How to create a class in python

Creating a class in Python involves defining a class using the `class` keyword, adding methods (functions) and attributes (variables) to the class, and then creating instances of the class. Here's a basic example:

```python
class Dog:
    # Class attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        # Instance attributes
        self.name = name
        self.age = age

    # Instance method
    def describe(self):
        return f"{self.name} is {self.age} years old."

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}."
```

Explanation:

1. **Class Definition:**

```python
class Dog:
```

This line defines a new class named `Dog`.

2. **Class Attribute:**

```python
species = "Canis familiaris"
```

This is a class attribute shared by all instances of the `Dog` class.

3. **Constructor Method (`__init__`):**

```python
def __init__(self, name, age):
    self.name = name
    self.age = age
```

The `__init__` method is a special method that gets called when a new instance of the class is created. It initializes the instance attributes `name` and `age`.

4. **Instance Method (`describe`):**

```python
def describe(self):
    return f"{self.name} is {self.age} years old."
```

This is an instance method that describes the dog.

5. **Another Instance Method (`speak`):**

```python
def speak(self, sound):
    return f"{self.name} says {sound}."
```

This method allows the dog to "speak" a given sound.

Creating an Instance:

To create an instance of the `Dog` class, you would do the following:

```python
my_dog = Dog("Buddy", 3)

print(my_dog.describe())  # Output: Buddy is 3 years old.
print(my_dog.speak("Woof"))  # Output: Buddy says Woof.
```

Full Example:

Here is the complete code:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def describe(self):
        return f"{self.name} is {self.age} years old."

    def speak(self, sound):
        return f"{self.name} says {sound}."

# Create an instance of the Dog class
my_dog = Dog("Buddy", 3)

# Call the instance methods
print(my_dog.describe())  # Output: Buddy is 3 years old.
print(my_dog.speak("Woof"))  # Output: Buddy says Woof.
```

This example demonstrates how to define a class, create an instance of it, and use its methods.

more info

Certainly! Let's delve deeper into the concepts of classes and objects in Python, including additional features such as inheritance, class methods, static methods, and special methods.

More on Class Attributes and Instance Attributes

1. **Class Attributes:**
   - Class attributes are attributes that are shared across all instances of a class. They are defined outside the `__init__` method.
   - Example:

```python
class Dog:
    species = "Canis familiaris"  # Class attribute
```

2. **Instance Attributes:**
   - Instance attributes are unique to each instance of the class. They are defined within the `__init__` method.
   - Example:

```python
def __init__(self, name, age):
    self.name = name  # Instance attribute
    self.age = age  # Instance attribute
```

Methods in Classes

1. **Instance Methods:**
   - Instance methods are functions defined within a class and are used to manipulate the data of the instance. They take `self` as the first parameter.
   - Example:

```python
def describe(self):
    return f"{self.name} is {self.age} years old."
```

2. **Class Methods:**
   - Class methods are used to access or modify class state. They take `cls` as the first parameter, which represents the class itself.
   - Use the `@classmethod` decorator.
   - Example:

```python
@classmethod
def change_species(cls, new_species):
    cls.species = new_species
```

3. **Static Methods:**
   - Static methods do not operate on an instance or class, they are utility functions. They do not take `self` or `cls` as the first parameter.
   - Use the `@staticmethod` decorator.
   - Example:

```python
@staticmethod
def is_dog(name):
    return name.lower() in ["buddy", "max", "bella"]
```

Inheritance

Inheritance allows one class (child class) to inherit attributes and methods from another class (parent class).

Example:

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow"

my_dog = Dog("Buddy")
print(my_dog.speak())  # Output: Buddy says Woof

my_cat = Cat("Whiskers")
print(my_cat.speak())  # Output: Whiskers says Meow
```

Special Methods

Special methods (also known as magic methods or dunder methods) allow us to define the behavior of instances of the class with respect to built-in functions and operators.

