Chapter 22
Operator Overloading

22.1 Introduction

We will explore Operator Overloading in this chapter; what it is, how it works and why we want it.

22.2 Operator Overloading

22.2.1 Why Have Operator Overloading?

Operator overloading allows user defined *classes* to appear to have a natural way of using operators such as +, −, <, > or == as well as logical operators such as & (and) and | (or).

This leads to more succinct and readable code as it is possible to write code such as:

```
q1 = Quantity(5)
q2 = Quantity(10)
q3 = q1 + q2
```

It feels more natural for both developers and those reading the code. The alternative would be to create methods such as add and write code such as

```
q1 = Quantity(5)
q2 = Quantity(10)
q3 = q1.add(q2)
```

Which semantically might mean the same thing but feel less *natural* to most people.

© Springer Nature Switzerland AG 2019

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_22

22.2.2 Why Not Have Operator Overloading?

If operator overloading is such a good idea, why don't all programming languages support it? Interestingly Java, a very widely used programming language, does not support operator overloading!

One answer is because it can be abused! For example, what is the meaning of the following code:

```
p1 = Person('John')
p2 = Person('Denise')
p3 = p1 + p2
```

It is not clear what '+' means in this context; in what way is Denise being added to John; does it imply they are getting married? If so, what is the result that is held in `p3`?

The problem here is that from a design perspective (which in this case may be purely intuitive but in other cases may relate to the intention of an application) the plus operator does not make sense for the type `Person`. However, there is nothing in the Python language to indicate this and thus anyone can code any operator into any class!

As a general design principle; developers should follow the semantics of built in types and thus should only implement those operators which are appropriate for the type being developed. For example, for arithmetic value types such as `Quantity` it makes perfect sense to provide a plus operator but for domain specific data-oriented types such as `Person` it does not.

22.2.3 Implementing Operator Overloading

To implement operators such as '+' in a user defined class it is necessary to implement specific methods that are then mapped to the arithmetic or logical operators used by users of the class.

These methods are considered *special* in that they start with and end with a double underscore (`'__'`). Such methods are considered private and usually restricted for Python oriented implementations (we have seen these already with methods such as `__init__()` and `__str__()`).

As an example, let us assume that we want to implement the '+' and '−' operators for our `Quantity` type. We also want our `Quantity` type to hold an actual value and be able to be converted into a string for printing purposes.

To implement the '+' and '−' operators we need to provide two special methods one will provide the implementation of the '+' operator and one will provide the implementation of the '−' operator:

- '+' operator is implemented by a method with the signature **def** `__add__(self, other):`
- '−' operator is implemented by a method with the signature **def** `__sub__(self, other):`

Where `other` represents another `Quantity` or other suitable type which will be either added to, or subtracted from, the current `Quantity` object.

The methods will be mapped by Python to the operators '+' and '−'; such that if someone attempts to add to quantities together then the `__add__()` method will be called etc.

The definition of the class `Quantity` is given below; note that the class actually just wraps a number held in the attribute `value`.

```
class Quantity:
    def __init__(self, value=0):
        self.value = value
    def __add__(self, other):
        new_value = self.value + other.value
        return Quantity(new_value)
    def __sub__(self, other):
        new_value = self.value - other.value
        return Quantity(new_value)
    def __str__(self):
        return 'Quantity[' + str(self.value) + ']'
```

Using this class definition, we can create two instances of the type `Quantity` and add them together:

```
q1 = Quantity(5)
q2 = Quantity(10)
print('q1 =', q1, ', q2 =', q2)

q3 = q1 + q2
print('q3 =', q3)
```

If we run this code snippet we get:

```
q1 = Quantity[5] , q2 = Quantity[10]
q3 = Quantity[15]
```

Note that we have made the class `Quantity` *immutable*; that is once a `Quantity` instance has been created its value cannot be changed (it is fixed).

This means that when two quantities are added tougher a new instance of the class `Quantity` is created. This is analogous to how integers work, if you add together `2 + 3` then you get `5`; neither `2` or `3` are modified however; instead a new integer `5` is generated—this is an example of the general design principle; developers should follow the semantics of built in types; `Quantity` objects act like number objects.

