Vector magnitude & normalization

Multiplication and division, as we just saw, are means by which the length of the vector can be changed without affecting direction. Perhaps you're wondering: "OK, so how do I know what the length of a vector is? I know the components (x and y), but how long (in pixels) is the actual arrow?" Understanding how to calculate the length (also known as ***magnitude***) of a vector is incredibly useful and important.

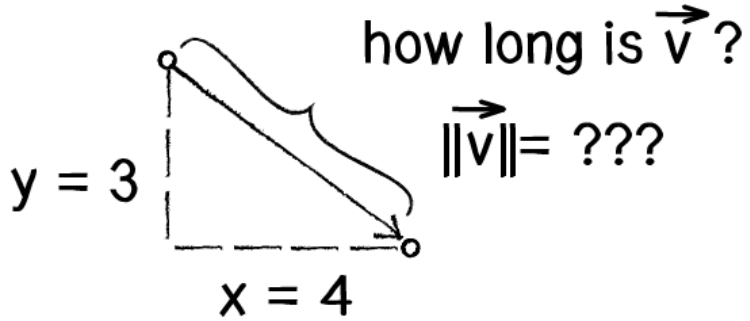
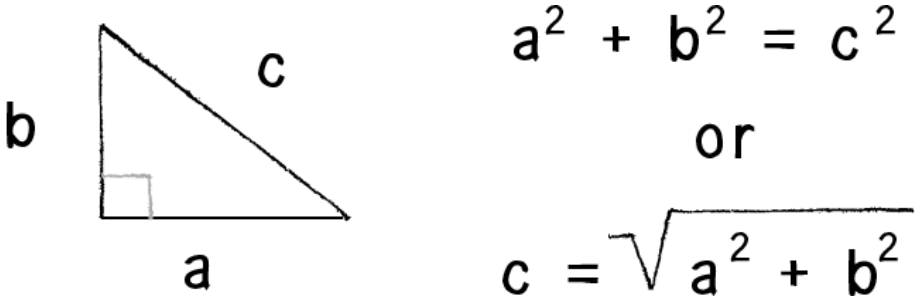


Figure 1.10: The length or "magnitude" of a vector \vec{v}$\vec{v}$v, with, vector, on top is often written as: ||\vec{v}||$\|\vec{v}\|$vertical bar, vertical bar, v, with, vector, on top, vertical bar, vertical bar

Notice in the above diagram how the vector, drawn as an arrow and two components (x and y), creates a right triangle. The sides are the components and the hypotenuse is the arrow itself. We're very lucky to have this right triangle, because once upon a time, a Greek mathematician named Pythagoras developed a lovely formula to describe the relationship between the sides and hypotenuse of a right triangle.

The Pythagorean theorem is a*a*a squared plus b*b*b squared equals c*c*c squared.



Armed with this formula, we can now compute the magnitude of \vec{v} $\vec{v}$v, with, vector, on top as follows:

||\vec{v}|| = \sqrt{v_x * v_x + v_y * v_y}||$\vec{v}$||=$\sqrt{v_x * v_x + v_y * v_y}$ vertical bar, vertical bar, v, with, vector, on top, vertical bar, vertical bar, equals, square root of, v, start subscript, x, end subscript, times, v, start subscript, x, end subscript, plus, v, start subscript, y, end subscript, times, v, start subscript, y, end subscript, end square root

The code for implementing in the `PVector` object would thus be:

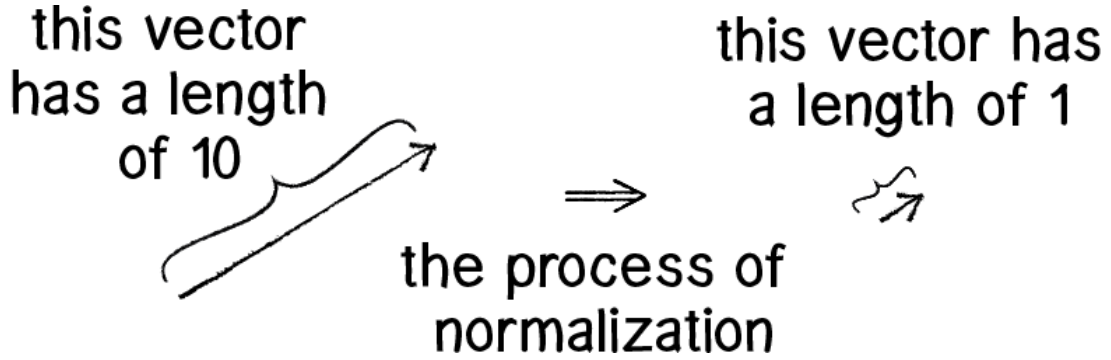
```
PVector.prototype.mag = function() {
    return sqrt(this.x*this.x + this.y*this.y);
};
```