Example:

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(2, 3)
v2 = Vector(5, 7)
v3 = v1 + v2
print(v3)  # Output: Vector(7, 10)
```

Full Example with All Concepts

Here's a comprehensive example that incorporates class attributes, instance attributes, instance methods, class methods, static methods, inheritance, and special methods:

```python
class Animal:
    species = "Animalia"  # Class attribute

    def __init__(self, name):
        self.name = name  # Instance attribute

    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")

    @classmethod
    def change_species(cls, new_species):
        cls.species = new_species

    @staticmethod
    def is_animal(name):
        return isinstance(name, str)

class Dog(Animal):
    def __init__(self, name, age):
        super().__init__(name)
        self.age = age

    def speak(self):
        return f"{self.name} says Woof"

    def __repr__(self):
        return f"Dog(name={self.name}, age={self.age})"

# Create instances
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Call instance methods
print(dog1.speak())  # Output: Buddy says Woof

# Call class methods
Dog.change_species("Canis lupus")
print(Dog.species)  # Output: Canis lupus

# Call static methods
print(Dog.is_animal("Buddy"))  # Output: True

# Special methods
print(dog1)  # Output: Dog(name=Buddy, age=3)
```

This example provides a comprehensive view of defining and using classes in Python, covering basic to advanced features.

sql server connection + data insert in sql server + implemented through Python class

To connect to a SQL Server database and perform data insertion using a Python class, you'll need to use the `pyodbc` library, which allows Python to connect to ODBC databases. Below is a step-by-step guide on how to set this up, including creating a Python class for the connection and data insertion.

Step 1: Install `pyodbc`

First, you need to install the `pyodbc` library. You can do this using pip:

```sh
pip install pyodbc
```

Step 2: Define the Python Class

Here's how you can define a Python class to handle the connection and data insertion:

```python
import pyodbc

class SQLServerDatabase:
    def __init__(self, server, database, username, password):
        self.server = server
        self.database = database
        self.username = username
        self.password = password
        self.connection = None

    def connect(self):
        try:
            self.connection = pyodbc.connect(
                f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={self.server};DATABASE={
            )
            print("Connection successful")
        except Exception as e:
            print("Error connecting to database:", e)

    def insert_data(self, table, data):
        cursor = self.connection.cursor()
        placeholders = ', '.join(['?'] * len(data))
        columns = ', '.join(data.keys())
        sql = f"INSERT INTO {table} ({columns}) VALUES ({placeholders})"
        try:
            cursor.execute(sql, list(data.values()))
            self.connection.commit()
            print("Data inserted successfully")
        except Exception as e:
            print("Error inserting data:", e)

    def close(self):
        if self.connection:
            self.connection.close()
            print("Connection closed")

# Example usage
if __name__ == "__main__":
    server = 'your_server'
    database = 'your_database'
    username = 'your_username'
    password = 'your_password'

    db = SQLServerDatabase(server, database, username, password)
    db.connect()

    # Insert example data
    table = 'your_table'
    data = {
        'column1': 'value1',
        'column2': 'value2',
        'column3': 'value3'
    }
    db.insert_data(table, data)

    db.close()
```

Explanation:

1. **Class Initialization:**

```python
def __init__(self, server, database, username, password):
    self.server = server
    self.database = database
    self.username = username
    self.password = password
    self.connection = None
```

The `__init__` method initializes the class with the server details and sets the connection to `None`.

2. **Connecting to the Database:**

```python
def connect(self):
    try:
        self.connection = pyodbc.connect(
            f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={self.server};DATABASE={
        )
        print("Connection successful")
    except Exception as e:
        print("Error connecting to database:", e)
```

The `connect` method establishes a connection to the SQL Server using `pyodbc`.

3. **Inserting Data:**

```python
def insert_data(self, table, data):
    cursor = self.connection.cursor()
    placeholders = ', '.join(['?'] * len(data))
    columns = ', '.join(data.keys())
    sql = f"INSERT INTO {table} ({columns}) VALUES ({placeholders})"
    try:
        cursor.execute(sql, list(data.values()))
        self.connection.commit()
        print("Data inserted successfully")
    except Exception as e:
        print("Error inserting data:", e)
```

The `insert_data` method prepares an SQL `INSERT` statement and executes it to insert data into the specified table.

4. **Closing the Connection:**

