

# Vectors and Forces

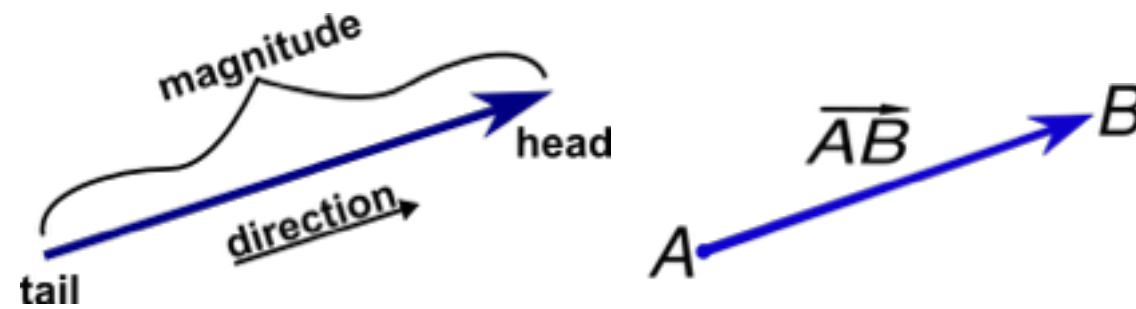
oh my!

Look at NOC\_1\_1

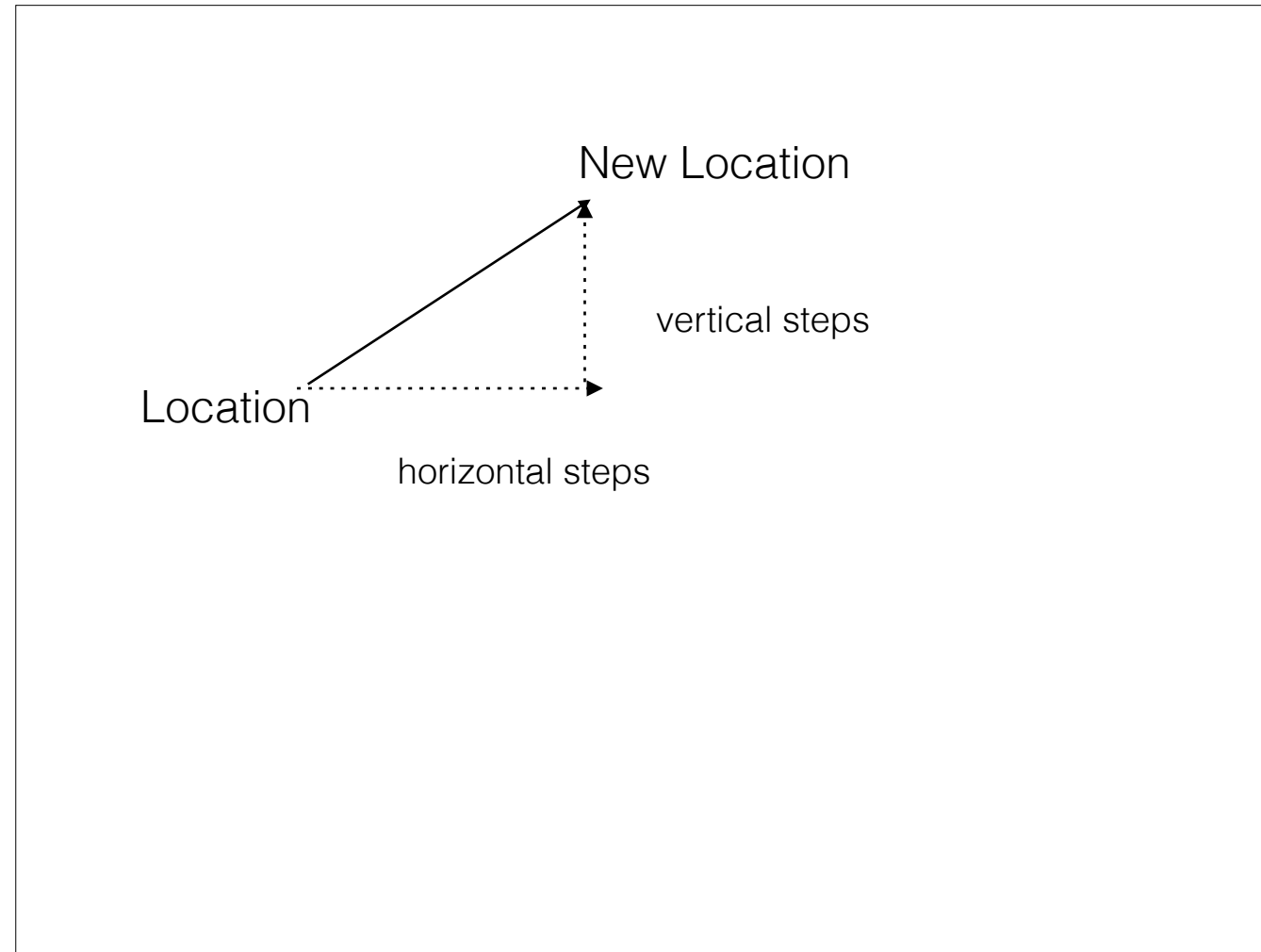
- ball traveling around
- properties represented in variables
- variables for location, speed, acceleration, target location, wind, friction —> x + y for each!!! God forbid we are working in 3D and need a variable also for the z axis for each

