תוכן העניינים

C++

Python

Design Patterns

- **Behavioral Design Patterns,** these patterns define how objects interact with each other and manage their communication in a clean and organized way. They focus on the flow of control and information among different objects to achieve a specific task. Examples of commonly used **Behavioral patterns:**
  - **Iterator:**
    Purpose: Provides a way to access elements of a collection (like arrays or lists) sequentially without exposing the underlying structure.
    Use Case: When you need to iterate over a collection of objects without knowing the details of its implementation
    Key Words (for when to use iterators):
    מעבר על אוסף, גישה רציפה, רשימות, סדרה של אובייקטים.
    Example:

```
template <typename T>
//The base class we intend using iterator for
class MyClass{
public:
    class Iterator {
    public:
        // Constructor
        Iterator(T* ptr) : ptr(ptr) {}

        // Overloading * operator to access the value
        T& operator*() {return *ptr;}

        // Overloading ++ operator for pre-increment
        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        // Overloading != operator to compare two iterators
        bool operator!=(const Iterator& other) const {
            return ptr != other.ptr;
        }

private:
        T* ptr;  // Pointer to the current element
    };
    // We are outside iterator class
    // Begin iterator (points to the first element)
    Iterator begin() {return Iterator(data);}

    // End iterator (points to one past the last element)
    Iterator end() {return Iterator(data + size);}
};
```

- **Strategy:**

  Purpose: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. The strategy pattern lets the algorithm vary independently from clients that use it.

  Use Case: When a class needs to perform a behavior (algorithm), but you want to allow different behaviors to be defined and swapped at runtime.

  Key Words (for when to use strategy):

  שינוי התנהגות, בחירה בזמן ריצה, פעולות משתנות לפי ההתנהגות הנקבעת.

  Example:

```
// Strategy Interface
class Strategy {
public:
    virtual ~Strategy() = default;
    virtual int execute(int a, int b) const = 0; // Pure virtual
function
};
// First strategy
class AddStrategy : public Strategy {
public:
    int execute(int a, int b) const override {
        return a + b;
    }
};
// Second strategy
class MultiplyStrategy : public Strategy {
public:
    int execute(int a, int b) const override {
        return a * b;
    }
};
// A Class that uses Strategy
class Calculator {
private:
    std::unique_ptr<Strategy> strategy; // Pointer to Strategy
public:
    // Constructor that accepts a strategy
    explicit Calculator(std::unique_ptr<Strategy> strategy) :
strategy(std::move(strategy)) {}
    // Set a new strategy at runtime
    void setStrategy(std::unique_ptr<Strategy> newStrategy) {
        strategy = std::move(newStrategy);
    }
    // Execute the strategy
    int calculate(int a, int b) const {
        return strategy->execute(a, b);
    }
};
```

- ## Command:

  Purpose: Encapsulates a request (פעולה) as an object that contains all the needed information to perform this request, therefore allowing us to create store these requests objects in lists, queues....
  Use Case: Used when different objects handle some requests differently, or what requests an object should do or is allowed to do.
  Key Words (for when to use command):
  תורים של בקשות, ביטול פעולה (Undo), עצירת ביצוע.

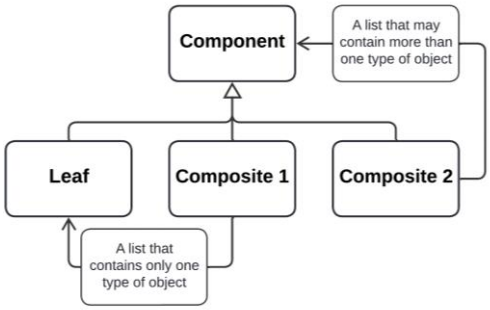
  Example:

```
// Receiver Class
class Light {
public:
    void turnOn() {}
    void 3urnoff() {}
};
// Command Interface
class Command {
public:
    virtual ~Command() = default;
    virtual void execute() const = 0; // Pure virtual method for
executing the command
};
// Concrete Command to turn on the light
class LightOnCommand : public Command {
private:
    Light& light;
public:
    explicit LightOnCommand(Light& light) : light(light) {}
    void execute() const override {light.turnOn();}
};
// Concrete Command to turn off the light
class LightOffCommand : public Command {
private:
    Light& light;
public:
    explicit LightOffCommand(Light& light) : light(light) {}
    void execute() const override {light.turnOff();}
};
// Invoker Class
class RemoteControl {
private:
    std::vector<std::unique_ptr<Command>> commands;
public:
    void addCommand(std::unique_ptr<Command> command) {
        commands.push_back(std::move(command));
    }
    void pressButton() const {
        for (const auto& command : commands) {command->execute();}
    }
};
```

