Massive Information & Knowledge Engineering

Python OOP

part I

204113 Computer & Programming

Dr. Arnon Rungsawang
Dept. of computer engineering
Kasetsart University
https://mike.cpe.ku.ac.th/204113

Version 2024

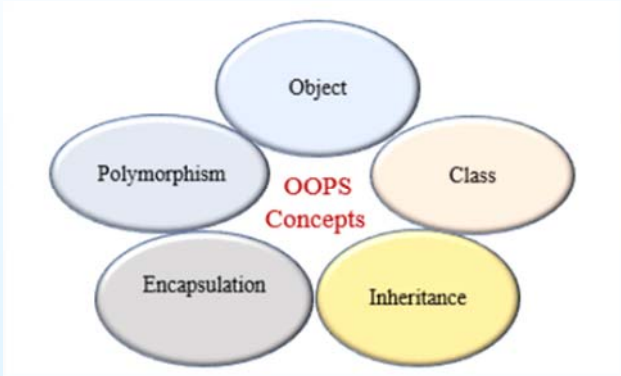
What's OOP?

What's OOP?

- Python programming language supports different programming approaches like functional programming, modular programming.
- One of the popular approaches is Object-Oriented Programming (OOP) to solve a programming problem is by creating objects.
- OOP is a programming paradigm based on the concept of "objects".
- The object contains both data and code:
  - Data in the form of properties (often known as attributes), and
  - Code, in the form of methods (actions object can perform).
- An object-oriented paradigm is to design the program using classes and objects.

OOP concept

- OOP concepts include object, class, encapsulation, inheritance and polymorphism.



https://pynative.com/python/object-oriented-programming

OOP concept

- An object has two following characteristics:
  - State, defined by its attributes, and
  - Behavior, defined by its method.
- For example, a car is an object, as it has the following properties:
  - `name`, `price`, `color` as attributes.
  - `breaking`, `accelerating` as behavior.
- One important aspect of OOP is to create reusable code using concept of inheritance.
  - This concept is also known as DRY (Don't Repeat Yourself).

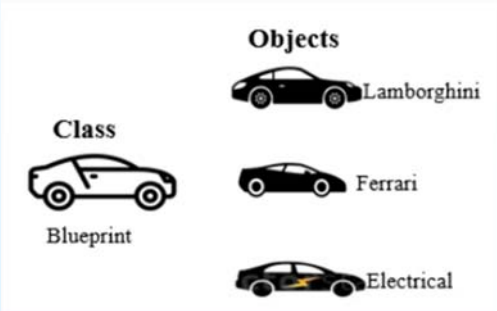
https://pynative.com/python/object-oriented-programming

Everything is an object!

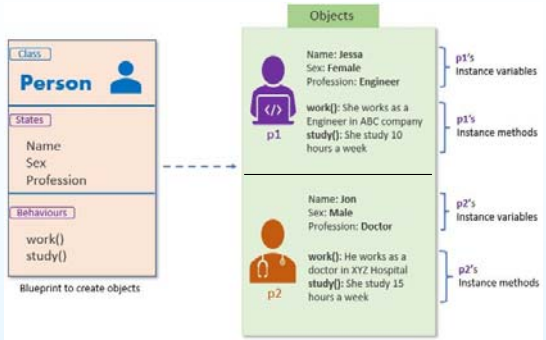
- In Python, everything is an object.
- A class is a blueprint for the object.
- To create an object, we require a model or plan or blueprint which is nothing but class.
  - For example, you can create a vehicle according to the `Vehicle` blueprint (template).
  - The model (blueprint) contains all dimensions and structure.
  - Based on these descriptions, we can construct a car, truck, bus, or any vehicle.
  - Here, a car, truck, bus are objects of `Vehicle` class.
- A class contains the properties (attribute) and action (behavior) of the object.
- Properties represent variables, and the methods represent actions. Hence class includes both variables and methods.

Instance of a class

- Object is an instance of a class.
- Object is the physical existence of a class.
- Object is an entity that has a state and behavior.
  - It may be any real-world object like the mouse, keyboard, laptop, etc.



Class and Object in Python



- Jessa is female, and she works as a Software engineer.
- On the other hand, Jon is a male, and he is a doctor.
- Here, both objects are created from the same class, but they have different states and behaviors.

- In Python, everything is treated as an object.
- An object is a real-life entity. It is the collection of various data and functions that operate on those data.
  - For example, if we design a class based on the state and behaviors of a Person, then states can be represented as instance variables and behaviors as class methods.