```python
def close(self):
    if self.connection:
        self.connection.close()
        print("Connection closed")
```

The `close` method closes the database connection.

Usage Example:

In the `if __name__ == "__main__":` block, you can see an example of how to use this class to connect to a database, insert data, and close the connection.

Make sure to replace `'your_server'`, `'your_database'`, `'your_username'`, `'your_password'`, and `'your_table'` with your actual SQL Server details and table information. The `data` dictionary should be updated with the appropriate column names and values you wish to insert.

more generic

Sure! Let's make the class more generic by adding features such as dynamic SQL query execution, handling different types of queries (like SELECT, UPDATE, DELETE), and better error handling. We'll also include a method to fetch data from the database.

Enhanced SQLServerDatabase Class

```python
import pyodbc

class SQLServerDatabase:
    def __init__(self, server, database, username, password):
        self.server = server
        self.database = database
        self.username = username
        self.password = password
        self.connection = None

    def connect(self):
        try:
            self.connection = pyodbc.connect(
                f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={self.server};DATABASE={
            )
            print("Connection successful")
        except Exception as e:
            print(f"Error connecting to database: {e}")

    def execute_query(self, query, params=None):
        cursor = self.connection.cursor()
        try:
            if params:
                cursor.execute(query, params)
            else:
                cursor.execute(query)
            self.connection.commit()
            print("Query executed successfully")
        except Exception as e:
            print(f"Error executing query: {e}")

    def fetch_data(self, query, params=None):
        cursor = self.connection.cursor()
        try:
            if params:
                cursor.execute(query, params)
            else:
                cursor.execute(query)
            results = cursor.fetchall()
            columns = [column[0] for column in cursor.description]
            data = [dict(zip(columns, row)) for row in results]
            return data
        except Exception as e:
            print(f"Error fetching data: {e}")
            return None

    def close(self):
        if self.connection:
            self.connection.close()
            print("Connection closed")

# Example usage
if __name__ == "__main__":
    server = 'your_server'
    database = 'your_database'
    username = 'your_username'
    password = 'your_password'

    db = SQLServerDatabase(server, database, username, password)
    db.connect()

    # Insert example data
    insert_query = "INSERT INTO your_table (column1, column2, column3) VALUES (?, ?, ?)"
    insert_data = ('value1', 'value2', 'value3')
    db.execute_query(insert_query, insert_data)

    # Fetch example data
    select_query = "SELECT * FROM your_table"
    data = db.fetch_data(select_query)
    for row in data:
        print(row)

    db.close()
```

Explanation:

1. **Class Initialization:**

```python
def __init__(self, server, database, username, password):
    self.server = server
    self.database = database
    self.username = username
    self.password = password
    self.connection = None
```

Initializes the class with the server details and sets the connection to `None`.

2. **Connecting to the Database:**

```python
def connect(self):
    try:
        self.connection = pyodbc.connect(
            f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={self.server};DATABASE={
        )
        print("Connection successful")
    except Exception as e:
        print(f"Error connecting to database: {e}")
```

Establishes a connection to the SQL Server using `pyodbc`.

3. **Executing Queries:**

```python
def execute_query(self, query, params=None):
    cursor = self.connection.cursor()
    try:
        if params:
            cursor.execute(query, params)
        else:
            cursor.execute(query)
        self.connection.commit()
        print("Query executed successfully")
    except Exception as e:
        print(f"Error executing query: {e}")
```

Executes a provided SQL query with optional parameters.

4. **Fetching Data:**

```python
def fetch_data(self, query, params=None):
    cursor = self.connection.cursor()
    try:
        if params:
            cursor.execute(query, params)
        else:
            cursor.execute(query)
        results = cursor.fetchall()
        columns = [column[0] for column in cursor.description]
        data = [dict(zip(columns, row)) for row in results]
        return data
    except Exception as e:
        print(f"Error fetching data: {e}")
        return None
```

Fetches data from the database and returns it as a list of dictionaries, where each dictionary represents a row with column names as keys.

5. **Closing the Connection:**