- **Structural Design Patterns,** these patterns focus on how objects and classes are structured and composed to form larger structures. They help to organize code by forming relationships between entities, making systems easier to understand and maintain. Examples of commonly used **Structural patterns:**
  - **Composite:**

    Purpose: Composing objects that have a tree-like structure (explained in **structure** section). This pattern allows clients to treat individual objects that represent different items in the same category uniformly.

    Structure:

    - ❖ **Component**: A class (can be Abstract or interface) that defines the common interface for both **leaf** nodes and **composite** nodes.
    - ❖ **Leaf:** Represents the end objects in the hierarchy that do not have any children. They implement the Component's interface.
    - ❖ **Composite:** Represents an object that contains a list of **leaves** and/or **composites**. It also implements the Component's interface.

    Use Case: When you want to treat different but related objects in the same way, or group them in one list.

    Key Words (for when to use composite):

    טיפול אחיד, מכיל רשימת אובייקטים שונים.

    Example:

```
class Musicians{
    public:
    virtual string playSong(string songName) const = 0;
    private:
    string name;
};
class Singer : public Musicians{
    public:
    string playSong(string songName) const override{}
};
class PianoPlayer : public Musicians{
    public:
    string playSong(string songName) const override{}
};
class Band : public Musicians{
    public:
    string playSong(string songName) const override{
        string song = "";
        for (int © = 0; © < int(bandMembers.size()); i++)
        {song += bandMembers[i].playSong(songName);}
        return song;
    }
    private:
    vector<Musicians> bandMembers;
};
```

- **Adapter:**

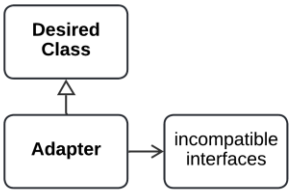
  Purpose: Allowing incompatible interfaces to work together.

  Use Case: When you have existing classes that don't match the interface you need, and you cannot or don't want to modify them.

  Key Words (for when to use adapter):

  חוסר תאימות, אין לשנות את המחלקה, שינוי ממשק. חיבור ממשקים,

  Example:




```
// Existing Class (cannot be changed)
class Car {
public:
    virtual ~Car() = default;
    virtual void drive() const = 0; // Pure virtual function
};

// New Class (cannot be changed)
class ElectricCar {
public:
    void startElectricEngine() const {
        std::cout << "Electric engine started." << std::endl;
    }

    void driveElectric() const {
        std::cout << "Driving electric car silently." << std::endl;
    }
};

// Adapter Class that makes ElectricCar compatible with Car
class ElectricCarAdapter : public Car {
public:
    explicit ElectricCarAdapter(ElectricCar* ec):electricCar(ec){}

    void drive() const override {
        // Adapting ElectricCar's functionality
        electricCar->startElectricEngine();

        // Adapted method to fit the Car interface
        electricCar->driveElectric();
    }

private:
    // Pointer to an ElectricCar instance
    ElectricCar* electricCar;

};
```

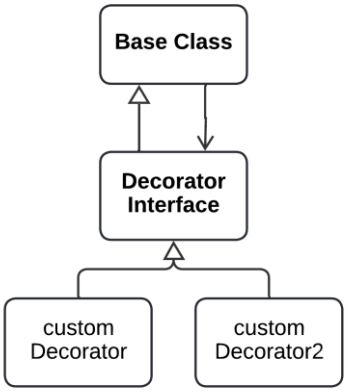
- **Decorator:**

  Purpose: Adds behavior or responsibilities to an object dynamically without altering its structure.