Create a class in Python

- In Python, class is defined by using the `class` keyword.
- The syntax to create a class is given below:

```
class class_name:
  '''This is a docstring. I have created a new class'''
  <statement 1>
  <statement 2>
  .
  .
  <statement N>
```

- `class_name`: It is the name of the class.
- `Docstring`: It is the first string inside the class and has a brief description of the class.
  - Although not mandatory, this is highly recommended.
- `statements`: Attributes and methods.

Create a class in Python -- example

```
1  class Person:
2    def __init__(self, name, sex, profession):
3      # data members (instance variables)
4      self.name = name
5      self.sex = sex
6      self.profession = profession
7
8    # Behavior (instance methods)
9    def show(self):
10     print('Name:', self.name, 'Sex:', self.sex,\
11                 'Profession:', self.profession)
12
13   # Behavior (instance methods)
14   def work(self):
15     print(self.name, 'working as a', self.profession)
```


Create Object of a Class

- The object is created using the class name.
- When we create an object of the class, it is called instantiation.
- The object is also called the instance of a class.
- A constructor is a special method used to create and initialize an object of a class.
- In Python, Object creation is divided into two parts in object creation and object initialization.
  - Internally, the `__new__` is the method that creates the object.
  - And, using the `__init__()` method, we can implement constructor to initialize the object.

Create Object of a Class -- example

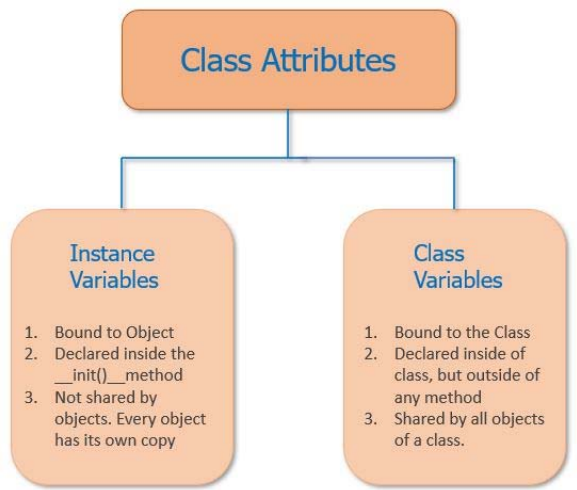
```
1  ## The complete example:
2  class Person:
3    def __init__(self, name, sex, profession):
4      # data members (instance variables)
5      self.name = name
6      self.sex = sex
7      self.profession = profession
8    # Behavior (instance methods)
9    def show(self):
10     print('Name:', self.name, 'Sex:', self.sex,\
11                 'Profession:', self.profession)
12   # Behavior (instance methods)
13   def work(self):
14     print(self.name, 'working as a', self.profession)
15
16 # create object of a class
17 jessa = Person('Jessa', 'Female', 'Software Engineer')
18 # call methods
19 jessa.show()
20 jessa.work()
```

```
Name: Jessa Sex: Female Profession: Software Engineer
Jessa working as a Software Engineer
```

Class Attributes

- In a class, attributes can be defined into two parts:
  - Instance variables are the attributes attached to an instance of a class. We define instance variables in the constructor, i.e., the `__init__()` method of a class.
  - Class variables is variables that are declared inside of a class, but outside of any instance method or `__init__()` method.
- Objects do not share instance attributes. Instead, every object has its own copy of the instance attribute and is unique to each object.
- All instances of a class share the same class variables. However, unlike instance variables, the value of a class variable is not varied from object to object.
  - Only one copy of the static (aka., class) variable will be created and shared between all objects of the class.

Class Attributes (2)



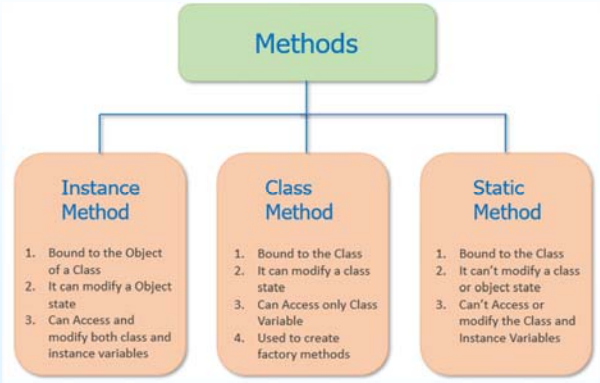
Accessing properties and assigning values

```
1  class Student:
2    # class variables
3    school_name = 'ABC School'
4    # constructor
5    def __init__(self, name, age):
6      # instance variables
7      self.name = name
8      self.age = age
9
10 s1 = Student("Harry", 12)
11 # access instance variables
12 print('Student:', s1.name, s1.age)
13 # access class variable
14 print('School name:', Student.school_name)
15 # Modify instance variables
16 s1.name = 'Jessa'
17 s1.age = 14
18 print('Student:', s1.name, s1.age)
19 # Modify class variables
20 Student.school_name = 'XYZ School'
21 print('School name:', Student.school_name)
22 print(s1.__dict__)
```

