Pygame Part 1

Objectives

- Import and initialize pygame
- Create a surface
- Understand the geometry of a surface
- Draw lines
- Draw circles and pixels
- Write a simple game loop to animate a graphical element
- Write simple functions

Modules, Functions, Objects, Methods, and Variables

You already know **variables**. The following declares an integer variable, string variable, a float variable, and a boolean variable:

```
num_lives = 3
name = "John Doe"
speed = 5.5
alive = True
```

The above are names (or identifiers): `num_lives`, `name`, `speed`, `alive`.

There are other "names" such as modules, functions, objects, methods, etc.

Recall that a **function** is like a regular function from math. For a function, input or inputs are placed inside parentheses. For instance you have already seen the `range` function.

```
x = range(10)
print x
```

The input into the `range` function is `10`. The output is the list of integers from 0 to 9 (inclusive), i.e. [0,1,2,3,4,5,6,7,8,9] and that is then given to variable `x`.

The type of math functions you have seen in your math classes are numerical engines: you put numbers in and you get numbers out.

In Python (and many other programming languages), besides returning values (example: `range` returns lists), they can perform other tasks including drawing, playing music, etc.

Now for modules ...

You can think of **modules** as files. For instance you

already know there is a python program file called random.py and you have already used two functions in random.py. But to access the names in random.py, you need to execute `import random`. Furthermore, the function `seed` in random.py is accessed by the name `random.seed` and ***not*** `seed`.

```
import random
random.seed()
print random.randrange(0, 1000)
```

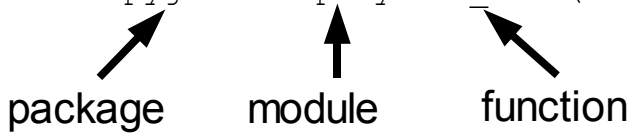
(If you have been reading our reference you know that there is a way to make `seed` just `seed` and not `random.seed`. But in order not to confuse you, I do not want to talk about that now.)

There can be all kinds of names in a module, not just functions. In your previous worksheets you have created examples of your modules containing variables.

A **package** is a just bunch of modules. I will not show you how to create packages at all.

You have already seen the **pygame** package. Recall that to open a screen for drawing we do the following:

```
surface = pygame.display.set_mode(SIZE)
```



The `surface` is the screen for drawing. As you can see, the `pygame` package contains the `display` module (there are other modules) which contains the `set_mode` function.

Objects are like variables. You execute a function by passing in values (for instance `range(10)`). In the case of objects, the "function" can appear ***behind*** the object. So for instance to sort a list, instead of calling `sort(x)` (where `x` is a list), you would execute `x.sort()`. Remember?

(For the experts: almost everything is an object in Python.

This is different from languages such as C++.)

So those are some of the “types” of things you will see.

Don't worry about the details. I know it can be overwhelming. I'm just introducing some terms here so that you won't be too surprised when we talk about them later. For instance, don't worry too much about objects because you will only be using them and not writing code for them until much, much, much, much later.

How to “view” Pygame as a Game Programmer

As you have seen above, to create a drawing surface (which we will call `surface`) you execute:

```
surface = pygame.display.set_mode(SIZE)
```

In this case the variable `surface` is an object. To create a variable for playing a sound file you would do this:

```
explode = pygame.mixer.Sound("explode.wav")
```

Exercise. For the above statement, what is the module used?

Once you have the surface you still need to know how to blit an image to the surface; you need to know how to make `explode` play the sound it refers to.

Etc., etc., etc.

Wow! That's a lot to remember!!! Like a ton!!!

Wrong. You ***don't*** have to remember them.

You should view pygame as a ***tool***. If you can remember all the tools in pygame, then of course you can program faster. But if you do not program games in Python (using pygame) everyday you will forget most of them.

You should only try to remember the main concepts that the package (the tool) pygame provides. For instance you should remember that you need a surface for drawing.

The most important thing is still the basic Python language constructs such as the `if`, the `if-else`, the `while`-loop, the `for`-loop, etc.

How does a software programmer or developer or engineer work? For instance suppose I want to draw a spaceship moving from the middle of the screen to the right and then stop. I would first need to know that I must have a drawing

surface. I must then load the image from an image file into a variable. Of course I must know how to copy (blit) the image the variable refers to onto the surface at a region of my choice – in the middle. I must also know how to repeat the above process of clearing the screen, changing the region a little to the right, blit the image until the region reaches the right.