  Use Case: When you want to add functionality to objects without affecting other objects of the same class.

  Key Words (for when to use decorator):

  הוספת תכונות, הרחבה דינמית, גמישות בתכנון, שיפור התנהגות



Example:

```
// Simple coffee class
class Coffee {
public:
    virtual string getDescription() const{
        return “Simple Coffee”;
    }
};

// Base Decorator
class CoffeeDecorator : public Coffee {
public:
    explicit CoffeeDecorator(Coffee* c) : coffee© {}

    // Delegate to the wrapped coffee
    string getDescription() const override {
        return coffee->getDescription();
    }

protected:
    Coffee* coffee; // Pointer to a Coffee object
};

// Concrete Decorator for Milk
class MilkDecorator : public CoffeeDecorator {
public:
    explicit MilkDecorator(Coffee* c) : CoffeeDecorator© {}

    //Add milk to description
    string getDescription() const override {
        return CoffeeDecorator::getDescription() + “, Milk”;
    }
};

// you can add more concrete decorators
```

- **Creational Design Patterns,** these patterns provide ways to create different objects according to some parameters and with some conditions. They help manage object creation, ensuring that the right types of objects are created and that the process is efficient and flexible. Examples of commonly used **Creational patterns:**
  - **Factory:**
    Purpose: Providing a method for creating objects that let subclasses of **Factory** class change the type of objects that will be created.
    Use Case: When we want to create different classes (most of the times subclasses for a Base class, for example, the Base class Job and his subclasses teacher, writer, singer) according to inputs or parameters that are changing during runtime.
    Key Words (for when to use factory):
    יצירת אובייקטים לפי פרמטרים, מחלקות שונים, סוגים משתנים

    Example: creating a factory that generates candy based on the inputted type:

```
enum CandyTypes{gummybears, icecream};

class Candy{};
class GummyBears:public Candy{};
class IceCream:public Candy{};

class CandyFactory{
    public:
    Candy* createCandy(CandyTypes type) const{
        Candy* candy;
        switch (type){
            case (CandyTypes)gummybears:
                candy = new GummyBears();
            case (CandyTypes)icecream:
                candy = new IceCream();
            return candy;
        }
    }
};
```

```
int main() {
    CandyFactory* candyFactory = new CandyFactory();
    Candy* gummyBears = candyFactory-
>createCandy((CandyTypes)gummybears);
    Candy* iceCream = candyFactory-
>createCandy((CandyTypes)icecream);
     return 0;
}
```

- **Abstract Factory:**

  Purpose: Providing an interface for creating related objects from the same family or dependent objects without specifying their concrete classes.

  Use Case: When the system needs to be independent of how its products are created, composed, or represented.

  Key Words (for when to use abstract factory):

