UNIT-I

ABSTRACT DATA TYPES

Abstract Data Types (ADTs):-

ADT is a high-level concept in data structures that deffined a set of values and operations that can be performed on those values, without specifying how the values are stored or implemented.

In other word, an ADT is a way of describing what a data structures can do, without specifying how the values are stored or implemented.

Examples of ADTs:

1. stack:

Last In, First out (LIFO) data structures with push and pop operations.

2. Queue:

Fist In, First out (FIFO) data structures with enqueue and dequeue operations.

3. set:

Unordered collection of unique elements with add, remove, and search operations.

4. Map (or) Dictionary:

Key-value pairs with insert, delete and lookup operations.

5. Graph:

Non-linear data structure with nodes and edges, supporting traversal and search operations.

6. List or Array:

ordered collection of elements with indexing. Insertion and deletion operations.

ADTs and classes:

ADTs and classes are related concepts in programming but they serve different purposes;

Abstract Data Type (ADT):

* A high-level concept that defines a set of values and operations that can be performed on those values.
* Specifies what a data structures can do, without worrying about how its done.
* Defines the interface or contract of a data structures.

Class:

* A programming construct that implements an ADT.
* A blueprint or template that defines the properties (data) and methods (operations) of an object.
* Encapsulate the data and behavior of an object, making it a self-contained unit.

Examples:

* ADT:

Stack (Last In, First out data structures with push and pop operations).

* Class:

StackImplementation (a class that implements the stack ADT using an array or linked list).

Introduction to oop:

Key concepts:

1. **objects:**

Represent real-world entities or abstract concepts with properties (data) and Methods (functions that operate on that data).

2. **classes:**

Blueprints or templates that defines the properties and Methods of an object.

3. **Inheritance:**

A class can inherit properties and methods from a parent class allowing for code reuse and a hierachical relationship.

4. **polymorphism:**

objects of different classes can be treated as object of a common super class, enabling more flexibility in programming.

5. **Encapsulation:**

objects hide their internal details and expose only necessary information through public methods.

6. **Absbraction:**

Focus on essential features whiles hiding non-essential details.

Benefits of oop:

1. Modularity:

Code is organized into self-contained units (objects) that are easy to maintain and reuse.

2. Reusability:

Inheritance and polymorphism enables code reuse and reduce duplication.

3. Easier Maintenance:

Encapsulation and abstraction make it easier to modify and extend code without affecting other parts of the program.

4. Improved Readability:

oop concepts help make code more intuitive and easier to understand.

Classes in python:

A class is a blueprint or template that defines the properties and behavior of an objects.

Key components of a class:

1. Class name:

A unique name for the class.

2. Attributes (data):

Variables that are defined inside

the class, representing the object's properties.

3. Methods/functions:

Functions that are defined inside the class, representing the object's behavior.

Syntax:

```
class ClassName:
    #attributes
    attribute1 = value1
    attribute2 = value2
    # Methods
    def Method1 (self):
        #code here
    def Method2 (self):
        #code here
....
```

Example:

```
class BankAccount:
    #attributes
    account_number = "  "
    account_name = "  "
    balance = 0.0

    #methods
    def deposit(self, amount):
        self.balance += amount
        print("Deposited $ {amount}. New balance: $
                    {self.balance"}")
```

```
def withdraw (self, amount):
    if amount > self.balance:
        print ("Insufficient funds!")
    else
        self.balance -= amount
        print ("withdraw $ {amount}. New balance:
               $ {self.balance}")
def display_details (self):
    print (" Account Number : {self.account_number}")
    print ("Account Name : {self.account_name}")
    print (" Balance : $ {self.balance}")
...
# Create an object of this class

my_account = BankAccount ()
my_account.account_Number = " 1234567890"
my_account.account_name = " Karthi "
my_account.balance = 10,000

my_account.display_details ()
my_account.deposit (5,000)
my_account.withdraw (7,000)
....
```

output:

```
Account Number: 1234567890
Account Name : Karthi
Balance : 10,000
Deposited : 5000
New Balance : 15,000
withdraw : 7,000
New Balance : 8,000
```

Inheritance:

Inheritance in python is a mechanism that allow one class to inherit the attributes and Methods of another class.

