'blue ', 'orange']

new\_squares = []

i = 0

while(i < len(squares) and squares[i] == 'orange'):

new\_squares.append(squares[i])

i = i + 1

print (new\_squares)

## **Step 4:- -** Implement a code to count the frequency of all unique words[¶](https://cf-courses-data.static.labs.skills.network/jupyterlite/2.5.5/lab/index.html?mode=learn&env_type=jupyterlite&notebook_url=https%3A%2F%2Fcf-courses-data.static.labs.skills.network%2FIBMDeveloperSkillsNetwork-PY0101EN-SkillsNetwork%2Flabs%2Fjupyterlite%2Ffiles%2FModule_3%2FPY0101EN-3-5-Practice_lab-20230526-1685059200.jupyterlite.ipynb&file_path=PY0101EN%2Fjupyterlite%2Ffiles%2FModule+3%2FPY0101EN-3-5-Practice+lab.ipynb#Step-4:-Implement-a-code-to-count-the-frequency-of-all-unique-words)

class TextAnalyzer(object):

    def \_\_init\_\_ (self, text):

        # remove punctuation

        formattedText = text.replace('.','').replace('!','').replace('?','').replace(',','')

        # make text lowercase

        formattedText = formattedText.lower()

        self.fmtText = formattedText

    def freqAll(self):

        # split text into words

        wordList = self.fmtText.split(' ')

        # Create dictionary

        freqMap = {}

        for word in set(wordList): # use set to remove duplicates in list

            freqMap[word] = wordList.count(word)

        return freqMap

## Implement a code to count the frequency of a specific word[¶](https://cf-courses-data.static.labs.skills.network/jupyterlite/2.5.5/lab/index.html?mode=learn&env_type=jupyterlite&notebook_url=https%3A%2F%2Fcf-courses-data.static.labs.skills.network%2FIBMDeveloperSkillsNetwork-PY0101EN-SkillsNetwork%2Flabs%2Fjupyterlite%2Ffiles%2FModule_3%2FPY0101EN-3-5-Practice_lab-20230526-1685059200.jupyterlite.ipynb&file_path=PY0101EN%2Fjupyterlite%2Ffiles%2FModule+3%2FPY0101EN-3-5-Practice+lab.ipynb#Step-5:-Implement-a-code-to-count-the-frequency-of-a-specific-word)

class TextAnalyzer(object):

    def \_\_init\_\_ (self, text):

        # remove punctuation

        formattedText = text.replace('.','').replace('!','').replace('?','').replace(',','')

        # make text lowercase

        formattedText = formattedText.lower()

        self.fmtText = formattedText

    def freqAll(self):

        # split text into words

        wordList = self.fmtText.split(' ')

        # Create dictionary

        freqMap = {}

        for word in set(wordList): # use set to remove duplicates in list

            freqMap[word] = wordList.count(word)

        return freqMap

    def freqOf(self,word):

        # get frequency map

        freqDict = self.freqAll()

        if word in freqDict:

            return freqDict[word]

        else:

            return 0

#Copy a File

#Let's copy the file Example2.txt to the file Example3.txt:

# Copy file to another

with open('/Example2.txt','r') as readfile:

    with open('/Example3.txt','w') as writefile:

          for line in readfile:

                writefile.write(line)

# Verify if the copy is successfully executed

with open('/Example3.txt','r') as testwritefile:

    print(testwritefile.read())

**File Operations**

# Write line to file

exmp2 = '/Example2.txt'

with open(exmp2, 'w') as writefile:

    writefile.write("This is line A")

------------------

# Read file

with open(exmp2, 'r') as testwritefile:

    print(testwritefile.read())

---------------

# Write lines to file

with open(exmp2, 'w') as writefile:

    writefile.write("This is line A\n")

    writefile.write("This is line B\n")

----------------

# Check whether write to file

with open(exmp2, 'r') as testwritefile:

    print(testwritefile.read())

---------------

# Sample list of text

Lines = ["This is line A\n", "This is line B\n", "This is line C\n"]

Lines

--------------

# Write the strings in the list to text file

with open('/Example2.txt', 'w') as writefile:

    for line in Lines:

        print(line)

        writefile.write(line)

------------

# Verify if writing to file is successfully executed

with open('/Example2.txt', 'r') as testwritefile:

    print(testwritefile.read())

----------------

with open('/Example2.txt', 'w') as writefile:

    writefile.write("Overwrite\n")

with open('/Example2.txt', 'r') as testwritefile:

    print(testwritefile.read())

----------

Appending Files

We can write to files without losing any of the existing data as follows by

setting the mode argument to append: a. you can append a new line as follows:

# Write a new line to text file

with open('/Example2.txt', 'a') as testwritefile:

    testwritefile.write("This is line C\n")

    testwritefile.write("This is line D\n")

    testwritefile.write("This is line E\n")

You can verify the file has changed by running the following cell:

# Verify if the new line is in the text file

with open('/Example2.txt', 'r') as testwritefile:

    print(testwritefile.read())

----------

# Copy file to another

with open('/Example2.txt','r') as readfile:

    with open('/Example3.txt','w') as writefile:

          for line in readfile:

                writefile.write(line)

----------

Random records Generations

#Run this prior to starting the exercise

from random import randint as rnd

memReg = '/members.txt'

exReg = '/inactive.txt'

fee =('yes','no')

def genFiles(current,old):

    with open(current,'w+') as writefile:

        writefile.write('Membership No  Date Joined  Active  \n')

        data = "{:^13}  {:<11}  {:<6}\n"

        for rowno in range(20):

            date = str(rnd(2015,2020))+ '-' + str(rnd(1,12))+'-'+str(rnd(1,25))

            writefile.write(data.format(rnd(10000,99999),date,fee[rnd(0,1)]))

    with open(old,'w+') as writefile:

        writefile.write('Membership No  Date Joined  Active  \n')

        data = "{:^13}  {:<11}  {:<6}\n"

        for rowno in range(3):

            date = str(rnd(2015,2020))+ '-' + str(rnd(1,12))+'-'+str(rnd(1,25))

            writefile.write(data.format(rnd(10000,99999),date,fee[1]))

genFiles(memReg,exReg)

---

Clean and divide files:-

def cleanFiles(currentMem,exMem):

    with open(currentMem,'r+') as writeFile:

        with open(exMem,'a+') as appendFile:

            #get the data

            writeFile.seek(0)

            members = writeFile.readlines()

            #remove header

            header = members[0]

            members.pop(0)

            inactive = [member for member in members if ('no' in member)]

            '''

            The above is the same as

            for member in members:

            if 'no' in member:

                inactive.append(member)

            '''

            #go to the beginning of the write file

            writeFile.seek(0)

            writeFile.write(header)

            for member in members:

                if (member in inactive):

                    appendFile.write(member)

                else:

                    writeFile.write(member)

            writeFile.truncate()

memReg = 'members.txt'

exReg = 'inactive.txt'

cleanFiles(memReg,exReg)

# code to help you see the files

headers = "Membership No  Date Joined  Active  \n"

with open(memReg,'r') as readFile:

    print("Active Members: \n\n")

    print(readFile.read())

with open(exReg,'r') as readFile:

    print("Inactive Members: \n\n")

    print(readFile.read())

----

TestFiles:-

def testMsg(passed):

    if passed:

       return 'Test Passed'

    else :

       return 'Test Failed'

testWrite = "/testWrite.txt"

testAppend = "/testAppend.txt"

passed = True

genFiles(testWrite,testAppend)

with open(testWrite,'r') as file:

    ogWrite = file.readlines()

with open(testAppend,'r') as file:

    ogAppend = file.readlines()

try:

    cleanFiles(testWrite,testAppend)

except:

    print('Error')

with open(testWrite,'r') as file:

    clWrite = file.readlines()

with open(testAppend,'r') as file:

    clAppend = file.readlines()

# checking if total no of rows is same, including headers

if (len(ogWrite) + len(ogAppend) != len(clWrite) + len(clAppend)):

    print("The number of rows do not add up. Make sure your final files have the same header and format.")

    passed = False

for line in clWrite:

    if  'no' in line:

        passed = False

        print("Inactive members in file")

        break

    else:

        if line not in ogWrite:

            print("Data in file does not match original file")

            passed = False

print ("{}".format(testMsg(passed)))

