Working with Excel files in *Python*

- pandas
- openpyxl

pandas

To read or create Excel files in Python, you can use the pandas library, which provides powerful and convenient functionalities for working with tabular data, including reading and writing Excel files. Before proceeding, make sure you have installed the pandas library. If you haven't installed it yet, you can do so using the following command:

```
pip install pandas
```

Reading Excel Files

To read data from an Excel file, you can use the `pandas.read_excel()` function. The function allows you to read data from specific sheets within the Excel file.

Suppose you have an Excel file named data.xlsx with the following content in the sheet named Sheet1:

Name	Age	Occupation
John	30	Engineer
Alice	25	Teacher
Bob	28	Doctor

Reading Excel Files

The `pd.read_excel()` function reads the data from the specified sheet in the Excel file and stores it in a pandas **DataFrame**, which is a two-dimensional tabular data structure

```
import pandas as pd

try:
    df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
    print(df)
except FileNotFoundError:
    print("File not found!")
```

Creating Excel Files

To create an Excel file and write data to it, you can also use the `pandas` library. First, you need to prepare the data in a pandas **DataFrame**, and then you can use the `DataFrame.to_excel()` method to save it to an Excel file.

Suppose you have the following data that you want to write to an Excel file:

```
data_to_write = {
    'Name': ['John', 'Alice', 'Bob'],
    'Age': [30, 25, 28],
    'Occupation': ['Engineer', 'Teacher', 'Doctor']
}
```

Creating Excel Files

To create an Excel file named `new_data.xlsx` and write the data to it as follows:

```
import pandas as pd

try:
    df = pd.DataFrame(data_to_write)
    df.to_excel('new_data.xlsx', index=False)
except Exception as e:
    print(f"An error occurred: {e}")
```

pandas alternative: openpyxl

`openpyxl` is a powerful library for working with Excel files in Python. It allows you to read and modify Excel files, including creating new files and writing data to them.

To read and create Excel files using the `openpyxl` library, you'll need to install it if you haven't already. You can install `openpyxl` using the following command:

```
pip install openpyxl
```

Reading Data from Excel File

```
import openpyxl


try:
    workbook = openpyxl.load_workbook('data.xlsx')
    sheet = workbook['Sheet1']
        for row in sheet.iter_rows(min_row=2, values_only=True):
            name, age, occupation = row
            print(f"Name: {name}, Age: {age}, Occupation: {occupation}")
except FileNotFoundError:
    print("File not found!")
```

Creating Excel File and Writing Data

```
import openpyxl
data_to_write = [
    ['Name', 'Age', 'Occupation'],
    ['John', 30, 'Engineer'],
    ['Alice', 25, 'Teacher'],
    ['Bob', 28, 'Doctor']]
try:
    workbook = openpyxl.Workbook()
    sheet = workbook.active
    for row_data in data_to_write:
        sheet.append(row_data)
    workbook.save('new_data.xlsx')
except Exception as e:
    print(f"An error occurred: {e}")
```

Object-Oriented programming (*hour 11*)

- Introduction to object-oriented programming (OOP).
- Classes, objects, and attributes.
- Creating and using methods.

Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects, which are instances of classes. It aims to model real-world entities and their interactions, making the code more modular, flexible, and maintainable. OOP is based on the concepts of classes, objects, attributes, and methods.

OOP enables you to organize your code into reusable and self-contained components, promoting code reusability and making your programs easier to manage and understand. It is a powerful paradigm widely used in modern programming languages to build complex systems and applications.

Classes

A class is a blueprint or a template for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects of that class will have. It serves as a design for creating multiple instances of the same type of object.

For example, the Car class is a blueprint for creating car objects.

```
class Car:
    # Class definition
    pass
```

Objects

An object is an instance of a class. It represents a specific entity or data structure with its own unique attributes and behaviors. Each object created from a class can have different values for its attributes while sharing the same methods defined in the class.

An object is an instance of a class.

For example, it represents a specific `Car` with its own unique characteristics and capabilities.

```
# Creating objects (instances) of the Car class
car1 = Car()
car2 = Car()
```

Attributes

Attributes are the data members or variables defined within a class. They represent the characteristics or properties of the objects created from the class. Each object has its own set of attribute values.

Attributes are the properties or characteristics of an object. They describe the state of the object. In the context of a `Car`, attributes could include `color`, `make`, `model`, `year`, and `mileage`.

