Week Zero Notes

Class	
Created	@November 19, 2021 5:43 PM
Materials	
Reviewed	☐
Type	

Vector Algebra

Scalars

Certain physical quantities are described completely by numerical value and units. These are known as *Scalars*.

Examples: Mass, time, Energy, etc.

They can be added algebraically. For example, a massless basket containing a 5kg ball and a 7 kg ball will together weigh 12 kgs.

Vectors

Some physical quantities can't be described fully with just a numerical value and units. They have magnitude, and a direction. Telling a particle is moving at a speed of 5 m/s does not tell us which way it is going.

Such quantities are known as *Vectors* They're represented by a letter with an arrow on top, like $\vec{a}$

Examples include Displacement, velocity, acceleration, force, Gravitational Field, etc etc.

They cannot be added algebraically. Two forces, of 3 and 4 newtons each will result in a different valued resultant force, depending on the angle between the two vectors.

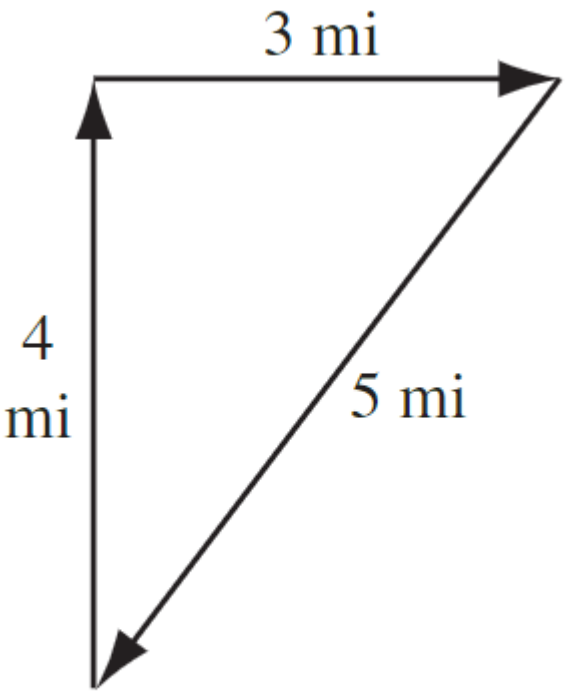


FIGURE 1.1

Standard Example of a 3-4-5 vector triangle, but with Miles as units, rather than Newtons like the example above.

Properties of Vectors

Equality of Vectors

Two vectors are equal only if they both have the same magnitude and direction.

Unit Vector

A vector with a magnitude of one, and any direction.

Any vector can be represented as a product of a unit vector, and a scalar. The Unit Vector gives it the direction, the scalar gives it it's magnitude.

A unit vector is represented by a cap, ie $\hat{a}$. The Standard directions, x, y and z axes are represented by $\hat{i}, \hat{j}$, and $\hat{k}$

Resolution of Vectors

Any vector can be written as a sum of multiple vectors. The process of splitting a vector into a sum of two or more vectors is called Resolution of Vectors.

One popular method of representing the vectors is by resolving them into perpendicular components along the coordinate axes. Hence, any general vector can be written as

$$\vec{A} = x\hat{i} + y\hat{j} + z\hat{k}$$

Finding Magnitude and Direction

A vector given as $\vec{A} = x\hat{i} + y\hat{j} + z\hat{k}$ has a magnitude and unit vector showing it's direction.

The magnitude of the vector, represented by $|\vec{A}|$ is given by

$$\vec{A} = x\hat{i} + y\hat{j} + z\hat{k}$$
$$|\vec{A}| = \sqrt{x^2 + y^2 + z^2}$$

If a Vector $\vec{A}$ has a magnitude of $|\vec{A}|$, and it's unit vector has a magnitude of 1, to find the unit vector $\hat{A}$, all we have to do is

$$\hat{A} = \frac{\vec{A}}{|\vec{A}|}$$

Addition of Vectors

Two vectors that are pointing in the same direction can be added Algebraically.