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Package/Method** | **Description** | **Syntax and Code Example** | | |
| File opening modes | Different modes to open files for specific operations. | 1. Examples:  with open("data.txt", "r") as file: content = file.read() print(content)  with open("output.txt", "w") as file: file.write("Hello, world!")  with open("log.txt", "a") as file: file.write("Log entry: Something happened.") with open("data.txt", "r+") as file: content = file.read() file.write("Updated content: " + content)</td> | | |
|  |
|  |
|  |
| File reading methods | Different methods to read file content in various ways. | Syntax: | | |  |
| 1 | | |  |
| 2 | | |  |
| 3 | | |  |
| 1. file.readlines() # reads all lines as a list | | |  |
| 2. readline() # reads the next line as a string | | |  |
| 3. file.read() # reads the entire file content as a string | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 2 | | |  |
| 3 | | |  |
| 4 | | |  |
| 1. with open("data.txt", "r") as file: | | |  |
| 2. lines = file.readlines() | | |  |
| 3. next\_line = file.readline() | | |  |
| 4. content = file.read() | | |  |
| Copied! | | |  |
| File writing methods | Different write methods to write content to a file. | Syntax: | | |  |
| 1 | | |  |
| 2 | | |  |
| 1. file.write(content) # writes a string to the file | | |  |
| 2. file.writelines(lines) # writes a list of strings to the file | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 2 | | |  |
| 3 | | |  |
| 1. lines = ["Hello\n", "World\n"] | | |  |
| 2. with open("output.txt", "w") as file: | | |  |
| 3. file.writelines(lines) | | |  |
| Copied! | | |  |
| Iterating over lines | Iterates through each line in the file using a `loop`. | Syntax: | | |  |
| 1 | | |  |
| 1. for line in file: # Code to process each line | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 2 | | |  |
| 1. with open("data.txt", "r") as file: | | |  |
| 2. for line in file: print(line) | | |  |
| Copied! | | |  |
| Open() and close() | Opens a file, performs operations, and explicitly closes the file using the close() method. | Syntax: | | |  |
| 1 | | |  |
| 2 | | |  |
| 1. file = open(filename, mode) # Code that uses the file | | |  |
| 2. file.close() | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 2 | | |  |
| 3 | | |  |
| 1. file = open("data.txt", "r") | | |  |
| 2. content = file.read() | | |  |
| 3. file.close() | | |  |
| Copied! | | |  |
| with open() | Opens a file using a with block, ensuring automatic file closure after usage. | Syntax: | | |  |
| 1 | | |  |
| 1. with open(filename, mode) as file: # Code that uses the file | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 2 | | |  |
| 1. with open("data.txt", "r") as file: | | |  |
| 2. content = file.read() | | |  |
| Copied! | | |  |
|  | | |  |
| **Package/Method** | **Description** | **Syntax and Code Example** | | |  |
| .read\_csv() | Reads data from a `.CSV` file and creates a DataFrame. | Syntax: dataframe\_name = pd.read\_csv("filename.csv") Example: df = pd.read\_csv("data.csv") | | |  |
| .read\_excel() | Reads data from an Excel file and creates a DataFrame. | Syntax: | | |  |
| 1 | | |  |
| 1. dataframe\_name = pd.read\_excel("filename.xlsx") | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. df = pd.read\_excel("data.xlsx") | | |  |
| Copied! | | |  |
| .to\_csv() | Writes DataFrame to a CSV file. | Syntax: | | |  |
| 1 | | |  |
| 1. dataframe\_name.to\_csv("output.csv", index=False) | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. df.to\_csv("output.csv", index=False) | | |  |
| Copied! | | |  |
| Access Columns | Accesses a specific column using [] in the DataFrame. | Syntax: | | |  |
| 1 | | |  |
| 2 | | |  |
| 1. dataframe\_name["column\_name"] # Accesses single column | | |  |
| 2. dataframe\_name[["column1", "column2"]] # Accesses multiple columns | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 2 | | |  |
| 1. df["age"] | | |  |
| 2. df[["name", "age"]] | | |  |
| Copied! | | |  |
| describe() | Generates statistics summary of numeric columns in the DataFrame. | Syntax: | | |  |
| 1 | | |  |
| 1. dataframe\_name.describe() | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. df.describe() | | |  |
| Copied! | | |  |
| drop() | Removes specified rows or columns from the DataFrame. axis=1 indicates columns. axis=0 indicates rows. | Syntax: | | |  |
| 1 | | |  |
| 2 | | |  |
| 1. dataframe\_name.drop(["column1", "column2"], axis=1, inplace=True) | | |  |
| 2. dataframe\_name.drop(index=[row1, row2], axis=0, inplace=True) | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 2 | | |  |
| 1. df.drop(["age", "salary"], axis=1, inplace=True) # Will drop columns | | |  |
| 2. df.drop(index=[5, 10], axis=0, inplace=True) # Will drop rows | | |  |
| Copied! | | |  |
| dropna() | Removes rows with missing NaN values from the DataFrame. axis=0 indicates rows. | Syntax: | | |  |
| 1 | | |  |
| 1. dataframe\_name.dropna(axis=0, inplace=True) | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. df.dropna(axis=0, inplace=True) | | |  |
| Copied! | | |  |
| duplicated() | Duplicate or repetitive values or records within a data set. | Syntax: | | |  |
| 1 | | |  |
| 1. dataframe\_name.duplicated() | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. duplicate\_rows = df[df.duplicated()] | | |  |
| Copied! | | |  |
| Filter Rows | Creates a new DataFrame with rows that meet specified conditions. | Syntax: | | |  |
| 1 | | |  |
| 1. filtered\_df = dataframe\_name[(Conditional\_statements)] | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. filtered\_df = df[(df["age"] > 30) & (df["salary"] < 50000) | | |  |
| Copied! | | |  |
| groupby() | Splits a DataFrame into groups based on specified criteria, enabling subsequent aggregation, transformation, or analysis within each group. | Syntax: | | |  |
| 1 | | |  |
| 2 | | |  |
| 1. grouped = dataframe\_name.groupby(by, axis=0, level=None, as\_index=True, | | |  |
| 2. sort=True, group\_keys=True, squeeze=False, observed=False, dropna=True) | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. grouped = df.groupby(["category", "region"]).agg({"sales": "sum"}) | | |  |
| Copied! | | |  |
| head() | Displays the first n rows of the DataFrame. | Syntax: | | |  |
| 1 | | |  |
| 1. dataframe\_name.head(n) | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. df.head(5) | | |  |
| Copied! | | |  |
| Import pandas | Imports the Pandas library with the alias pd. | Syntax: | | |  |
| 1 | | |  |
| 1. import pandas as pd | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. import pandas as pd | | |  |
| Copied! | | |  |
| info() | Provides information about the DataFrame, including data types and memory usage. | Syntax: | | |  |
| 1 | | |  |
| 1. dataframe\_name.info() | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. df.info() | | |  |
| Copied! | | |  |
| merge() | Merges two DataFrames based on multiple common columns. | Syntax: | | |  |
| 1 | | |  |
| 1. merged\_df = pd.merge(df1, df2, on=["column1", "column2"]) | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. merged\_df = pd.merge(sales, products, on=["product\_id", "category\_id"]) | | |  |
| Copied! | | |  |
| print DataFrame | Displays the content of the DataFrame. | Syntax: | | |  |
| 1 | | |  |
| 1. print(df) # or just type df | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 2 | | |  |
| 1. print(df) | | |  |
| 2. df | | |  |
| Copied! | | |  |
| replace() | Replaces specific values in a column with new values. | Syntax: | | |  |
| 1 | | |  |
| 1. dataframe\_name["column\_name"].replace(old\_value, new\_value, inplace=True) | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. df["status"].replace("In Progress", "Active", inplace=True) | | |  |
| Copied! | | |  |
| tail() | Displays the last n rows of the DataFrame. | Syntax: | | |  |
| 1 | | |  |
| 1. dataframe\_name.tail(n) | | |  |
| Copied! | | |  |
| Example: | | |  |
| 1 | | |  |
| 1. df.tail(5) | | |  |

**Extracting Data from an API**

URL=[**"https://web.archive.org/web/20230902185326/https://en.wikipedia.org/wiki/List\_of\_countries\_by\_GDP\_%28nominal%29"**](https://web.archive.org/web/20230902185326/https:/en.wikipedia.org/wiki/List_of_countries_by_GDP_%28nominal%29)

### Exercise 1

Extract the required GDP data from the given URL using Web Scraping.

# Extract tables from webpage using Pandas. Retain table number 3 as the required dataframe.

tables = pd.read\_html(URL)

df = tables[3]

# Replace the column headers with column numbers

df.columns = range(df.shape[1])

# Retain columns with index 0 and 2 (name of country and value of GDP quoted by IMF)

df = df[[0,2]]

# Retain the Rows with index 1 to 10, indicating the top 10 economies of the world.

df = df.iloc[1:11,:]

# Assign column names as "Country" and "GDP (Million USD)"

df.columns = ['Country','GDP (Million USD)']

### Exercise 2

Modify the GDP column of the DataFrame, converting the value available in Million USD to Billion USD. Use the round() method of Numpy library to round the value to 2 decimal places. Modify the header of the DataFrame to GDP (Billion USD).

# Change the data type of the 'GDP (Million USD)' column to integer. Use astype() method.

df['GDP (Million USD)'] = df['GDP (Million USD)'].astype(int)

# Convert the GDP value in Million USD to Billion USD

df[['GDP (Million USD)']] = df[['GDP (Million USD)']]/1000

# Use numpy.round() method to round the value to 2 decimal places.

df[['GDP (Million USD)']] = np.round(df[['GDP (Million USD)']], 2)

# Rename the column header from 'GDP (Million USD)' to 'GDP (Billion USD)'

df.rename(columns = {'GDP (Million USD)' : 'GDP (Billion USD)'})

### Exercise 3

Load the DataFrame to the CSV file named "Largest\_economies.csv"

# Load the DataFrame to the CSV file named "Largest\_economies.csv"

df.to\_csv('./Largest\_economies.csv')

**## Example 1: RandomUser API**

* Bellow are Get Methods parameters that we can generate. For more information on the parameters, please visit this [documentation](https://randomuser.me/documentation) page.

## **Get Methods**

* get\_cell()
* get\_city()
* get\_dob()
* get\_email()
* get\_first\_name()
* get\_full\_name()
* get\_gender()
* get\_id()
* get\_id\_number()
* get\_id\_type()
* get\_info()
* get\_last\_name()
* get\_login\_md5()
* get\_login\_salt()
* get\_login\_sha1()
* get\_login\_sha256()
* get\_nat()
* get\_password()
* get\_phone()
* get\_picture()
* get\_postcode()
* get\_registered()
* get\_state()
* get\_street()
* get\_username()
* get\_zipcode()

##Example 1: RandomUser API

Bellow are Get Methods parameters that we can generate.

For more information on the parameters, please visit this [documentation](https://randomuser.me/documentation) page.