Attributes

```
class Car:
    def __init__(self, color, make, model, year, mileage):
        self.color = color
        self.make = make
        self.model = model
        self.year = year
        self.mileage = mileage
# Creating car objects with attributes
car1 = Car("Red", "Toyota", "Corolla", 2022, 10000)
car2 = Car("Blue", "Honda", "Civic", 2021, 8000)
# Accessing attributes of car objects
print(car1.color)    # Output: "Red“
print(car2.make)     # Output: "Honda"
```

Methods

Methods are the functions defined within a class. They represent the actions or behaviors that the objects created from the class can perform. Methods can operate on the object's attributes or perform specific actions related to the object.

Methods are the actions or behaviors that an object can perform. In the context of a `car`, methods could include `start()`, `accelerate()`, `brake()`, and `get_mileage()`.

Creating and Using Methods

```
class Car:
    def __init__(self, color, make, model, year, mileage):
        self.color = color
        self.make = make
        self.model = model
        self.year = year
        self.mileage = mileage
    def start(self):
        return "The car engine is running.“
    def accelerate(self):
        return "The car is accelerating.“
    def brake(self):
        return "The car is braking.“
    def get_mileage(self):
        return f"The car has {self.mileage} miles.“
```

Creating and Using Methods

```
# Creating car objects
car1 = Car("Red", "Toyota", "Corolla", 2022, 10000)

# Calling methods of the car object
print(car1.start())            # Output: "The car engine is running."
print(car1.accelerate())       # Output: "The car is accelerating."
print(car1.brake())            # Output: "The car is braking."
print(car1.get_mileage())      # Output: "The car has 10000 miles."
```

Creating and Using Methods

Creating an object Dog

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def bark(self):
        return "Woof!“
    def get_age_in_human_years(self):
        return self.age * 7
```

Creating and Using Methods

Object usage:

```
# Creating objects (instances) of the class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)
# Accessing attributes and calling methods
print(dog1.name)  # Output: "Buddy“
print(dog2.age)   # Output: 5
print(dog1.bark())   # Output: "Woof!“
print(dog2.bark())   # Output: "Woof!“
print(dog1.get_age_in_human_years())  # Output: 21
print(dog2.get_age_in_human_years())  # Output: 35
```

Object-Oriented programming(*hour 12*)

- Inheritance and polymorphism.
- Creating derived classes.
- Method overriding and class inheritance.

Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows a class (called the subclass or derived class) to inherit properties and behaviors from another class (called the base class or parent class). The subclass can reuse the code and functionalities defined in the base class, enhancing code reusability and maintainability.

Inheritance is represented by an "is-a" relationship.

For example, if we have a `Vehicle` base class, we can create subclasses such as `Car`, `Bike`, and `Truck`, which "are-a" kind of vehicle.

Inheritance

```
# Base class (parent class)
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
    def start(self):
        return "The vehicle engine is running.“
```

Inheritance

```
# Derived class (subclass)
class Car(Vehicle):
    def __init__(self, make, model, year, num_doors):
        # Call the constructor of the base class
        super().__init__(make, model, year)
        self.num_doors = num_doors

    def start(self):
        # Method overriding: Overrides the start() method of the
base class
        return f"The {self.make} {self.model} car engine is
running."
```

Polymorphism

- Polymorphism is the ability of different classes to be treated as objects of a common base class, allowing objects of different classes to be used interchangeably. It enables a single interface to represent different data types or objects.
- Polymorphism is often achieved through method overriding and method overloading.

Method Overriding

Method overriding allows a derived class to provide a specific implementation for a method that is already defined in the base class. When a method is overridden in the derived class, the version defined in the derived class is executed, rather than the one in the base class.

In the example above, we have overridden the `start()` method in the `Car` class. When we call the `start()` method on a `Car` object, the overridden method in the `Car` class is executed.

Method Overriding

```
car1 = Car("Toyota", "Corolla", 2022, 4)
print(car1.start())  # Output: "The Toyota Corolla car engine
is running."
```

Class Inheritance

In the example above, Car is a derived class that inherits from the base class Vehicle. We use class Car(Vehicle): to indicate that Car inherits from Vehicle. This means that Car has access to the attributes and methods defined in the Vehicle class.