Those are the things that pop into my mind. I must then look for the functions, modules, etc to use. I do know that there is a repetition there: clear surface, change region, blit, clear surface, change region, blit, etc.

This is a repetition – a loop. So that comes from the Python language itself. It will be either a for-loop or a while-loop.

Concepts such as variables, integers, operations on integers, if, if-else, for-loop, while-loop, etc. appear in many programming languages.

Concepts such as a surface, blitting images, pixels, animation techniques, etc. are common to all games. However the functions are specific to the package(s) you use.

In other words when you write another game in say C++ you will still see variables, for-loops, etc. You do have to open a surface. But the function used for opening a surface has a different name.

Therefore it's more important to know the Python **language** well rather than trying to memorize the tools inside pygame.

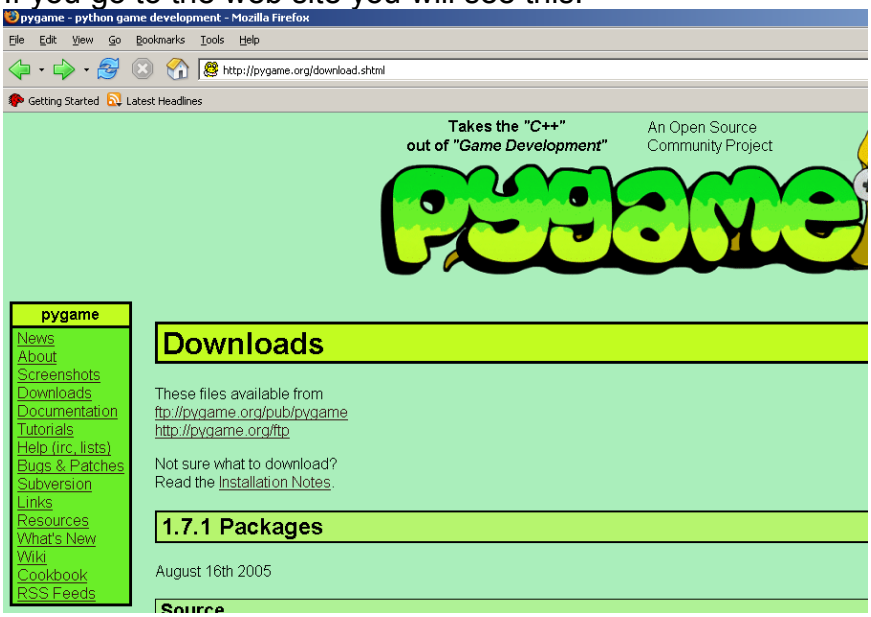
I will make a list of the most commonly used tools in pygame. Use this list as a **reference**. You do not have to memorize it. I certainly don't!

Another thing to remember is this: you are using the tools in pygame. The goal is not the tools; it's the game. For instance you need not worry about how `pygame.display.set_mode works`. You only need to know how to use it.

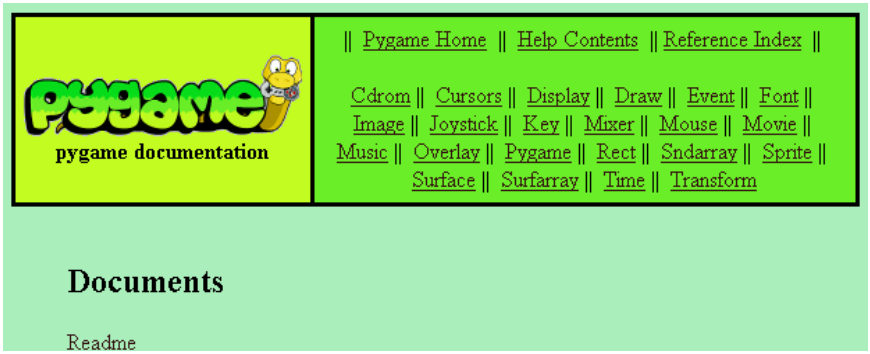
Resources

Before I point out the most useful features in pygame, I should tell to you that almost everything you need to know in order to write a pygame program can be found on pygame's web site: http://www.pygame.org.