The class that is being inherited from is called the parent or super class and the class that is doing the inheriting is called child or subclass.

Example:

```
#parent class
class shape:
    def _int_(Self, color):
        Self.color = color

    def. get_color (Self):
        return self.color
#child class

class circle (Shape):
    def _init_(self, color, radius):
        super()._int_(color)
        self.radius = radius

    def get_area(self):
        return 3.14 * (self.radius ** 2)
#child class
class Rectangle (Shape):
    def _int_(self, color, width, height):
        super().__init__(color)
        self.width = width
```

```
        self.height = height

    def get_area (self):
        return self.width * self.height

circle = Circle ("red", 5)
print (circle.get_color())
print (circle.get_area())

rectangle = Rectangle ("blue", 4, 6)
print (rectangle.get_color())
print (rectangle.get_area())
```

<u>output:</u>

red
78.5
blue
24

<u>Types of inheritance:</u>

1. <u>single Inheritance:</u>

one child class inherits from one parent class.

2. <u>Multiple Inheritance:</u>

one child class inherits from multiple parent classes.

3. <u>Multilevel Inheritance:</u>

A child class inherits from a parent class that itself inherits from another parent class.

4. Hierarchical Inheritance:

Multiple subclasses inherit from a single parent class.

5. Hybrid inheritance:

Combination of Multiple and Multilevel inheritance.

Namespace:

In python, a namespace is a Mapping of names to objects. It's a way to organized and scope names to avoid name clashes.

Types of Namespaces:

1. Global Namespace:

The global namespace contains built-in names such as len, print and range.

2. Local Namespace:

Each function or Method has its own local namespace, which contains names defined within that function

3. Module Namespace:

Each Module has its own namespace which contains names defined in that Module.

4. class Namespace:

Each class has its own namespace which contains names defined in that class.

5. Instance Namespace:

Each Instance of a class has its own namespace, which contains names defined in that instance

Example:

```
# Global namespace
x=10
def my_function():
    # local namespace
    y=20
    print ("Local:", locals())
    print ("Global:", globals())

class myclass:
    # class namespace
    z=30
    def _init_(self):
        # Instance namespace
        self.w=40

obj = myclass()
print ("class:", myclass._dict_)
print ("Instance:", obj._dict_)
print ("Global x:", x)
print ("Local y (not accessible):", y)
print ("class z:", Myclass.z)
print ("Instance w:", obj.w)
```

Output:

```
Global: {...,'x': 10, ...}
Local : {'y': 20}
class {...'z': 30, ...}
Instance: {..., 'w': 40, ...}
Global x: 10
Local y (not accessible):
```

class 2:30
Instance 10:40

shallow and deep copying:

In python, when you assign a new variable to an existing variable, it doesn't create a new copy of the original variable.

Instead, it creates a new reference to the same object. This can lead to unexpected behavior when modifying the new variable.

To create a true copy of an object, you can use the copy module, which provides two types of copying:

1. shallow copy:

creates a new object and then (to the extend possible) inserts reference into it to the objects found in the original object.

2. Deep copy:

creates a new object and then, recursively, inserts copies into it of the objects found in the original object.

Example:

```
import copy
original_list=[[1,2],[3,4]]
#shallow copy
shallow_copied_list = copy.copy (original_list)
# Deep copy
deep_copied_list = copy.deepcopy (original_list)
```

```
original_list.append([5,6])
original_list[0][0]='x'
print("original List:", original_list)
print("shallow Copied List:", shallow_copied_list)
print("Deep copied List:", deep_copied_list)
```

Output:

```
original List : [['x',2],[3,4],[5,6]]
Shallow Copied List : [['x',2],[3,4]]
Deep Copied List : [[1,2],[3,4]]
```

Introduction to analysis of algorithms:

Analysis of Algorithms is the process of evaluating the performance of an algorithm, typically in terms of its ~~too~~ time and space complexity. This involves understanding how the algorithms running time and memory usage scale as the input size increases.