  משפחות של מוצרים, יצירה מופשטת, גמישות במימוש, הפרדה ממימוש קונקרטי

  Example: Given the classes: in the first section you will have to build an Abstract factory class like shown in the second section, and then we can run the program as shown in the third section.

```
Class Weapon{};
class Bow:public Weapon{};
class Wand:public Weapon{};

class Shield{};
class WoodenShield:public Shield{};
class MagicShield:public Shield{};

class Warrior{
    public:
    void setWaeopn(Weapon* weapon){
        this->w_weapon = weapon;
    }
    void setShield(Shield* shield){
        this->w_shield = shield;
    }
    private:
    Weapon* w_weapon;
    Shield* w_shield;
};
class Archer:public Warrior{};
class Magician:public Warrior{};
```

```
class WarriorFactory{
    public:
    virtual ~WarriorFactory() = default;
    virtual Weapon* createWeapon() const = 0;
    virtual Shield* createShield() const = 0;
};

class ArcherFactory:public WarriorFactory{
    public:
    Weapon* createWeapon() const override {
        return new Bow();
    }
    Shield* createShield() const override {
        return new WoodenShield();
    }
};

class MagicianFactory:public WarriorFactory{
    public:
    Weapon* createWeapon() const override {
        return new Wand();
    }
    Shield* createShield() const override {
        return new MagicShield();
    }
};
```

```
int main() {
    WarriorFactory* factory = new ArcherFactory();
    Weapon* weapon = factory->createWeapon();
    Warrior* archer = new Archer();
    archer->setWaeopn(weapon);
    delete weapon;
    delete factory;

    factory = new MagicianFactory();
    Shield* shield = factory->createShield();
    Warrior* magician = new Archer();
    magician->setShield(shield);
    delete shield;
    delete factory;
    return 0;
}
```

- **Singleton:**

  Purpose: Ensures that a class has only one instance and provides a global point of access to that instance.

  Use Case: When exactly one object is needed to coordinate actions across the system.

  Key Words (for when to use singleton):

  מופע יחיד, גישה גלובלית, ניהול מצב משותף, בקרת גישה

  How to use:

```
//get instance give you a reference to the object,
//and then you can use any method that is in the Game class
Game::getInstance().function();
```

How to create a Singleton class: there is two common ways,

```
//option 1:
class Game{
    public:
    static Game& getInstance() {
        if (instance == nullptr) {
            instance = new Game(); // this is called lazy initializion
        }
        return *instance;
    }
    //you cant clear this object but using this way
    static void destroy() {
        delete instance;
        instance = nullptr;
    }
    //there can be only one object, so we forbid users from copying and
    // assign new one
    Game(const Game& other) = delete;
    Game& operator=(const Game& other) = delete;

    private:
    static Game* instance; // this will be initialized to nullptr
     // the constructor should be private so users can't create new Game objects
    Game(){}
};

//option 2:
class Game {
    public:
    static Game& getInstance() {
        static Game instance; //created on first use (lazy initialization)
        return instance; // guaranteed to be destroyed after exiting the block
    }
    //there can be only one object, so we forbid users from copying and
    // assign new one
    Game(const Game& other) = delete;
    Game& operator=(const Game& other) = delete;

    private:
     // the constructor should be private so users can't create new Game objects
    Game(){}
};
```

STL

Iterators(Vector, Set, Map):

- object.begin() // returns an iterator to the beginning of the vector
- object.end() // returns an iterator to the end of the vector
- object.rbegin() // returns a reverse iterator starts from the end
- object.rend() // returns a reverse iterator to the end of the reversed vector
- object.cbegin() // returns a const iterator to the beginning of the vector
- object.cend() // returns a const iterator to the beginning of the vector

```
// how to use iterators
for(auto it = object.begin(); it != object.end(); ++it){cout << *it << endl}
```

Vector: usually used when we need a dynamic array.

- **Initializing**:

```
vector<int> vector;//Empty vector of integers
vector<int> vector (5);//Vector with 5 default-initialized elements
vector<int> vector (5, 10);//Vector with 5 elements, each initialized to 10
```

- **Adding Elements**:
  - vector.push_back(value) // Adds an element to the end of the vector.
- **Accessing Elements**:
  - int first = vector.front(); // First element
  - int last = vector.back(); // Last element
  - int element = vector [index]; // the (index + 1) element (0-based indexing)
- **Size:**
  - int count = int(vector.size()); // Number of element
    // Warning! if we don't cast it to int vector.size() will return a size_t object
  - bool isEmpty = vector.empty(); // Check if empty
- **Modifying Elements**:
  - vector.insert(vec.begin(), value); // Inserts a value at the beginning
  - vector.erase(vec.begin() + 1); // Remove the second element
  - vector.clear(); // Remove all elements
  - vector.resize(10); // Resize to 10 elements

For Set and Map: we can use the .count(value) method for sets it returns the amount of values that match the input and for map it trurns the amount of keys that match the input but we know that set values and map values are unique therefore we can use them to check if the input is already in the set or map or not .