!pip install randomuser

!pip install pandas

from randomuser import RandomUser

import pandas as pd

r = RandomUser()

some\_list = r.generate\_users(10)

for user in some\_list:

    print (user.get\_full\_name()," ",user.get\_email())

def get\_users():

    users =[]

    for user in RandomUser.generate\_users(10):

        users.append({"Name":user.get\_full\_name(),"Gender":user.get\_gender(),"City":user.get\_city(),"State":user.get\_state(),"Email":user.get\_email(), "DOB":user.get\_dob(),"Picture":user.get\_picture()})

    return pd.DataFrame(users)

get\_users()

df1 = pd.DataFrame(get\_users())

##Example 2: Fruityvice API

import requests

import json

data = requests.get("https://web.archive.org/web/20240929211114/https://fruityvice.com/api/fruit/all")

results = json.loads(data.text)

pd.DataFrame(results)

df2 = pd.json\_normalize(results)

cherry = df2.loc[df2["name"] == 'Cherry']

(cherry.iloc[0]['family']) , (cherry.iloc[0]['genus'])

#Table of Contents

#Beautiful Soup Object

#Tag

#Children, Parents, and Siblings

#HTML Attributes

#Navigable String

#Filter

#find All

#find

#HTML Attributes

#Navigable String

#Downloading And Scraping The Contents Of A Web

!pip install bs4

!pip install requests pandas html5lib

from bs4 import BeautifulSoup # this module helps in web scrapping.

import requests  # this module helps us to download a web page

html="<!DOCTYPE html><html><head><title>Page Title</title></head><body><h3><b id='boldest'>Lebron James</b></h3><p> Salary: $ 92,000,000 </p><h3> Stephen Curry</h3><p> Salary: $85,000, 000 </p><h3> Kevin Durant </h3><p> Salary: $73,200, 000</p></body></html>"

soup = BeautifulSoup(html, 'html5lib')

print(soup.prettify())

Tags

Let's say we want the title of the page and the name of the top paid player.

We can use the Tag.

The Tag object corresponds to an HTML tag in the original document, for example, the tag title.

tag\_object=soup.title

print("tag object:",tag\_object)

print("tag object type:",type(tag\_object))

tag\_object=soup.h3

tag\_object

-------------------------------

import piplite

await piplite.install(['seaborn', 'lxml', 'openpyxl'])

import pandas as pd

from pyodide.http import pyfetch

filename = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-PY0101EN-SkillsNetwork/labs/Module%205/data/addresses.csv"

async def download(url, filename):

    response = await pyfetch(url)

    if response.status == 200:

        with open(filename, "wb") as f:

            f.write(await response.bytes())

await download(filename, "addresses.csv")

df = pd.read\_csv("addresses.csv", header=None)

df.columns =['First Name', 'Last Name', 'Location ', 'City','State','Area Code']

#### Selecting rows using .iloc and .loc

##Now, let's see how to use .loc for selecting rows from our DataFrame.

##\*\*loc() : loc() is label based data selecting method which means that we have to pass the name of the row or column which we want to select.\*\*

# To select the first row

df.loc[0]

# To select the 0th,1st and 2nd row of "First Name" column only

df.loc[[0,1,2], "First Name" ]

# To select the 0th,1st and 2nd row of "First Name" column only

df.iloc[[0,1,2], 0]

Reading the data from XLSX file

Let's load the data from XLSX file and define the sheet name. For loading the data you can use the Pandas library in python.

import pandas as pd

# Not needed unless you're running locally

# import urllib.request

# urllib.request.urlretrieve("https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-PY0101EN-SkillsNetwork/labs/Module%205/data/file\_example\_XLSX\_10.xlsx", "sample.xlsx")

filename = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-PY0101EN-SkillsNetwork/labs/Module%205/data/file\_example\_XLSX\_10.xlsx"

async def download(url, filename):

    response = await pyfetch(url)

    if response.status == 200:

        with open(filename, "wb") as f:

            f.write(await response.bytes())

await download(filename, "file\_example\_XLSX\_10.xlsx")

df = pd.read\_excel("file\_example\_XLSX\_10.xlsx")

##XML file format

##XML is also known as Extensible Markup Language.

##As the name suggests, it is a markup language. It has certain rules for encoding data.

##XML file format is a human-readable and machine-readable file format.

##Pandas does not include any methods to read and write XML files.

##Here, we will take a look at how we can use other modules to read data from an XML file, and load it into a Pandas DataFrame.

#Writing with xml.etree.ElementTree

#The xml.etree.ElementTree module comes built-in with Python. It provides functionality for parsing and creating XML documents. ElementTree represents the XML document as a tree. We can move across the document using nodes which are elements and sub-elements of the XML file.

#For more information please read the xml.etree.ElementTree documentation.

import xml.etree.ElementTree as ET

# create the file structure

employee = ET.Element('employee')

details = ET.SubElement(employee, 'details')

first = ET.SubElement(details, 'firstname')

second = ET.SubElement(details, 'lastname')

third = ET.SubElement(details, 'age')

first.text = 'Shiv'

second.text = 'Mishra'

third.text = '23'

# create a new XML file with the results

mydata1 = ET.ElementTree(employee)

# myfile = open("items2.xml", "wb")

# myfile.write(mydata)

with open("new\_sample.xml", "wb") as files:

    mydata1.write(files)

#Reading with xml.etree.ElementTree

#Let's have a look at a one way to read XML data and put it in a Pandas DataFrame.

#You can see the XML file in the Notepad of your local machine.

# Not needed unless running locally

# !wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-PY0101EN-SkillsNetwork/labs/Module%205/data/Sample-employee-XML-file.xml

import xml.etree.ElementTree as etree

filename = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-PY0101EN-SkillsNetwork/labs/Module%205/data/Sample-employee-XML-file.xml"

async def download(url, filename):

    response = await pyfetch(url)

    if response.status == 200:

        with open(filename, "wb") as f:

            f.write(await response.bytes())

await download(filename, "Sample-employee-XML-file.xml")

#You would need to firstly parse an XML file and create a list of columns for data frame, then extract useful information from the XML file and add to a pandas data frame.

#Here is a sample code that you can use.:

# Parse the XML file

tree = etree.parse("Sample-employee-XML-file.xml")

# Get the root of the XML tree

root = tree.getroot()

# Define the columns for the DataFrame

columns = ["firstname", "lastname", "title", "division", "building", "room"]

# Initialize an empty DataFrame

datatframe = pd.DataFrame(columns=columns)

# Iterate through each node in the XML root

for node in root:

    # Extract text from each element

    firstname = node.find("firstname").text

    lastname = node.find("lastname").text

    title = node.find("title").text

    division = node.find("division").text

    building = node.find("building").text

    room = node.find("room").text

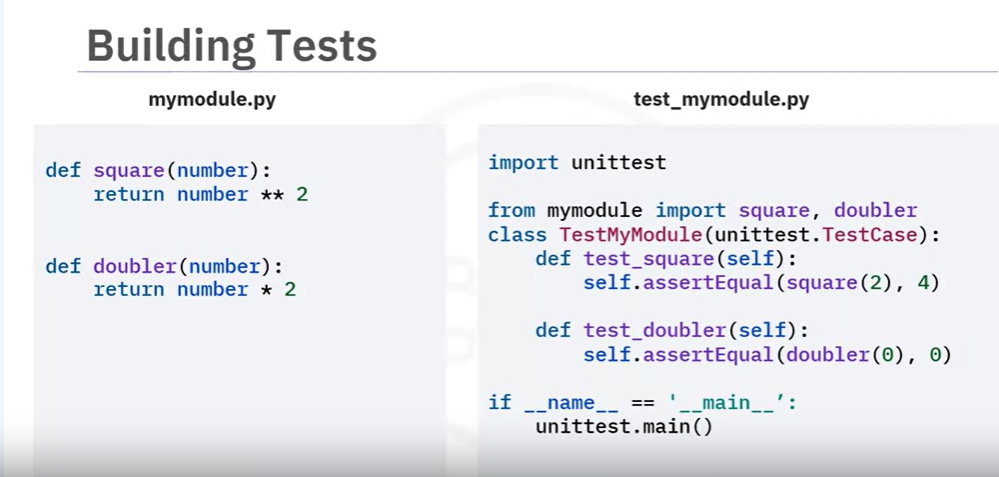
    # Create a DataFrame for the current row

    row\_df = pd.DataFrame([[firstname, lastname, title, division, building, room]], columns=columns)

    # Concatenate with the existing DataFrame

    datatframe = pd.concat([datatframe, row\_df], ignore\_index=True)