By that logic, when two vectors $\vec{A}\ and\ \vec{B}$ are given in their component forms, they can be added as:

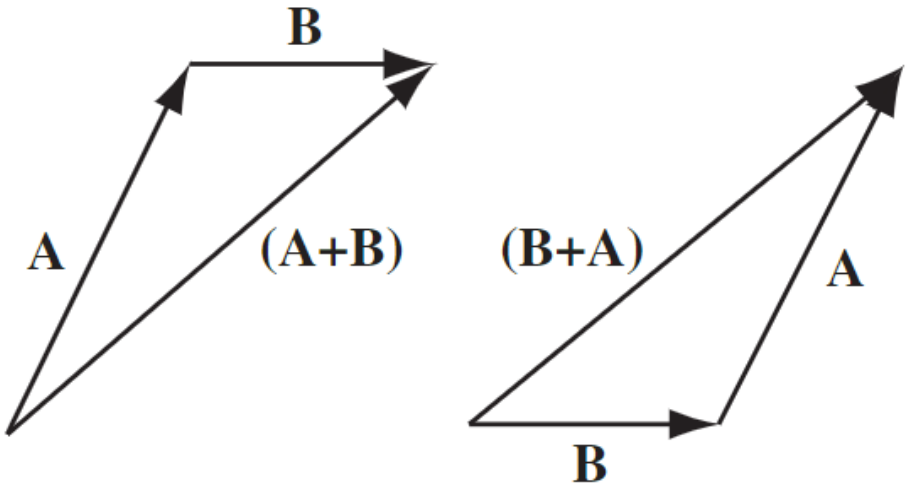
$$\vec{A} = x_1\hat{i} + y_1\hat{j} + z_3\hat{k}$$
$$\vec{B} = x_2\hat{i} + y_2\hat{j} + z_2\hat{k}$$
$$\vec{A} + \vec{B} = (x_1 + x_2)\hat{i} + (y_1 + y_2)\hat{j} + (z_1 + z_2)\hat{k}$$

If the vectors are not pointing in the same direction, or are not given in the component form, we follow the *Triangle Law of Vector Addition*.

Triangle Law of Vector Addition:

> Given two vectors $\vec{A}$ and $\vec{B}$ of Arbitrary Directions, Place the tail of $\vec{B}$ at the head of $\vec{A}$

The sum, $\vec{A} + \vec{B}$, is the vector from the tail of $\vec{A}$ to the head of $\vec{B}$



Scalar Multiplication

If a vector is multiplied by a scalar k, the direction of the new vector is going to remain the same, while it's magnitude increases by a factor of k

Multiplying a vector with a negative scalar,

▼ Exercise: Figure out how Scalar Division would work.

Dividing a vector by k is going to be the same as multiplying the vector by $\frac{1}{k}$

▼ Exercise: Figure out how Subtraction of Vectors would work.

$$\vec{A} - \vec{B} => \vec{A} + (-\vec{B})$$

Dot Product / Scalar Product

Dot Product, otherwise known as scalar product is helpful in finding the angle between two vectors. The output of this operation is a scalar.

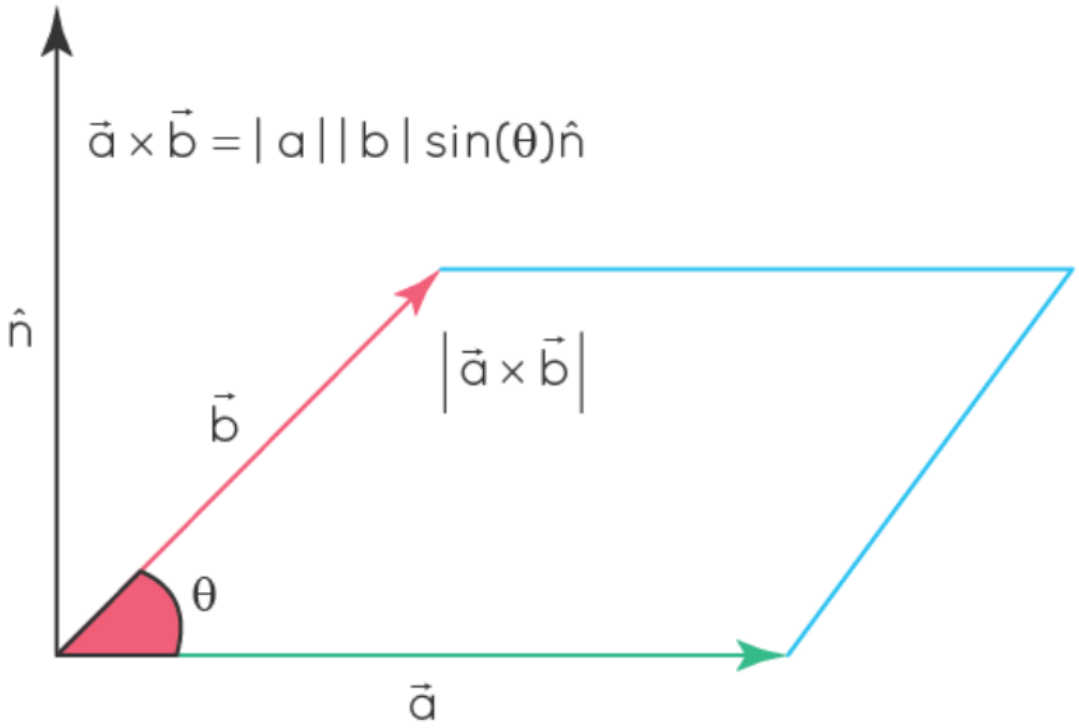
A dot product essentially tells how much the two vectors are in the same direction.