Map: Usually used when we need key-value pairs with unique keys.

- **Initializing:**

```
map<int, std::string> myMap;   // Empty map
map<int, std::string> myMap = {{1, “one”}, {2, “two”}};
// Map with initial values
```

- **Adding/Modifying Elements:**
  - map.insert({3, “three”}); // Adds the key-value pair (3, “three”)
  - map [4] = “four”; // Adds or updates the key-value pair (4, “four”)
  - map[‘key’]; // if a key with the inserted value exists return a reference to its value otherwise it creates a new value using the value default constructor and the return a reference to its value.
- **Removing Elements:**
  - map.erase(2); // Removes the element with key 2
  - map.clear(); // Removes all elements
- **Searching:**

```
auto it = map.find(3);
if (it != map.end()) {
    cout << “Key 3 found with the value: “ << it->second << endl;
}
//it->first points to the key, and it->second points to the value
```

- **Accessing Elements:**
  - string value = map.at(1); // Access the value associated with key 1
- **Size:**
  - int count = int(map.size()); // Number of elements
    // Warning! if we don’t cast it to int map.size() will return a size_t object
  - bool isEmpty = map.empty(); // Check if empty

Set: usually used when we need a collection of unique elements.

- **Initializing:**

```
std::set<int> mySet;                // Empty set of integers
std::set<int> mySet = {1, 2, 3};    // Set initialized with elements 1, 2, 3
```

- **Adding Elements:**
  - set.insert(10); // Adds 10 to the set (if not already in set)
- **Removing Elements:**
  - set.erase(5); // Removes the element 5
  - set.clear(); // Removes all elements
- **Searching:**

```
if(mySet.find(value)!= mySet.end()){cout << "value in set" << std::endl;}
```

- **Size:**
  - Int count = int(set.size()); // Number of elements
    // Warning! if we don't cast it to int set.size() will return a size_t object
  - bool isEmpty = set.empty(); // Check if empty

Queue:

- **Initializing:**

```
queue<int> q;
```

- **Adding/Modifying Elements:**
  - q.push(10);
- **Removing Elements:**
  - q.pop();
- **Searching:**

```
auto it = map.find(3);
if (it != map.end()) {
    cout << "Key 3 found with the value: " << it->second << endl;
}
```

- **Accessing Elements:**

```
int frontElement = q.front(); // Gets the element at the front of the queue
int backElement = q.back();  // Gets the element at the back of the queue
```

- **Size:**
  - int queueSize = int(q.size()); // Gets the number of elements in the queue
    // Warning! if we don't cast it to int q.size() will return a size_t object
  - bool isEmpty = q.empty(); // Checks if empty

Pair: A simple container to store two different objects as a single unit.

```
Pair<int, string> myPair(1, “one”);
int num = myPair.first;    // Accessing the first element
string str = myPair.second; // Accessing the second element
```

Tuple: Similar to pair but can store more than two objects.

```
Tuple<int, string, double> myTuple(1, “one”, 3.14);
auto [num, str, pi] = myTuple; // Structured binding
```

Smart Pointers

unique_ptr: Owns a dynamically allocated object, and no two unique pointers can own the same object. Transfers ownership using move().

```
Unique_ptr<int> p1 = make_unique<int>(10);
unique_ptr<int> p2 = move(p1); // p1 is now empty
```

shared_ptr: Allows multiple pointers to manage the same object. Keeps a reference count.

```
Shared_ptr<int> sp1 = make_shared<int>(20);
shared_ptr<int> sp2 = sp1; // Both point to the same object
```

Lambda Expression

Basic syntax:

Capture List [capture]: Specifies which variables from the surrounding scope are accessible inside the lambda. This can include by value ([x]), by reference ([&x]), or a combination ([=, &x]).
Parameters (parameters): A list of parameters for the lambda, similar to a function’s parameters.
Return Type -> return_type: Specifies the return type of the lambda. This is optional; if omitted, the compiler infers the return type.
Function Body: The code to be executed when the lambda is called.

```
[capture](parameters) -> return_type {
      // function body
}
```

3. **Basic Lambda:**

```
auto greet = []() {
    cout << “Hello, World!” << endl;
};
greet();
```

2. **Lambda with Parameters:**

```
auto add = [](int a, int b) -> int {
 return a + b;
};
int result = add(5, 3); // result is 8
```

3. **Lambda with Capture List:**

```
int x = 10;
auto byValue = [x]() {
    cout << “Captured by value: “ << x << endl;
};

auto byReference = [&x]() {
    cout << “Captured by reference: “ << x << endl;
};

byValue();          // Outputs: Captured by value: 10
byReference();    // Outputs: Captured by reference: 10

x = 20;
byReference();
// Outputs: Captured by reference: 20
(captured by reference, so value changes)
```

Function as a parameter

To use a function as a parameter first you should define the function to accept another function

```
template<class Function>
void process(int x, Function func) {
     func(x);
}
```

And then you can put a lambda as you inputted function

i/o stream

When using a file, you should always make sure to open the file before you use it and close the file after you are done with it. Bad file management can cause similar problems as bad memory management.

We read and write from files using >> and << operators

For reading from a file we can use the getline() function that has two variations:

istream& getline (istream& istream, string& str);

// istream, is the file we want to read from, str is where the line will be copied.

istream& getline (istream& is, string& str, char delimiter);

//it is almost like the function up there, but you can pick to stop after reading a specific //char instead of the default being (new line (‘\n’)).

Constructors: //constructors get the filename and open the requested file so we can use it.

ofstream file("output.txt");

ifstream file("input.txt");

Warning! fstream objects can’t be copied, if we want to send the object to a function, we send it as a reference or send a pointer to the object.

Exceptions

```
class MyException : public exception {
       const char* what() const override {
              return “Error message”;
       }
}
```

Python

Exciptions

```
class UserNameAlreadyExists(Exception):
    def __init__(self, username):
       super().__init__(f"Username '{username}' already exists.")
       self.username = username
```

JSON

to store an object in a JSON file you should cast the object to a dictionary and then saving each item inside this dictionary, and then you use json.dump() to put your item inside a file:

```
// storing each item in a dictionary
def item_to_dict(item):
    item_dect = {'name': item.name, 'weight': item.weight}


// storing the bag detaild and the items list in a disctionary
def store_bag(bag, path)
    data = {
        'capacity': bag.capacity,
        'weight': bag.weight,
        'items': [{'name':item.name, 'weight':item.weight}
                  for item in bag.items]
    }
    with open(path, 'w') as file:
        json.dump(data, file)
```

File Handling

Opening a file:

```
file=open('filename.txt', 'r')# Open file for reading (default mode)

//using with.
with open('filename.txt', 'r') as file:
    content = file.read()
    # Process the content
# File is automatically closed after the block
```

Closing a file:

```
file.close()
```

Reading from a file:

```
# Read the entire file
content = file.read()

# Read one line at a time
line = file.readline()

# Read all lines into a list
lines = file.readlines()
```

Writing to a file:

```
file = open('filename.txt', 'w')  # Open file for writing

file.write('Hello, World!\n')
file.writelines(['Line 1\n', 'Line 2\n'])

file.close()  # Always close the file when done
```

os.path(file_name); //returns the path to the file name

os.listdir(dir_name); //returns a list of the files that are in the dir

if you has a folder that contains files you can access each file using list comprehension as follows:

[os.path.join(dir_name, r) for r in os.listdir(dir_name)];

List Comprehension

Basic Syntax:

expression: The value or computation to include in the new list.
item: The variable representing each element in the iterable.
iterable: The collection or sequence to iterate over (e.g., list, range).
condition (optional): A filter that determines if the item should be included in the new list.

```
[expression for item in iterable if condition]
```

Dictionary: A collection of key-value pairs, where each key is unique.