- An instance attribute can be accessed or modified by using the dot notation, i.e., `instanceName.attributeName`
- A class variable is accessed or modified using the class name.

```
Student: Harry 12
School name: ABC School
Student: Jessa 14
School name: XYZ School
{'name': 'Jessa', 'age': 14}
```


Class Methods



- Inside a class, we can define the following 3 type of methods.
  - Instance method: Used to access or modify the object state. If we use instance variables inside a method, such methods are called instance methods.
  - Class method: Used to access or modify the class state. In method implementation, if we use only class variables, then such type of methods we should declare as a class method.
  - Static method: It is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variable because this static method doesn't have access to the class attributes.

Class Methods (2)

```
class Student:
  # class variable
  school_name = 'ABC School'
  # constructor
  def __init__(self, name, age):
    # instance variables
    self.name = name
    self.age = age
  # instance method
  def show(self):
    # access instance variables and class variables
    print('Student:', self.name, self.age,\
          Student.school_name)
  # instance method
  def change_age(self, new_age):
    # modify instance variable
    self.age = new_age

  # class method
  @classmethod
  def modify_school_name(cls, new_name):
    # modify class variable
    cls.school_name = new_name

s1 = Student("Harry", 12)
# call instance methods
s1.show()
s1.change_age(14)
# call class method
Student.modify_school_name('XYZ School')
# call instance methods
s1.show()
```

- Instance methods work on the instance level (object level). For example, if we have two objects created from the student class, They may have different names, marks, roll numbers, etc. Using instance methods, we can access and modify the instance variables.
- A class method is bound to the class and not the object of the class. It can access only class variables.

```
Student: Harry 12 ABC School
Student: Harry 14 XYZ School
```


Class Naming Convention

- Writing readable code is one of the guiding principles of the Python language.
- We should follow specific rules while we are deciding a name for the class in Python.
  - Rule-1: Class names should follow the `UpperCaseCamelCase` convention
  - Rule-2: Exception classes should end in `Error`.
  - Rule-3: If a class is callable (Calling the class from somewhere), in that case, we can give a class name like a function.
  - Rule-4: Python's built-in classes are typically lowercase words.

Object Properties

- Every object has properties with it. In other words, we can say that object property is an association between name and value.
  - For example, a car is an object, and its properties are car's color, sunroof, price, manufacture, model, engine, and so on.
  - Here, color is the name and red is the value.
  - Object properties are represented by instance variables.

Modify Object Properties

```
class Fruit:
  def __init__(self, name, color):
    self.name = name
    self.color = color

  def show(self):
    print("Fruit is", self.name,\
          "and Color is", self.color)

# creating object of the class
obj = Fruit("Apple", "red")

# Modifying Object Properties
obj.name = "strawberry"

# calling the instance method
# using the object obj
obj.show()
# Output Fruit is strawberry and Color is red
```

```
Fruit is strawberry and Color is red
```

- Every object has properties associated with them.
- We can set or modify the object's properties after object initialization by calling the property directly using dot operator.

Delete Object Properties

```
1  class Fruit:
2    def __init__(self, name, color):
3      self.name = name
4      self.color = color
5
6    def show(self):
7      print("Fruit is", self.name,\
8            "and Color is", self.color)
9
10 # creating object of the class
11 obj = Fruit("Apple", "red")
12 # Deleting Object Properties
13 del obj.name
14
15 # Accessing object properties after deleting
16 print(obj.name)
```

```
AttributeError: 'Fruit' object has no attribute 'name'
```

- We can delete the object property by using the del keyword.
- After deleting it, if we try to access it, we will get an error.

Delete an Object

```
1  class Employee:
2    depatment = "IT"
3
4    def show(self):
5      print("Department is ", self.depatment)
6
7  emp = Employee()
8  emp.show()
9
10 # delete object
11 del emp
12
13 # Accessing after delete object
14 emp.show()
```

```
NameError: name 'emp' is not defined
```

- In Python, we can also delete the object by using a del keyword.
- An object can be anything like, class object, list, tuple, set, etc.