If you go to the web site you will see this:



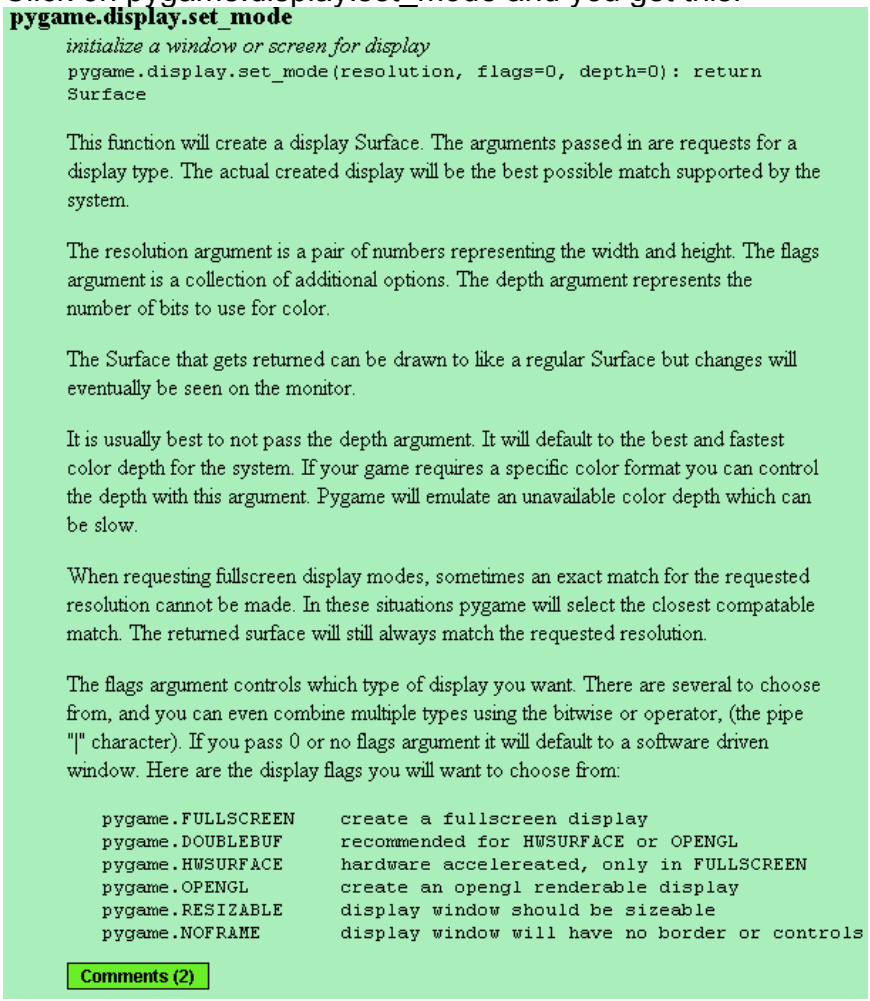
Now click on “Documentation” on the left side bar and you will see this:



The list of items in the green square contains modules, some of which you have seen before. Remember pygame.display.set_mode that we used to create a surface? Let's checkout the display module.

Go ahead and click on “Display” and you will see this:

Click on pygame.display.set_mode and you get this:



Remember: Google knows everything.

Importing and Initializing Pygame

There are two things you must do to use pygame. You must import and initialize pygame like this:

```
import pygame
pygame.init()

# the other game code
```

(Remember that everything to the right of # is ignored by Python. It's a comment.)

At some point we will need to close the program. This is done by calling the `exit()` function in the `sys` module. So I might as well tell you now that you need to import that as well. You can import pygame and sys separately like this:

```
import pygame
import sys
pygame.init()

# the other game code
```

Or you can import them both in one line like this:

```
import pygame, sys
pygame.init()

# the other game code
```

By the way it's a common practice to put all **import statements** at the **top** of the program.

Exercise. Write the above program, save it, run it by double-clicking on the file's icon.

Oops ... there's nothing to see. Python imported pygame, initialized the package, and stopped, because there was nothing else to execute.

Creating a Surface

What's a game without a graphical screen??