# Vector Review:

- Euclidean Vector
- entity with magnitude and direction

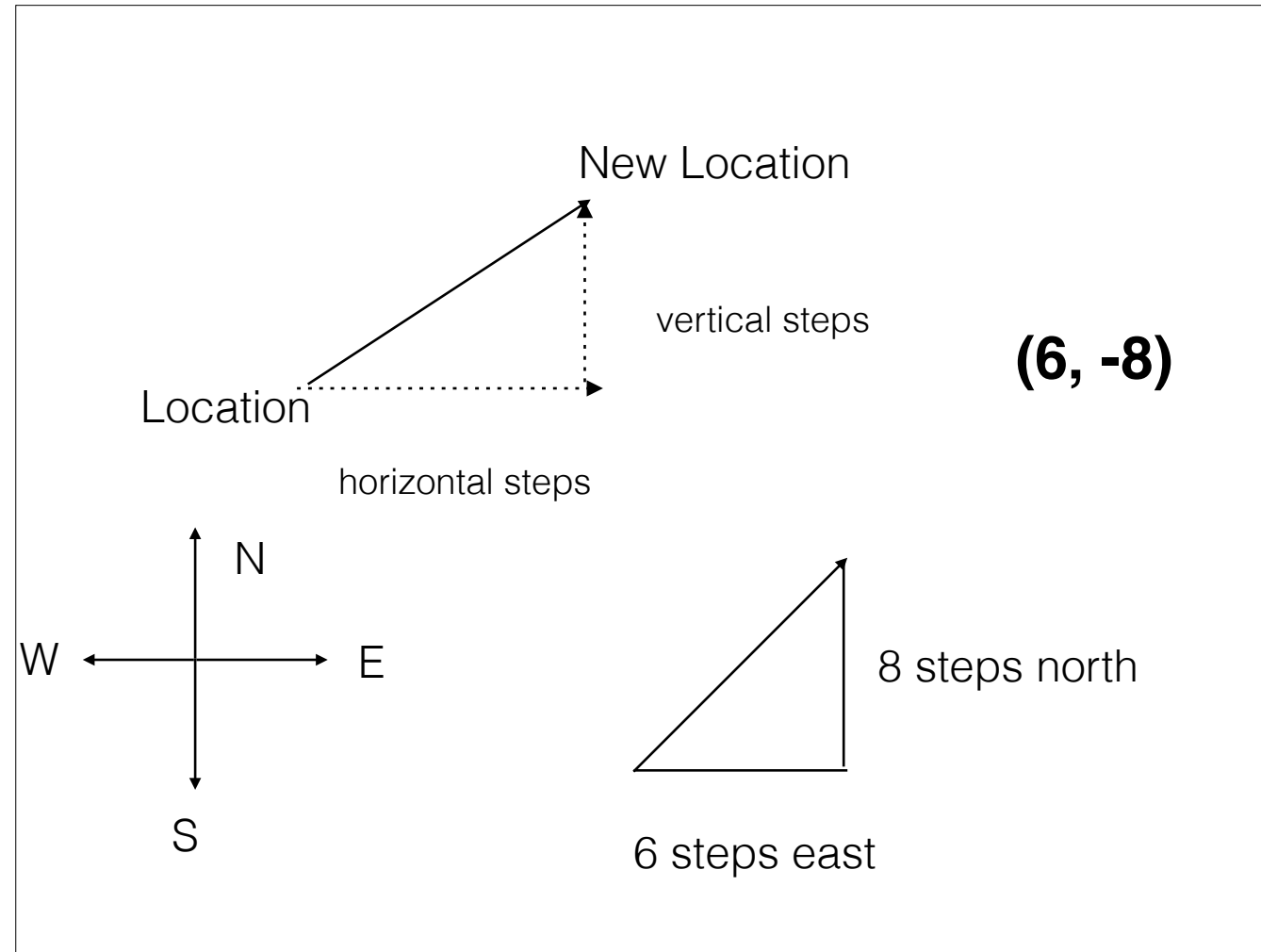


entity with magnitude and direction



location is a singular point in space

- think of it as the difference between two points



the path taken from the origin to reach that location

As a result: a location can be the vector representing the difference between the location + origin

