ADVANCED ASSIGNMENT 4

Q1. Which two operator overloading methods can you use in your classes to support iteration?

ANSWER: To support iteration, a custom class needs to define two special methods:

- __iter__ and __next__ : The iter method returns an iterator object that has a next method. The next method returns the next value of the iteration.
- __getitem__ : This is evoked only if iter is not defined in a class and uses indexing and slicing for iteration.

```
class Counter:
    def __init__(self, limit:int, string:str):
        self.limit = limit
        self.st = string
        self.count = 0

    def __iter__(self):
        return self

    def __getitem__(self, i):
        return self.st[i]

    def __next__(self):
        if self.count < self.limit:
            self.count += 1
            return self.count
        else:
            raise StopIteration

c = Counter(5, "hey")
for i in c:
    print(i, end=" ")
```

Output: 1 2 3 4 5

Note: clearly, only __iter__ was called when both iter and getitem were present in the class. If __iter__ is commented out then the __getitem__ will be called and the output will be : h e y

Q2. In what contexts do the two operator overloading methods manage printing?

ANSWER:

- __str__ : it is tried first for the **print** operation and the str built-in function (the internal equivalent of which print runs). It generally returns a user-friendly display.
- __repr__ : The goal here to be unambiguous i.e., __repr__ generally returns an as-code string that could be used to re-create the object, or a detailed display for developers.
    - __repr__ is used everywhere for displaying data, except by **print** and **str** when a __str__ is defined. If no __str__ is defined, then the printing operation falls back on __repr__.

Q3. In a class, how do you intercept slice operations?

ANSWER: Slice operations are intercepted by __getitem__ by calling a slice object.

```
class Custom:
    data = [1,2,3,4,5,6,7,8,9]
    def __getitem__(self, ind):
        print('getitem: ', ind)
        return self.data[ind]
SliceObj = Custom()
SliceObj[2:4]             # Slicing sends __getitem__ a slice object

OUTPUT:getitem:  slice(2, 4, None)
[3, 4]
```

Q4. In a class, how do you capture in-place addition?

ANSWER: In-place addition += can be captured using __iadd__, but if it's not defined then __add__ can also be used.

```
class Add:
    def __init__(self, val) -> None:
        self.val = val
```

```
  def __iadd__(self, i):       # __iadd__ explicitly uses a+=b
        self.val += i
        return self

    def __add__(self, i):         # __add__ uses a = a + b
        return Add(self.val + i)

obj = Add(10)
obj += 2
obj.val
OUTPUT = 12
```

NOTE: If __iadd__ is commented out then the In-place addition will fall back on __add__.

Q5. When is it appropriate to use operator overloading?

ANSWER:

- Operator overloading can be useful when you're making a new class that falls into an existing "Abstract Base Class".
- It can improve code readability by allowing the use of familiar operators, ensure that objects of a class behave consistently with built-in types and other user-defined types, and make it simpler to write code, especially for complex data types.
- It is also used to add or customise the functionality of an existing method in the class.

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __str__(self):
        return f"{self.real}+{self.imag}i"

    def __add__(self, other):
        return Complex(self.real + other.real, self.imag + other.imag)
```

```
c1 = Complex(1, 2)
c2 = Complex(3, 4)
print(c1 + c2)
```

OUTPUT: 4+6i

Here, the __add__ has been overloaded to define how to add two complex numbers and __str__ is defined to display the summation as complex numbers.