Here's how you create a screen

```
# Don't forget to import pygame and
# initialize it here!!!
# You also need to import sys.

WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)
```

(You have to fill in the blanks wherever there are comments. From now on I might not include the reminder to import and initialize pygame and to import sys.)

Of course you can choose your own names for the `WIDTH`, `HEIGHT`, `SIZE` , and `surface` variables. (For instance you can use `W` for `WIDTH`, `H` for `HEIGHT`, etc.)

Note that `WIDTH` and `HEIGHT` are integer variables while `SIZE` is a tuple. Remember that tuples are like lists except that you can't change the values in a tuple.

By the way, it's a common practice among all programmers that if a variable's **value does not change**, then the name of that variable should be **uppercase**. Such variables are called **constants**.

Exercise. Run the above program by double-clicking on the program's icon. Do not run it in IDLE.

Oops again ... you see the window opens ... but it closes immediately. Well ... of course. That's exactly what the program does. It opens a screen and then ends.

OK. Let's move on.

Once you have the `surface` you can start drawing on it.

You can also open a surface in fullscreen mode like this:

```
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE, \
                      pygame.FULLSCREEN)
```

Again the screen closes immediately because that what the program does.

Remember that when a statement is too long you can tell Python you want to continue to the next line using \. You can write the last statement on one line – I just ran out of space in my notes.

The Simplest Game Loop

Let's make the surface stay. To do that we need the program to run continually. Well not forever – just until we close the window by clicking the X in the top right corner.

Remember that the following while-loop will run forever:

```
# Did you import pygame, sys, initialize
# pygame, and create a surface?

while 1:
    print "ahhhh ..... i can't stop ..."
```

Try it. Don't forget that Python will run the body of the while-loop when the controlling boolean expression is `True`. In this case the boolean expression is `1`. Recall that `1` is converted to `True`.

Exercise. Run ***this*** program and make sure you see that this program does ***not*** run forever:

```
i = 10;
while i > 0:
    print i, "..."
    i = i - 1
print "blast off!"
```

Run the program in your head and then compare what you expect with the output you get when you run the program through Python.

So if we do this:

```
# what do you need to do here?

while 1:
    print "ahhhh ..... i can't stop ..."
```

the window will stay open. Try it.

But the problem is ... we can't close the window!!! Arrgh.

To kill this window, do Ctrl-Alt-Del (hold down the Ctrl key, the Alt key and press the delete key). You will see the Windows Task Manager. Look for the “pygame window”, select it, and click on “End Task” to kill it.

Phew ...

The problem is that we did not tell Python what do to when we click X in the top right corner of the window. Recall that to tell Python to stop the program when we do so we need to write this into the while-loop:

```
while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    print "ahhhh ..... i can't stop"
```

The statement

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

runs over all the events created by the user (mouse clicks, mouse cursor motion, etc) and if the event is the “click on X” event Python will execute `sys.exit()` which will stop the program immediately.

Now run the program. You see that it will stop when you click on the X. (It might not respond well or immediately but you can still close it.)

Exercise. Try this program. What's new here? And what does that new “thing” do?

```
# final reminder ...

pygame.display.set_caption('Hello world!')

while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    print "ahhhh ..... i can't stop"
```

Pixels

Recall that the surface is divided up into “dots” (**pixels** or picture elements) ... really tiny dots. If you have a 640-by-480 surface, then the width (left to right) is made up of 640 pixels while the height (top to bottom) is made up of 480 pixels. The position of the top-left is usually denoted (0,0). The pixel to its right has position (1,0). The pixel below the one at (0,0) has position (0,1).

So here I'm showing you a grid of the positions of pixels of a 640-by-480 surface: Each cell represents a pixel.

(0,0)	(1,0)	(2,0)	...	(639,0)
(0,1)	(1,1)	(2,1)	...	(639,1)
(0,2)	(1,2)	(2,2)	...	(639,1)
...	...	...	...	...
(0,479)	(1,479)	(2,479)	...	(479,639)

(Of course the pixels are a lot smaller than that!)

It's important to note that if you have a width of 640, since you start with 0, the last pixel on the top row is (639,0). Right?

Exercise. What are the locations of the pixels marked X, Y and Z?

(0,0)									
					X				
								Z	
	Y								

Understanding the layout (or the geometry) of the surface is important since you're going to draw on it.