$$\vec{A} * \vec{B} = |\vec{A}||\vec{B}|cos\theta$$
$$\vec{A} * \vec{B} = x_a x_b + y_a y_b + z_a z_b$$

Cross Product

A cross product is also known as a vector product, because it outputs a vector perpendicular to both the input vectors.

If you draw the parallelogram with the vectors $\vec{a}$ and $\vec{b}$, the result of the cross product is the area vector of the parellogram. The Area vector has a magnitude of the area of the parellogram, and has a direction perpendicular to the surface of the parellogram.



There are two formulas to caluclate the Cross product of a vector, depending on which form of the vectors you have.

$$\vec{a} \times \vec{b} = \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = (a_2b_3 - a_3b_2)i + (a_3b_1 - a_1b_3)j + (a_1b_2 - a_2b_1)k$$

$$\vec{a} \times \vec{b} = |\vec{a}||\vec{b}|sin\theta$$

Why do we need Vectors for Physical simulations?

If you remember your 11th Grade Physics class, to describe motion of any kind, you will need Vectors. To display an object on screen, we need to have a position, which is a vector. Position changes according to velocity, which again is a vector. Velocity again changes with Acceleration, which again is a vector.

We're going to be doing calculations on the x component, and y component separately, while using the same formula. Working with vectors means we can do

calculations on them at the same time.

Representing Vectors in Python

Python allows us to use Linear Lists where we can store a set of numbers.

We can define a list to be three elements long, so that we can store the x component, y component, and z component of the 3D vector, in the same variable.

```
list = [1,2,3,4,5]
vec = [x, y, z]
```

However, the disadvantage with lists in python is that we can't work with all the elements at once. Multiplying the whole list by a number, or adding two lists elementwise is not supported by python's lists.

To get around this, we use the numpy library, which let's us use a datatype called an Array. An Array can be made by typing np.asarray(list), and supports two important features for us.

1. Scalar Multiplication: If you multiply a Numpy array by a constant value k, every element is multiplied by that constant k, like how vectors behave.
2. Array Additions: If two arrays of the same dimensions are added, the two arrays are added element-wise, pretty much like matrices, or vectors.

Thanks to these two properties, Arrays are going to be the ideal choice for representing vectors in python for us.

```
import numpy as np
vec1 = np.asarray([1,2,3]) #This vector is now equivalent to i + 2j + 3k
vec2 = np.asarray(2, 5, 1]) #This vector is now equivalent to 2i + 5j + k

#ADDITION OF TWO VECTORS
print(vec1 + vec2) #The output should be an array that is [3, 7, 4]

#Scalar Multiplication
print(5*vec1) #The output should be an array that is [5,10,15]
```

Euler's Method of Solving Differential Equations

Vast majority of Differential Equations cannot be solved by the methods we learn in 12th grade. The ones we saw so far are exceptions, rather than the rule.

When faced with a tough Differential Equation that's not possible to solve easily, we look at numerical methods that allow us to approximate solutions. One of the oldest method for it is the Euler's Method, derived by Leonard Euler himself.

