A Vector Class

*Mathematical objects have a lot in common conceptually with Python objects – both are defined in terms of what they do, and these definitions are precise and rigorous. In the same way that mathematicians have a well-defined idea of what an integer does (and coders what an **int** object does), we can create our own custom objects that behave in the way we would expect. Vectors are a perfect example.*

Classes: a recap

The programming idea of *encapsulation* in essence means separating the parts of a program into smaller, independent and more easily managed sections. Building a custom object often helps to organize our code more efficiently, and significantly improves the readability of a program, not to mention making the debugging process easier. Custom objects have to be created in order for Python to know what they do. The code that spells out what a new type of object has and does is called a **class**. Think of it as the blueprint for a new object. The cookie-cutter, if you like, where a created *instance* of a given class is the cookie itself:

Class code

```
class Triangle:
  def __init__(self, a, b, c):
    self.a = a
    self.b = b
    self.c = c

  def perimeter(self):
    return self.a + self.b + self.c

  def area(self):
    s = self.perimeter() / 2
    return (s*(s - self.a)*(s - self.b)*(s - self.c))**0.5

  def enlarge(self, sf):
    self.a *= sf
    self.b *= sf
    self.c *= sf
```

Example usage

```
t = Triangle(3, 4, 5)
print(t.area())
t.enlarge(2)
print(t.area())
```

An *instance* of the **Triangle** class is created by passing the values 3, 4 and 5 to the built-in constructor function (__init__). This particular triangle is called **t** and saved in memory, and we can apply its methods or retrieve its attributes as needed.

*The class code includes a **constructor** function which initializes a new object (__init__). Every method contains the key word **self** as the first argument (this is missed out when you invoke the method using the .method approach: note that Triangle.area(t) does the same thing as t.area(), calling the area method).*

What do vectors have and do?

Before writing the code, we should clarify what properties and behaviour (attributes and methods) vectors have. In fact, most of this comes down to how they interact *with one another*. Vectors can be added or subtracted (provided they have the same rank, such as 3D and 3D), they can be multiplied or divided by a scalar (ie, a **float** or **int**), they can be compared (to determine whether they are equal or not) and we can calculate their size. If all of that sounds a lot like the way **int**s and **float**s behave in Python, that's good news: we can not only define our own methods, but we can take advantage of Python's built-in 'standard operators' so that our code can use the same notation as ordinary numbers!

Read on to find out what Python's standard operators are and how to encode them…

The Standard Operators

When we use operators like **+** or **==**, behind the scenes Python is calling a method on the objects used. That's why we can 'add' strings or lists as well as numeric types ('addition' is defined in a way suitable for the type involved, and they're usually not cross-compatible):

```
>>> 3 + 4
7
>>> "three" + "four"
'threefour'
>>> [1, 2, 3] + [1, 2, 3, 4]
[1, 2, 3, 1, 2, 3, 4]
```

When we write our Vector class, we will be able to choose what addition means for us.

How is it done?

Just like the **__init__** method (a key word sandwiched between double underscores), Python has a bunch of standard methods we can include in our code, including:

```
def __add__(self, other):
    ...

def __sub__(self, other):
    ...

def __abs__(self):
    ...
```

If I run the code **a + b**, this will be interpreted by Python as **a.__add__(b)** In other words, it will apply the **__add__** method to **a** with **b** as the argument. Some methods, like **__abs__**, only take one argument: **a.__abs__()** has exactly the same effect as **abs(a)**.

Setting up a Vector class

```
1  class Vector:
2      def __init__(self, elements):
3          self.elements = elements
4          self.rank = len(self.elements)
```

There are a few different ways we could do this, but I'm going for a single input: a list of the vector's components. This should allow for 2D, 3D or any other dimension.

*Note: all the functions that follow should be in line with this, within the **Vector** class block.*

Printing

```
def __str__(self):
    return "v" + str(self.elements)
```

We get to decide here how our custom **Vector** object will be displayed (eg if we **print** it).

I've chosen to use the corresponding method for the underlying list of elements, but by adding a letter v to the front it will be clear that it is a different object under the hood.

Equal is Equal

Next, we need to define the most basic function: equality:

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

We could test each element in turn, but it's easier to make use of Python's built-in equality method for lists to do the same job.

Not Equal is Not Equal

And along with equality (**==**), we should define inequality (**!=**). Python doesn't make any assumptions here, but often it's as simple as this: if they're not equal, they're… not equal:

```
def __ne__(self, other):
    return not self == other
```

If we have vectors **a** and **b**, when we run **a != b** this code will run (in vector **a**, with vector **b** as the argument **other**).

Opposite is Opposite

Next, we define negation (what happens when we put a **–** in front):

```
15      def __neg__(self):
16          return Vector([-e for e in self.elements])
```

This uses a list comprehension to change the sign of each element, returning a new **Vector** object.