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Code Example** |
| Accessing element attribute | Access the value of a specific attribute of an HTML element. | Syntax: 1. attribute = element[(attribute)]  Example: 1. href = link\_element[(href)] |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| BeautifulSoup() | Parse the HTML content of a web page using BeautifulSoup. The parser type can vary based on the project. | Syntax: 1. soup = BeautifulSoup(html, (html.parser))  Example: 1. html = (https://api.example.com/data) soup = BeautifulSoup(html, (html.parser)) |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| delete() | Send a DELETE request to remove data or a resource from the server. DELETE requests delete a specified resource on the server. | Syntax: 1. response = requests.delete(url)  Example: 1. response = requests.delete((https://api.example.com/delete)) |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| find() | Find the first HTML element that matches the specified tag and attributes. | Syntax: 1. element = soup.find(tag, attrs)  Example:  1. first\_link = soup.find((a), {(class): (link)}) |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| find\_all() | Find all HTML elements that match the specified tag and attributes. | Syntax: 1. elements = soup.find\_all(tag, attrs)  Example:  1. all\_links = soup.find\_all((a), {(class): (link)})</td> |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| findChildren() | Find all child elements of an HTML element. | Syntax: 1. children = element.findChildren()  Example:  1. child\_elements = parent\_div.findChildren() |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| get() | Perform a GET request to retrieve data from a specified URL. GET requests are typically used for reading data from an API. The response variable will contain the server's response, which you can process further. | Syntax: 1. response = requests.get(url)  Example:  1. response = requests.get((https://api.example.com/data)) |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| Headers | Include custom headers in the request. Headers can provide additional information to the server, such as authentication tokens or content types. | Syntax: 1. headers = {(HeaderName): (Value)}  Example:  1. base\_url = (https://api.example.com/data) headers = {(Authorization): (Bearer YOUR\_TOKEN)} response = requests.get(base\_url, headers=headers) |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| Import Libraries | Import the necessary Python libraries for web scraping. | Syntax: 1. from bs4 import BeautifulSoup |  |
|  |
|  |
|  |
| json() | Parse JSON data from the response. This extracts and works with the data returned by the API. The response.json() method converts the JSON response into a Python data structure (usually a dictionary or list). | Syntax: 1. data = response.json()  Example:  1. response = requests.get((https://api.example.com/data))  2. data = response.json() |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| next\_sibling() | Find the next sibling element in the DOM. | Syntax: 1. sibling = element.find\_next\_sibling() Example:  1. next\_sibling = current\_element.find\_next\_sibling() |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| parent | Access the parent element in the Document Object Model (DOM). | Syntax: 1. parent = element.parent  Example:  1. parent\_div = paragraph.parent |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| post() | Send a POST request to a specified URL with data. Create or update POST requests using resources on the server. The data parameter contains the data to send to the server, often in JSON format. | Syntax: 1. response = requests.post(url, data)  Example:  1. response = requests.post((https://api.example.com/submit), data={(key): (value)}) |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| put() | Send a PUT request to update data on the server. PUT requests are used to update an existing resource on the server with the data provided in the data parameter, typically in JSON format. | Syntax: 1. response = requests.put(url, data)  Example:  1. response = requests.put((https://api.example.com/update), data={(key): (value)}) |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| Query parameters | Pass query parameters in the URL to filter or customize the request. Query parameters specify conditions or limits for the requested data. | Syntax: 1. params = {(param\_name): (value)}  Example:  1. base\_url = "https://api.example.com/data" 2. params = {"page": 1, "per\_page": 10} 3. response = requests.get(base\_url, params=params) |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| select() | Select HTML elements from the parsed HTML using a CSS selector. | Syntax: 1. element = soup.select(selector)  Example:  1. titles = soup.select((h1)) |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| status\_code | Check the HTTP status code of the response. The HTTP status code indicates the result of the request (success, error, redirection). Use the HTTP status codeIt can be used for error handling and decision-making in your code. | Syntax: 1. response.status\_code  Example:  1. url = "https://api.example.com/data" 2. response = requests.get(url) 3. status\_code = response.status\_code |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| tags for find() and find\_all() | Specify any valid HTML tag as the tag parameter to search for elements of that type. Here are some common HTML tags that you can use with the tag parameter. | "Tag Example: 1 - (a): Find anchor () tags. 2 - (p): Find paragraph ((p)) tags. 3 - (h1), (h2), (h3), (h4), (h5), (h6): Find heading tags from level 1 to 6 ( (h1),n (h2)). 4 - (table): Find table () tags. 5- (tr): Find table row () tags. 6- (td): Find table cell ((td)) tags. 7- (th): Find table header cell ((td))tags. 8- (img): Find image ((img)) tags. 9- (form): Find form ((form)) tags. 10- (button): Find button ((button)) tags." |  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| text | Retrieve the text content of an HTML element. | Syntax: 1. text = element.text Example:  1.title\_text = title\_element.text |  |
|  |
|  |
|  |
|  |
|  |
|  |



#Import the `unittest` module to create unit tests

import unittest

# Import the 'square' and 'double' functions from the 'mymodule' module.

from mymodule import square, double

class Testsquare(unittest.TestCase):

    def test1(self):

        self.assertEqual(square(2),4)

        self.assertEqual(square(3.0),9.0)

        self.assertNotEqual(square(-3),-9)

class TestDouble(unittest.TestCase):

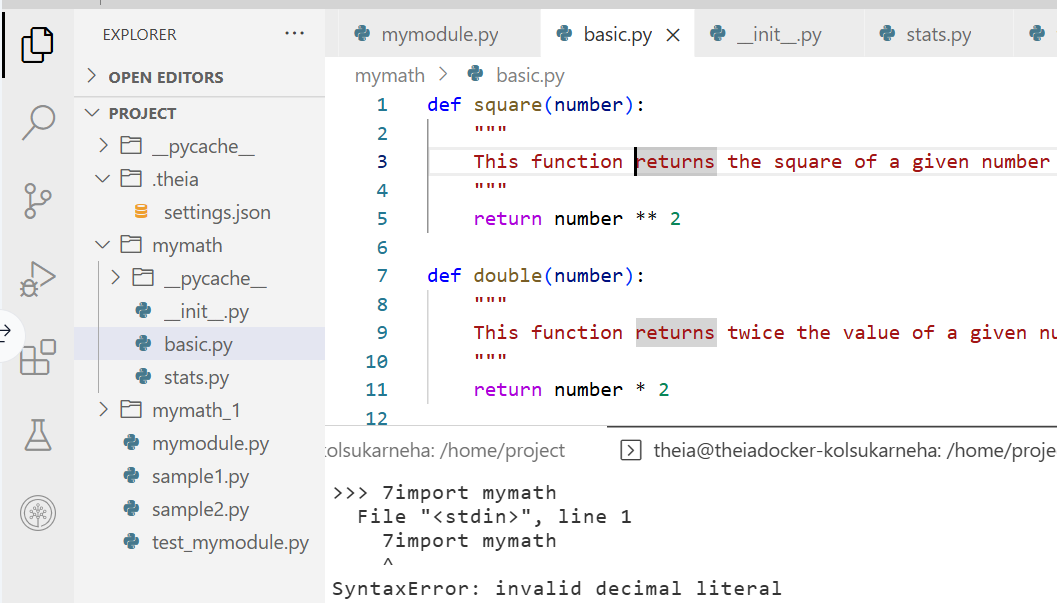
    def test1(self):

        self.assertEqual(double(2),4)

        self.assertEqual(double(-3.1),-6.2)

        self.assertEqual(double(0),0)

unittest.main()



Step 1: Set response status code

from flask import Flask, make\_response

# Create an instance of the Flask class, passing in the name of the current module

app = Flask(\_\_name\_\_)

# Define a route for the root URL ("/")

@app.route("/")

def index():

    # Function that handles requests to the root URL

    # Return a plain text response

    return "hello world"

# Define a route for the "/no\_content" URL

@app.route("/no\_content")

def no\_content():

    """Return 'no content found' with a status of 204.

    Returns:

        tuple: A tuple containing a dictionary and a status code.

    """

    # Create a dictionary with a message and return it with a 204 No Content status code

    return ({"message": "No content found"}, 204)

# Define a route for the "/exp" URL

@app.route("/exp")

def index\_explicit():

    """Return 'Hello World' message with a status code of 200.

    Returns:

        response: A response object containing the message and status code 200.

    """

    # Create a response object with the message "Hello World"

    resp = make\_response({"message": "Hello World"})

    # Set the status code of the response to 200

    resp.status\_code = 200

    # Return the response object

    return resp

----------------

Step 2: Process input arguments

server.py

from flask import Flask, make\_response

from flask import request

app = Flask(\_\_name\_\_)

data = [

    {

        "id": "3b58aade-8415-49dd-88db-8d7bce14932a",

        "first\_name": "Tanya",

        "last\_name": "Slad",

        "graduation\_year": 1996,

        "address": "043 Heath Hill",

        "city": "Dayton",

        "zip": "45426",

        "country": "United States",

        "avatar": "http://dummyimage.com/139x100.png/cc0000/ffffff",

    },

    {

        "id": "d64efd92-ca8e-40da-b234-47e6403eb167",

        "first\_name": "Ferdy",

        "last\_name": "Garrow",

        "graduation\_year": 1970,

        "address": "10 Wayridge Terrace",

        "city": "North Little Rock",

        "zip": "72199",

        "country": "United States",

        "avatar": "http://dummyimage.com/148x100.png/dddddd/000000",

    },

    {

        "id": "66c09925-589a-43b6-9a5d-d1601cf53287",

        "first\_name": "Lilla",

        "last\_name": "Aupol",

        "graduation\_year": 1985,

        "address": "637 Carey Pass",

        "city": "Gainesville",

        "zip": "32627",

        "country": "United States",

        "avatar": "http://dummyimage.com/174x100.png/ff4444/ffffff",

    },

    {

        "id": "0dd63e57-0b5f-44bc-94ae-5c1b4947cb49",

        "first\_name": "Abdel",

        "last\_name": "Duke",

        "graduation\_year": 1995,

        "address": "2 Lake View Point",

        "city": "Shreveport",

        "zip": "71105",

        "country": "United States",

        "avatar": "http://dummyimage.com/145x100.png/dddddd/000000",

    },

    {

        "id": "a3d8adba-4c20-495f-b4c4-f7de8b9cfb15",

        "first\_name": "Corby",

        "last\_name": "Tettley",

        "graduation\_year": 1984,

        "address": "90329 Amoth Drive",

        "city": "Boulder",

        "zip": "80305",

        "country": "United States",

        "avatar": "http://dummyimage.com/198x100.png/cc0000/ffffff",

    }

]

@app.route("/data")

def get\_data():

    try:

        # Check if 'data' exists and has a length greater than 0

        if data and len(data) > 0:

            # Return a JSON response with a message indicating the length of the data

            return {"message": f"Data of length {len(data)} found"}

        else:

            # If 'data' is empty, return a JSON response with a 500 Internal Server Error status code

            return {"message": "Data is empty"}, 500

    except NameError:

        # Handle the case where 'data' is not defined

        # Return a JSON response with a 404 Not Found status code

        return {"message": "Data not found"}, 404

##curl -X GET -i -w '\n' localhost:5000/data

@app.route("/name\_search")

def name\_search():

    """Find a person in the database based on the provided query parameter.