Concepts of Analysis of Algorithms:

1. Time Complexity:

The amount of time an algorithm takes to complete usually expressed as a function of the input size.

2. Space Complexity:

The amount of memory an algorithm uses usually expressed as a function of the input size.

3. Big O Notation:

A mathematical notation used to describe the upper bound of an algorithm's time or space complexity.

4. <u>Best Case:</u>

The minimum amount of time or space an algorithm requires, usually occurring when the input is in a specific order or has a specific structure.

5. <u>Average case:</u>

The expected amount of time or space an algorithm requires, usually ~~occurs~~ calculated by averaging the time or space required for a large number of inputs.

6. <u>Worst case:</u>

The maximum amount of time or space an algorithm requires usually occurring when the input is in a specific order or has a specific structure that causes the algorithm to perform poorly.

<u>Some common time complexities include:</u>

* O(1) - constant time
* O(log n) - Logarithmic time
* O(n) - Linear time
* O(n log n) - Linearithmic time
* O(n^2) - quadratic time
* O(2^n) - exponential time

By analyzing an algorithm's time and space complexity, we can:

* predict its performance on large inputs
* Compare the efficiency of different algorithms
* Identify bottlenecks and areas for optimization

Asymptotic notations:

Asymptotic notations are used to describe the growth rate of an algorithm's time or space complexity as the input size increases.

Here are the most common asymptotic notations:

1. Big O ($O(n)$):

Upper bound, worst case scenario. It gives the maximum time or space an algorithm can take.

2. Big Ω ($\Omega(n)$):

Lower bound, best case scenario. It gives the minimum time or space an algorithms can take.

3. Big θ ($\theta(n)$):

Exact bound, average-case scenario. It gives the exact time or space an algorithm takes.

4. Little o ($o(n)$):

Loose upper bound. It gives a upper bound that is not tight.

5. Little ω ($\omega(n)$):

Loose lower bound. It gives a

lower bound that is not tight.

Important points;

1. Big O is the most commonly used notation.
2. Big O and Big Ω can be used to describe both time and space complexity.
3. Big Θ is used when the upper and lower bounds are the same.
4. Little o and Little ω are used when the bounds are not tight.

Example;

```
....
def best_case (n):
    # Best-case scenario: Ω(n)
    for i in range (n):
        print (i)
def average_case (n):
    # Average-case scenario: Θ (n log n)
    for i in range (n):
        for j in range (int (n/2)):
            print (i,j)
def worst_case (n):
    # worst-case scenario: O (n^2)
    for i in range (n):
        for j in range (n):
            print(i,j)
```

Test the functions

```
best_case (5)
average_case (5)
worst_case (5)
```

Benefits :

1. simplifies analysis:

Asymptotic notations simplify the analysis of algorithms by focusing on the growth rate rather than the exact running time.

2. ~~Hides Constants~~ Focus on scalability:

Asymptotic notations help us understand how an algorithm's running time or space usage scales as the input size increases.

3. Hides Constant:

Asymptotic notations ignore constant factors, which can vary depending on the implementation and hardware

4. Comparability:

Asymptotic notations provide a common language for comparing the efficiency of different algorithms.

5. predicts performance:

Asymptotic notations help predict an algorithm's performance on large inputs even if we can't measure it directly.

6. Helps in optimization:

Asymptotic notations identify performance bottlenecks and guide optimization efforts.

7. Theoretical foundations:

Asymptotic notations provide a theoretical foundation for understanding algorithmic complexity.

8. platform independence:

Asymptotic notations are platform independent, making them useful for analyzing algorithms across different hardware and software environment.

Divide and Conquer:

It is a popular algorithmic paradigm used to solve Complex problems by breaking them down into smaller sub-problems, solving each sub program, and then combining the solutions to the original problem.

Three Main steps:

1. Divide:

Divide the problem into smaller sub problems that are more manageable and similar in structure to the original problems.