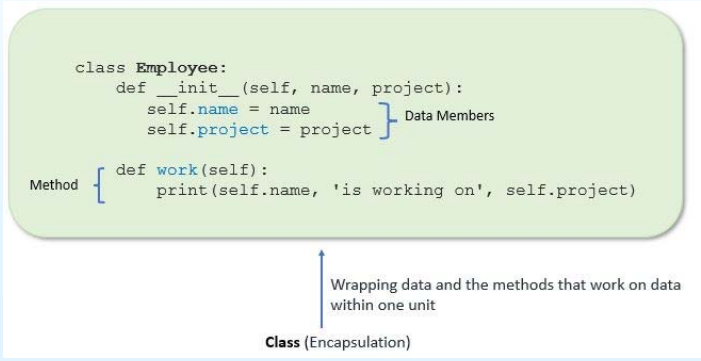
Encapsulation

Encapsulation

- Encapsulation is a method of wrapping data and functions into a single entity.
- Encapsulation means the internal representation of an object is generally hidden from outside of the object's definition.
  - A class encapsulates all the data (variables and methods).

Encapsulation in Python

- Encapsulation in Python describes the concept of bundling data and methods within a single unit.
  - When we create a class, it means we are implementing encapsulation.
  - A class is an example of encapsulation as it binds all the data members (instance variables) and method into a single unit.



Encapsulation in Python -- Example

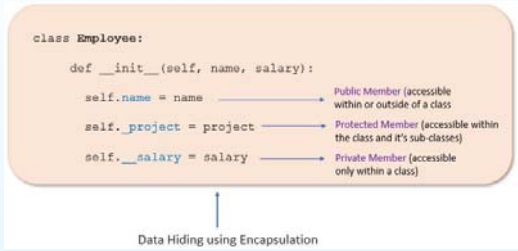
```
1  class Employee:
2      # constructor
3      def __init__(self, name, salary, project):
4          # data members
5          self.name = name
6          self.salary = salary
7          self.project = project
8
9      # method to display employee's details
10     def show(self):
11         # accessing public data member
12         print("Name: ", self.name, 'Salary:', self.salary)
13
14     # method
15     def work(self):
16         print(self.name, 'is working on', self.project)
17
18 # creating object of a class
19 emp = Employee('Jessa', 8000, 'NLP')
20
21 # calling public method of the class
22 emp.show()
23 emp.work()
```

```
Name:  Jessa Salary: 8000
Jessa is working on NLP
```


Usage of Encapsulation

- Using encapsulation, we can hide an object's internal representation from the outside. This is called information hiding.
- Also, encapsulation allow us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.
- Suppose we have an attribute that is not visible from the outside of an object and bundle it with methods that provide read or write access. In that case, we can hide specific information and control access to the object's internal state.
- Encapsulation offers a way for us to access the required variable without providing the program full-fledged access to all variables of a class. This mechanism is used to protect the data of an object from other objects.

Access Modifiers in Python



- In Python, we do not have direct access modifiers like `public`, `protected`, and `private`, but we achieve this by using single underscore and double underscore.
- Python provides three types of access modifier:
  - Public Member: Accessible anywhere from outside the class.
  - Protected Member: Accessible within the class and its sub-classes.
  - Private Member: Accessible within the class.

Public Member

- Public data members are accessible within and outside of a class. All member variables of the class are by default public.

```
1  class Employee:
2    # constructor
3    def __init__(self, name, salary):
4      # public data members
5      self.name = name
6      self.salary = salary
7
8    # public instance methods
9    def show(self):
10     # accessing public data member
11     print("Name: ", self.name, 'Salary:', self.salary)
12
13 # creating object of a class
14 emp = Employee('Jessa', 10000)
15 # accessing public data members
16 print("Name: ", emp.name, 'Salary:', emp.salary)
17 # calling public method of the class
18 emp.show()
```

```
Name:  Jessa Salary: 10000
Name:  Jessa Salary: 10000
```


Private Member

- Private members are accessible only within the class, and we can't access them directly from the class objects.

```
1  class Employee:
2      # constructor
3      def __init__(self, name, salary):
4          # public data member
5          self.name = name
6          # private member
7          self.__salary = salary
8
9  # creating object of a class
10 emp = Employee('Jessa', 10000)
11
12 # accessing private data members
13 print('Salary:', emp.__salary)
```

```
AttributeError: 'Employee' object has no attribute '__salary'
```