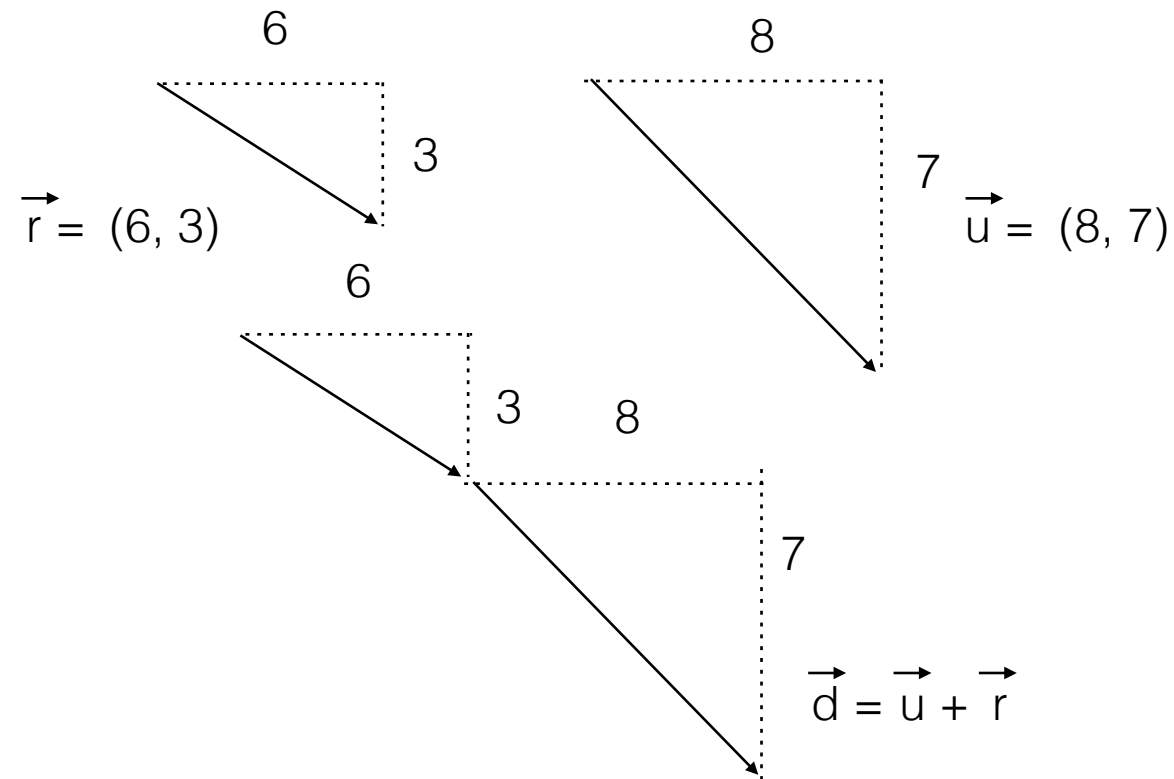
Vector = convenient way to store 2 values

# p5.Vector

```
sketch.js x +
1 var p1, p2;
2 var w = 40;
3
4 function setup() {
5   createCanvas(400, 400);
6   p1 = createVector(42, 104);
7   p2 = createVector(110, 260);
8 }
9
10 function draw() {
11   fill(87, 219, 232);
12   noStroke();
13   ellipse(p1.x, p1.y, w, w);
14
15   fill(255, 121, 97);
16   ellipse(p2.x, p2.y, w, w);
17 }
```

