CSCI 111, Lab 10
Vector Class

Due date: Midnight, Tuesday, November 29, on Canvas. No late work accepted.

File names: Names of files, functions, and variables, when specified, must be EXACTLY as specified. This includes simple mistakes such as capitalization.

Individual work: All work must be your own. Do not share code with anyone other than the instructor and teaching assistants. This includes looking over shoulders at screens with the code open. You may discuss ideas, algorithms, approaches, *etc.* with other students but NEVER actual code.

A 3d vector type: Write a module called `vector.py` such that the following interaction is possible:

```
>>> from vector import V
>>> v1 = V(1, 2, 3)
>>> v1
<vector.V object at 0x00000189527E4808>
>>> print(v1)
V(1, 2, 3)
>>> print(10 * v1)
V(10, 20, 30)
>>> print(v1 / 5)
V(0.2, 0.4, 0.6)
>>> v2 = V(10,20,30)
>>> print(v2)
V(10, 20, 30)
>>> print(v1 + v2)
V(11, 22, 33)
>>> print(v2 - v1)
V(9, 18, 27)
>>> v1 * v1
14
>>> v1 * v2
140
>>> v2 * v2
1400
```

And so on. Clearly you will be defining a new class and overloading the operators. Make sure you overload all of the following:

`+`	`__add__(self, other)`	`-=`	`__isub__(self, other)`
`-`	`__sub__(self, other)`	`+=`	`__iadd__(self, other)`
`*`	`__mul__(self, other)`	`*=`	`__imul__(self, other)`
`/`	`__truediv__(self, other)`	`/=`	`__idiv__(self, other)`
`==`	`__eq__(self, other)`	`-`	`__neg__(self)`
`!=`	`__ne__(self, other)`	`+`	`__pos__(self)`

Adding and subtracting is done with two vectors. They are computed componentwise.

Multiplying and dividing is done with a vector and a number (float or int).

Two vectors are equal if all their components are equal, and not equal otherwise. Because we will have lots of floats in our vectors, use `math.isclose` to compare components instead of `==`. You can also use your own function to determine if two floats are almost equal.

There is also vector multiplication to be supported. This is the inner product or dot product of two vectors:

$$(a_x, a_y, a_z) \cdot (b_x, b_y, b_z) = a_x b_x + a_y b_y + a_z b_z$$

Clearly the dot product of two vectors is a scalar (int or float). Since there is no "·" available in python, you will override the multiplication symbol, "`+`". This means we have three kinds of multiplication, where `v` is a vector and `x` is a scalar:

```
x * v                    v * x                    v * v
```

One of these can be handled with `__rmul__`. To handle all three you will need to use type-based dispatch, where you examine the type of one or more of the arguments and take action accordingly.

`__imul__` obviously does not apply to vector multiplication. Why not?

The last two operators in the list above are the unary operators, so we can write, for example, `v + (-w)` as well as `v - w` or `v - -w` instead of `v+w`.

None of the above operators should be destructive. They should all be pure functions that return new objects and not modify their arguments.

Testing: Write a `unittest` module called `test_vector.py` to accompany your `vector` module. Make sure your unit tests test every operator!

Normalize: Write a destructive method `normalize(self)` that will normalize a vector, *i.e.* stretch or shrink the vector so that its length is 1. If v is a vector, length of the vector, $|v|$, can be computed by

$$|v| = \sqrt{v \cdot v}$$

A vector can be made unit length by dividing by its length. If the length of the vector is zero, you should raise a `RunTimeError` with the message `'cannot normalize zero vector'` rather than the normal divide by zero error.

This method should be destructive, the vector itself should be modified. No new vector is returned.

Testing: Add tests to your `test_vector` unittest module to test normalization. Make sure all tests are passed before you continue!

Project: Write a nondesctructive method `project(self, other)` that will find the projection of a vector onto another. Mathematically the projection of v onto w is

$$\text{proj}_w(v) = \frac{v \cdot w}{w \cdot w} w$$

This method is nondestructive, returns a new vector and does not modify either of the input vectors.

Testing: Add tests to your `test_vector` module to test projection. Make sure all tests are passed before you continue!

Gram-Schmidt process: The Gram-Schmidt process starts with any three linearly independent vectors and ends up with three orthonormal vectors. Orthonormal means they are all of unit length and all perpendicular to each other.

You can test whether two vectors v and w are perpendicular by testing whether $v \cdot w = 0$. Since we're using floating point numbers a lot with vectors, don't use equality, `==` to compare to zero, use `math.isclose`, or your own function to compare two floats to see if they are almost the same.

If v_1, v_2, v_3 are the original vectors, the Gram-Schmidt process works as follows:

$$
\begin{aligned}
u_1 &= v_1 \\
u_2 &= v_2 - \text{proj}_{u_1}(v_2) \\
u_3 &= v_3 - \text{proj}_{u_1}(v_3) - \text{proj}_{u_2}(v_3)
\end{aligned}
$$

If any one of these vectors u_1, u_2, u_3 is (close to) zero, the original vectors are not (really) linearly indepedent.

After this, the three vectors u_1, u_2, u_3 are normalized to unit length.

Write a method `gram_schmidt(self, v2, v3)` that will desctructively perform the above algorithm on the three input vectors, ending up with all of them orthonormal.

If you find that the original vectors are not orthonormal, raise the `RuntimeError` with the message `'cannot orthonormalize linearly dependent vectors'`

Testing: Add tests to your `test_vector` module to test the Gram-Schmidt process. Make sure all tests are passed before you turn in the assignment!