Public method to access private member

```
1  class Employee:
2    # constructor
3    def __init__(self, name, salary):
4      # public data member
5      self.name = name
6      # private member
7      self.__salary = salary
8    # public instance methods
9    def show(self):
10     # private members are accessible from a class
11     print("Name:", self.name, 'Salary:', self.__salary)
12
13 # creating object of a class
14 emp = Employee('Jessa', 10000)
15 # calling public method of the class
16 emp.show()
```

```
Name: Jessa Salary: 10000
```


Name mangling to access private member

```
1  class Employee:
2    # constructor
3    def __init__(self, name, salary):
4      # public data member
5      self.name = name
6      # private member
7      self.__salary = salary
8
9  # creating object of a class
10 emp = Employee('Jessa', 10000)
11
12 print('Name:', emp.name)
13 # direct access to private member
14 # using name mangling
15 print('Salary:', emp._Employee__salary)
```

```
Name: Jessa
Salary: 10000
```

- We can directly access private and protected variables from outside of a class through name mangling.
- The name mangling is created on an identifier by adding one leading underscore and two trailing underscores, like `_classname__dataMember`, where class name is the current class, and data member is the private variable name.

Protected Member

```python
# base class
class Company:
  def __init__(self):
    # Protected member
    self._project = "NLP"
# child class
class Employee(Company):
  def __init__(self, name):
    self.name = name
    Company.__init__(self)
  def show(self):
    print("Employee name :", self.name)
    # Accessing protected member in child class
    print("Working on project :", self._project)

c = Employee("Jessa")
c.show()
```

```
Employee name : Jessa
Working on project : NLP
```

- Protected members are accessible within the class and also available to its sub-classes.
- To define a protected member, prefix the member's name with a single underscore _.
- Protected data members are used when you implement inheritance, and you want to allow data members access to only child classes.

Getters and Setters in Python

- To implement proper encapsulation in Python, we need to use `setters` and `getters`.
- The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. We use
  - the getter method to access data members
  - the setter methods to modify the data members.
- In Python, private variables are not hidden fields like in other programming languages.
- The getters and setters methods are often used when:
  - When we want to avoid direct access to private variables.
  - To add validation logic for setting a value.

Getters and Setters in Python (2)

```python
class Student:
  def __init__(self, name, age):
    # private member
    self.name = name
    self.__age = age
  # getter method
  def get_age(self):
    return self.__age
  # setter method
  def set_age(self, age):
    self.__age = age

stud = Student('Jessa', 14)
# retrieving age using getter
print('Name:', stud.name, stud.get_age())
# changing age using setter
stud.set_age(16)
# retrieving age using getter
print('Name:', stud.name, stud.get_age())
```

```
Name: Jessa 14
Name: Jessa 16
```


Getters and Setters in Python (3)

```python
class Student:
  def __init__(self, name, roll_no, age):
    self.name = name
    # private members to restrict access
    # avoid direct data modification
    self.__roll_no = roll_no
    self.__age = age
  def show(self):
    print('Student Details:', self.name, self.__roll_no)
  # getter methods
  def get_roll_no(self):
    return self.__roll_no
  # setter method to modify data member
  # condition to allow data modification with rules
  def set_roll_no(self, number):
    if number > 50:
      print('Invalid roll no!')
    else:
      self.__roll_no = number

jessa = Student('Jessa', 10, 15)
# before Modify
jessa.show()
# changing roll number using setter
jessa.set_roll_no(120)
jessa.set_roll_no(25)
jessa.show()
```

```
ident Details: Jessa 10
alid roll no!
ident Details: Jessa 25
```

- Example of information hiding and conditional logic for setting an object attributes.

Getters and Setters in Python (4)

```
class Student:
  def __init__(self, name, roll_no, age):
    self.name = name
    # private members to restrict access
    # avoid direct data modification
    self.__roll_no = roll_no
    self.__age = age
  def show(self):
    print('Student Details:', self.name, self.__roll_no)
  # getter methods
  def get_roll_no(self):
    return self.__roll_no
  # setter method to modify data member
  # condition to allow data modification with
  def set_roll_no(self, number):
    if number > 50:
      print('Invalid roll no!')
    else:
      self.__roll_no = number

jessa = Student('Jessa', 10, 15)
# before Modify
jessa.show()
# changing roll number using setter
jessa.set_roll_no(120)
jessa.set_roll_no(25)
jessa.show()
```

```
[2]  1  jessa.__dict__
```

```
{'name': 'Jessa', '_Student__roll_no': 25, '_Student__age': 15}
```

```
[3]  1  # try to modify the internal private member state?
     2  # note that in Python, we can still modify the internal
     3  # private member value, but it's bad prog. practice!
     4  jessa.__roll_no = 1000
     5  jessa.__dict__
```

```
{'name': 'Jessa',
 '_Student__roll_no': 25,
 '_Student__age': 15,
 '__roll_no': 1000}
```

```
     1  jessa._Student__roll_no = 2000
     2  jessa.__dict__
```

```
{'name': 'Jessa',
 '_Student__roll_no': 2000,
 '_Student__age': 15,
 '__roll_no': 1000}
```


