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

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

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

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

Q5. When is it appropriate to use operator overloading?

Solutions :

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

**Ans:** **\_\_iter\_\_** and **\_\_next\_\_** are the operator overloading methods in python that support iteration and are collectively called iterator protocol.

* **\_\_iter\_\_** returns the iterator object and is called at the start of loop in our respective class.
* **\_\_next\_\_** is called at each loop increment, it returns the incremented value. Also Stopiteration is raised when there is no value to return.

In [1]:

**class** Counter:

**def** \_\_init\_\_(self,low,high):

self**.**current **=** low

self**.**high **=** high

**def** \_\_iter\_\_(self):

**return** self

**def** \_\_next\_\_(self):

**if** self**.**current **>** self**.**high:

**raise** StopIteration

**else**:

self**.**current **+=** 1

**return** self**.**current **-** 1

**for** ele **in** Counter(5,15):

print(ele, end**=**" ")

5 6 7 8 9 10 11 12 13 14 15

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

**Ans:** **\_\_str\_\_** and **\_\_repr\_\_** are two operator overloading methods that manage printing.

* In Short, the difference between both these operators is the goal of **\_\_repr\_\_** is to be unambiguous and **\_\_str\_\_** is to be readable.
* Whenever we are printing any object reference internally **\_\_str\_\_** method will be called by default.
* The main purpose of **\_\_str\_\_** is for readability. it prints the informal string representation of an object, one that is useful for printing the object. it may not be possible to convert result string to original object.
* **\_\_repr\_\_** is used to print official string representation of an object, so it includes all information and development.

In [2]:

**class** Student:

**def** \_\_init\_\_(self,name,roll\_no):

self**.**name **=** name

self**.**roll\_no **=** roll\_no

s1 **=** Student("Mano",1)

print(str(s1))

**class** Student:

**def** \_\_init\_\_(self,name,roll\_no):

self**.**name **=** name

self**.**roll\_no **=** roll\_no

**def** \_\_str\_\_(self):

**return** f'Student Name: {self**.**name} and Roll No: {self**.**roll\_no}'

s1 **=** Student("Mano",1)

print(str(s1))

**import** datetime

today **=** datetime**.**datetime**.**now()

s **=** str(today) *# converting datetime object to presentable str*

print(s)

**try**: d **=** eval(s) *# converting str back to datetime object*

**except**: print("Unable to convert back to original object")

u **=** repr(today) *# converting datetime object to str*

print(u)

e **=** eval(u) *# converting str back to datetime object*

print(e)

<\_\_main\_\_.Student object at 0x000001FDD5B05518>

Student Name: Mano and Roll No: 1

2021-11-20 18:13:47.885746

Unable to convert back to original object

datetime.datetime(2021, 11, 20, 18, 13, 47, 885746)

2021-11-20 18:13:47.885746

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

**Ans:** In a class use of slice() in \_\_getitem\_\_ method is used for intercept slice operation. This slice method is provided with start integer number, stop integer number and step integer number.

**Example:** \_\_getitem\_\_(slice(start,stop,step))

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

**Ans:** **a+b** is normal addition. Whereas **a+=b** is inplace addition operation. In this in-place addition **a** itself will store the value of addition. In a class **\_\_iadd\_\_** method is used for this in-place operation

In [3]:

**class** Book:

**def** \_\_init\_\_(self,pages):

self**.**pages **=** pages

**def** \_\_iadd\_\_(self,other):

self**.**pages **+=** other**.**pages

**return** self**.**pages

b1 **=** Book(100)

b2 **=** Book(200)

b1**+=**b2

print(b1)

300

#### Q5. When is it appropriate to use operator overloading?

**Ans:** Operator overloading is used when we want to use an operator other than its normal operation to have different meaning according to the context required in user defined function.

In [4]:

**class** Book:

**def** \_\_init\_\_(self,pages):

self**.**pages **=** pages

**def** \_\_add\_\_(self,other):

**return** self**.**pages**+**other**.**pages

b1 **=** Book(100)

b2 **=** Book(200)

print(f'Total Number of Pages -> {b1**+**b2}')

Total Number of Pages -> 300