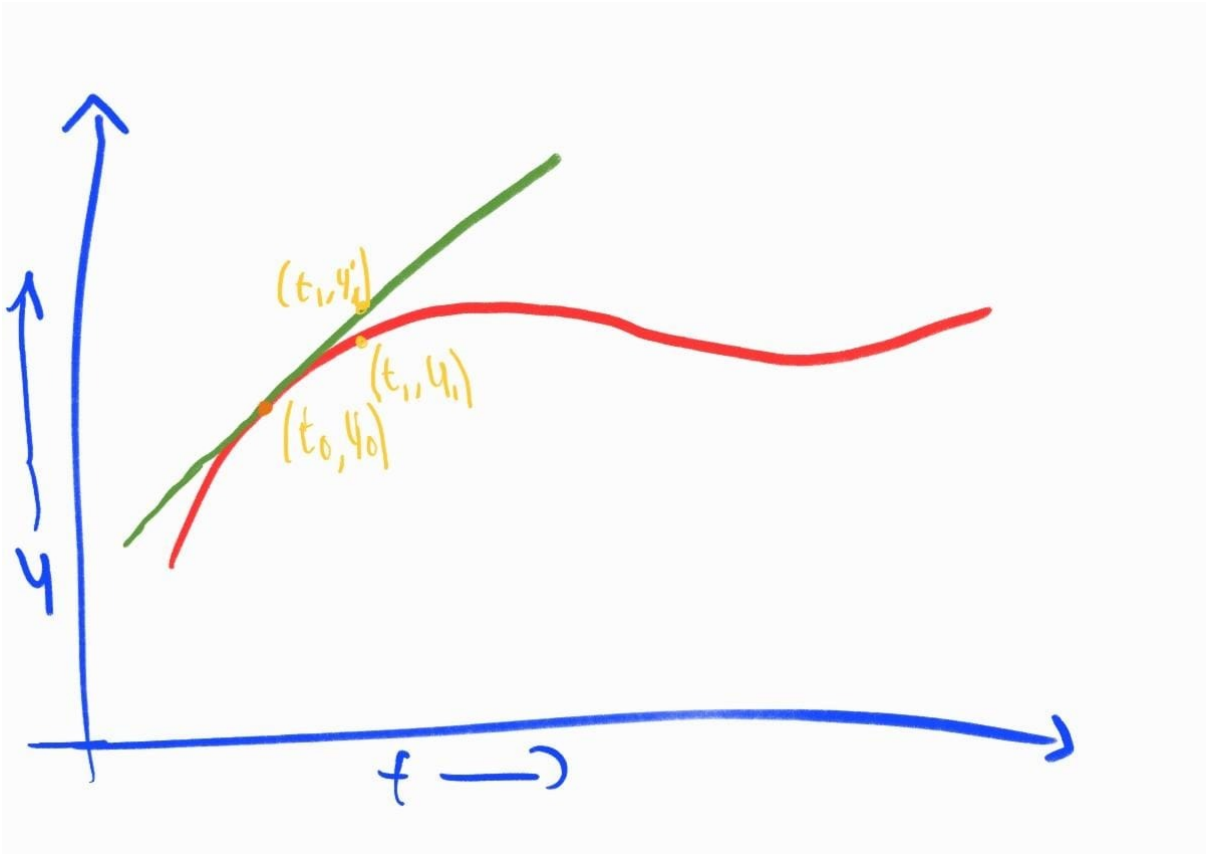
Let us look at a general first order Differential Equation, where we know the derivative function, and the initial conditions.

$$\frac{dy}{dt} = f(t, y)$$
$$y_0(t_0) = y_0$$

From this information, we can easily write the tangent line to the curve, at $y = y_0$ and $t = t_0$.

The equation of the tangent comes out to be

$$y'(t) = y_0 + f(t_0, y_0)(t - t_0)$$



The Red line represents the curve. (Here randomly Drawn). The Green line shows the tangent to the curve at the initial point.

The challenge here, is to find the next point (t_1, y_1) which satisfy the given equation. The trick to the solution lies in the observation that plugging in t_1 in the equation, and the resulting output is going to be approximately equal to the value of the the curve at t_1, ie:

$$(t_1, y_1) = \lim_{t_1 \to t_0} (t_1, y_1')$$

With that in mind, we can find that the value of y_1 will be given by the equation

$$y_1 = y_0 + f(t_0, y_0)(t_1 - t_0)$$

Similarly, now that we know y_1, we can use the same logic to find (t_2, y_2) using the formula

$$y_2 = y_1 + f(t_1, y_1)(t_2 - t_1)$$

This can be generalized into the equation,

$$y_{n+1} = y_n + f(t_n, y_n)(t_{n+1} - t_n)$$

That's a bit of a long equation and doesn't look cute, so we can define a term called step, represented by h, to replace $t_{n+1} - t_n$. While we're at that, we can shorten the derivative $f(t_n, y_n)$ into f_n, finally giving us a simple equation that looks like this

$$y_{n+1} = y_n + f_n * h$$

As you can probably guess, the smaller the value of h, the better the approximation you get from Euler's method is. But it's not uncommon that people set $h = 1$ for simplicity, to understand the rough trend of the curve.