Advantages of Encapsulation

- Security: The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
- Data Hiding: The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
- Simplicity: It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.
- Aesthetics: Bundling data and methods within a class makes code more readable and maintainable.

Sample Problem Solving

Circle class

- Write a program to create a class representing a Circle. Include methods to calculate its area and perimeter.

```python
import math
class Circle:
  def __init__(self, radius):
    self.radius = radius
  def calculate_circle_area(self):
    return math.pi * self.radius**2
  def calculate_circle_perimeter(self):
    return 2 * math.pi * self.radius

# Example usage
radius = float(input("Input the radius of the circle: "))
circle = Circle(radius)
area = circle.calculate_circle_area()
perimeter = circle.calculate_circle_perimeter()
print("Area of the circle:", area)
print("Perimeter of the circle:", perimeter)
```

Person class

- Write a program to create a person class. Include attributes like name, country and date of birth. Implement a method to determine the person's age.

```python
class Person:
  ThisYear = 2023
  def __init__(self, name, country, date_of_birth):
    self.name = name
    self.country = country
    self.date_of_birth = date_of_birth
  def calculate_age(self):
    age = Person.ThisYear - self.date_of_birth[0]
    return age

# Example usage
person1 = Person("Ferdi Odilia", "France", (1962, 7, 12))
person2 = Person("Shweta Maddox", "Canada", (1982, 10, 20))
person3 = Person("Elizaveta Tilman", "USA", (2000, 1, 1))
```


Person class (2)

```python
# Accessing attributes and calculating age
print("Person 1:")
print("Name:", person1.name)
print("Country:", person1.country)
print("Date of Birth:", person1.date_of_birth)
print("Age:", person1.calculate_age())

print("\nPerson 2:")
print("Name:", person2.name)
print("Country:", person2.country)
print("Date of Birth:", person2.date_of_birth)
print("Age:", person2.calculate_age())

print("\nPerson 3:")
print("Name:", person3.name)
print("Country:", person3.country)
print("Date of Birth:", person3.date_of_birth)
print("Age:", person3.calculate_age())
```


Calculator class

- Write a program to create a calculator class. Include methods for basic arithmetic operations.

```python
class Calculator:
  def add(self, x, y):
    return x + y
  def subtract(self, x, y):
    return x - y
  def multiply(self, x, y):
    return x * y
  def divide(self, x, y):
    if y != 0:
      return x / y
    else:
      return ("Cannot divide by zero.")

# Example usage
calculator = Calculator()
# Addition
result = calculator.add(7, 5)
print("7 + 5 =", result)
# Subtraction
result = calculator.subtract(34, 21)
print("34 - 21 =", result)
# Multiplication
result = calculator.multiply(54, 2)
print("54 * 2 =", result)
# Division
result = calculator.divide(144, 2)
print("144 / 2 =", result)
# Division by zero (raises an error)
result = calculator.divide(45, 0)
print("45 / 0 =", result)
```


ShoppingCart class

- Write a program to create a class representing a shopping cart. Include methods for adding and removing items and calculating the total price.

```python
class ShoppingCart:
  def __init__(self):
    self.items = []
  def add_item(self, item_name, qty):
    item = (item_name, qty)
    self.items.append(item)
  def remove_item(self, item_name):
    for item in self.items:
      if item[0] == item_name:
        self.items.remove(item)
        break
  def calculate_total(self):
    total = 0
    for item in self.items:
      total += item[1]
    return total
```

ShoppingCart class (2)

```
# Example usage
cart = ShoppingCart()

cart.add_item("Papaya", 100)
cart.add_item("Guava", 200)
cart.add_item("Orange", 150)

print("Current Items in Cart:")
for item in cart.items:
    print(item[0], "-", item[1])

total_qty = cart.calculate_total()
print("Total Quantity:", total_qty)

cart.remove_item("Orange")

print("\nUpdated Items in Cart after removing Orange:")
for item in cart.items:
    print(item[0], "-", item[1])

total_qty = cart.calculate_total()
print("Total Quantity:", total_qty)
```