*See https://docs.python.org/3/library/operator.html for a full list of standard operators. Try to implement **__add__** and **__sub__** to your vector class next…*

Adding and subtracting

```
def __add__(self, other):
    return Vector([self.elements[i] + other.elements[i] for i in range(self.rank)])
```

This code uses list comprehensions, pulling out each element in turn from the vector being operated upon and the one doing the operating, and adding them to form new elements. Notice that the result is itself a **Vector** object, so that running **a + b** on a pair of vectors will produce a completely new **Vector** object in memory (just like what happens with **int**s).

```
def __sub__(self, other):
    return self + -other
```

The **__sub__** code makes use of our **__neg__** code, which makes things simpler. Note that **a – b** needs to subtract each element of **b** from the corresponding element of **a**. When we type **a – b** it is interpreted as **a.__sub__(b)**, which adds **a** to the negative of **b** as required.

Multiplying

Vectors can be added to or subtracted from other vectors, but (until we learn about the different forms of vector product) we can only multiply them by scalars (**int**s or **float**s):

```
def __mul__(self, scalar):
    return Vector([scalar * e for e in self.elements])
```

This uses list comprehensions to generate an output vector with each element scaled up. Note that **a * k** is interpreted as **a.__mul__(k)** which will perform this custom **Vector** method on **a**, taking **k** as the scalar argument.

Multiplying… backwards?

But if we try it with **k * a**, we get an error:

```
a = Vector([2, 3, 1])
k = 5
print(a * k)
print(k * a)
```

```
v[10, 15, 5]
Traceback (most recent call last):
  File "main.py", line 49, in <module>
    print(k * a)
TypeError: unsupported operand type(s) for *: 'int' and 'Vector'
```

The problem occurs because **k * a** is interpreted by Python as **k.__mul__(a)**. It is invoking the **__mul__** method on the object **k**, which, in this case, was an **int**. Since the code within the **int** class doesn't contain information on how to deal with our newly minted **Vector** object, it has a problem. Fortunately, there is a work-around:

```
def __rmul__(self, scalar):
    return self * scalar
```

Reverse multiplication allows for non-commutative multiplication. When Python sees **k * a** now, it first tries to multiply the **int** by a **Vector** object, but when it fails it next looks for **__rmul__** within the **Vector** object, and does that instead. Essentially, **k * a** ends up being interpreted as **a.__rmul__(k)** because **k.__mul__(a)** returned a value of **NotImplemented** from the **int** class.

Division

```
def __truediv__(self, scalar):
  return self * (1 / scalar)
```

This is equivalent to the normal **a / k** operator (called **truediv** to distinguish it from integer division, or **floordiv**). To keep things simple, I'm using multiplication by the reciprocal of the scalar.

Size

```
def __abs__(self):
  return sum([e ** 2 for e in self.elements]) ** 0.5
```

The **abs** function returns the size of an object. For vectors, that means Pythagoras. Notice that list comprehensions come in handy here as well.

Direction

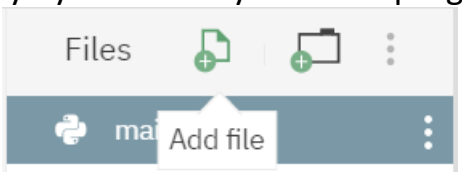
This one is not a standard operation, but still a very common thing to do with vectors: finding a vector of length 1 which points in the same direction as a given vector. Known as a *unit vector*, it is a handy way to extract information about the *direction* of a vector without having the *size* come into the picture.

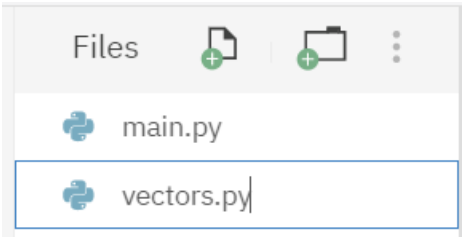
```
def unit(self):
  return Vector([e / abs(self) for e in self.elements])
```

Having defined **abs**, this is also relatively straightforward, especially using list comprehensions.

Now what?

We've successfully built the code required to construct and manipulate vectors in Python. This is likely to be useful in a range of different applications (kinematics, geometry, rocket science, etc) so it might be worth making this file into a standalone **module**. Like the **random** module, this is just a body of code which can sit in a separate file and is only read by Python from your main program if you import it:





vectors.py saved

```
1  class Vector:
2    def __init__(self, elements):
3      self.elements = elements
4      self.rank = len(self.elements)
5
6    def __eq__(self, other):
7      return self.elements == other.elements
```

Click *Add file*, and name it *vectors.py*. You can now cut and paste your code from *main.py*. All that you need in the main file is an import statement. The most common options are:

Conscious Import

Good for large files if you want to make it clear where a function or object is from.

```
1  import vectors
2
3  g = vectors.Vector([0, 0, -9.8])
4  m = 65
5  weight = m * g
```

Blanket Import

Simpler, and more convenient, if you want to just use the objects with less hassle.

```
1  from vectors import *
2
3  g = Vector([0, 0, -9.8])
4  m = 65
5  weight = m * g
```