- **Initializing:**

```
dictionary = {}  # Empty dictionary
dictionary = {'key1': 'value1', 'key2': 'value2'}
# Dictionary with initial key-value pairs
```

- **Adding/Modifying Elements:**

```
dictionary['new_key'] = 'new_value'
# Adds a new key-value pair or updates an existing key
```

- **Removing Elements:**

```
dictionary.pop('key1')  # Removes the key-value pair by key
dictionary.clear()  # Removes all elements
```

- **Accessing Elements:**

```
value = dictionary['key1']  # Access value by key
value = dictionary.get('key1', 'default_value')
# Access value with a default if key doesn't exist
```

- **Size:**

```
count = len(dictionary)  # Number of key-value pairs
isEmpty = not dictionary  # Check if empty
```

List:

- **Initializing:**

```
lst = []  # Empty list
lst = [1, 2, 3, 4]  # List with initial elements
```

- **Adding/Modifying Elements:**

```
lst.append(value)  # Adds an element to the end of the list
lst.insert(index, value) # Inserts an element at a specific position
lst[index] = new_value  # Updates the value at a specific index
lst.sort()  # Sorts the list in ascending order
```

- **Removing Elements:**

```
lst.remove(value)  # Removes the first occurrence of the value
lst.pop(index)  # Removes the element at the specified index
lst.clear()  # Removes all elements
```

- **Accessing Elements:**

```
first = lst[0]  # First element
last = lst[-1]  # Last element
element = lst[index]  # The (index + 1) element (0-based indexing)
```

- **Size:**

```
count = len(lst)  # Number of elements
isEmpty = not lst  # Check if empty
```

Set:

- **Initializing:**

```
my_set = set()  # Empty set
my_set = {1, 2, 3, 4}  # Set with initial elements
```

- **Adding/Modifying Elements:**

```
my_set.add(value)  # Adds an element to the set (duplicates are
ignored)
```

- **Removing Elements:**

```
my_set.remove(value)  # Removes the specified value (raises KeyError
if not found)
my_set.discard(value)  # Removes the value if it exists (no error if
not found)
my_set.clear()  # Removes all elements
```

- **Accessing Elements:**

```
exists = value in my_set  # Check if a value exists in the set
```

- **Size:**

```
count = len(my_set)  # Number of elements
isEmpty = not my_set  # Check if empty
```

String:

- **Initializing:**

```
str1 = ""  # Empty string
str1 = "Hello, World!"  # String with initial content
```

- **Adding Elements:**

```
str2 = str1 + " More text"  # Concatenate strings
```

- **Modifying Elements:**

```
new_str = str1.replace("Hello", "Hi")  # Replace part of the string
new_str = str1.lower()  # Convert to lowercase
new_str = str1.upper()  # Convert to uppercase
```

- **Accessing Elements:**

```
first_char = str1[0]  # First character
last_char = str1[-1]  # Last character
substring = str1[start:end]  # Slice the string from 'start' to
'end'
```

- **Size:**

```
count = len(str1)  # Number of characters
isEmpty = not str1  # Check if empty
```

Tuple:

- **Initializing:**

```
tpl = ()  # Empty tuple
tpl = (1, 2, 3, 4)  # Tuple with initial elements
tpl = (1,)  # Single-element tuple (comma is required)
```

- **Modifying Elements:**

```
new_tpl = tpl + (5, 6)  # Concatenate tuples to add new elements
new_tpl = tpl[:2] + (5,) + tpl[2:]  # Insert an element at a
specific index
```

- **Accessing Elements:**

```
first = tpl[0]  # First element
last = tpl[-1]  # Last element
element = tpl[index]  # The (index + 1) element (0-based indexing)
```

- **Size:**

```
count = len(tpl)  # Number of elements
isEmpty = not tpl  # Check if empty
```

Matam-100

by: sari zrieq