Bank class

- Write a program to create a class representing a bank. Include methods for managing customer accounts and transactions.

```
class Bank:
  def __init__(self):
    self.customers = {}

  def create_account(self, account_number, initial_balance=0):
    if account_number in self.customers:
      print("Account number already exists.")
    else:
      self.customers[account_number] = initial_balance
      print("Account created successfully.")

  def make_deposit(self, account_number, amount):
    if account_number in self.customers:
      self.customers[account_number] += amount
      print("Deposit successful.")
    else:
      print("Account number does not exist.")

  def make_withdrawal(self, account_number, amount):
    if account_number in self.customers:
      if self.customers[account_number] >= amount:
        self.customers[account_number] -= amount
        print("Withdrawal successful.")
      else:
        print("Insufficient funds.")
    else:
      print("Account number does not exist.")
```


Bank class (2)

```
  def check_balance(self, account_number):
    if account_number in self.customers:
      balance = self.customers[account_number]
      print(f"Account balance: {balance}")
    else:
      print("Account number does not exist.")

# Example usage
bank = Bank()
acno1= "SB-123"
damt1 = 1000
print("New a/c No.: ",acno1,"Deposit Amount:",damt1)
bank.create_account(acno1, damt1)
acno2= "SB-124"
damt2 = 1500
print("New a/c No.: ",acno2,"Deposit Amount:",damt2)
bank.create_account(acno2, damt2)
wamt1 = 600
print("\nDeposit Rs.",wamt1,"to A/c No.",acno1)
bank.make_deposit(acno1, wamt1)
wamt2 = 350
print("Withdraw Rs.",wamt2,"From A/c No.",acno2)
bank.make_withdrawal(acno2, wamt2)
print("A/c. No.",acno1)
bank.check_balance(acno1)
print("A/c. No.",acno2)
bank.check_balance(acno2)
wamt3 = 1200
print("Withdraw Rs.",wamt3,"From A/c No.",acno2)
bank.make_withdrawal(acno2, wamt3)
acno3 = "SB-134"
print("A/c. No.",acno3)
bank.check_balance(acno3)  # Non-existent account number
```

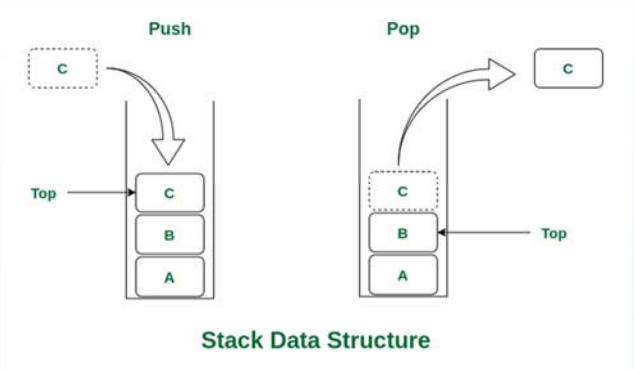

To be continue..

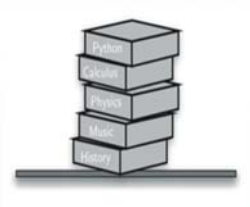
つづく

Extra..

Stack

- A stack is defined as a linear data structure that is open at one end and the operations follow the Last-In-First-Out (LIFO) order..





Ref: https://www.geeksforgeeks.org/stack-meaning-in-dsa/


```python
class myStack:
    ''' references
    https://www.geeksforgeeks.org/stack-meaning-in-dsa/
    https://www.geeksforgeeks.org/stack-in-python/
    '''
    def __init__(self):
        self.m = []
    def push(self, n):
        self.m.append(n)
    def pop(self):
        length = len(self.m)
        if length==0: return None
        res = self.m[length-1]
        self.m = self.m[:length-1]
        return res
    def peek(self):
        length = len(self.m)
        if length > 0: return self.m[length-1]
        return None
    def top(self):
        return myStack.peek(self)
    def empty(self):
        if len(self.m) > 0: return False
        return True
    def size(self):
        return len(self.m)
    def __str__(self):
        return str(self.m)
    def __repr__(self):
        return str(self.m)
```

```python
def test_myStack():
    m = myStack()
    print('Pushing test:')
    for i in range(5):
        m.push(i)
        print(m)
    print('Poping test:')
    while True:
        k = m.pop()
        if k == None:
            print()
            break
        print(k, end=' ')
```


Infix to Postfix

- Infix expression: The expression of the form "a operator b" (a + b) i.e., when an operator is in-between every pair of operands.
- Postfix expression: The expression of the form "a b operator" (ab+) i.e., When every pair of operands is followed by an operator.

