task-oops

May 18, 2024

0.0.1 Q1: Read and write about classes and objects, self, init, magic methods, constructor.

Classes and Objects Classes: A class in Python is a blueprint for creating objects (a particular data structure). A class encapsulates data for the object and methods to manipulate that data.

Objects: An object is an instance of a class. When a class is defined, no memory is allocated until an object of that class is instantiated.

[1]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return f"{self.name} is barking."

# Creating an object of the Dog class
my_dog = Dog("Buddy", 3)
print(my_dog.bark())  # Output: Buddy is barking.
```

```
Buddy is barking.
```

self: self is a reference to the current instance of the class. It is used to access variables that belong to the class. Within a method, self refers to the object calling the method

[2]:
```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"Hello, my name is {self.name}."

# Creating an object of the Person class
person = Person("Alice")
print(person.greet())  # Output: Hello, my name is Alice.
```

```
Hello, my name is Alice.
```

init Method The **init** method is a special method called a constructor. It is automatically invoked when an object of the class is instantiated. The primary purpose of the **init** method is to initialize

the object's attributes.

[3]:
```python
class Car:
    def __init__(self, model, year):
        self.model = model
        self.year = year

    def display_info(self):
        return f"Model: {self.model}, Year: {self.year}"

# Creating an object of the Car class
my_car = Car("Toyota", 2020)
print(my_car.display_info())  # Output: Model: Toyota, Year: 2020
```

```
Model: Toyota, Year: 2020
```

Magic Methods Magic methods in Python are special methods that start and end with double underscores, also known as dunder methods. They are used to override or add special functionality to classes. Common magic methods include **init**, **str**, **repr**, **add**, **len**, etc.

[4]:
```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f"'{self.title}' by {self.author}"

    def __len__(self):
        return len(self.title)

# Creating an object of the Book class
my_book = Book("1984", "George Orwell")
print(my_book)        # Output: '1984' by George Orwell
print(len(my_book))   # Output: 4
```

```
'1984' by George Orwell
4
```

Constructor A constructor in Python is a special method used to initialize objects. The **init** method is the most commonly used constructor method. When a new object is created, the **init** method is called to set up the initial state of the object.

[5]:
```python
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id

    def display_student(self):
```

```
        return f"Name: {self.name}, ID: {self.student_id}"

# Creating an object of the Student class
student = Student("John", "S12345")
print(student.display_student())  # Output: Name: John, ID: S12345
```

```
Name: John, ID: S12345
```

0.0.2 Q2: Create your own real-time class, objects, and methods.

Let's create a class called BankAccount to represent a bank account. This class will include methods to deposit money, withdraw money, and check the balance.

[6]:
```
class BankAccount:
    def __init__(self, account_holder, balance=0):
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            return f"Deposited {amount}. New balance is {self.balance}."
        else:
            return "Deposit amount must be positive."

    def withdraw(self, amount):
        if amount > 0 and amount <= self.balance:
            self.balance -= amount
            return f"Withdrew {amount}. New balance is {self.balance}."
        elif amount > self.balance:
            return "Insufficient funds."
        else:
            return "Withdrawal amount must be positive."

    def check_balance(self):
        return f"Account holder: {self.account_holder}, Balance: {self.balance}"

# Example usage
account1 = BankAccount("Alice", 100)
print(account1.check_balance())  # Output: Account holder: Alice, Balance: 100

print(account1.deposit(50))      # Output: Deposited 50. New balance is 150.
print(account1.withdraw(30))     # Output: Withdrew 30. New balance is 120.
print(account1.check_balance())  # Output: Account holder: Alice, Balance: 120

account2 = BankAccount("Bob")
print(account2.check_balance())  # Output: Account holder: Bob, Balance: 0
```

```
print(account2.deposit(200))      # Output: Deposited 200. New balance is 200.
print(account2.withdraw(100))     # Output: Withdrew 100. New balance is 100.
print(account2.check_balance())   # Output: Account holder: Bob, Balance: 100
```

```
Account holder: Alice, Balance: 100
Deposited 50. New balance is 150.
Withdrew 30. New balance is 120.
Account holder: Alice, Balance: 120
Account holder: Bob, Balance: 0
Deposited 200. New balance is 200.
Withdrew 100. New balance is 100.
Account holder: Bob, Balance: 100
```

Explanation Class Definition:

BankAccount: Represents a bank account with an account holder and balance. **Constructor (init method):**

Initializes the account_holder and balance attributes. The balance defaults to 0 if not provided. Methods:

deposit(amount): Adds the specified amount to the balance if the amount is positive. **withdraw(amount):** Deducts the specified amount from the balance if the amount is positive and does not exceed the current balance. **check_balance():** Returns the account holder's name and the current balance. Example Usage:

Creating two BankAccount objects (account1 for Alice and account2 for Bob). Demonstrating deposits, withdrawals, and balance checks on both accounts.

0.0.3 Q3: Create a class named Powers with property number. Include various methods like squares, cubes, and exponents.

[7]:
```
class Powers:
    def __init__(self, number):
        self.number = number

    def squares(self):
        return self.number ** 2

    def cubes(self):
        return self.number ** 3

    def exponent(self, exp):
        return self.number ** exp

# Example usage
p = Powers(5)
print(f"p.squares()----{p.squares()}")      # Output: p.squares()----25
print(f"p.cubes()----{p.cubes()}")          # Output: p.cubes()----125
```

