Basic Programming with Python

Prepared By:

Professor Dr. Md. Mijanur Rahman
Department of Computer Science & Engineering
Jatiya Kabi Kazi Nazrul Islam University, Bangladesh.
www.mijanrahman.com

OOP Concepts in Python

CONTENTS

8.1 Overview....2
8.2 Procedural Oriented Approach....2
8.3 OOP Concepts....3
8.4 Class and Object....4
 8.4.1 Creating Classes in Python....5
 8.4.2 Creating Objects of Classes....6
 8.4.3 Accessing Attributes of Objects....7
 8.4.4 Built-In Class Attributes in Python....8
 8.4.5 Destroying Objects in Python....9
8.5 Methods in Python....11
8.6 Program Examples Using Class, Object and Method....12
8.7 Encapsulation in Python....16
 8.7.1 Public Access Modifier....16
 8.7.2 Private Access Modifier....16
 8.7.3 Protected Access Modifier....17
8.8 Static Variable and Static Method....17
 8.8.1 Static Variables....18

8.8.2 Static Methods 18

8.1 OVERVIEW

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects, which are instances of classes. Python supports OOP and provides several key concepts that enable the implementation of object-oriented principles.

The following are the main OOP concepts in Python:

1. **Class:** A class is a blueprint or a template that defines the properties (attributes) and behaviors (methods) that objects of that class should have. It serves as a blueprint for creating multiple objects with similar characteristics.
2. **Object:** An object is an instance of a class. It represents a specific entity that possesses the attributes and behaviors defined by its class.
3. **Encapsulation:** Encapsulation is the bundling of data and methods together within a class. It allows data hiding and provides control over how the data can be accessed or modified.
4. **Inheritance:** Inheritance is a mechanism that allows a class (derived or child class) to inherit properties and behaviors from another class (base or parent class). It promotes code reuse and supports the concept of the "is-a" relationship.
5. **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common base class. It allows methods to be overridden in the derived classes, providing different implementations while maintaining a common interface.
6. **Abstraction:** Abstraction focuses on providing a simplified representation of complex systems. It hides unnecessary details and exposes only relevant information to the user.
7. **Method:** A method is a function defined within a class. It represents the behaviors associated with the objects of that class.
8. **Attribute:** An attribute is a variable that belongs to an object or a class. It represents the data associated with the objects or the characteristics of the class.

These concepts work together to provide a powerful and flexible way to structure and organize code in Python. They promote modularity, reusability, and maintainability.

8.2 PROCEDURAL ORIENTED APPROACH

Early programming languages developed in 50s and 60s are recognized as procedural (or procedure oriented) languages.

A computer program describes procedure of performing certain task by writing a series of instructions in a logical order. Logic of a more complex program is broken down into smaller but independent and reusable blocks of statements called functions.

Every function is written in such a way that it can interface with other functions in the program. Data belonging to a function can be easily shared with other in the form of arguments, and called function can return its result back to calling function.

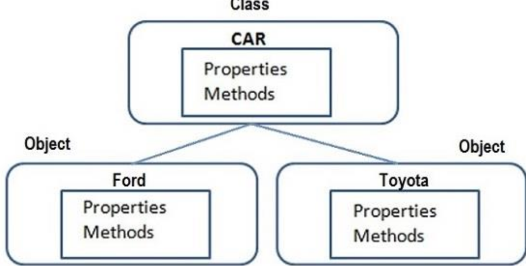
Prominent problems related to procedural approach are as follows −

- Its top-down approach makes the program difficult to maintain.
- It uses a lot of global data items, which is undesired. Too many global data items would increase memory overhead.
- It gives more importance to process and doesn't consider data of same importance and takes it for granted, thereby it moves freely through the program.
- Movement of data across functions is unrestricted. In real-life scenario where there is unambiguous association of a function with data it is expected to process.

8.3 OOP CONCEPTS

OOP is an abbreviation that stands for Object-oriented programming paradigm. It is defined as a programming model that uses the concept of objects which refers to real-world entities with state and behavior.

In the real world, we deal with and process objects, such as student, employee, invoice, car, etc. Objects are not only data and not only functions, but combination of both. Each real-world object has attributes and behavior associated with it.