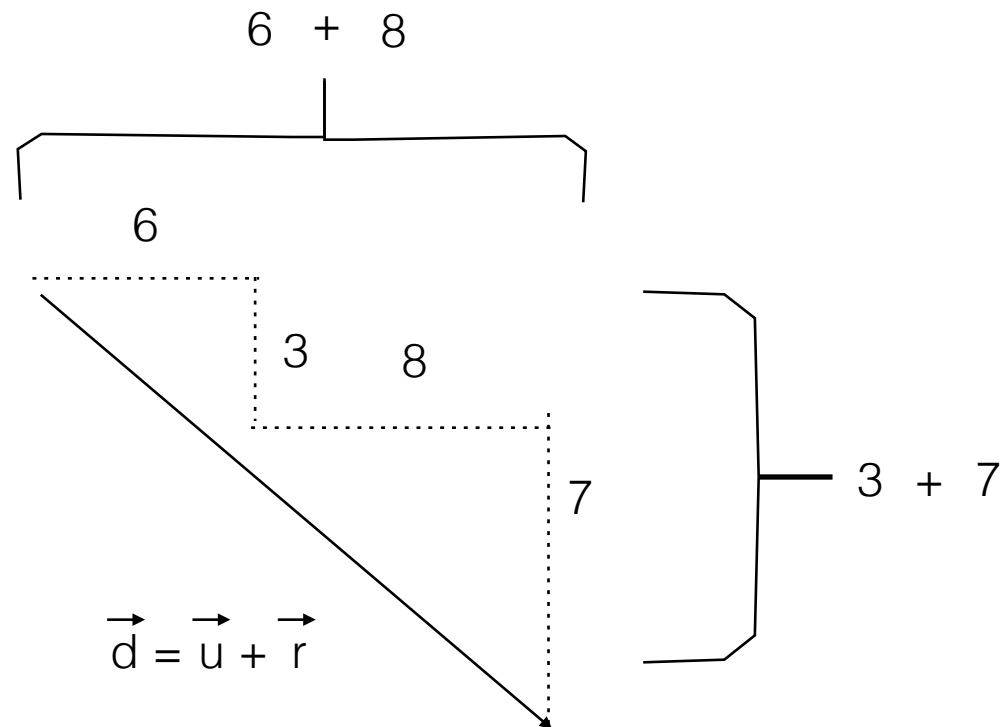
When we want to use a vector for placing graphics, we still need to independently reference it's to components to apply them to the x + y of the graphics independently

# Mathematical Operations



you have two vectors + you want to add them together...

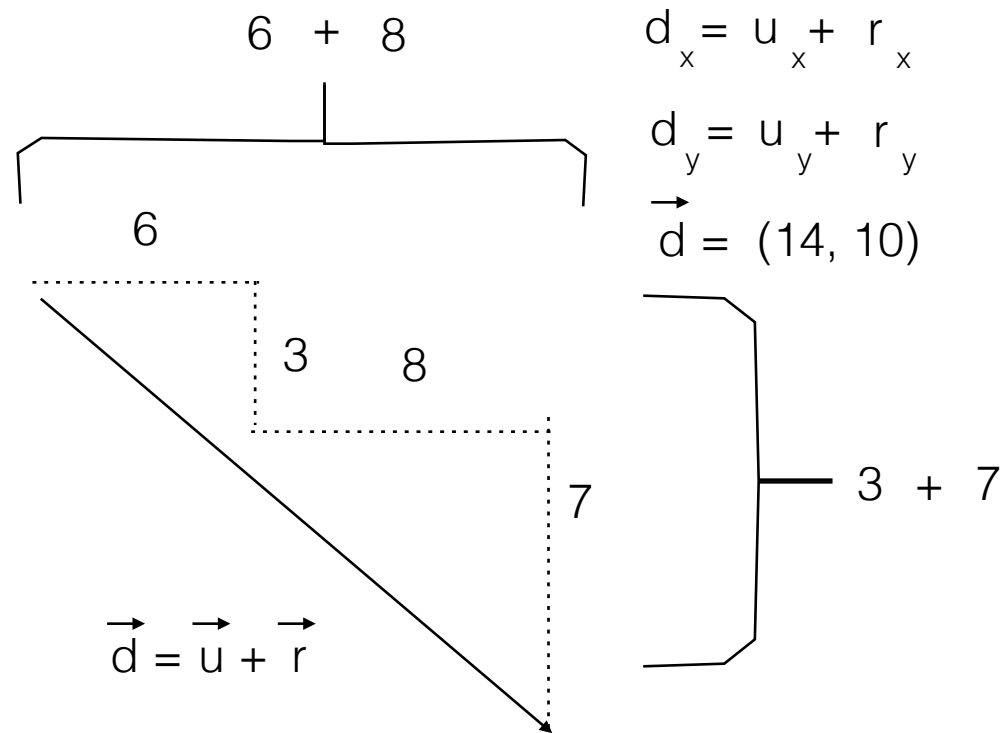
# Mathematical Operations



to add vectors, put them end to end

addition operator (+) is reserved for primitive values (int, float, etc)