Here's a warning: In your math geometry classes, you learn that your teacher prefers to label locations on the xy-plane so that the y-axis goes **up**. But for the computer surface the y-axis goes **down**.

Drawing a Line

You can specify a line segment with a starting and an ending position. The following is a line from (0,0) to (4,4):

X									
	X								
		X							
			X						
				X					

(I've marked the pixels on the line with X's).

It's important to note that in computer graphics, you don't get a perfect line because you have to work with pixels.

To draw a line using pygame, you need to specify the color of the line. A **color** can be specified using a tuple of three values (R, G, B). The R, G, B values are the red, green, blue intensities of the color you want. The intensities range from 0 to 255.

For white you need all the colors with the highest intensities, i.e. you need (255, 255, 255). The color black is created with (0,0,0).

Try this:

```
while 1:

    # close the program if X is clicked

    pygame.draw.line(surface, \
                     (255,0,0), \
                     (0,0), (10,10))

    pygame.display.flip()
```

Go ahead and run the above program.

The above program will draw a red line from (0,0) to (10,10). You can see that it's going to be red because of (255,0,0): maximum red component and no blue or green.

The `pygame.display.flip()` transfers data to the video – the important thing you need to know is that you must call it after you're done with drawing everything you need to draw in the body of the while-loop. If you need to draw 2 lines, you need to call the flip function ***after*** calling pygame.draw.line twice – you need not call the flip function twice:

```
while 1:

    # close the program if X is clicked

    # draw first line
    # draw second line

    pygame.display.flip()
```

Exercise. Add one more line to the above program. You get to choose the color and the positions of the beginning and ending point of the line.

Exercise. Let's draw lots of random red lines.

```
import random
random.seed()

# create a 640-by-480 surface

while 1:

    # close the program if X is clicked

    x0 = random.randrange(640)
    y0 = random.randrange(480)
    x1 = random.randrange(640)
    y1 = random.randrange(480)
    pygame.draw.line(surface, (255,0,0), \
                     (x0,y0), (x1,y1))
    pygame.display.flip()
```

Exercise. Modify the above program to draw random lines of random colors:

```
import random
random.seed()

# Create a 640-by-480 surface

while 1:

    # close the program if X is clicked

    x0 = random.randrange(640)
    y0 = random.randrange(480)
    x1 = random.randrange(640)
    y1 = random.randrange(480)

    # Create a variable R with random value
    # between 0 and 255 (inclusive)

    # Create a variable G with random value
    # between 0 and 255 (inclusive)

    # Create a variable B with random value
    # between 0 and 255 (inclusive)

    pygame.draw.line(surface, (R,G,B), \
                     (x0,y0), (x1,y1))
    pygame.display.flip()
```

Exercise. Let's go back to the first line drawing program:

```
while 1:

    # close the program if X is clicked

    pygame.draw.line(surface, \
                     (255,0,0), \
                     (0,0), (10,10), 1)

    pygame.display.flip()
```

Hmmm ... there's an extra value in the line drawing function. See that? Run the program. Change the 1 to 5 and run the program again. Try 10. So ... what's the point of this last extra value?

Drawing Circles

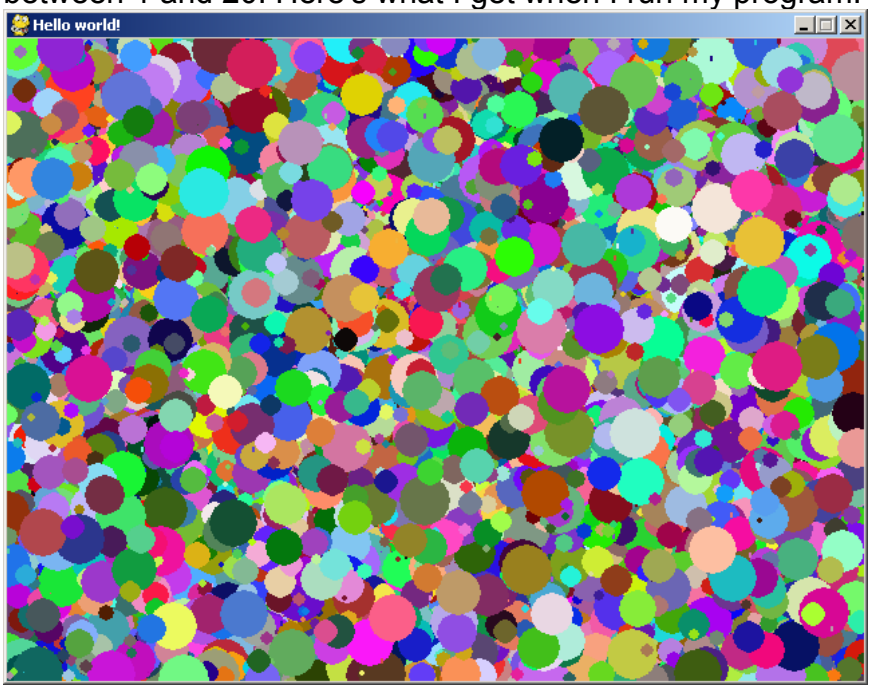
I'm just going to tell you that the function to draw a circle is

```
pygame.draw.circle(surface, color, pos, \
                   radius)
```