Attributes:

- Name, class, subjects, marks, etc., of student
- Name, designation, department, salary, etc., of employee
- Invoice number, customer, product code and name, price and quantity, etc., in an invoice
- Registration number, owner, company, brand, horsepower, speed, etc., of car

Each attribute will have a value associated with it. Attribute is equivalent to data.

Behavior: Processing attributes associated with an object.

- Compute percentage of student's marks
- Calculate incentives payable to employee
- Apply GST to invoice value
- Measure speed of car

Behavior is equivalent to function. In real life, attributes and behavior are not independent of each other, rather they co-exist.

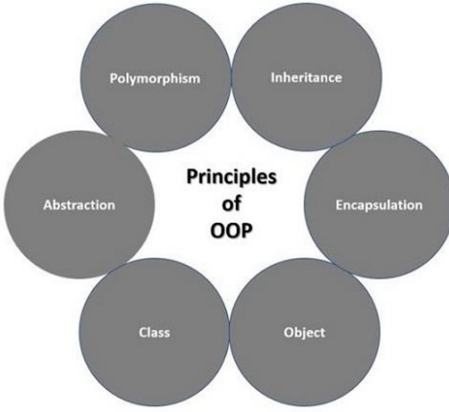
The most important feature of object-oriented approach is defining attributes and their functionality as a single unit called class. It serves as a blueprint for all objects having similar attributes and behavior.

In OOP, class defines what are the attributes its object has, and how is its behavior. Object, on the other hand, is an instance of the class. Python is a programming language that supports object-oriented programming. This makes it simple to create and use classes and objects.

Principles of OOPs Concepts:

Object-oriented programming paradigm is characterized by the following principles −

- Class
- Object
- Encapsulation
- Inheritance
- Polymorphism



8.4 CLASS AND OBJECT

A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle. A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Python is an OOP language, which means that each and every element used within a Python program is an object of one or another class. For instance, numbers, strings, lists, dictionaries, and other similar entities of a program are objects of the corresponding built-in class.

In Python, the Object class is the base or parent class for all the classes, built-in as well as user defined.

Example:

If we want to see which attribute belongs to which built-in class, we can use the Python type() function as demonstrated in the below example:

```
num1 = 20
print (type(num1))
```

```
num2 = 55.50
print (type(num2))
name = "Python OOP Language"
print (type(name))
dct = {'a':1,'b':2,'c':3}
print (type(dct))
def SayHello():
  print ("Hello Python World!")
  return
print (type(SayHello))
```

When you execute this code, it will produce the following output:

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'dict'>
<class 'function'>
```

8.4.1 Creating Classes in Python

Classes allow us to create new types of objects with their own attributes (data) and methods (functions). The **class** keyword is used to define a new class. The name of the class immediately follows the keyword class followed by a colon as follows:

```
class ClassName:
  'Optional class documentation string'
  class_suite
```

- The class has a documentation string, which can be accessed via ClassName.__doc__.
- The class_suite consists of all the component statements defining class members, data attributes and functions.

Following is the example of a simple Python class:

```
class Employee:
  'Common base class for all employees'
  empCount = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1

  def displayCount(self):
   print("Total Employee: %d" % Employee.empCount)

  def displayEmployee(self):
    print("Name : ", self.name,  ", Salary: ", self.salary)
```

8.4.2 Creating Objects of Classes

Creating objects of classes in Python involves instantiating the class to create individual instances. To create instances of a class, we call the class using class name and pass in whatever arguments its __init__ method accepts.

For example:

```
# This would create first object of Employee class
emp1 = Employee("Zara", 2000)
# This would create second object of Employee class
emp2 = Employee("Manni", 5000)
```

The following is a step-by-step guide on how to create objects of classes in Python:

1. **Define the class:** Define the blueprint for the objects you want to create. This includes specifying attributes and methods.
2. **Instantiate the class:** Create individual instances (objects) of the class by calling the class name followed by parentheses. You can pass arguments to the class's constructor method (__init__) if it requires any.
3. **Access attributes and methods:** Once objects are created, you can access their attributes and call their methods using dot notation.

Here's a simple example:

```
# Define the class
class Car:
  def __init__(self, brand, model, year):
    self.brand = brand
    self.model = model
    self.year = year

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

# Instantiate the class to create objects
car1 = Car("Toyota", "Corolla", 2020)
car2 = Car("Honda", "Civic", 2018)

# Access attributes and call methods
car1.display_info()  # Output: Brand: Toyota, Model: Corolla, Year: 2020
print(car2.model)    # Output: Civic
```

In this example:

- We define a class called Car with attributes brand, model, and year, and a method display_info() to print information about the car.
- We instantiate the class twice to create two objects (car1 and car2) with different attributes.
- We access the attributes of car1 and car2 using dot notation and call the display_info() method on car1.

8.4.3 Accessing Attributes of Objects

Accessing attributes of objects in Python is straightforward. Once we have created an object of a class, we can access its attributes using dot notation (object.attribute). The following is a brief explanation and an example:

- **Dot notation:** We use dot notation to access attributes of an object. It involves typing the object's name followed by a dot (.) and then the attribute's name.
- **Instance attributes:** These are variables that belong to a specific instance of a class. Each object of the class can have its own unique values for these attributes.

Here's an example:

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

# Creating an object (instance) of the Person class
person1 = Person("Rahman", 30)

# Accessing attributes using dot notation
print(person1.name)  # Output: Rahman
print(person1.age)   # Output: 30
```

In this example:

- We define a class called Person with two attributes: name and age.
- We create an object person1 of the Person class with the name "Alice" and age 30.
- We access the name and age attributes of person1 using dot notation (person1.name, person1.age).

Here's another example:

```
class Employee:
   "Common base class for all employees"
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print ("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print ("Name : ", self.name,  ", Salary: ", self.salary)
```

```
# This would create first object of Employee class
emp1 = Employee("Zara", 2000)
# This would create second object of Employee class
emp2 = Employee("Manni", 5000)

# Accessing attributes using dot notation
emp1.displayEmployee()
emp2.displayEmployee()
print ("Total Employee %d" % Employee.empCount)
```

When the above code is executed, it produces the following result:

```
Name :  Zara , Salary:  2000
Name :  Manni , Salary:  5000
Total Employee 2
```

We can add, remove, or modify attributes of classes and objects at any time, as follows:

```
# Add an 'age' attribute
emp1.age = 7
# Modify 'age' attribute
emp1.age = 8
# Delete 'age' attribute
del emp1.age
```

8.4.4 Built-In Class Attributes in Python

In Python, there are several built-in class attributes that are available by default for all classes. These attributes provide useful information about the class and its objects. Some of the common built-in class attributes include:

- **__doc__:** This attribute contains the documentation string of the class. It typically provides information about the class and its usage.
- **__module__:** This attribute contains the name of the module in which the class is defined.
- **__name__:** This attribute contains the name of the class.
- **__dict__:** This attribute contains a dictionary that holds the namespace of the class.
- **__bases__:** This attribute contains a tuple that holds the base classes of the class.

Here's an example demonstrating the use of these built-in class attributes:

```
class Employee:
  'Common base class for all employees'
  empCount = 0

  def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1
```

```
  def displayCount(self):
    print ("Total Employee %d" % Employee.empCount)

  def displayEmployee(self):
     print ("Name : ", self.name,  ", Salary: ", self.salary)

print ("Employee.__doc__:", Employee.__doc__)
print ("Employee.__name__:", Employee.__name__)
print ("Employee.__module__:", Employee.__module__)
print ("Employee.__bases__:", Employee.__bases__)
print ("Employee.__dict__:", Employee.__dict__)
```

When the above code is executed, it produces the following result −

```
Employee.__doc__: Common base class for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__', '__doc__': 'Common base class for all employees', 'empCount': 0, '__init__': <function Employee.__init__ at 0x7f50fb93b880>, 'displayCount': <function Employee.displayCount at 0x7f50fb93b910>, 'displayEmployee': <function Employee.displayEmployee at 0x7f50fb93b9a0>, '__dict__': <attribute '__dict__' of 'Employee' objects>, '__weakref__': <attribute '__weakref__' of 'Employee' objects>}
```

8.4.5 Destroying Objects in Python

Python deletes unneeded objects (built-in types or class instances) automatically to free the memory space. The process by which Python periodically reclaims blocks of memory that no longer are in use is termed Garbage Collection.