# Mathematical Operations



addition operator (+) is reserved for scalar values (int, float, etc — single value numbers)



# Mathematical Operations

```
4 var loc;  
5  
6 function setup() {  
7   createCanvas(400, 400);  
8   loc = createVector(6, 3);  
9   velocity = createVector(8, 7);  
10  loc.add(velocity);  
11 }  
12
```

So, instead of +, we use the add method from the vector's class  
doing so allows us to add two vectors together  
We can use this to create motion of objects  
and to allow forces to effect our objects motion

# Mathematical Operations

- `add()`
- `sub()`
- `mult()`
- `div()`
- `mag()`
- `normalize()`
- `limit()`
- `dot()`

These all come from making an object from the `P5.Vector` class  
methods of the class

they do different types of mathematical operations on a vector.

some pass vectors in, some pass scalar values

Take for instance

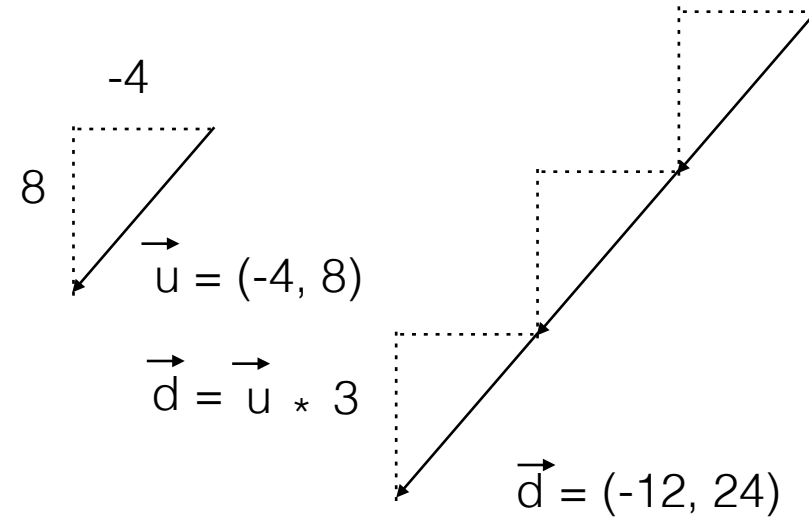
multiplication + division

# Mathematical Operations

$$\vec{d} = \vec{u} * k$$

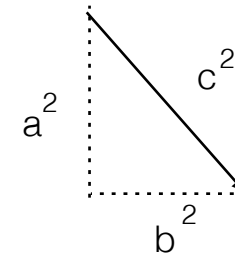
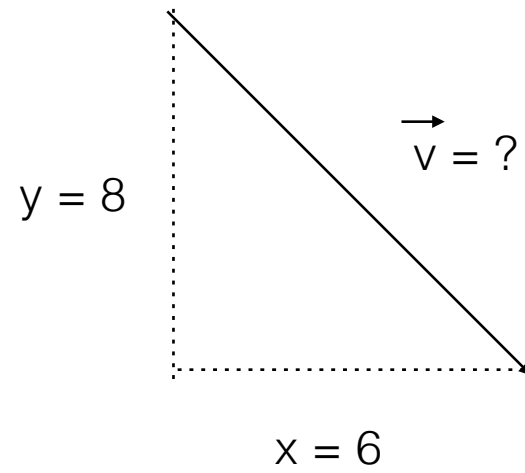
$$d_x = u_x * k$$