```
print(f"p.exponent(7)----(5,7)={p.exponent(7)}")  # Output: p.
 ↪exponent(7)----(5,7)=78125

k = Powers(12)
print(f"k.squares() --- {k.squares()}")      # Output: k.squares() --- 144
print(f"k.cubes() --- {k.cubes()}")          # Output: k.cubes() --- 1728
print(f"k.exponent(9) --- (12,9)={k.exponent(9)}")  # Output: k.exponent(9) ---␣
 ↪(12,9)=5159780352
```

```
p.squares()----25
p.cubes()----125
p.exponent(7)----(5,7)=78125
k.squares() --- 144
k.cubes() --- 1728
k.exponent(9) --- (12,9)=5159780352
```

0.0.4 Q4: Write a Python program to create a Vehicle class with static variables max_speed and mileage instance attributes. Print the attributes.

[8]:
```
class Vehicle:
    # Static variable
    max_speed = 240

    def __init__(self, mileage):
        # Instance variable
        self.mileage = mileage

# Creating an object of the Vehicle class
vehicle = Vehicle(18)

# Printing the attributes
print(f"max_speed = {Vehicle.max_speed}")
print(f"mileage = {vehicle.mileage}")
```

```
max_speed = 240
mileage = 18
```

0.0.5 Q5a: Create a Python class called Car with attributes make and model. Instantiate an object of this class and print its attributes.

[9]:
```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

# Creating an object of the Car class
my_car = Car("Toyota", "Corolla")
```

```
# Printing the attributes
print(f"Make: {my_car.make}")
print(f"Model: {my_car.model}")
```

```
Make: Toyota
Model: Corolla
```

0.0.6 Q5b: Expand the Car class by adding a method called display_info that prints the make and model of the car. Create an object and call this method

[10]:
```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def display_info(self):
        print(f"Make: {self.make}, Model: {self.model}")

# Creating an object of the Car class
my_car = Car("Toyota", "Corolla")

# Calling the display_info method
my_car.display_info()
```

```
Make: Toyota, Model: Corolla
```

0.0.7 Q6a: Modify the Car class to include a constructor (init) that initializes the make and model attributes. Instantiate an object using the constructor.

0.0.8 Q6b: Add a method get_total_cars to the Car class that returns the total number of cars created.

[11]:
```
class Car:
    # Class variable to keep track of the number of cars
    total_cars = 0

    def __init__(self, make, model):
        self.make = make
        self.model = model
        # Increment the total number of cars each time a new car is created
        Car.total_cars += 1

    def display_info(self):
        print(f"Make: {self.make}, Model: {self.model}")

    @classmethod
    def get_total_cars(cls):
```

```
        return cls.total_cars

# Creating objects of the Car class
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")

# Calling the display_info method
car1.display_info()  # Output: Make: Toyota, Model: Corolla
car2.display_info()  # Output: Make: Honda, Model: Civic

# Getting the total number of cars
print(f"Total cars: {Car.get_total_cars()}")  # Output: Total cars: 2
```

```
Make: Toyota, Model: Corolla
Make: Honda, Model: Civic
Total cars: 2
```

0.0.9 Q7: Create a class for arithmetic operations using different methods like addition(), subtraction(), etc.

[12]:
```
class Arithmetic:
    def __init__(self, num1, num2):
        self.num1 = num1
        self.num2 = num2

    def addition(self):
        return self.num1 + self.num2

    def subtraction(self):
        return self.num1 - self.num2

    def multiplication(self):
        return self.num1 * self.num2

    def division(self):
        if self.num2 != 0:
            return self.num1 / self.num2
        else:
            return "Division by zero is not allowed."

# Example usage
arithmetic = Arithmetic(10, 5)

print(f"Addition: {arithmetic.addition()}")        # Output: Addition: 15
print(f"Subtraction: {arithmetic.subtraction()}")  # Output: Subtraction: 5
print(f"Multiplication: {arithmetic.multiplication()}")  # Output:␣
 ↪Multiplication: 50
```

```
print(f"Division: {arithmetic.division()}")        # Output: Division: 2.0
```

```
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2.0
```

0.0.10 Q8: Define a base class Vehicle with attributes color and model. Create a derived class Car that inherits from Vehicle and adds an attribute num_doors.

[13]:
```
class Vehicle:
    def __init__(self, color, model):
        self.color = color
        self.model = model

class Car(Vehicle):
    def __init__(self, color, model, num_doors):
        super().__init__(color, model)
        self.num_doors = num_doors

    def display_info(self):
        print(f"Car: {self.model}, {self.color}; Doors: {self.num_doors}")

# Example usage
my_car = Car("Red", "MG Hector", 4)
my_car.display_info()
```

```
Car: MG Hector, Red; Doors: 4
```

0.0.11 Q 9)Create two base classes and define a derived class

[20]:
```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

class Address:
  def __init__(self, city, country):
    self.city = city
    self.country = country

class Employee(Person, Address):
  def __init__(self, name, age, city, country, job):
    Person.__init__(self, name, age)  # Initialize Person attributes
    Address.__init__(self, city, country)  # Initialize Address attributes
    self.job = job
```

```
# Example usage
emp1 = Employee("John Doe", 30, "New York", "USA", "Software Engineer")
print(f"Employee: {emp1.name}, {emp1.age}, {emp1.city}, {emp1.country}, {emp1.
  ↪job}")
```

```
Employee: John Doe, 30, New York, USA, Software Engineer
```