There are more complex methods of solving Differential Equations, that are more accurate, however this is the simplest, and can get our job done to a reasonable degree of accuracy, and more importantly, this is easy to represent in code using a simple loop.

Mathematical Representation of Motion.

While there are a lot of equations related to motion, with various terms and topics, here we will only be focusing on the small bit that's nessecary for making a physics simulation. The remaining equations emerge as a consequnce of these equations

itself. For this section, we will consider 1 Dimensional motion, before generalizing them with help of vectors.

Case 1: Constant Velocity

A particle starts at time t_0 at a position x_0 with a velocity of v. We know that Velocity is the Rate of Change of Position, and hence, we can find the position of the particle at any given time by using that definition.

$$v = \frac{\Delta x}{\Delta t}$$

The quantity Δt can be defined as $t - t_0$, and Δx can be written as $x - x_0$

By substituting so in the equation, we get that

$$\begin{aligned} \frac{x - x_0}{t - t_0} &= v \\ x - x_0 &= v(t - t_0) \\ x &= x_0 + v(t - t_0) \end{aligned}$$

From this, we can easily find the equation that gives us the change in position in the second t_n, or in other words, $x_n - x_{n-1}$

$$x_n = x_{n-1} + v(t_n - t_{n-1})$$

Getting the equations into this specific form is going to be extremely important, and you can understand why once we begin with the code.

This equation, even though we calculated for the x axis, it is equally valid for y axis as well, as long as the velocity is along that axis. Hence, we can generalize this equation and write it using vectors, as

$$\vec{P}_n = \vec{P}_{n-1} + \vec{v}(t_n - t_{n-1})$$

where $\vec{P}$ denotes the position vector, and $\vec{v}$ denotes the velocity vector.

Case 2: Variable Velocity, Constant Acceleration.

Velocity is the rate of change of position, and Acceleration is the rate of change of Velocity. Hence, they can be represented as diffrential equations.

$$\frac{dx}{dt} = v(t) = v_t$$
$$x(t_0) = x_0$$

where $v(t)$ is given by another diffrential equation,

$$\frac{d}{dt}v(t) = a$$
$$v(t_0) = v_0$$

From this differential equation, we can find the velocity of the particle to be governed by the equation

$$v_t = v_0 + a(t - t_0)$$

This might seem familiar to the equations of motion covered in NCERT's 9th grade textbook. The derivation largely follows the same logic as case one to arrive at this equation, and you can try it yourself.

We can also derive the velocity at time t_n to be given by the equation

$$v_n = v_{n-1} + a(t_n - t_{n-1})$$

From the Euler's method, we know that the first differential equation describing the position can be written as

$$x_{n+1} = x_n + v_n * h$$

By calculating v_n from the above equation, we can then substitute it into this equation to find x_{n+1}

Obviously these equations can be extended to the other axes, and even in vector form by replacing x, v, and a by position, velocity and acceleration vectors.

Case 3: Variable Acceleration, Variable Velocity.

▼ I'm too lazy to type this now. This section is left as an exercise to the readers. If Acceleration is given as a function of time, how will you calculate the position of the particle and bring it to this form?

> Hint: Write a DE to describe the velocity, and find v_n. After that, the last few steps are the same as case 2.

Basics of Pygame

Pygame is a popular library to make games using Python. In it, you make a window, and draw something to the screen. Then, the frame refreshes, and you draw the next image.