22.3 Numerical Operators

There are nine different numerical operators that can be implemented by special methods; these operators are listed in the following table:

Operator	Expression	Method
Addition	`q1 + q2`	`__add__(self, q2)`
Subtraction	`q1 – q2`	`__sub__(self, q2)`
Multiplication	`q1 * q2`	`__mul__(self, q2)`
Power	`q1 ** q2`	`__pow__(self, q2)`
Division	`q1 / q2`	`__truediv__(self, q2)`
Floor Division	`q1 // q2`	`__floordiv__(self, q2)`
Modulo (Remainder)	`q1 % q2`	`__mod__(self, q2)`
Bitwise Left Shift	`q1 << q2`	`__lshift__(self, q2)`
Bitwise Right Shift	`q1 >> q2`	`__rshift__(self, q2)`

We have already seen examples of add and subtract; this table indicates how we can also provide operators for multiplication and division etc.

The above table also presents Bitwise shift operators (both left and right). These operate at the bit level used to represent numbers under the hood and can be a very efficient way of manipulating numeric values; however, we do not want to support these operators for our `Quantity` class therefore we will only implement the core numeric operators of multiplication, division and power.

Also note that the names of the division methods are not div but `__truediv__()` and `__floordiv__()` indicating the difference in behaviour between '`/`' and '`//`'.

The updated `Quantity` class is given below:

```
class Quantity:
    def __init__(self, value=0):
        self.value = value

    def __add__(self, other):
        new_value = self.value + other.value
        return Quantity(new_value)

    def __sub__(self, other):
        new_value = self.value - other.value
        return Quantity(new_value)

    def __mul__(self, other):
        new_value = self.value * other.value
        return Quantity(new_value)

    def __pow__(self, other):
        new_value = self.value ** other.value
        return Quantity(new_value)

    def __truediv__(self, other):
        new_value = self.value / other.value
        return Quantity(new_value)

    def __floordiv__(self, other):
        new_value = self.value // other.value
        return Quantity(new_value)

    def __mod__(self, other):
        new_value = self.value % other.value
        return Quantity(new_value)

    def __str__(self):
        return 'Quantity[' + str(self.value) + ']'
```

This means that we can now extend our simple application that uses the `Quantity` class to include some of these additional numerical operators:

```
q1 = Quantity(5)
q2 = Quantity(10)
print('q1 =', q1, ', q2 =', q2)

q3 = q1 + q2
print('q3 =', q3)
print('q2 - q1 =', q2 - q1)

print('q1 * q2 =', q1 * q2)
print('q1 / q2 =', q1 / q2)
```

The output from this is now:

```
q1 = Quantity[5] ,q2 = Quantity[10]
q3 = Quantity[15]
q2 - q1 = Quantity[5]
q1 * q2 = Quantity[50]
q1 / q2 = Quantity[0.5]
```

One interesting point to note is that the multiple and divide style methods, we might want to multiple a `Quantity` by an integer or divide a `Quantity` by an integer. There is nothing to stop us doing this and indeed this might be a very useful behaviour. This would allow a `Quantity` to be multiplied by 2 or divided by 2, for example:

```
print('q1 * 2', q1 * 2)
print('q2 / 2', q2 / 2)
```

At the moment if we tried to run the above code we would generate an error telling us that an int does not have a value attribute. However, we can test to see if the argument passed into the `__mult__()` and `__truediv__()` methods is an `int` or not using the `isinstance` function. This function takes a variable and the name of a class and returns `True` if the contents of the variable is an instance of the named class, for example:

```
class Quantity:
    # Code ommitted for brevity

    def __mul__(self, other):
        if isinstance(other, int):
            new_value = self.value * other
        else:
            new_value = self.value * other.value
        return Quantity(new_value)

    def __truediv__(self, other):
        if isinstance(other, int):
            new_value = self.value / other
        else:
            new_value = self.value / other.value
        return Quantity(new_value)
```

Now when we run the earlier print statements we generate the output:

```
q1 * 2 Quantity[10]
q2 / 2 Quantity[5.0]
```

22.4 Comparison Operators

Numerical types (such as integers and real numbers) also support comparison operators such as equals, not equals, greater than, less than as well as greater than or equal to and less than or equal to.

Python allows these comparison operators to be defined for user defined types/classes as well.