$$d_y = u_y * k$$



Check out code

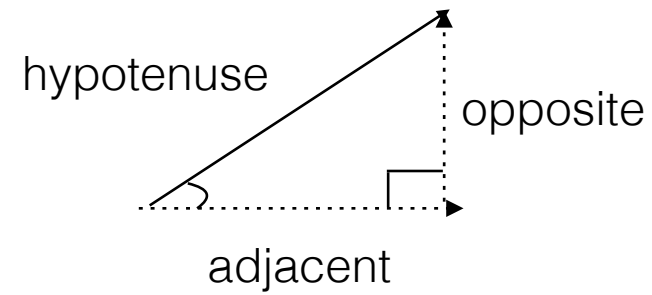
# Vector Magnitude



$$c = \sqrt{a^2 + b^2}$$

$$c = \sqrt{a_x^2 + a_y^2}$$

length of a vector



## **SOH CAH TOA**

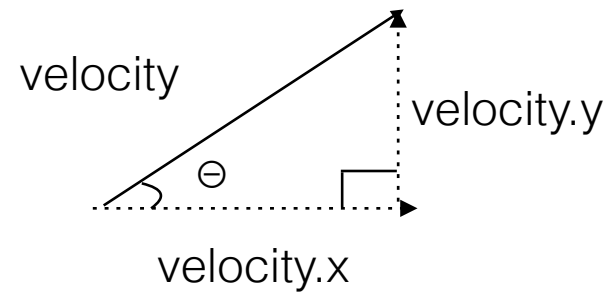
sine = opposite / hypotenuse

cosine = adjacent / hypotenuse

tangent = opposite / adjacent

Provided the triangle as a 90degree angle, we can use Trig to find out other information about the triangle:

- if we know 2 sides, we can figure out the angle
- if we know the angle and one side, we can figure out the other sides
-



$$\text{tangent}(\text{angle}) = \text{velocity.y} / \text{velocity.x}$$

$$\text{angle} = \text{arctan}(\text{velocity.y} / \text{velocity.x})$$

$$\text{angle} = \text{atan}(\text{velocity.y} / \text{velocity.x})$$

$$\text{angle} = \text{atan2}(\text{velocity.y} / \text{velocity.x})$$

### **SOH CAH TOA**

sine = opposite / hypotenuse

cosine = adjacent / hypotenuse

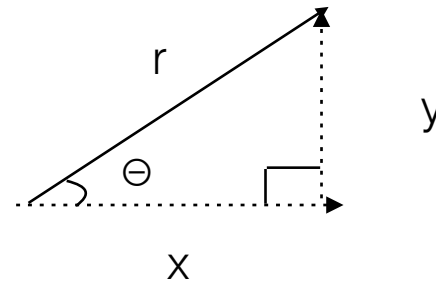
tangent = opposite / adjacent

atan2 for all directions

tangent is great for finding the angle

$$\cos(\Theta) = x / r \longrightarrow x = r * \cos(\Theta)$$

$$\sin(\Theta) = y / r \longrightarrow y = r * \sin(\Theta)$$



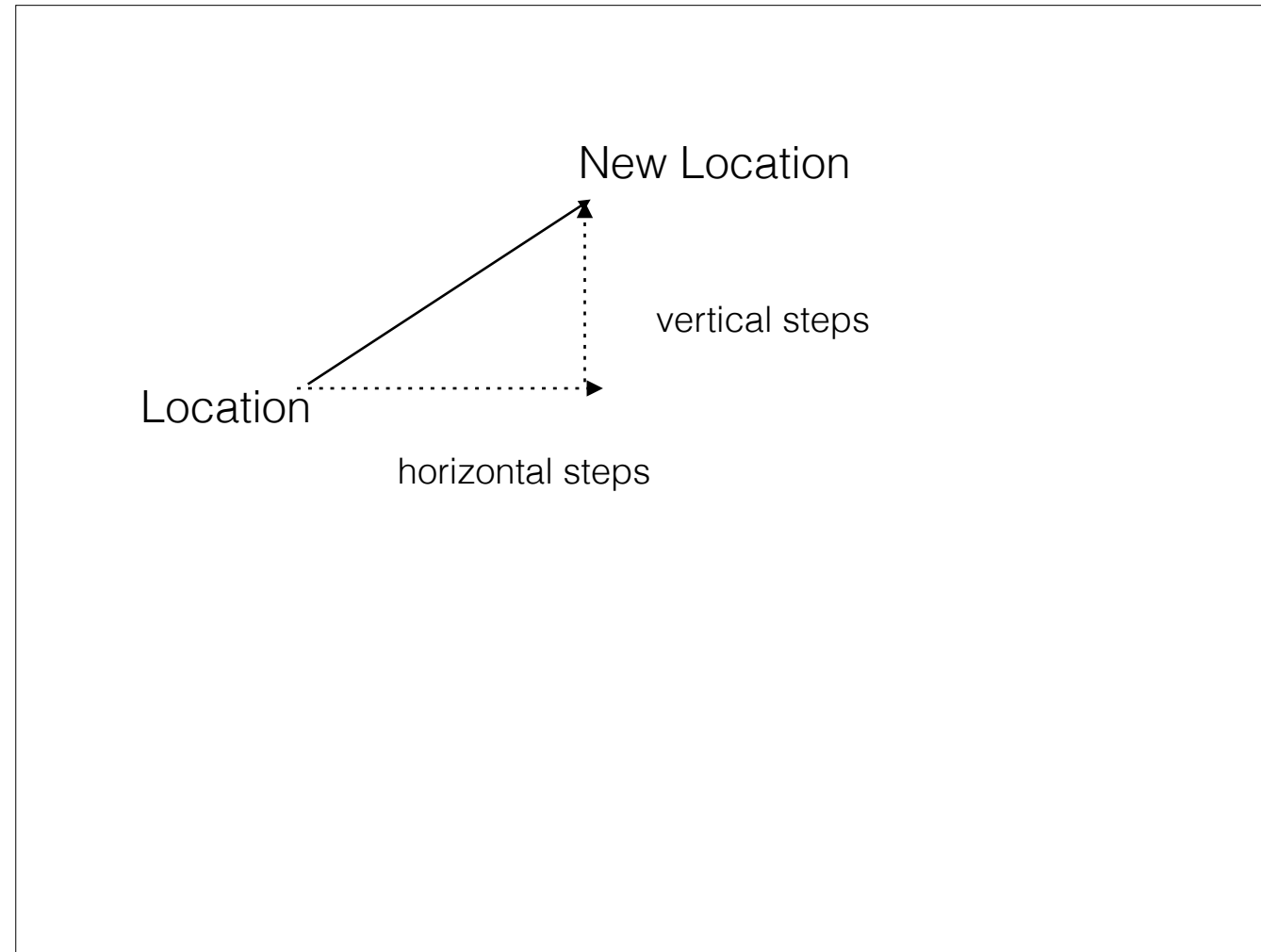
cartesian coordinate = the x,y component of a vector