    Returns:

        json: Person if found, with status of 200

        404: If not found

        422: If the argument 'q' is missing

    """

    # Get the 'q' query parameter from the request URL

    #curl -X GET -i -w '\n' "localhost:5000/name\_search?q=Abdel"

    query = request.args.get("q")

    # Check if the query parameter 'q' is missing or empty

    if not query:

        # Return a JSON response with a message indicating invalid input and a 422 Unprocessable Entity status code

        return {"message": "Invalid input parameter"}, 422

    # Iterate through the 'data' list to search for a matching person

    for person in data:

        # Check if the query string is present in the person's first name (case-insensitive)

        if query.lower() in person["first\_name"].lower():

            # Return the matching person as a JSON response with a 200 OK status code

            return person

    # If no matching person is found, return a JSON response with a message and a 404 Not F

---------

Step 3: Add dynamic URLs

@app.route("/count")

def count():

    try:

        # Attempt to return the count of items in 'data' as a JSON response

        return {"data count": len(data)}, 200

    except NameError:

        # Handle the case where 'data' is not defined

        # Return a JSON response with a message and a 500 Internal Server Error status code

        return {"message": "data not defined"}, 500

#curl -X GET -i -w '\n' "localhost:5000/count"

@app.route("/person/<uuid:id>")

def find\_by\_uuid(id):

    # Iterate through the 'data' list to search for a person with a matching ID

    for person in data:

        # Check if the 'id' field of the person matches the 'id' parameter

        if person["id"] == str(id):

            # Return the matching person as a JSON response with a 200 OK status code

            return person

    # If no matching person is found, return a JSON response with a message and a 404 Not Found status code

    return {"message": "person not found"}, 404

#curl -X GET -i localhost:5000/person/66c09925-589a-43b6-9a5d-d1601cf53287

#curl -X GET -i localhost:5000/person/not-a-valid-uuid

@app.route("/person/<uuid:id>", methods=['DELETE'])

def delete\_by\_uuid(id):

    # Iterate through the 'data' list to search for a person with a matching ID

    for person in data:

        # Check if the 'id' field of the person matches the 'id' parameter

        if person["id"] == str(id):

            # Remove the person from the 'data' list

            data.remove(person)

            # Return a JSON response with a message confirming deletion and a 200 OK status code

            return {"message": f"Person with ID {id} deleted"}, 200

    # If no matching person is found, return a JSON response with a message and a 404 Not Found status code

    return {"message": "person not found"}, 404

#curl -X DELETE -i localhost:5000/person/66c09925-589a-43b6-9a5d-d1601cf53287

-------

Step 4: Parse JSON from Request body

from flask import request

@app.route("/person", methods=['POST'])

def add\_by\_uuid():

    new\_person = request.json

    if not new\_person:

        return {"message": "Invalid input parameter"}, 422

    # code to validate new\_person ommited

    try:

        data.append(new\_person)

    except NameError:

        return {"message": "data not defined"}, 500

    return {"message": f"{new\_person['id']}"}, 200

#curl -X POST -i -w '\n' \

# --url http://localhost:5000/person \

# --header 'Content-Type: application/json' \

# --data '{

#       "id": "4e1e61b4-8a27-11ed-a1eb-0242ac120002",

#       "first\_name": "John",

#       "last\_name": "Horne",

#       "graduation\_year": 2001,

#       "address": "1 hill drive",

#       "city": "Atlanta",

#       "zip": "30339",

#       "country": "United States",

#       "avatar": "http://dummyimage.com/139x100.png/cc0000/ffffff"

#}'

---------------

Step 5: Add error handlers

#curl -X POST -i -w '\n' http://localhost:5000/notvalid

@app.errorhandler(404)

def api\_not\_found(error):

    # This function is a custom error handler for 404 Not Found errors

    # It is triggered whenever a 404 error occurs within the Flask application

    return {"message": "API not found"}, 404

----

# Flask redirect

from flask import redirect

@app.route('/admin')

def admin():

    return redirect('/login')

# URL for

from flask import url\_for

@app.route('/admin')

def admin():

    return redirect(url\_for('login'))

@app.route('/login')

def login():

    return "<Login Page>"

# Handling different HTTP request types

@app.route('/data', methods=['GET', 'POST'])

def data():

    if request.method == 'POST':

        # process POST request

    if request.method == 'GET':

        # process GET request

-----------

Python code:-

@app.route('/create', methods=['GET', 'POST'])

def create():

    if request.method == 'POST':

        # Access form data

        name = request.form['name']

        # Create a new record with the name

        record = create\_new\_record(name)  # Assuming you have this function defined

        # Redirect user to the new record

        return redirect(url\_for('read', id=record.id))

    # Render the form for GET request

    return render\_template('create.html')

@app.route('/read/<int:id>', methods=['GET'])

def read(id):

    # Get the record by id

    record = get\_record(id)  # Assuming you have this function defined

    # Render a template with the record

    return render\_template('read.html', record=record)

@app.route('/update/<int:id>', methods=['GET', 'POST'])

def update(id):

    if request.method == 'POST':

        # Access form data

        name = request.form['name']

        # Update the record with the new name

        update\_record(id, name)  # Assuming you have this function defined

        # Redirect user to the updated record

        return redirect(url\_for('read', id=id))

    # Render the form for GET request with current data

    record = get\_record(id)  # Assuming you have this function defined

    return render\_template('update.html', record=record)

# Flask with html forms

flask.request.form is used to retrieve form data that has been sent to the server via an HTTP POST request.

Here’s a simple example that demonstrates how to use request.form in Flask:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Form Example</title>

</head>

<body>

    <form action="/submit" method="POST">

        <label for="name">Name:</label>

        <input type="text" id="name" name="name"><br><br>

        <label for="email">Email:</label>

        <input type="email" id="email" name="email"><br><br>

        <button type="submit">Submit</button>

    </form>

</body>

</html>

#app.py

from flask import Flask, render\_template, request, redirect, url\_for

app = Flask(\_\_name\_\_)

# Route to display the form

@app.route('/')

def index():

    return render\_template('form.html')

# Route to handle form submission

@app.route('/submit', methods=['POST'])

def submit():

    # Access form data using request.form

    name = request.form.get('name')

    email = request.form.get('email')

    # Process the data or save it to a database

    # Here, we simply print it

    print(f"Name: {name}, Email: {email}")

    # Redirect to a success page or render a response

    return f"Received data - Name: {name}, Email: {email}"

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

Explanation

1. Accessing Form Data: In the submit route, request.form.get('name') retrieves the value associated with the name input field,

    and request.form.get('email') retrieves the value for email.

2. Printing Data (or further processing): The data can be printed, stored in a database, or used in any other processing.

3. Form Submission Route: After the form is submitted, this route processes the form data

    and displays it back or redirects to another page.

----------------

#Delete operation

1. Deleting data involves removing a record based on its ID.

   The Delete operation will typically require the ID to be passed, as reported by the HTML page,

    in the form of an argument to the function.

Sample HTML form for deleting data:

    <form method="POST" action="/delete/{{record.id}}">

    <input type="submit" value="Delete">

    </form>

    Python Code:-

    @app.route('/delete/<int:id>', methods=['POST'])

    def delete(id):

    # Delete the record

    delete\_record(id)  # Assuming you have this function defined

    # Redirect user to the homepage

    return redirect(url\_for('home'))

## Backend code for transactions

from flask import Flask, redirect, request, render\_template, url\_for

from flask\_sqlalchemy import SQLAlchemy

# Initialize Flask app

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///transactions.db'  # SQLite database

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

# Initialize SQLAlchemy

db = SQLAlchemy(app)

# Define Transaction model

class Transaction(db.Model):

    id = db.Column(db.Integer, primary\_key=True)

    date = db.Column(db.String(10), nullable=False)  # YYYY-MM-DD format

    amount = db.Column(db.Float, nullable=False)

# Create the database tables

with app.app\_context():

    db.create\_all()

# Read operation: List all transactions

@app.route("/")

def get\_transactions():

    transactions = Transaction.query.all()

    return render\_template("transactions.html", transactions=transactions)

# Create operation: Display add transaction form

@app.route("/add", methods=["GET", "POST"])

def add\_transaction():

    if request.method == 'POST':

        # Start a database transaction

        try:

            # Create a new transaction object using form field values

            new\_transaction = Transaction(

                date=request.form['date'],

                amount=float(request.form['amount'])

            )

            db.session.add(new\_transaction)  # Add to session

            db.session.commit()  # Commit transaction to database

        except Exception as e:

            db.session.rollback()  # Roll back the transaction if there's an error

            print("Error:", e)

        finally:

            db.session.close()  # Close the session

        return redirect(url\_for("get\_transactions"))

    return render\_template("form.html")

# Update operation: Display edit transaction form

@app.route("/edit/<int:transaction\_id>", methods=["GET", "POST"])

def edit\_transaction(transaction\_id):

    transaction = Transaction.query.get(transaction\_id)

    if not transaction:

        return {"message": "Transaction not found"}, 404

    if request.method == 'POST':

        try:

            # Update the transaction fields

            transaction.date = request.form['date']

            transaction.amount = float(request.form['amount'])

            db.session.commit()  # Commit changes to the database

        except Exception as e:

            db.session.rollback()  # Roll back the transaction if there's an error

            print("Error:", e)

        finally:

            db.session.close()  # Close the session

        return redirect(url\_for("get\_transactions"))

    return render\_template("edit.html", transaction=transaction)