Hence, Python has a garbage collector that automatically deallocates memory for objects that are no longer referenced.

However, if we want to explicitly remove a reference to an object or delete an object, we can use the **del** statement. Here's how it works:

```
# Create an object
x = [1, 2, 3]

# Delete the reference to the object
del x

# Now, the object may be garbage collected if there are no other references to it
```

In this example, x was a reference to a list object [1, 2, 3]. When del x is executed, the reference x is removed, and if there are no other references to the list object, it becomes eligible for garbage collection.

Here's another example illustrating the automatic garbage collection:

```
# Create a circular reference
x = [1, 2, 3]
y = [x, x]  # y references a list containing x twice
x.append(y) # x now references y

# Delete references to x and y
del x
del y
```

In this example, even though there are no direct references to [1, 2, 3], the cyclic garbage collector in Python can detect and collect cyclically referenced objects.

We normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method **__del__(), called a destructor**, that is invoked when the instance is about to be destroyed. This method might be used to clean up any non memory resources used by an instance.

For example, the __del__() destructor prints the class name of an instance that is about to be destroyed as shown in the below code block:

```
class Point:
  def __init__( self, x=0, y=0):
    self.x = x
    self.y = y
  def __del__(self):
    class_name = self.__class__.__name__
    print (class_name, "destroyed")

pt1 = Point()
pt2 = pt1
pt3 = pt1
# prints the ids of the obejcts
print (id(pt1), id(pt2), id(pt3))
del pt1
del pt2
del pt3
```

On executing, the above code will produce following result:

```
135007479444176 135007479444176 135007479444176
Point destroyed
```

8.5 METHODS IN PYTHON

In Python, a method is a function that is defined inside a class and operates on instances of that class. Methods are associated with objects and are used to perform operations on the data contained within those objects. Methods are defined similarly to regular functions, but they are always associated with a class and have access to the instance variables (attributes) of the class through the self parameter.

Here's an example of a simple class with methods:

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def display(self):
        print("The value is:", self.value)

    def increment(self, increment_by):
        self.value += increment_by

# Create an object of the class
obj = MyClass(10)

# Call the methods on the object
obj.display()       # Output: The value is: 10
obj.increment(5)
obj.display()       # Output: The value is: 15
```

In this example:

- __init__() is a special method called the constructor, which is automatically called when a new instance of the class is created. It initializes the object's attributes.
- display() and increment() are methods defined within the class. They can access and operate on the object's attributes using the self parameter.

Python methods can also take additional parameters along with self, just like regular functions. For example:

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)
```

```
# Create an object of the class
rect = Rectangle(5, 3)

# Call methods on the object
print("Area:", rect.area())       # Output: Area: 15
print("Perimeter:", rect.perimeter()) # Output: Perimeter: 16
```

In this example, area() and perimeter() are methods that calculate the area and perimeter of a rectangle based on its length and width attributes.

8.6 PROGRAM EXAMPLES USING CLASS, OBJECT AND METHOD

Following are some examples of Python programs that use classes, methods, and objects:

1. Bank Account Management:

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited {amount} into {self.owner}'s account. New balance: {self.balance}")

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            print(f"Withdrew {amount} from {self.owner}'s account. New balance: {self.balance}")
        else:
            print("Insufficient funds.")

# Create objects
account1 = BankAccount("Rahman")
account2 = BankAccount("Islam", 1000)

# Deposit and withdraw
account1.deposit(500)
account2.withdraw(200)
```

Terminal

```
Deposited 500 into Rahman's account. New balance: 500
Withdrew 200 from Islam's account. New balance: 800
```

2. Student Record Management:

```
class Student:
    def __init__(self, name, roll_number, marks):
        self.name = name
        self.roll_number = roll_number
        self.marks = marks

    def display_details(self):
        print(f"Name: {self.name}, Roll Number: {self.roll_number}, Marks: {self.marks}")