polar coordinate = the magnitude (length) and direction (angle)

$$\vec{v} = (x, y) \qquad \vec{v} = (r, \Theta)$$

sine & cosine are great for converting back and forth between polar and cartesian coordinates





location is a singular point in space

- think of it as the difference between two points

# Static vs Non-Static Functions

```
float x = 4;
```

```
float y = 7;
```

```
x = x + y;
```

value of x changes

```
float x = 0;
```

```
float y = 5;
```

```
float z = x + y;
```

value of x does not change

Seems obvious, but not so obvious when working with vectors

Are we manipulating a vector?

Or do we want to create a new vector?

# Static vs Non-Static Functions

```
var v = createVector(0, 0);
```

```
var g = createVector(4, 5);
```

```
var w = v.add(u);
```

# Static vs Non-Static Functions

```
var v = createVector(0, 0);
```

```
var g = createVector(4, 5);
```

```
var w = v.add(u);
```

```
add(v){
```

```
    this.x = this.x + v.x;
```

```
    this.y = this.y + v.y
```

```
}
```

add method doesn't return a new vector; manipulates a given vector  
not only that, but it changes the value of the vector which it was called

# Static vs Non-Static Functions

Functions that we call from the class name itself (rather than a specific object instance) are known as *static functions*

`v.add(u);`      **Not static: called from an object instance**  
`p5.Vector.add(v, u);`      **Static: called from the class name**

```
var w = p5.Vector.add(v, u);
```

in order to get a new vector, we must use the static add function.

static functions allow us to perform generic math operations on vector objects, without having to adjust the value of one of the input vectors

use when you want a new vector returned, or you don't want to effect the current vector. make a copy

# Static vs Non-Static Functions

```
var v = createVector(0, 0);
```

```
var g = createVector(4, 5);
```

```
var w = v.add(u);
```

```
var w = p5.Vector.add(v, u);
```

p5.Vector returns a new vector

# Motion 101

1. Add velocity to location
2. draw object at location

# Motion 101

1. Add velocity to location
2. draw object at location

```
6 var Mover = function() {  
7  
8   this.position = createVector(random(width), random(height));  
9   this.velocity = createVector(random(-2, 2), random(-2, 2));  
10  
11   this.update = function() {  
12     this.position.add(this.velocity);  
13   };  
14  
15   this.display = function() {  
16     stroke(0);  
17     strokeWeight(2);  
18     fill(127);  
19     ellipse(this.position.x, this.position.y, 48, 48);  
20   };
```



