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

Answer:

The following two operator overloading methods can be used in our classes to support iteration:

🡪 \_\_iter\_\_ it returns the iterator object and is implicitly called at the start of loop in our respective class

🡪 \_\_next\_\_ it returns the next value and is implicitly called at each loop increment

|  |
| --- |
| class Counter:  def \_\_init\_\_(self,n\_start, n\_stop):  self.start = n\_start  self.end = n\_stop    def \_\_iter\_\_(self):  return self    def \_\_next\_\_(self):  if self.start > self.end:  raise StopIteration  else:  self.start += 1  return self.start - 1    for ele in Counter(1, 25):  print(ele, end=" ") |

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

Answer:

Following two operator overloading methods manage printing:

🡪 \_\_str\_\_ this method is called by the print and str built-in functions. A \_\_str\_\_ is usually used for user-friendly displays

🡪 \_\_repr\_\_ this method is called by print and str if there is no \_\_str\_\_, and always by the repr built-in, interactive echoes, and nested appearances. A \_\_repr\_\_ gives extra details or the object’s as-code form.

|  |
| --- |
| class Student1:  def \_\_init\_\_(self,name, roll\_no):  self.name = name  self.roll\_no = roll\_no    s1 = Student1("A", 1)  print(str(s1))  class Student2:  def \_\_init\_\_(self,name, roll\_no):  self.name = name  self.roll\_no = roll\_no    # overeloading  def \_\_str\_\_(self):  return f'Student Name: {self.name} and Roll No: {self.roll\_no}'    s2 = Student2("A", 1)  print(str(s2))  import datetime  today = datetime.datetime.now()  # method-1  # converting datetime object to str  s = str(today)  print(s)  # converting str back to datetime object  try: d = eval(s)  except: print("Unable to convert back to original object")  # method-2: repr  # converting datetime object to str  u = repr(today)  print(u)  # converting str back to datetime object  e = eval(u)  print(e) |

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

Answer:

The \_\_getitem\_\_ method is used for accessing or intercept slice operation in a class. This slice is a constructor in Python that creates slice object to represent set of indices that the range (start, stop, step) specifies. The \_\_getitem\_\_ method can be implement in a class, and the behavior of slicing can be defined inside it.

Example:

s ='debabrata'.\_\_getitem\_\_(slice(0, 4, 1))

print(s)

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

Answer:

As we can see that Python language provides the operator x += y to add two objects or variable in-place by calculating the sum of x and y and assigning the result to the first operands variable called x. We can set up the same in-place addition behavior for a class by overriding the dunder method named \_\_iadd\_\_(self, other) in the class definition.

|  |
| --- |
| class BookPrice:  def \_\_init\_\_(self, price):  self.price = price    def \_\_iadd\_\_(self, other):  self.price += other.price  return self.price    b1 = BookPrice(150)  b2 = BookPrice(200)  b1 += b2  print(b1) |

Q5. When is it appropriate to use operator overloading?

Answer:

With the help of operator overloading, we can redefine the majority of the operators and also we can use operator overloading to perform different operations using one operator.

|  |
| --- |
| class BookPrice:  def \_\_init\_\_(self, price):  self.price = price    def \_\_add\_\_(self, other):  return self.price + other.price    b1 = BookPrice(150)  b2 = BookPrice(200)  tot = b1 + b2  print(f'total book price: {tot}') |