The following example visualizes the magnitude of a vector with a bar at the top:

```
mouseMoved = function() {
  background(255, 255, 255);
  resetMatrix();

  // Two PVectors, one for the mouse location and
  //  one for the center of the window
  var mouse  = new PVector(mouseX, mouseY);
  var corner = new PVector(width/2, height/2);
  // PVector subtraction!
  mouse.sub(corner);

  var m = mouse.mag();
  fill(0, 0, 0);
```

```
    rect(0, 0, m, 10);


    // Draw a line to represent the vector.
    translate(width/2, height/2);
    stroke(255, 0, 0);
    strokeWeight(3);
    line(0, 0, mouse.x, mouse.y);
};
```

Calculating the magnitude of a vector is only the beginning. The magnitude function opens the door to many possibilities, the first of which is ***normalization***. Normalizing refers to the process of making something "standard" or, well, "normal." In the case of vectors, let's assume for the moment that a standard vector has a length of 1. To normalize a vector, therefore, is to take a vector of any length and, keeping it pointing in the same direction, change its length to 1, turning it into what is called a ***unit vector***.

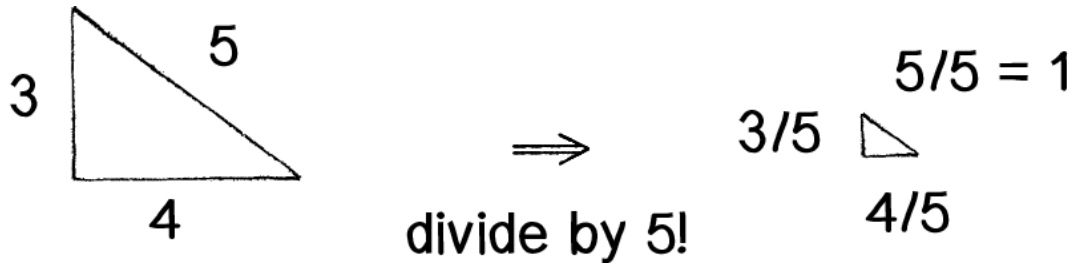


Since it describes a vector's direction without regard to its length, it's useful to have the unit vector readily accessible. We'll see this come in handy once we start to work with forces in the next section.

For any given vector \vec{u}$\vec{u}$u, with, vector, on top, its unit vector (written as \hat{u}$\hat{u}$^) is calculated as follows:

$$\hat{u} = \dfrac{\vec{u}}{\|\vec{u}\|}$$

In other words, to normalize a vector, simply divide each component by its magnitude. This is pretty intuitive. Say a vector is of length 5. Well, 5 divided by 5 is 1. So, looking at our right triangle, we then need to scale the hypotenuse down by dividing by 5. In that process the sides shrink, divided by 5 as well.



In the PVector object, we therefore write our normalization function as follows:

```
PVector.prototype.normalize = function() {
  var m = this.mag();
  this.div(m);
};
```

Of course, there's one small issue. What if the magnitude of the vector is 0? We can't divide by 0! Some quick error checking will fix that right up:

```
PVector.prototype.normalize = function() {
  var m = this.mag();
  if (m > 0) {
    this.div(m);
  }
};
```

Here's a program where we always normalize the vector that represents the mouse position from the centre (and then multiply it so we can see it, since 1 pixel is tiny!):

```
mouseMoved = function() {
  background(255, 255, 255);

  // Two PVectors, one for the mouse location and
  //  one for the center of the window
  var mouse  = new PVector(mouseX, mouseY);
  var corner = new PVector(width/2, height/2);
  // PVector subtraction!
  mouse.sub(corner);

  // In this example, after the vector is normalized, it is multiplied by 50 so that it is viewable onscreen. Note that no matter where the mouse is, the vector will have the same length (50) due to the normalization process.
  mouse.normalize();
  mouse.mult(50);

  // Draw a line to represent the vector.
  resetMatrix();
  translate(width/2, height/2);
  stroke(255, 0, 0);
  strokeWeight(3);
  line(0, 0, mouse.x, mouse.y);
};
```