2. Conquer:

Solve each sub-problem recursively or iteratively using the same approach.

3. Combine:

Combine the solutions to the sub-problems to solve the original problem.

Example:

```
def merge_sort (arr):
  # Divide
  if len(arr)<=1:
      return arr

  mid = len(arr) //2
  left_half = arr[:Mid]
  right_half = arr[mid:]

  # Conquer
  left_half = merge_sort (left_half)
  right_half = merge_sort(right_half)

  # Combine
  return merge (left_half, right_half)

def merge (left, right):
    merged = []
    left_index = right_index = 0
  while left_index < len (left) and right_index < len(right
    if left [left_index] <= right [right_index]:
        merged.append (left [left_index])
        left_index += 1
  else:
        merged.append(right[right_index])
        right_index += 1
```

```
    merged.extend(left[left_index:])
    merged.extend(right[right_index:])
    return merged

# Test the program
arr = [5, 2, 8, 3, 1, 6, 4]
print("original array:", arr)
print("sorted array:", merge_sort(arr))
```

output:

original Array: [5, 2, 8, 3, 1, 6, 4]
Sorted Array: [1, 2, 3, 4, 5, 6, 8]

Benefits:

1. Efficient: often have a time complexity of O(n log n) or better.
2. Scalable: can be applied to large problems by breaking them down into smaller sub-problems.
3. Easy to implement: Recursive solutions can be easier to understand and implements.

drawbacks:

1. Recursive overhead: Recursive solutions can have overhead due to functions calls and returns.
2. Difficulty in solving small problems: Some algorithms may have difficulty solving small sub-problems efficiently.

Recursion:

Recursion is a programming technique where a function calls itself repeatedly until it reaches a base case that stops the recursion. Here's a breakdown of recursion:

1. Base case:

A trivial case that can be solved without recursion.

2. Recursive case:

A case that can be broken down into smaller sub-problems of the same type which are then solved recursively.

3. Recursive call:

The function calls itself with a smaller input or a modified version of the original input.

Benefits:

1. Simplified code:

Recursion can lead to more elegant and concise code.

2. Divide and conquer:

Recursion naturally implements the divide and conquer approach.

3. Tree traversals:

Recursion is well-suited for traversing tree-like data structure.

drawbacks:

1. stack overflow:

Deep recussion can lead to a stack overflow error.

2. Inefficiency:

Recursion can be slower than iterative solutions due to function call overhead.

3. Difficulty in debugging:

Recursion can make debugging more challenging due to the repeated function calls.

Example:

Calculates the factorial of a number:

```
def factorial (n):
    # Base case
if n==0 or n==1
    return 1
 # Recursive case
 else:
    return n*factorial (n-1)
 print (factorial (5))
```

output:

120

The factorial function calls itself with decreasing values of n until it reaches the base case (n==0 or n==1) at which point it starts returning the results back up the call stack.

Analysing recursive Algorithms:

1: **Base case:**

Identify the trivial case that stops the recursion

2. **Recursive case:**

Understand how the problem is broken down into smaller sub-problems.

3. **Recursive call:**

Analyze the function call with a smaller input or modified original input.

4. **Termination:**

Ensure the recursion terminates (base case is reached).

5. **Time complexity:**

Calculate the time complexity (E.g. O(n), O(2^n)).

6. **Space complexity:**

Determine the space complexity (E.g. O(n), O(1)).

7. **optimization:**

Look for opportunities to optimize the recursive algorithm. (E.g. memoization, dynamic programming.)

Some common techniques for analyzing recursive algorithms include;

1. <u>Recursion tree:</u>

Visualize the recursive calls as a tree to understand the algorithm's behavior,

2. <u>Recursion depth:</u>

Analyze the maximum depth ~~of~~ the recursive calls.

3. <u>Function call overhead:</u>

~~Evaluate the~~ consider the overhead of repeated function calls.

4. <u>Cache performance:</u>

Evaluate the impact of recursion on cache performance.