# Delete operation: Delete a transaction

@app.route("/delete/<int:transaction\_id>")

def delete\_transaction(transaction\_id):

    transaction = Transaction.query.get(transaction\_id)

    if not transaction:

        return {"message": "Transaction not found"}, 404

    try:

        db.session.delete(transaction)  # Mark for deletion

        db.session.commit()  # Commit deletion

    except Exception as e:

        db.session.rollback()  # Roll back if there's an error

        print("Error:", e)

    finally:

        db.session.close()  # Close the session

    return redirect(url\_for("get\_transactions"))

# Run the Flask app

if \_\_name\_\_ == "\_\_main\_\_":

    app.run(debug=True)

#transcations HTML

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Transactions</title>

</head>

<body>

    <h1>Transactions</h1>

    <a href="{{ url\_for('add\_transaction') }}">Add New Transaction</a>

    <table border="1" cellpadding="10" cellspacing="0">

        <tr>

            <th>ID</th>

            <th>Date</th>

            <th>Amount</th>

            <th>Actions</th>

        </tr>

        {% for transaction in transactions %}

        <tr>

            <td>{{ transaction.id }}</td>

            <td>{{ transaction.date }}</td>

            <td>{{ transaction.amount }}</td>

            <td>

                <a href="{{ url\_for('edit\_transaction', transaction\_id=transaction.id) }}">Edit</a> |

                <a href="{{ url\_for('delete\_transaction', transaction\_id=transaction.id) }}" onclick="return confirm('Are you sure you want to delete this transaction?');">Delete</a>

            </td>

        </tr>

        {% endfor %}

    </table>

</body>

</html>

##form.html

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Add Transaction</title>

</head>

<body>

    <h1>Add New Transaction</h1>

    <form action="{{ url\_for('add\_transaction') }}" method="POST">

        <label for="date">Date:</label>

        <input type="date" id="date" name="date" required><br><br>

        <label for="amount">Amount:</label>

        <input type="number" id="amount" name="amount" required step="0.01"><br><br>

        <button type="submit">Add Transaction</button>

    </form>

    <a href="{{ url\_for('get\_transactions') }}">Back to Transactions</a>

</body>

</html>

edit.html

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Edit Transaction</title>

</head>

<body>

    <h1>Edit Transaction</h1>

    <form action="{{ url\_for('edit\_transaction', transaction\_id=transaction.id) }}" method="POST">

        <label for="date">Date:</label>

        <input type="date" id="date" name="date" value="{{ transaction.date }}" required><br><br>

        <label for="amount">Amount:</label>

        <input type="number" id="amount" name="amount" value="{{ transaction.amount }}" required step="0.01"><br><br>

        <button type="submit">Update Transaction</button>

    </form>

    <a href="{{ url\_for('get\_transactions') }}">Back to Transactions</a>

</body>

</html>

<!DOCTYPE html>

<html>

<head>

    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

    <script src="./static/mywebscript.js"></script>

    <style>

        .center-heading {

            text-align: center;

        }

    </style>

</head>

<body>

    <div class="card">

    <h1 class="center-heading">Basic Mathematical Application</h1>

        <div style="padding: 25px 25px 25px 25px;">

        <div class="card-body">

            <div class="mb-3">

                <label class="form-label">Number 1</label>

                <input type="number" class="form-control" id="num1">

            </div>

            <div class="mb-3">

                <label class="form-label">Number 2</label>

                <input type="number" class="form-control" id="num2">

            </div>

            <div style="padding: 25px;">

                <button onclick="runAddition()" class="btn btn-secondary">Add</button>

                <button onclick="runSubtraction()" class="btn btn-secondary">Subtract</button>

                <button onclick="runMultiplication()" class="btn btn-secondary">Multiply</button>

            </div>

    <h1 class="center-heading">Response from server</h1>

            <div style="padding: 25px 25px 25px 25px;">

            <div id="system\_response" style="padding: 25px;"></div>

        </div>

    </div>

</body>

</html>

**Django:-**

wget "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-CD0251EN-SkillsNetwork/labs/m3\_django\_orm/lab2\_template.zip"

unzip lab2\_template.zip

rm lab2\_template.zip

"""

This is the emotion detector() function using watson NLP library

"""

import request

import jason

def emotion\_detector(text\_to\_analyse):

    url = 'https://sn-watson-emotion.labs.skills.network/v1/watson.runtime.nlp.v1/NlpService/EmotionPredict'

    myobj = { "raw\_document": { "text": text\_to\_analyse } }

    header = {"grpc-metadata-mm-model-id": "emotion\_aggregated-workflow\_lang\_en\_stock"}

    response = requests.post(url, json = myobj, headers=header)

    # Step 1: convert the response text into dictionary

    formatted\_response  = json.loads(response.text)

    print(formatted\_response)

    # Step 2: Extract required set of emotions and their scores

    if response.status\_code == 200:

        anger\_score = formatted\_response["emotionPredictions"][0]["emotion"]["anger"]

        disgust\_score = formatted\_response["emotionPredictions"][0]["emotion"]["disgust"]

        fear\_score = formatted\_response["emotionPredictions"][0]["emotion"]["fear"]

        joy\_score = formatted\_response["emotionPredictions"][0]["emotion"]["joy"]

        sadness\_score = formatted\_response["emotionPredictions"][0]["emotion"]["sadness"]

    # Step 3: Find the dominant emotion

        response.status\_code == 400:

        emotion\_list = [anger\_score, ,disgust\_score, fear\_score, joy\_score, sadness\_score]

        dominant\_emotion\_index = emotion\_list.index(max(emotion\_list))

        emotion keys =  ["anger", "disgust", "fear", "joy", "sadness"]

        dominant\_emotion\_key = emotion\_key[dominant\_emotion\_index]

    elif response.status\_code == 400:

        anger\_score = None

        disgust\_score = None

        fear\_score = None

        joy\_score = None

        sadness\_score = None

        dominant\_emotion\_key = None

    #Step 4: Modify function to return the required output format

    result = {

        'anger': anger\_score,

        'disgust': disgust\_score,

        'fear': fear\_score,

        'joy': joy\_score,

        'sadness': sadness\_score,

        'dominant\_emotion': dominant\_emotion\_key

    }

    return result

    # User model

class User(models.Model):

    first\_name = models.CharField(null=False, max\_length=30, default='john')

    last\_name = models.CharField(null=False, max\_length=30, default='doe')

    dob = models.DateField(null=True)

    # Create a toString method for object string representation

    def \_\_str\_\_(self):

        return self.first\_name + " " + self.last\_name

    # Instructor model

class Instructor(User):

    full\_time = models.BooleanField(default=True)

    total\_learners = models.IntegerField()

    # Create a toString method for object string representation

    def \_\_str\_\_(self):

        return "First name: " + self.first\_name + ", " + \

               "Last name: " + self.last\_name + ", " + \

               "Is full time: " + str(self.full\_time) + ", " + \

               "Total Learners: " + str(self.total\_learners)

    # Course model

class Course(models.Model):

    name = models.CharField(null=False, max\_length=100, default='online course')

    description = models.CharField(max\_length=500)

    # Many-To-Many relationship with Instructor

    instructors = models.ManyToManyField(Instructor)

    # Create a toString method for object string representation

    def \_\_str\_\_(self):

        return "Name: " + self.name + "," + \

            "Description: " + self.description

    # Lesson

class Lesson(models.Model):

    title = models.CharField(max\_length=200, default="title")

    course = models.ForeignKey(Course, null=True, on\_delete=models.CASCADE)

    content = models.TextField()

    # Learner model

class Learner(User):

    STUDENT = 'student'

    DEVELOPER = 'developer'

    DATA\_SCIENTIST = 'data\_scientist'

    DATABASE\_ADMIN = 'dba'

    OCCUPATION\_CHOICES = [

        (STUDENT, 'Student'),

        (DEVELOPER, 'Developer'),

        (DATA\_SCIENTIST, 'Data Scientist'),

        (DATABASE\_ADMIN, 'Database Admin')

    ]

    # Occupation Char field with defined enumeration choices

    occupation = models.CharField(

        null=False,

        max\_length=20,

        choices=OCCUPATION\_CHOICES,

        default=STUDENT

    )

    # Social link URL field

    social\_link = models.URLField(max\_length=200)

    # Create a toString method for object string representation

    def \_\_str\_\_(self):

        return "First name: " + self.first\_name + ", " + \

                "Last name: " + self.last\_name + ", " \

                "Date of Birth: " + str(self.dob) + ", " + \

                "Occupation: " + self.occupation + ", " + \

                "Social Link: " + self.social\_link

   # Course model

class Course(models.Model):

    name = models.CharField(null=False, max\_length=100, default='online course')

    description = models.CharField(max\_length=500)

    # Many-To-Many relationship with Instructors

    instructors = models.ManyToManyField(Instructor)

    # Many-To-Many relationship with Learner via Enrollment relationship

    learners = models.ManyToManyField(Learner, through='Enrollment')

    def \_\_str\_\_(self):

        return "Name: " + self.name + "," + \

                 "Description: " + self.description

    def \_\_str\_\_(self):

        return "Name: " + self.name + "," + \

                 "Description: " + self.description

#Then generate migration scripts for app crud

- python3 manage.py makemigrations crud --> generate migration scripts

- python3 manage.py migrate --> Run the migrations.

def write\_instructors():

    # Add instructors

    # Create a user

    user\_john = User(first\_name='John', last\_name='Doe', dob=date(1962, 7, 16))

    user\_john.save()

    instructor\_john = Instructor(full\_time=True, total\_learners=30050)