```python
def close(self):
    if self.connection:
        self.connection.close()
        print("Connection closed")
```

Closes the database connection.

Usage Example:

In the `if __name__ == "__main__":` block, the class is instantiated, and the connection is established. Then, it inserts data into a table, fetches data from the table, and finally closes the connection.

Replace `'your_server'`, `'your_database'`, `'your_username'`, `'your_password'`, and `'your_table'` with your actual SQL Server details and table information. The `insert_data` tuple should be updated with the appropriate values you wish to insert.

more example

Sure! Let's expand the example to include more operations such as updating data, deleting data, and handling transactions. We'll also include detailed comments for better understanding.

Enhanced SQLServerDatabase Class with More Examples

```python
import pyodbc

class SQLServerDatabase:
    def __init__(self, server, database, username, password):
        self.server = server
        self.database = database
        self.username = username
        self.password = password
        self.connection = None

    def connect(self):
        try:
            self.connection = pyodbc.connect(
                f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={self.server};DATABASE={
            )
            print("Connection successful")
        except Exception as e:
            print(f"Error connecting to database: {e}")

    def execute_query(self, query, params=None):
        cursor = self.connection.cursor()
        try:
            if params:
                cursor.execute(query, params)
            else:
                cursor.execute(query)
            self.connection.commit()
            print("Query executed successfully")
        except Exception as e:
            print(f"Error executing query: {e}")

    def fetch_data(self, query, params=None):
        cursor = self.connection.cursor()
        try:
            if params:
                cursor.execute(query, params)
            else:
                cursor.execute(query)
            results = cursor.fetchall()
            columns = [column[0] for column in cursor.description]
            data = [dict(zip(columns, row)) for row in results]
            return data
        except Exception as e:
            print(f"Error fetching data: {e}")
            return None

    def close(self):
        if self.connection:
            self.connection.close()
            print("Connection closed")

# Example usage
if __name__ == "__main__":
    server = 'your_server'
    database = 'your_database'
    username = 'your_username'
    password = 'your_password'

    db = SQLServerDatabase(server, database, username, password)
    db.connect()

    # Insert example data
    insert_query = "INSERT INTO your_table (column1, column2, column3) VALUES (?, ?, ?)"
    insert_data = ('value1', 'value2', 'value3')
    db.execute_query(insert_query, insert_data)

    # Fetch example data
    select_query = "SELECT * FROM your_table"
    data = db.fetch_data(select_query)
    for row in data:
        print(row)

    # Update example data
    update_query = "UPDATE your_table SET column2 = ? WHERE column1 = ?"
    update_data = ('new_value2', 'value1')
    db.execute_query(update_query, update_data)

    # Delete example data
    delete_query = "DELETE FROM your_table WHERE column1 = ?"
    delete_data = ('value1',)
    db.execute_query(delete_query, delete_data)

    db.close()
```

Explanation:

1. **Insert Data:**

```python
insert_query = "INSERT INTO your_table (column1, column2, column3) VALUES (?, ?, ?)"
insert_data = ('value1', 'value2', 'value3')
db.execute_query(insert_query, insert_data)
```

This code inserts a new row into `your_table` with the values `'value1'`, `'value2'`, and `'value3'`.

2. **Fetch Data:**

```python
select_query = "SELECT * FROM your_table"
data = db.fetch_data(select_query)
for row in data:
    print(row)
```

This code retrieves all rows from `your_table` and prints them.

3. **Update Data:**

```python
update_query = "UPDATE your_table SET column2 = ? WHERE column1 = ?"
update_data = ('new_value2', 'value1')
db.execute_query(update_query, update_data)
```

This code updates `column2` to `'new_value2'` for the row where `column1` is `'value1'`.

4. **Delete Data:**

```python
delete_query = "DELETE FROM your_table WHERE column1 = ?"
delete_data = ('value1',)
db.execute_query(delete_query, delete_data)
```

This code deletes the row from `your_table` where `column1` is `'value1'`.

Using Transactions

For more advanced usage, let's see how to handle transactions. This can be useful when you need to ensure multiple operations either all succeed or all fail (e.g., in the case of a bank transfer).