# Create objects
student1 = Student("Rahman", 101, 95)
student2 = Student("Islam", 102, 85)

# Display details
student1.display_details()
student2.display_details()
```

Terminal

```
Name: Rahman, Roll Number: 101, Marks: 95
Name: Islam, Roll Number: 102, Marks: 85
```

3. Employee Management System:

```
class Employee:
    def __init__(self, name, emp_id, salary):
        self.name = name
        self.emp_id = emp_id
        self.salary = salary

    def display_employee_details(self):
        print(f"Name: {self.name}, Employee ID: {self.emp_id}, Salary: {self.salary}")

# Create objects
emp1 = Employee("Rahman", 1001, 50000)
emp2 = Employee("Islam", 1002, 60000)

# Display employee details
emp1.display_employee_details()
emp2.display_employee_details()
```

```
Terminal
Name: Rahman, Employee ID: 1001, Salary: 50000
Name: Islam, Employee ID: 1002, Salary: 60000
```

4. Book Management System:

```
class Book:
   def __init__(self, title, author, year_published):
      self.title = title
      self.author = author
      self.year_published = year_published

   def display_book_details(self):
      print(f"Title: {self.title}, Author: {self.author}, Year Published: {self.year_published}")

# Create objects
book1 = Book("Pandemic vs Technology", "M.M. Rahman", 2023)
book2 = Book("Programming & Software Development", "M.M. Rahman", 2024)

# Display book details
book1.display_book_details()
book2.display_book_details()
```

```
Terminal
Title: Pandemic vs Technology, Author: M.M. Rahman, Year Published: 2023
Title: Programming & Software Development, Author: M.M. Rahman, Year
    Published: 2024
```

5. Geometry: Circle and Rectangle:

```
class Circle:
   def __init__(self, radius):
      self.radius = radius

   def area(self):
      return 3.14 * self.radius ** 2

class Rectangle:
   def __init__(self, length, width):
      self.length = length
      self.width = width

   def area(self):
      return self.length * self.width
```

```
# Create objects
circle = Circle(5)
rectangle = Rectangle(4, 6)

# Calculate and display areas
print("Area of the circle:", circle.area())
print("Area of the rectangle:", rectangle.area())
```

Terminal

```
Area of the circle: 78.5
Area of the rectangle: 24
```

6. To-Do List:

```
class ToDoList:
    def __init__(self):
        self.tasks = []

    def add_task(self, task):
        self.tasks.append(task)

    def display_tasks(self):
        print("Tasks:")
        for index, task in enumerate(self.tasks, start=1):
            print(f"{index}. {task}")

# Create object
todo_list = ToDoList()

# Add tasks
todo_list.add_task("Complete assignment")
todo_list.add_task("Buy groceries")
todo_list.add_task("Go for a run")

# Display tasks
todo_list.display_tasks()
```

Terminal

```
Tasks:
1. Complete assignment
2. Buy groceries
3. Go for a run
```

8.7 ENCAPSULATION IN PYTHON

Data members of class are available for processing to functions defined within the class only. Functions of class on the other hand are accessible from outside class context. So, object data is hidden from environment that is external to class. Class function (also called method) encapsulates object data so that unwarranted access to it is prevented.

Encapsulation is one of the fundamental concepts of object-oriented programming (OOP) and it refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit called a class. Encapsulation helps in hiding the internal state of an object from the outside world and allows controlled access to the object's attributes and methods.

Encapsulation helps in maintaining the integrity of the data by preventing unintended modifications and providing a clean interface for interacting with objects.

In Python, encapsulation is implemented using access modifiers, such as public, private and protected.

8.7.1 Public Access Modifier

By default, all attributes and methods in a class are public, which means they can be accessed from outside the class.

```
class MyClass:
    def __init__(self):
        self.public_attribute = 10

    def public_method(self):
        return "This is a public method."

# Accessing public attributes and methods
obj = MyClass()
print(obj.public_attribute)
print(obj.public_method())
```

8.7.2 Private Access Modifier

Python uses a convention to make an attribute or method private by prefixing its name with double underscores (__). This makes it inaccessible from outside the class.

```
class MyClass:
    def __init__(self):
        self.__private_attribute = 20

    def __private_method(self):
        return "This is a private method."

