CS 162P
Operator Overloading

Overview

When creating a new class, it is often useful to overload some of the existing operators to work with the new class. There are three different groups of operators to consider: arithmetic, comparisons, and built-in functions. For a given class, not all of them make sense. The most common overloads are built-in functions followed by comparison operators.

Here is a link to the Python Operators https://docs.python.org/3/library/operator.html.

Example Class

To demonstrate how arithmetic and comparison operators can be overloaded, consider the following simple class Square.

```
class Square:
    def __init__(self, side = 0):
        self.__side = side

    def setSide(self, side):
        self.__side = side

    def getSide(self):
        return self.__side

    def getArea(self):
        return self.__side * self.__side

    def getPerimeter(self):
        return self.__side * 4
```

Arithmetic Operators

There are many arithmetic operators available in Python that can be overloaded. The most common are addition, subtraction, multiplication, division, and integer division. Since the overloading of these are essentially the same, this example will focus on addition. The result of adding two squares is defined as a new square with sides equal to the sum of the sides of the two squares being added.

Method:

```
def __add__(self, other):
    side = self.__side + other.__side
    return Square(side)
```

Usage:

```
def main():
    square1 = Square(3)
    square2 = Square(5)
    print("square1 side is " + str(square1.getSide()))
    print("square2 side is "+ str(square2.getSide()))

    square3 = square1 + square2
    print("square3 side is "+ str(square3.getSide()))
```

Output

```
square1 side is 3
square2 side is 5
square3 side is 8
```

Explanation:

When using a binary operator such as addition, it is necessary to have two instances of the class pass to the operator. Since the operator is defined as an object method, the self refers to the instance of the class on the left side of the operator and the other parameter refers to the instance of the class on the right side.

This allows using the arithmetic operator on a data type that it has previously been defined for.

Using the object variables for self and other allows creating a new object that is returned.

Comparison Operators

It is very common to want to check if two objects are equal or if one is greater than the other. Unfortunately, using the equality operator, ==, on two objects normally results in a check to see if they are the same instance. That is, do they have the same address in memory. To fix this problem requires overriding the equality operator. Likewise, if the greater than operator is overloaded, then it allows sorting of the objects by some characteristic.

Since the Square method has a side, the comparisons will simply compare the sides of the two objects.

Methods:

```
def __eq__(self, other):
    return self.__side == other.__side

def __gt__(self, other):
    return self.__side > other.__side
```

Usage:

```
def main():
    square1 = Square(3)
    square2 = Square(5)
    square3 = Square(3)

    print("equal to")
    print(square1 == square2)
    print(square1 == square3)

    print("greater than")
    print(square1 > square3)
    print(square2 > square3)
```

Output

```
equal to
False
True
greater than
False
True
```

Explanation:

Like the binary arithmetic operators, the comparison operators require two arguments and self refers to the object to the left of the operator when used and other refers to the object to the right.

For checking equality, it is often necessary to check multiple attributes and return true only if all of them are equal. For greater than or less than, normally only a single attribute is compared. The same comparison operator is used in the body of the function as is being implemented by the method. As with arithmetic operators, this allows the operator to be used on a data type that it has previously been defined for.

Built-in Functions

Unlike the arithmetic and comparison operators, the implementation of the various built-in functions differs depending on the class. The most common overloads are for the str, repr, and len functions. Here is an example of overloading str for the Square class.

Method:

```
def __str__(self):
    return f"a square with sidelength {self.__side}"
```

Usage:

```
def main():
    square1 = Square(3)

    print(str(square1))
```

Output

```
a square with sidelength 3
```

Explanation:

The purpose of the __str__ function is to return a string that captures whatever information is needed about the class.

Enumerated Class

The LinkedList class has length and can be enumerated, so the following example will use the following definition of the class. Note that this class has been stripped of much of the functionality defined in the linked list document. It also does not deal with errors. It is a minimal version used only for these examples.

```
class LinkedList:
    def __init__(self):
        self.__head = None

    # return true if the list is empty
    def isEmpty(self):
        return self.__head is None

    # add a new link containing value at the head of the list
    def addHead(self, value):
        temp = Link(value)
        temp.setNext(self.__head)
        self.__head = temp
```

```
    # return the value contained in the first link of the list
    def getHead(self):
        return self.__head.getValue()

    # change head to point at the next link in the list
    def removeHead(self):
        self.__head = self.__head.getNext()
```

Representation, String, and Length

The following examples show adding a __repr__, __str__, and __len__ method to LinkedList.

Methods:

```
    # return a printable representation of the list
    def __repr__(self):
        current = self.__head
        nodes = []
        while current is not None:
            nodes.append(str(current.getValue()))
            current = current.getNext()
        nodes.append("None")
        return " -> ".join(nodes)

    # return a string containing the elements in the list
    def __str__(self):
        current = self.__head
        nodes = []
        while current is not None:
            nodes.append(str(current.getValue()))
            current = current.getNext()
        return ", ".join(nodes)

    # return the number of links in the list
    def __len__(self):
        current = self.__head
        length = 0
        while current is not None:
            length += 1
            current = current.getNext()
        return length
```

Usage:

```
def main():
    theList = LinkedList()
    theList.addHead(3)
    theList.addHead(5)
    theList.addHead(7)

    print(repr(theList))
    print(str(theList))
    print(len(theList))
```

Output

```
7 -> 5 -> 3 -> None
7, 5, 3
3
```

Enumerating LinkedList

If a program wanted to do something with each link of the linked it, it is necessary to overload the iteration methods for it. This can be done as shown below.

Methods:

```
def __iter__(self):
    self.__current = self.__head
    return self

def __next__(self):
    if self.__current is None:
        raise StopIteration
    else:
        current = self.__current
        self.__current = self.__current.getNext()
        return current
```

Usage:

```
def main():
    theList = LinkedList()
    theList.addHead(3)
    theList.addHead(5)
    theList.addHead(7)

    for link in theList:
        print(link.getValue())
```

Output

```
7
5
3
```

Explanation:

The __iter__ method is called at the start of the enumeration and sets an object variable to indicate where the traversal of the list current is. It starts at head, like all traversals. Then the __next__ method is called each time the loop gets a new value. If it is at the end of the list, it raises StopIteration, which terminates the loop. If it is not at the end, it saves the current link, updates the object variable to point to the next link in the list and returns the current link.

This is identical to the loop that is used, for example, in the len method described above.