# Motion 101: acceleration

- rate of change of velocity

rate of change of velocity

# Motion 101: acceleration

```
5- function Mover() {  
6   this.position = createVector(width/2,height/2);  
7   this.velocity = createVector();  
8   this.acceleration = createVector(-0.001, 0.01);  
9   this.topspeed = 10;  
10  
11  this.update = function() {  
12    this.velocity.add(this.acceleration);  
13    this.velocity.limit(this.topspeed);  
14    this.position.add(this.velocity);  
15  }  
16  
17  this.display = function() {  
18    stroke(0);  
19    strokeWeight(2);  
20    fill(127);  
21    ellipse(this.position.x, this.position.y, 48, 48);  
22  }
```

.limit() keeps velocity within a reasonable range

# Forces

Force = Mass x Acceleration

$$\vec{F} = M \times \vec{A}$$

$$\vec{A} = \vec{F} / M$$

Acceleration is directly proportional to force  
and inversely proportional to mass

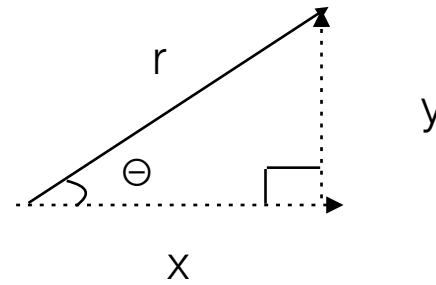
Mass = amount of matter in an object

Weight = the force of gravity on an object

density is the amount of mass per unit of volume

$$\cos(\Theta) = x / r \longrightarrow x = r * \cos(\Theta)$$

$$\sin(\Theta) = y / r \longrightarrow y = r * \sin(\Theta)$$



cartesian coordinate = the x,y component of a vector

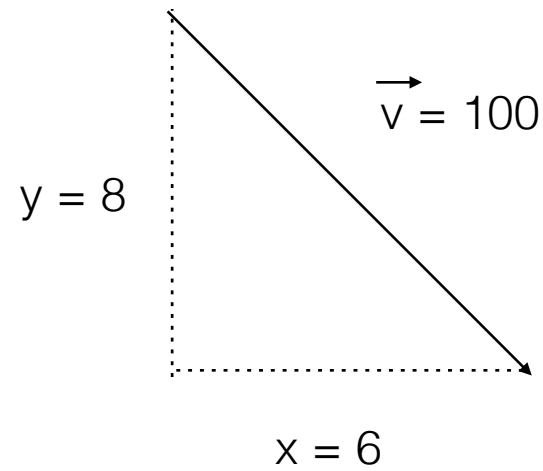
polar coordinate = the magnitude (length) and direction (angle)

$$\vec{v} = (x, y) \qquad \vec{v} = (r, \Theta)$$

Not only do we want a new vector, but we want that new vector to be small. We don't want to teleport. We want to apply this vector repeatedly to our location, to have us move there. Henceforth, we need to get the tiniest version of this vector, and run with that.

# Normalizing a Vector

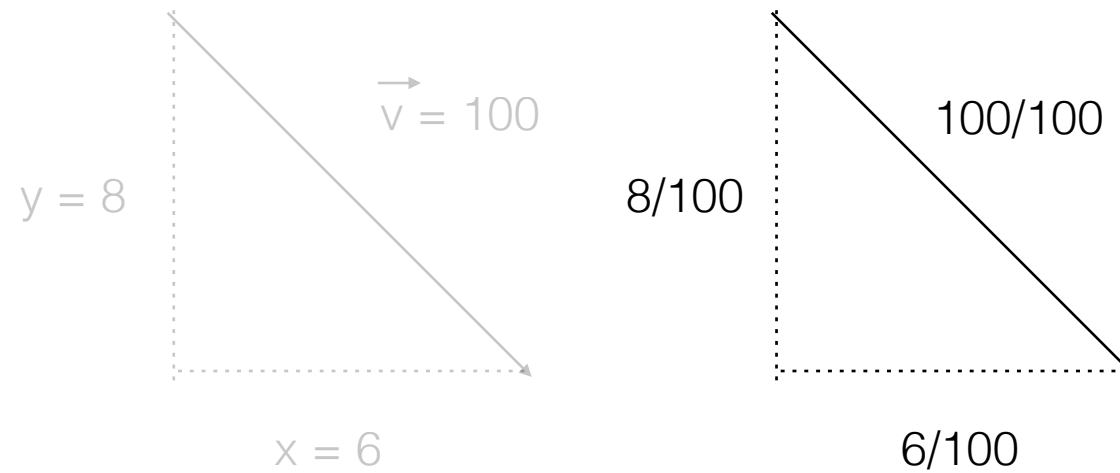
- Divide each component by it's magnitude



Unit vector

# Normalizing a Vector

- Divide each component by it's magnitude



Unit vector

# Steering