# Access private attributes and methods from outside the class will result in an AttributeError
```

```
obj = MyClass()
# print(obj.__private_attribute)  # This will raise an AttributeError
# print(obj.__private_method())   # This will raise an AttributeError
```

8.7.3 Protected Access Modifier

In Python, the protected access modifier is a convention that indicates that an attribute or method should not be accessed directly from outside the class or its subclasses, but it can still be accessed from within the class or its subclasses. Unlike private members, protected members can be accessed from outside the class, but it's generally discouraged.

To denote a member as protected in Python, a single underscore (_) prefix is used. However, this is more of a naming convention and does not enforce strict access control like private members.

Here's an example to demonstrate the use of protected members:

```
class MyClass:
    def __init__(self):
        self._protected_attribute = 20

    def _protected_method(self):
        return "This is a protected method."

# Accessing protected attributes and methods within the class
obj = MyClass()
print(obj._protected_attribute)
print(obj._protected_method())
```

Even though _protected_attribute and _protected_method() are marked as protected, they can still be accessed from outside the class:

```
obj = MyClass()
print(obj._protected_attribute)
print(obj._protected_method())
```

This prints the values of the protected attribute and calls the protected method without raising any error. However, using protected members from outside the class is generally considered bad practice as it breaks encapsulation.

8.8 STATIC VARIABLE AND STATIC METHOD

In Python, static variables and static methods belong to the class itself rather than to instances of the class. They are shared among all instances of the class and can be accessed using the class name. Static variables are shared data among instances, while static methods are methods that don't require access to instance attributes.

8.8.1 Static Variables

Static variables are declared inside a class but outside of any instance method. They are shared among all instances of the class and are accessed using the class name.

```
class MyClass:
    static_variable = 10  # Static variable shared among all instances

    def __init__(self, value):
        self.value = value

# Accessing static variable
print(MyClass.static_variable)  # Output: 10

# Modifying static variable
MyClass.static_variable = 20
print(MyClass.static_variable)  # Output: 20

# Creating instances
obj1 = MyClass(5)
obj2 = MyClass(8)

# Accessing static variable through instances
print(obj1.static_variable)  # Output: 20
print(obj2.static_variable)  # Output: 20
```

8.8.2 Static Methods

Static methods are methods that are bound to the class rather than to instances. They don't operate on instance data and don't have access to instance attributes (self). They are declared using the @staticmethod decorator.

```
class MyClass:
    @staticmethod
    def static_method():
        return "This is a static method."

# Calling static method
print(MyClass.static_method())  # Output: This is a static method.

# Calling static method through instance (not recommended)
obj = MyClass()
print(obj.static_method())  # Output: This is a static method.
```

Static methods can be called using the class name or through instances, but it's generally recommended to call them using the class name to make it clear that they are not dependent on instance data.

Static variables and static methods are useful when certain data or functionality is shared among all instances of a class and doesn't depend on individual instance data. They help in organizing code and improving readability.

Here's an example Python program that demonstrates the use of static variables and static methods:

```
class MathOperations:
    PI = 3.14159  # Static variable for storing the value of pi

    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def subtract(x, y):
        return x - y

    @staticmethod
    def multiply(x, y):
        return x * y

    @staticmethod
    def divide(x, y):
        if y != 0:
            return x / y
        else:
            return "Cannot divide by zero"

# Using static methods to perform mathematical operations
print("Addition:", MathOperations.add(5, 3))       # Output: 8
print("Subtraction:", MathOperations.subtract(10, 4))  # Output: 6
print("Multiplication:", MathOperations.multiply(7, 2)) # Output: 14
print("Division:", MathOperations.divide(15, 3))      # Output: 5.0

# Accessing static variable
print("Value of pi:", MathOperations.PI)  # Output: 3.14159
```

In this program:

- The MathOperations class contains static methods for performing basic arithmetic operations: addition, subtraction, multiplication, and division.
- The PI variable is a static variable that stores the value of pi.
- Static methods are used to perform operations without needing to create an instance of the class.
- The static variable PI is accessed directly using the class name.
- The static methods are called using the class name, and they operate independently of any instance data.

Terminal

```
Addition: 8
Subtraction: 6
Multiplication: 14
Division: 5.0
Value of pi: 3.14159
```

⌘⌘⌘