    # Update the user reference of instructor\_john to be user\_john

    instructor\_john.user = user\_john

    instructor\_john.save()

    instructor\_yan = Instructor(first\_name='Yan', last\_name='Luo', dob=date(1962, 7, 16), full\_time=True, total\_learners=30050)

    instructor\_yan.save()

    instructor\_joy = Instructor(first\_name='Joy', last\_name='Li', dob=date(1992, 1, 2), full\_time=False, total\_learners=10040)

    instructor\_joy.save()

    instructor\_peter = Instructor(first\_name='Peter', last\_name='Chen', dob=date(1982, 5, 2), full\_time=True, total\_learners=2002)

    instructor\_peter.save()

    print("Instructor objects all saved... ")

def write\_courses():

    # Add Courses

    course\_cloud\_app = Course(name="Cloud Application Development with Database",

                              description="Develop and deploy application on cloud")

    course\_cloud\_app.save()

    course\_python = Course(name="Introduction to Python",

                           description="Learn core concepts of Python and obtain hands-on "

                                       "experience via a capstone project")

    course\_python.save()

    print("Course objects all saved... ")

def write\_lessons():

    # Add lessons

    lession1 = Lesson(title='Lesson 1', content="Object-relational mapping project")

    lession1.save()

    lession2 = Lesson(title='Lesson 2', content="Django full stack project")

    lession2.save()

    print("Lesson objects all saved... ")

#saved as write.py

def clean\_data():

    # Delete all data to start from fresh

    Enrollment.objects.all().delete()

    User.objects.all().delete()

    Learner.objects.all().delete()

    Instructor.objects.all().delete()

    Course.objects.all().delete()

    Lesson.objects.all().delete()

python3 write.py

#read\_courses

#Find all courses

courses = Course.objects.all()

print(courses)

python3 read\_courses.py

#Use of get, filter and exclude /  read\_instructors.py

instructor\_yan = Instructor.objects.get(first\_name="Yan")

print("1. Find a single instructor with first name `Yan`")

print(instructor\_yan)

print("\n")

# Note that there is no instructor with first name `Andy`

# So the manager will throw an exception

try:

    instructor\_andy = Instructor.objects.get(first\_name="Andy")

except Instructor.DoesNotExist:

    print("2. Try to find a non-existing instructor with first name `Andy`")

    print("Instructor Andy doesn't exist")

print("\n")

part\_time\_instructors = Instructor.objects.filter(full\_time=False)

print("3. Find all part time instructors: ")

print(part\_time\_instructors)

print("\n")

full\_time\_instructors = Instructor.objects.exclude(full\_time=False).filter(total\_learners\_\_gt=30000).\

        filter(first\_name\_\_startswith='Y')

print("4. Find all full time instructors with First Name starts with `Y` and learners count greater than 30000")

print(full\_time\_instructors)

print("\n")

full\_time\_instructors = Instructor.objects.filter(full\_time=True, total\_learners\_\_gt=30000,

                                                      first\_name\_\_startswith='Y')

print("5. Find all full time instructors with First Name starts with `Y` and learners count greater than 30000")

print(full\_time\_instructors)

python3 read\_instructors.

#Few more variations

 # Find students with last name "Smith"

learners\_smith = Learner.objects.filter(last\_name="Smith")

print("1. Find learners with last name `Smith`:")

print(learners\_smith)

print("\n")

# Order by dob descending, and select the first two objects

learners = Learner.objects.order\_by('-dob')[0:2]

print("2. Find top two youngest learners:")

print(learners)

