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

The \_\_iter\_\_ returns the iterator object and is implicitly called at the start of loops. The \_\_next\_\_ method returns the next value and is implicitly called at each loop increment. \_\_next\_\_ raises a StopIteration exception when there are no more value to return, which is implicitly captured by looping constructs to stop iterating.

## 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 num in Counter(5, 15):

## print(num)

## 

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

If you’ve used the + or \* operator on a str object in Python, you must have noticed its different behavior when compared to int or float objects:

## >>> # Adds the two numbers

## >>> 1 + 2

## 3

## >>> # Concatenates the two strings

## >>> 'Real' + 'Python'

## 'RealPython'

## >>> # Gives the product

## >>> 3 \* 2

## 6

## >>> # Repeats the string

## >>> 'Python' \* 3

## 'PythonPythonPython'

**Operator Overloading**  : means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behaviour objects of different classes, this is called *Operator Overloading*.

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

The “ **\_\_getitem\_\_ “** method is used for accessing list items, array elements, dictionary entries etc. **slice** is a constructor in Python that creates slice object to represent set of indices that the range(start, stop, step) specifies. \_\_getitem\_\_ method can be implement in a class, and the behavior of slicing can be defined inside it.

**Syntax:**

\_\_getitem\_\_(slice(start, stop, step))

Program

sliced **=**'abcde'.\_\_getitem\_\_(slice(0, 2, 1))

print(sliced)

**Explanation:**  
The string “ **abcde “** is sliced with starting index 0 and stop index 2 with step index 1 hence it slices **ab** from the string and prints the output.

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

Python provides the operator x += y to add two objects in-place by calculating the sum x + y and assigning the result to the first operands variable name x . You can set up the in-place addition behavior for your own class by overriding the magic “dunder” method \_\_iadd\_\_(self, other) in your class definition.

## >>> x = **1**

## >>> x += **2**

## >>> x

## **3**

Integer example:

## >>> x = **2**

## >>> x += **40**

## >>> x

## **42**

Float Example:

## >>> x = **2**

## >>> x += **40.0**

## >>> x

## **42.0**

**To use the in-place addition operator += on custom objects, you need to define the \_\_iadd\_\_() method (“dunder method”, “magic method”) that takes two arguments self and other, updates the first argument self with the result of the addition, and returns the updated object.**

**class** Data:

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

self.data = data

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

self.data += other.data

**return** self

x = Data(**40**)

y = Data(**2**)

x += y

print(x.data)

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

Operator overloading is mostly useful when you're making a new class that falls into an existing "Abstract Base Class" (ABC) -- indeed, many of the ABCs in standard library module collections rely on the presence of certain special methods (and special methods, one with names starting and ending with double underscores .

Here is an example that uses the bitwise or operation to simulate a unix pipeline. This is intended as a counter example to most of the rules of thumb.

class pipely(object):

def \_\_init\_\_(self, \*args, \*\*kw):

self.\_args = args

self.\_\_dict\_\_.update(kw)

def \_\_ror\_\_(self, other):

return ( self.map(x) for x in other if self.filter(x) )

def map(self, x):

return x

def filter(self, x):

return True

class sieve(pipely):

def filter(self, x):

n = self.\_args[0]

return x==n or x%n

class strify(pipely):

def map(self, x):

return str(x)

class startswith(pipely):

def filter(self, x):

n=str(self.\_args[0])

if x.startswith(n):

return x

print"\*"\*80

for i in xrange(2,100) | sieve(2) | sieve(3) | sieve(5) | sieve(7) | strify() | startswith(5):

print i

print"\*"\*80

for i in xrange(2,100) | sieve(2) | sieve(3) | sieve(5) | sieve(7) | pipely(map=str) | startswith(5):

print i

print"\*"\*80

for i in xrange(2,100) | sieve(2) | sieve(3) | sieve(5) | sieve(7) | pipely(map=str) | pipely(filter=lambda x: x.startswith('5')):

print i

Python's overloading is "safer" in general than C++'s -- for example, the assignment operator can't be overloaded, and += has a sensible default implementation.

In some ways, though, overloading in Python is still as "broken" as in C++. Programmers should restrain the desire to "re-use" an operator for unrelated purposes, such as C++ re-using the bitshifts to perform string formatting and parsing. Don't overload an operator with different semantics from your implementation just to get prettier syntax.

Modern Python style strongly discourages "rogue" overloading, but many aspects of the language and standard library retain poorly-named operators for backwards compatibility. For example:

* %: modulus and string formatting
* +: addition and sequence concatenation
* \*: multiplication and sequence repetition

.