```python
class SQLServerDatabase:
    def __init__(self, server, database, username, password):
        self.server = server
        self.database = database
        self.username = username
        self.password = password
        self.connection = None

    def connect(self):
        try:
            self.connection = pyodbc.connect(
                f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={self.server};DATABASE={
            )
            print("Connection successful")
        except Exception as e:
            print(f"Error connecting to database: {e}")

    def execute_query(self, query, params=None):
        cursor = self.connection.cursor()
        try:
            if params:
                cursor.execute(query, params)
            else:
                cursor.execute(query)
            self.connection.commit()
            print("Query executed successfully")
        except Exception as e:
            print(f"Error executing query: {e}")

    def fetch_data(self, query, params=None):
        cursor = self.connection.cursor()
        try:
            if params:
                cursor.execute(query, params)
            else:
                cursor.execute(query)
            results = cursor.fetchall()
            columns = [column[0] for column in cursor.description]
            data = [dict(zip(columns, row)) for row in results]
            return data
        except Exception as e:
            print(f"Error fetching data: {e}")
            return None

    def begin_transaction(self):
        self.connection.autocommit = False

    def commit_transaction(self):
        self.connection.commit()
        self.connection.autocommit = True

    def rollback_transaction(self):
        self.connection.rollback()
        self.connection.autocommit = True

    def close(self):
        if self.connection:
            self.connection.close()
            print("Connection closed")

# Example usage
if __name__ == "__main__":
    server = 'your_server'
    database = 'your_database'
    username = 'your_username'
    password = 'your_password'

    db = SQLServerDatabase(server, database, username, password)
    db.connect()

    # Start a transaction
    db.begin_transaction()
    try:
        # Insert example data
        insert_query = "INSERT INTO your_table (column1, column2, column3) VALUES (?, ?, ?
        insert_data = ('value1', 'value2', 'value3')
        db.execute_query(insert_query, insert_data)

        # Update example data
        update_query = "UPDATE your_table SET column2 = ? WHERE column1 = ?"
        update_data = ('new_value2', 'value1')
        db.execute_query(update_query, update_data)

        # Commit the transaction
        db.commit_transaction()
        print("Transaction committed successfully")
    except Exception as e:
        print(f"Error in transaction: {e}")
        # Rollback the transaction in case of error
        db.rollback_transaction()
        print("Transaction rolled back")

    db.close()
```

Explanation of Transaction Handling:

1. **Begin Transaction:**

```python
def begin_transaction(self):
    self.connection.autocommit = False
```

This method disables autocommit, starting a transaction.

2. **Commit Transaction:**

```python
def commit_transaction(self):
    self.connection.commit()
    self.connection.autocommit = True
```

This method commits the transaction and re-enables autocommit.

3. **Rollback Transaction:**

```python
def rollback_transaction(self):
    self.connection.rollback()
    self.connection.autocommit = True
```

This method rolls back the transaction and re-enables autocommit.

4. **Transaction Usage Example:**

In the example, the `begin_transaction` method is called to start a transaction. The `execute_query` method is used to insert and update data. If any error occurs, the transaction is rolled back using the `rollback_transaction` method. If all operations succeed, the transaction is committed using the `commit_transaction` method.

Replace `'your_server'`, `'your_database'`, `'your_username'`, `'your_password'`, and `'your_table'` with your actual SQL Server details and table information. The `insert_data` and `update_data` tuples should be updated with the appropriate values you wish to insert and update.

convert dynamic json data to dataframe in class of python

To convert dynamic JSON data to a DataFrame within a Python class, you can use the `pandas` library, which provides powerful data manipulation and analysis capabilities. Below is a Python class that includes methods to connect to a SQL Server database, execute queries, fetch data, and convert JSON data to a DataFrame.

Step 1: Install `pandas`

First, ensure you have the `pandas` library installed. You can install it using pip if you haven't already:

```sh
pip install pandas
```

Step 2: Define the Python Class

Here's how you can define a Python class to handle the conversion of dynamic JSON data to a DataFrame, in addition to database operations:

```python
import pyodbc
import pandas as pd
import json

class SQLServerDatabase:
    def __init__(self, server, database, username, password):
        self.server = server
        self.database = database
        self.username = username
        self.password = password
        self.connection = None

    def connect(self):
        try:
            self.connection = pyodbc.connect(
                f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={self.server};DATABASE={
            )
            print("Connection successful")
        except Exception as e:
            print(f"Error connecting to database: {e}")

    def execute_query(self, query, params=None):
        cursor = self.connection.cursor()
        try:
            if params:
                cursor.execute(query, params)
            else:
                cursor.execute(query)
            self.connection.commit()
            print("Query executed successfully")
        except Exception as e:
            print(f"Error executing query: {e}")

    def fetch_data(self, query, params=None):
        cursor = self.connection.cursor()
        try:
            if params:
                cursor.execute(query, params)
            else:
                cursor.execute(query)
            results = cursor.fetchall()
            columns = [column[0] for column in cursor.description]
            data = [dict(zip(columns, row)) for row in results]
            return data
        except Exception as e:
            print(f"Error fetching data: {e}")
            return None

    def json_to_dataframe(self, json_data):
        try:
            data_dict = json.loads(json_data)
            df = pd.DataFrame(data_dict)
            return df
        except Exception as e:
            print(f"Error converting JSON to DataFrame: {e}")
            return None

    def close(self):
        if self.connection:
            self.connection.close()
            print("Connection closed")

# Example usage
if __name__ == "__main__":
    server = 'your_server'
    database = 'your_database'
    username = 'your_username'
    password = 'your_password'

    db = SQLServerDatabase(server, database, username, password)
    db.connect()

    # Example JSON data
    json_data = '''
    [
        {"column1": "value1", "column2": "value2", "column3": "value3"},
        {"column1": "value4", "column2": "value5", "column3": "value6"}
    ]
    '''
    df = db.json_to_dataframe(json_data)
    if df is not None:
        print(df)

    db.close()
```

Explanation:

1. **Class Initialization:**

```python
def __init__(self, server, database, username, password):
    self.server = server
    self.database = database
    self.username = username
    self.password = password
    self.connection = None
```

Initializes the class with the server details and sets the connection to `None`.

2. **Connecting to the Database:**

```python
def connect(self):
    try:
        self.connection = pyodbc.connect(
            f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={self.server};DATABASE={
        )
        print("Connection successful")
    except Exception as e:
        print(f"Error connecting to database: {e}")
```

Establishes a connection to the SQL Server using `pyodbc`.

3. **Executing Queries:**

```python
def execute_query(self, query, params=None):
    cursor = self.connection.cursor()
    try:
        if params:
            cursor.execute(query, params)
        else:
            cursor.execute(query)
        self.connection.commit()
        print("Query executed successfully")
    except Exception as e:
        print(f"Error executing query: {e}")
```

Executes a provided SQL query with optional parameters.

4. **Fetching Data:**

```python
def fetch_data(self, query, params=None):
    cursor = self.connection.cursor()
    try:
        if params:
            cursor.execute(query, params)
        else:
            cursor.execute(query)
        results = cursor.fetchall()
        columns = [column[0] for column in cursor.description]
        data = [dict(zip(columns, row)) for row in results]
        return data
    except Exception as e:
        print(f"Error fetching data: {e}")
        return None
```

Fetches data from the database and returns it as a list of dictionaries, where each dictionary represents a row with column names as keys.

5. **Converting JSON to DataFrame:**

```python
def json_to_dataframe(self, json_data):
    try:
        data_dict = json.loads(json_data)
        df = pd.DataFrame(data_dict)
        return df
    except Exception as e:
        print(f"Error converting JSON to DataFrame: {e}")
        return None
```

Converts JSON data to a pandas DataFrame. This method uses `json.loads` to parse the JSON string and `pd.DataFrame` to create the DataFrame.

6. **Closing the Connection:**

```python
def close(self):
    if self.connection:
        self.connection.close()
        print("Connection closed")
```

Closes the database connection.

Usage Example:

In the `if __name__ == "__main__":` block, the class is instantiated, and the connection is established. Then, it converts JSON data to a DataFrame and prints it. Finally, the connection is closed.

Replace `'your_server'`, `'your_database'`, `'your_username'`, and `'your_password'` with your actual SQL Server details. The example JSON data should be updated with the appropriate structure and values.