# Cheat sheet: ORM: Bridging the Gap Between the Real World and Relational Model

|  |  |  |
| --- | --- | --- |
| **Package/Method** | **Description** | **Code Example** |
| **django.db.models.Model** | Define a model. | from django.db import models   class MyModel(models.Model):   field1 = models.CharField(max\_length=100)   field2 = models.IntegerField() |
|  |
|  |
|  |
|  |
|  |
|  |
| **makemigrations/migrate** | Create database tables based on models. | python manage.py makemigrations  python manage.py migrate |  |
|  |
|  |
| **all()** | Retrieves all instances of the ‘MyModel’ model from the database. | MyModel.objects.all() |  |
| **filter()** | Filter objects using conditions. | MyModel.objects.filter(field1="value")  MyModel.objects.filter(field2\_\_gt=5) |  |
|  |
|  |
| **get()** | Retrieves a single instance of the ‘MyModel’ model from the database where the value of ‘field1’ is "value". | MyModel.objects.get(field1="value") |  |
| **obj = MyModel(field1="value", field2=5)  obj.save()** | Creates a new instance of the ‘MyModel’ model with the values "value" for ‘field1’ and 5 for ‘field2’, and then saves the instance to the database. | obj = MyModel(field1="value", field2=5)  obj.save() |  |
|  |
|  |
| **obj.field1 = "new value" obj.save()** | Updates the value of ‘field1’ for the ‘obj’ instance to "new value" and saves the changes to the database. | obj.field1 = "new value"  obj.save() |  |
|  |
|  |
| **delete()** | Deletes an object. | obj.delete() |  |
| **obj.related\_model** | Retrieves the related model associated with the ‘obj’ instance. Access related objects (Foreign Key or OneToOneField) | obj.related\_model |  |
| **obj.model\_set.all()** | Fetches all related objects associated with the ‘obj’ instance. Access related objects in reverse (ForeignKey) | obj.model\_set.all() |  |
| **field** | Performs a filtering operation on the ‘MyModel’ model instances based on a related model's field value. | MyModel.objects.filter(related\_model\_\_field="value") |  |
| **exact** | Retrieves instances of the ‘MyModel’ model from the database where the value of the ‘field’ attribute is exactly equal to "value". | MyModel.objects.filter(field\_\_exact="value") |  |
| **iexact** | The iexact lookup is case-insensitive, meaning it will match values regardless of whether they are uppercase or lowercase and provide a case-insensitive match. | MyModel.objects.filter(field\_\_iexact="value") |  |
| **contains** | Checks if the value is a substring within the field. | MyModel.objects.filter(field\_\_contains="value") |  |
| **startswith** | Determines whether a string begins with the characters of a specified string. | MyModel.objects.filter(field\_\_startswith="value") |  |
| **endswith** | Determines whether a string ends with the specified suffix. | MyModel.objects.filter(field\_\_endswith="value") |  |
| **in** | Checks if the value of the field is present in the given list of values. | MyModel.objects.filter(field\_\_in=["value1", "value2"]) |  |
| **gt** | Checks if the value of ‘field’ is numerically greater than 5. | MyModel.objects.filter(field\_\_gt=5) |  |
| **lt** | Checks if the value of ‘field’ is numerically less than 10. | MyModel.objects.filter(field\_\_lt=10) |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **JavaScript (JS)** | **Python** | **React** |
| **Variable Declaration** | let x = 10; | x = 10 | Follows JS syntax for variables, e.g., const [state, setState] = useState(initialValue); |
| const y = 20; | y = 20 (dynamically typed) |
| **Conditionals** | if (x > 10) { ... }   else { ... } | if x > 10: …  else: ... | Same as JavaScript for JSX: {condition ? <ComponentA /> : <ComponentB />} |
| **Loops** | for (let i = 0; i < 10; i++)   { ... } | for i in range(10): ... | Loops handled via JavaScript (e.g., array.map() in JSX): {list.map((item) => <li>{item}</li>)} |
| array.forEach((item) => { ... }) | for item in iterable: |
| **Functions** | function greet(name)   { return 'Hello ' + name; } | def greet(name):  return f"Hello {name}" | React uses JS functions for components: function MyComponent() { return <div>Hello</div>; } |
| const greet = (name) => ... |
| **Data Structures (List)** | let arr = [1, 2, 3]; | arr = [1, 2, 3] | Managed via state or props, e.g., const list = [1, 2, 3]; <ul>{list.map(item => <li>{item}</li>)}</ul> |
| **Data Structures (Dict)** | let obj = { key: 'value' }; | obj = { 'key': 'value' } | React uses JS objects for props/state: const obj = { key: 'value' }; <Component {...obj} /> |
| **Class Syntax** | class MyClass   { constructor() { ... } } | class MyClass:   def \_\_init\_\_(self): ... | React supports class-based components: class MyComponent extends React.Component { render() { return <div />; } } |
| **Modules/Imports** | import { func } from './module.js'; | from module import func | Same as JS: import Component from './Component'; |
| **Error Handling** | try { ... }  catch (err) { ... } | try: ...  Except Exception as e: ... | Same as JavaScript for React error boundaries. React also has specific lifecycle methods for error handling. |
| **Printing Output** | console.log('Hello'); | print('Hello') | React handles output in JSX: <div>Hello</div> |
| **Function Declaration** | function greet(name)   { return 'Hello ' + name; } | def greet(name):   return f"Hello {name}" | React uses JS functions or functional components: const MyComponent = () => <div>Hello</div>; |
| const greet = (name) => 'Hello ' + name; --> Arrow function |
| **Object-Oriented Programming** | class Animal {   constructor(name)   { this.name = name; }   speak()   { console.log(this.name); }  } | class Animal:   def \_\_init\_\_(self, name):   self.name = name   def speak(self):   print(self.name) | React supports class-based components: class MyComponent extends React.Component { render() { return <div />; } } |
| **Anonymous Functions** | const square = (x) => x \* x; | square = lambda x: x \* x | React heavily uses anonymous functions in JSX: {list.map((item) => <li key={item}>{item}</li>)} |
| **Handling JSON** | let obj = JSON.parse(jsonString); | import json obj = json.loads(json\_string)   json\_string = json.dumps(obj) | Same as JavaScript: JSON parsing is typically handled outside JSX. |
| let jsonString = JSON.stringify(obj); |
| **Error Handling** | try { ... }  catch (err) { ... }  finally { ... } | try: ...  except Exception as e: ...  finally: ... | Same as JavaScript for JS syntax, but React has Error Boundaries to catch errors in the component tree. |
| **Type Checking** | Dynamic typing. Can use TypeScript for static typing: let x: number = 5; | Dynamic typing. Use mypy for static typing: x: int = 5 | Prop validation with PropTypes: MyComponent.propTypes = { name: PropTypes.string }; |
| **String Interpolation** | let greeting = `Hello, ${name}!`; | greeting = f"Hello, {name}!" | React uses JavaScript for interpolation within JSX: <div>Hello, {name}!</div> |
| **Libraries/Packages** | Managed with npm or yarn: npm install package-name | Managed with pip: pip install package-name | React is itself a library. Additional libraries like React Router or Redux are also managed via npm or yarn. |
| **List Comprehension** | let squares = arr.map((x) => x \* x); | squares = [x \* x for x in arr] | React uses JavaScript methods like map within JSX: {list.map((item) => <li>{item}</li>)} |
| **Asynchronous Programming** | async function fetchData()  {  const response = await fetch(url); return await response.json();  } | import asyncio  async def fetch\_data():   response = await fetch(url)   return await response.json() | React uses JS async features for data fetching. Often paired with hooks like useEffect or libraries like axios for async calls. |
| **Default Parameters** | function greet(name = 'Guest')  { console.log(`Hello, ${name}`); } | def greet(name='Guest'):   print(f"Hello, {name}") | React functional components support default props: MyComponent.defaultProps = { name: 'Guest' }; |
| **Dependency Management** | Uses package.json and node\_modules for dependencies. | Uses requirements.txt or pyproject.toml for dependencies. | Uses JavaScript ecosystem (npm/yarn) for dependency management. |
| **Multithreading** | Web Workers for threads: const worker = new Worker('worker.js'); | threading module: from threading import Thread t = Thread(target=func) t.start() | Multithreading is handled outside React. React itself is single-threaded and uses virtual DOM for UI updates. |
| **Framework Integration** | Used in conjunction with libraries like React, Angular, Vue.js, etc. | Works with Django, Flask, FastAPI for backend development. | React integrates seamlessly with backend APIs (often Python-based) to fetch and display data. |
| **Class Definition** | javascript   class Animal {  constructor(name)  { this.name = name; }  speak()   { console.log(this.name); }  } | python class Animal:   def \_\_init\_\_(self, name):   self.name = name  def speak(self):   print(self.name) | javascript class Animal extends React.Component { render() { return <div>{this.props.name}</div>; } } |
| **Inheritance** | javascript   class Dog extends Animal  {  bark()  { console.log('Woof!'); }  } | python   class Dog(Animal):  def bark(self):   print("Woof!") | javascript class DogComponent extends AnimalComponent { render() { return <div>Bark!</div>; } } |
| **Encapsulation** | javascript  class Person { #age; constructor(name, age) { this.name = name; this.#age = age; } } | python   class Person:   def \_\_init\_\_(self, name, age):   self.\_\_age = age   self.name = name | React doesn't support private fields but uses state/props for encapsulation. |
| **Polymorphism** | // Shape class with a draw method class Shape {  draw() {  console.log('Drawing...');  } }  // Circle class that extends Shape and overrides the draw method class Circle extends Shape {  draw() {  console.log('Circle');  } }  // Example of usage: const myShape = new Shape(); // Create a Shape instance myShape.draw(); // Output: "Drawing..."  const myCircle = new Circle(); // Create a Circle instance myCircle.draw(); // Output: "Circle" | # Shape class with a draw method class Shape:  def draw(self):  print("Drawing...")  # Circle class that inherits from Shape and overrides the draw method class Circle(Shape):  def draw(self):  print("Circle")  # Example of usage: shape = Shape() # Create a Shape instance shape.draw() # Output: "Drawing..."  circle = Circle() # Create a Circle instance circle.draw() # Output: "Circle" | javascript class BaseComponent extends React.Component { render() { return <div>Base</div>; } } class ChildComponent extends BaseComponent { render() { return <div>Child</div>; } } |
| **Abstraction** | Using abstract classes via inheritance: javascript class Animal { speak() { throw "Not Implemented"; } } | Achieved via ABC module: python from abc import ABC, abstractmethod class Animal(ABC): @abstractmethod def speak(self): pass | No direct abstraction but enforced via prop validation and architectural design. |
| **Method Overriding** | javascript class Parent { greet() { console.log("Hello!"); } } class Child extends Parent { greet() { console.log("Hi!"); } } | python class Parent: def greet(self): print("Hello!") class Child(Parent): def greet(self): print("Hi!") | Class-based components override lifecycle methods like render, componentDidMount, etc. |
| **Static Methods** | javascript class MathUtil { static square(x) { return x \* x; } } | python class MathUtil: @staticmethod def square(x): return x \* x | React components can define static methods for utility, e.g., static helper() { return 'value'; } |
| **Multiple Inheritance** | Not supported directly. Achieved via mixins: javascript Object.assign(classA.prototype, classB.prototype); | Supported directly: python class Child(Parent1, Parent2): pass | Not directly applicable in React, but composition is preferred over inheritance for combining functionalities. |
| **Properties (Getters/Setters)** | javascript class Personclass Person {  constructor(name) {  this.\_name = name; // Private-like convention with underscore  }   // Getter for 'name'  get name() {  return this.\_name;  }   // Setter for 'name'  set name(newName) {  this.\_name = newName;  } }  // Example usage: const person = new Person('Alice');  // Using the getter console.log(person.name); // Output: Alice  // Using the setter person.name = 'Bob'; console.log(person.name); // Output: Bob { constructor(name) { this.\_name = name; } get name() { return this.\_name; } set name(newName) { this.\_name = newName; } } | class Person:  def \_\_init\_\_(self, name):  self.\_name = name    @property  def name(self):  return self.\_name    @name.setter  def name(self, value):  self.\_name = value | React doesn’t support getters/setters for props, but state and event handlers act as mutators. |
| **Aggregation (Has-A Relationship):** | // Engine class with a start method class Engine {  start() {  console.log("Engine started");  } }  // Car class containing an instance of Engine class Car {  constructor() {  this.engine = new Engine(); // Aggregation: Car has an Engine instance  }   drive() {  this.engine.start(); // Car uses the Engine's start method  console.log("Car is moving");  } }  // Example of usage: const myCar = new Car(); // Create a Car instance myCar.drive(); // Start the engine and move the car | # Engine class with a start method class Engine:  def start(self):  print("Engine started")  # Car class that contains an Engine instance class Car:  def \_\_init\_\_(self):  self.engine = Engine() # Aggregation: Car has an Engine instance   def drive(self):  self.engine.start() # Car uses the Engine's start method  print("Car is moving")  # Example of usage: my\_car = Car() # Create a Car instance my\_car.drive() # Start the engine and move the car | Key points: The Car class "has" an Engine (aggregation). The Engine object can exist independently of the Car, meaning if the Car object is destroyed, the Engine can still be used elsewhere. |
| **Constructor Chaining:** | class Parent { constructor(name) { this.name = name; } } class Child extends Parent { constructor(name, age) { super(name); this.age = age; } } |  |  |

**Django CBV Examlpe:-**

from django.shortcuts import render

from django.http import HttpResponse, HttpResponseRedirect

from django.shortcuts import get\_object\_or\_404, render, redirect

from .models import Course, Lesson, Enrollment

from django.urls import reverse

from django.views import generic, View

from django.http import Http404

# Create your class based views here.

# Note that we are subclassing CourseListView from base View class

class CourseListView(View):

    # Handles get request

    def get(self, request):

        context = {}

        course\_list = Course.objects.order\_by('-total\_enrollment')[:10]

        context['course\_list'] = course\_list

        return render(request, 'onlinecourse/course\_list.html', context)

class EnrollView(View):

    # Handles post request

    def post(self, request, \*args, \*\*kwargs):

        course\_id = kwargs.get('pk')

        course = get\_object\_or\_404(Course, pk=course\_id)

        # Increase total enrollment by 1

        course.total\_enrollment += 1

        course.save()

        return HttpResponseRedirect(reverse(viewname='onlinecourse:course\_details', args=(course.id,)))

# Function based views

# Function-based course list view

# def popular\_course\_list(request):

#    context = {}

#    if request.method == 'GET':

#        course\_list = Course.objects.order\_by('-total\_enrollment')[:10]

#        context['course\_list'] = course\_list

#        return render(request, 'onlinecourse/course\_list\_no\_css.html', context)

# Function-based course\_details view

# def course\_details(request, course\_id):

#    context = {}

#    if request.method == 'GET':

#        try:

#            course = Course.objects.get(pk=course\_id)

#            context['course'] = course

#            return render(request, 'onlinecourse/course\_detail.html', context)

#        except Course.DoesNotExist:

#            raise Http404("No course matches the given id.")

# Function-based enroll view

# def enroll(request, course\_id):

#    if request.method == 'POST':

#       course = get\_object\_or\_404(Course, pk=course\_id)

#       # Create an enrollment

#       course.total\_enrollment += 1

#       course.save()

#       return HttpResponseRedirect(reverse(viewname='onlinecourse:course\_details', args=(course.id,)))

**Django : Generic List View ..**

# views.py

from django.views.generic import ListView

from .models import MyModel

class MyModelListView(ListView):

    model = MyModel  # The model associated with this view

    template\_name = 'myapp/mymodel\_list.html'  # Template to render the view

    context\_object\_name = 'objects'  # Context variable name for the queryset

    # Optional: Customize the queryset

    def get\_queryset(self):

        # Add custom filtering or ordering logic here

        return MyModel.objects.filter(active=True).order\_by('-created\_at')

# Template Example

<!-- myapp/templates/myapp/mymodel\_list.html -->

<h1>My Model List</h1>

<ul>

  {% for object in objects %}

    <li>{{ object.name }}</li>

  {% endfor %}

</ul>

**jsfhbvjhsfhjfbb**