Any Piece of code that uses Pygame follows a basic template that can be customized to our needs.

```
import pygame

dimensions = [1000, 500] # X length, and then Y length
title = "Physics Simulation" #Title of your Window

pygame.init() #intializing the pygame engine
win = pygame.display.set_mode(dimensions) #Makign a pygame window
pygame.display.set_caption(title) #Giving the window a title
fpsClock = pygame.time.Clock() #Basically a clock.
#Prevents the program from speeding up too much.

run = True #A variable that we can modify to quit

while run:
    for event in pygame.event.get():
        if event.type== pygame.QUIT:
            run = False
        else:
            #pos = pygame.mouse.get_pos()
            pass
    pygame.display.flip() #You're telling to update the display with this.
    fpsClock.tick(60) #This is there to basically make sure that the while loop
    #go toooooo fast.

pygame.quit() #THe engine we intialized earlier is now quit.
```

Drawing a circle

To draw anything in Pygame, it is common practice to define a function called Draw, which can be called in the 'while run' loop.

This is the code for a basic draw function, that draws a circle.

```
def draw():
  global win
  win.fill((0,0,0)) #This line of code over-writes whatever was drawn on the last frame.
  pygame.draw.circle(win, colour, (x,y), radius)
```

To specifically draw a circle, we use a pygame function called draw.circle(), which takes four inputs.

1. win: The window where we're drawing the image to.
2. Colour: The colour of the ball. A tuple of three numbers, each between 0 and 255. (For R,G and B colour values)
3. (x,y) is a tuple containing the coordinates of the the center of the circle in the window.
4. radius is the radius of the circle, basically how big it is.

Making One Particle Move in Pygame

The Basic idea in simulating motion is going to be updating the position argument in the draw circle function. For that, we can define a position vector which can be given as an input to the draw circle function. While initializing the vector, we will give it random values, ie the x_0 values

```
pos = np.asarray(20, 35]) #Random values taken for position.
#You can put your own values or generate new ones each time using the random module
```

Let us try to work with the first case, of constant velocity and no acceleration. The Equation we got there was $\vec{P}_n = \vec{P}_{n-1} + \vec{v} * h$, which we now need to implement on this particle.

For that, we need to define a velocity vector,

```
vel = np.asarray([2,5]) #Smaller values choosen, so the particle moves slowly
```

Since we're giving position as an input for the pygame.draw.circle, we should update the variable position.

We can now define an update function to update the value of position, which can be called right before drawing the particle.

```
def update():
  global pos, vel
  h = 1 #For the sake of simplicity, we're settign the step to be equal to 1.
```

```
  #you can experiment with different values of the step to figure out how it impacts the
  #code
  pos = pos + vel*h
  #Here, the pos on the left is equivalent to P_n, and pos on the right is equivalent to
  #P_{n-1}.
```

So now, our draw function would look like this:

```
def draw():
  global win, pos
  win.fill((0,0,0))
  update()
  colour = (200,200,200)
  radius = 5
  pygame.draw.circle(win,colour, pos, radius)
```

And our overall code should look a little something like this. I am using Random module to randomly generate the position and velocity at the start of the program.

```
import pygame
import numpy as np
import random

def update():
  global pos, vel
  h = 1 #Taken so for the sake of simplicity
  pos = pos + vel*h
  print(pos) #Just there to debug and so we'll be able to see how it's moving around
  #Here, the pos on the left is equivalent to P_n, and pos on the right is equivalent to
  #P_{n-1}.

def draw():
  global win, pos
  win.fill((0,0,0))
  update() #Updating the value of the position before it is sent to the drawing function
  colour = (200,200,200) #colour of the ball. You can randomly generate this too
  radius = 5 #Radius as 5 sounded good so I did this.
  pygame.draw.circle(win,colour, pos, radius) #drawing the circle on the screen.

dimensions = [1000, 500] # X length, and then Y length
title = "Physics Simulation" #Title of your Window

pygame.init() #intializing the pygame engine
win = pygame.display.set_mode(dimensions) #Makign a pygame window
pygame.display.set_caption(title) #Giving the window a title
fpsClock = pygame.time.Clock()

pos = np.asarray([random.randint(0,dimensions[0]), random.randint(0, dimensions[1])])
vel = np.asarray([random.randint(-5,5), random.randint(-5,5)])


run = True #A variable that we can modify to quit
```

```
while run:
    for event in pygame.event.get():
        if event.type== pygame.QUIT:
            run = False
        else:
            pass

    draw() #Drawing the
    pygame.display.flip() #You're telling the display to update.
    fpsClock.tick(600) #prevents the loop from running toooo fast.

pygame.quit() #THe engine we intialized earlier is now quit.
```