Input: *A + B * C + D*
Output: *ABC*+D+*

Input: *((A + B) – C * (D / E)) + F*
Output: *AB+CDE/*-F+*

Ref: https://www.geeksforgeeks.org/convert-infix-expression-to-postfix-expression/

Infix to Postfix

Change the expression and converter will convert infix to postfix step by step.

Infix: A+B*C/(E-F)

Infix to Postfix

Postfix: ABC*EF-/+

Step by step output for '' expression

Input String	Output Stack	Operator Stack
A+B*C/(E-F)	A	
A+B*C/(E-F)	A	+
A+B*C/(E-F)	AB	+
A+B*C/(E-F)	AB	+*
A+B*C/(E-F)	ABC	+*
A+B*C/(E-F)	ABC*	+/
A+B*C/(E-F)	ABC*	+/(
A+B*C/(E-F)	ABC*E	+/(
A+B*C/(E-F)	ABC*E	+/(-
A+B*C/(E-F)	ABC*EF	+/(-
A+B*C/(E-F)	ABC*EF-	+/
A+B*C/(E-F)	ABC*EF-/+	

Ref: https://www.web4college.com/converters/infix-to-postfix-prefix.php


```python
def in2postFix(s='a+b*(c^d-e)^(f+g*h)-i'):
    '''
    https://www.geeksforgeeks.org/convert-infix-expression-to-postfix-expression/
    https://www.web4college.com/converters/infix-to-postfix-prefix.php
    '''
    #-----------------------------------
    def prec(op):
        if op=='^': return 3
        elif op in ['*','/']: return 2
        elif op in ['+','-']: return 1
        else: return -1
    def associativity(op):
        if op=='^': return 'R'
        return 'L'
    #-----------------------------------
    m, res = myStack(), []
    for i in range(len(s)):
        c = s[i]
        if 'a'<=c<='z' or 'A'<=c<='Z' or '0'<=c<='9':
            # operand found
            res.append(c)
        elif c=='(':
            # '(' found, push into stack
            m.push(c)
        elif c==')':
            # ')' found, pop elements in stack till the matching '(' end
            while not m.empty() and m.top() != '(':
                res.append(m.pop())
            # poping th matching '('
            m.pop()
        else:
            # operator found
            while not m.empty() and (prec(c)<=prec(m.top()) and associativity(c)=='L'):
                res.append(m.pop())
            m.push(c)
        print(f'c={c}, res={" ".join(res)}, m={m}')
    while not m.empty():
        res.append(m.pop())
    return ' '.join(res)
```


Evaluate the Infix

```
Shell
>>> %Run myStackOOP.py
  2 +  3* (2^ 3 -4) ^(8 -2 *3)- 27 +1
  2 3 2 3 ^ 4 - 8 2 3 * - ^ * + 27 - 1 +
  calculate 2^3, result=8, m=['2', '3', '8']
  calculate 8-4, result=4, m=['2', '3', '4']
  calculate 2*3, result=6, m=['2', '3', '4', '8', '6']
  calculate 8-6, result=2, m=['2', '3', '4', '2']
  calculate 4^2, result=16, m=['2', '3', '16']
  calculate 3*16, result=48, m=['2', '48']
  calculate 2+48, result=50, m=['50']
  calculate 50-27, result=23, m=['23']
  calculate 23+1, result=24, m=['24']
  Finally, result=24
  24
>>> 2 +  3* (2**3 -4)**(8 -2 *3)- 27 +1
24
>>>
```


Evaluate the Infix

```python
def cal(r):
    #-------------------------------
    def cal(a,b,c):
        a,b = int(a), int(b)
        if c=='^': return pow(a,b)
        if c=='*': return a*b
        if c=='/': return a/b
        if c=='+': return a+b
        if c=='-': return a-b
    #-------------------------------
    m = myStack()
    r = r.split()
    res = 0
    for i in range(len(r)):
        c = r[i]
        if c in ['^','*','/','+','-']:
            b,a = m.pop(),m.pop()
            res = cal(a,b,c)
            m.push(str(res))
            print(f"calculate {a}{c}{b}, result={res}, m={m}")
            continue
        m.push(c)
    print(f'Finally, result={res}')
    return res
```