Just as numerical operators such as '+' and '−' are implemented by special methods so are comparison operators. For example the '<' operator is implemented by a method called `__lt__(self, other)`.

The complete list of comparison operators and the associated special methods is given in the following table:

Operator	Expression	Method
Less than	q1 < q2	`__lt__(q1, q2)`
Less than or equal to	q1 <= q2	`__le__(q1, q2)`
Equal to	q1 == q2	`__eq__(q1, q2)`
Not Equal to	q1 != q2	`__ne__(q1, q2)`
Greater than	q1 > q2	`__gt__(q1, q2)`
Greater than or equal to	q1 >= q2	`__ge__(q1, q2)`

We can add these definitions to our `Quantity` class to provide a more complete type that can be used in comparison style tests (such as if statements).

The updated `Quantity` class is given below (with some of the numerical operators omitted for brevity):

```
class Quantity:
    def __init__(self, value=0):
        self.value = value
    def __add__(self, other):
        new_value = self.value + other.value
        return Quantity(new_value)

    # remaining numerical operators omitted for brevity ...

    def __eq__(self, other):
        return self.value == other.value

    def __ne__(self, other):
        return self.value != other.value

    def __ge__(self, other):
        return self.value >= other.value

    def __gt__(self, other):
        return self.value > other.value

    def __lt__(self, other):
        return self.value < other.value

    def __le__(self, other):
        return self.value <= other.value

    def __str__(self):
        return 'Quantity[' + str(self.value) + ']'
```

This now means that we can update out sample application to take advantage of these comparison operators:

```
q1 = Quantity(5)
q2 = Quantity(10)
print('q1 =', q1, ',q2 =', q2)
q3 = q1 + q2
print('q3 =', q3)
print('q1 < q2: ', q1 < q2)
print('q3 > q2: ', q3 > q2)
print('q3 == q1: ', q3 == q1)
```

The output from this is now:

```
q1 = Quantity[5] ,q2 = Quantity[10]
q3 = Quantity[15]
q1 < q2:  True
q3 > q2:  True
q3 == q1:  False
```

22.5 Logical Operators

The final category of operators that can be defined on a class are Logical operators. These are operators that can be used with and/or type tests; they typically return the values `True` or `False`.

As with the numerical operators and the comparison operators; the logical operators are implemented by a set of special methods.

The following table summarises the logical operators and the methods used to implement them:

Operator	Expression	Method
AND	`q1 & q2`	`__and__(q1, q2)`
OR	`q1 | q2`	`__or__(q1, q2)`
XOR	`q1 ^ q2`	`__xor__(q1, q2)`
NOT	`~q1`	`__invert__()`

As these operators do not really make sense for the Quantity type, we will not define them. For example, what would it mean to say:

```
q1 | q2
```

In what way is `q1` an alternative to `q2`?

22.6 Summary

Only use operators when they make sense and only implement those operators that work with the type you are defining. In general, this means

- Arithmetic operators should only be used for values types with a numeric property.

- Comparison operators typically only make sense for classes that can be ordered.
- Logical operators typically work for types that are similar in nature to Booleans.

22.7 Online Resources

Some online resources on operator overloading include:

- https://docs.python.org/3/reference/datamodel.html for information on operator overloading in Python.
- https://pythonprogramming.net/operator-overloading-intermediate-python-tutorial/ Tutorial on operator overloading.
- http://cafe.elharo.com/programming/operator-overloading-considered-harmful/ An article on why operagoer overloading may be harmful to good programming style.

22.8 Exercises

The aim of this exercise is to create a new numeric style class.

You should create a new user defined class called `Distance`. It will be very similar to `Quantity`.

You should be able to add two distances together, subtract one distance from another, divide a distance by an integer, multiply a distance by an integer etc.

You should therefore be able to support the following program:

```
d1 = Distance(6)
d2 = Distance(3)

print( d1 + d2)
print (d1 - d2)
print (d1 / 2)
print(d2 // 2)
print(d2 * 2)
```

Note that the division and multiplication operators work with a distance and an integer; you will therefore need to think about how to implement the special methods for these operators.

The output from this might be:

```
Distance[9]
Distance[3]
Distance[3.0]
Distance[1]
Distance[6]
```