```
car1 = Car("Toyota", "Corolla", 2022, 4)
print(car1.make)      # Output: "Toyota“
print(car1.model)     # Output: "Corolla“
print(car1.year)      # Output: 2022
print(car1.num_doors)  # Output: 4
```

Method Overloading

Method overloading is a form of polymorphism that allows a class to define multiple methods with the same name but different parameter lists. However, Python does not support traditional method overloading, as it allows methods with the same name, but only the last defined method is effective.

Instead, Python uses a single method name with optional arguments and default values to achieve method overloading-like behavior:

Method Overloading

```
# Example of "method overloading-like" behavior in Python
class MathOperations:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):
        return a + b + c

math_ops = MathOperations()

print(math_ops.add(2, 3))
# Output: TypeError: add() missing 1 required positional argument: 'c‘

print(math_ops.add(2, 3, 4))
# Output: 9 (2 + 3 + 4)
```

Magic methods

Magic methods allow you to define behavior for built-in operators and functions in Python when they are used with objects of your custom classes.

Magic methods allow you to make your classes more intuitive and user-friendly by defining custom behavior for built-in operators and functions.

For example, if you have a custom class and you want to define how instances of that class behave when you use operators like `+`, `-`, `*`, etc., you can do so by implementing specific magic methods.

Magic methods

Some commonly used magic methods include:

- `__init__(self, ...)`: Constructor method that initializes a new object of the class.
- `__str__(self)`: Returns a string representation of the object, usually used with the str() function or when printing the object.
- `__repr__(self)`: Returns a string representation of the object, typically used with the `repr()` function.
- `__add__(self, other)`: Defines the behavior for the `+` operator when used with instances of the class.
- `__sub__(self, other)`: Defines the behavior for the `-` operator when used with instances of the class.

Magic methods

- `__mul__(self, other)`: Defines the behavior for the `*` operator when used with instances of the class.
- `__eq__(self, other)`: Defines the behavior for the `==` operator to compare instances of the class for equality.
- `__lt__(self, other)`: Defines the behavior for the `<` operator to compare instances of the class.
- `__gt__(self, other)`: Defines the behavior for the `>` operator to compare instances of the class.
- `__len__(self)`: Returns the length of the object when used with the `len()` function.

Magic methods. Example

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
    def __str__(self):
        return f"Vector({self.x}, {self.y})“
v1 = Vector(1, 2)
v2 = Vector(3, 4)
result = v1 + v2  # Calls the __add__() method
print(result)     # Calls the __str__() method:
# Output - Vector(4, 6)
```

Public Methods and Attributes

By default, all methods and attributes in a class are considered public, meaning they can be accessed from outside the class. Public methods and attributes are intended to be part of the class's public interface, and they are accessible to all users of the class.

```
class MyClass:
    def public_method(self):
        return "This is a public method.“
    def public_attribute(self):
        return self.public_attribute
obj = MyClass()
print(obj.public_method())
# Output: This is a public method.
# Setting a public attribute
obj.public_attribute = "Public“
print(obj.public_attribute)      # Output: Public
```

Protected Methods and Attributes

In Python, we use a single leading underscore (e.g., `_protected_method`) to indicate that a method or attribute is intended for internal use within the class and its subclasses. While the interpreter does not prevent access to protected members from outside the class, it's a convention to not use them directly outside the class.

```
class MyClass:
    def __init__(self):
        self._protected_attribute = "Protected“
    def _protected_method(self):
        return "This is a protected method.“
obj = MyClass()
print(obj._protected_method())          # Output: This is a protected method.
print(obj._protected_attribute)         # Output: Protected
obj._protected_attribute = "New Protected"  # Modifying a protected attribute
print(obj._protected_attribute)         # Output: New Protected
```

Private Methods and Attributes

In Python, we use double leading underscores (e.g., `__private_method`) to indicate that a method or attribute is intended to be private to the class and should not be accessed or modified from outside the class. Python uses name mangling to make private attributes harder to access from outside the class.

```
class MyClass:
    def __init__(self):
        self.__private_attribute = "Private“
    def __private_method(self):
        return "This is a private method.“
obj = MyClass()# The following will raise an AttributeError since the attribute is name-mangled.
# print(obj.__private_attribute)
# The following will raise an AttributeError since the method is name-mangled.
# print(obj.__private_method())
# Accessing private attributes using name mangling
print(obj._MyClass__private_attribute)      # Output: Private
obj._MyClass__private_attribute = "Secret"  # Modifying a private attribute
print(obj._MyClass__private_attribute)      # Output: Secret
```