where color is a tuple of (R,G,B) for the red-green-blue intensities, pos is a tuple giving the position of the center of the circle, and radius is the (duh) radius of the circle, i.e. ½ the diameter.

Exercise. Draw a blue circle centered at position (100, 100) with radius of 10.

Exercise. Now use your previous program, and draw random circles (this means random centers and random radii) with random colors. Of course the center has to be in the surface and it's a good idea not to have radii which are too large or the circle will cover the whole surface. Try random radii between 1 and 20. Here's what I get when I run my program:



Exercise. Actually you can pass in one more value to the circle drawing function:

```
pygame.draw.circle(surface, color, pos, \
                   radius, width)
```

Experiment with different values for `width`. Do you see the purpose of `width`?

Exercise. From the name of the function `pygame.draw.circle`, you know that the circle function must come from the `draw` module. Go to pygame's documentation again and look for the draw module. See if you can find the circle function. What other shapes can pygame draw? Try some of them.

More on Colors

It's not easy trying to figure out what intensities of R, G, B you should use for certain colors. For instance what if you want yellow? Or pink?

No problem! Pygame comes with a function that will compute the (R,G,B) for various colors. You just need to pass a string describing your color to pygame.Color. Here's how you get the (R,G,B) for yellow and assign it to the variable `yellow`:

```
yellow = pygame.Color("yellow")
```

(Not all color strings are recognized by pygame.Color.)

Exercise. Test out the pygame.Color function by drawing a yellow line.

Clearing The Whole Surface

We need to know how to clear the surface. You need this to create motion. For instance to make a ball move to the left you would do this:

set (x,y) to (100, 100)
draw ball with center at (x,y)

clear whole surface with black color
change the value of x to 101
draw ball with center at (x,y)

clear whole surface with black color
change the value of x to 102
draw ball with center at (x,y)

etc.

Here's how you clear the whole surface. You just call

```
surface.fill(color)
```

where color is given by an (R,G,B) tuple.

Here's a simple test to create a flickering surface. I'm going to quickly swap between yellow and green. I will use a variable `isgreen` to remember which color I'm going to use. In other words `isgreen` will switch between `True` and `False`. When `isgreen` is `True` I will use green; otherwise I will use yellow.

Run this:

```
import pygame, sys

WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)
isgreen = True

green = pygame.Color("green")
yellow = pygame.Color("yellow")
```

```
while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    if isgreen:
        surface.fill(green)
        isgreen = False
    else:
        surface.fill(yellow)
        isgreen = True

    pygame.display.flip()
```

The screen was changing so rapidly that you don't get a perfect flicker – you see “bleeding” and “shearing”. You can slow down the program a little by inserting a delay. A delay simply tells your computer to stop running the program for a period of time. Try this:

```
import pygame, sys

WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)
isgreen = True

green = pygame.Color("green")
yellow = pygame.Color("yellow")

while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    if isgreen:
        surface.fill(green)
        isgreen = False
    else:
        surface.fill(yellow)
        isgreen = True

    pygame.display.flip()
    pygame.time.delay(200)
```

For each execution of the body of the while-loop, I'm pausing the program for 200 milliseconds. Now it looks nicer. You can almost use this to hypnotize someone ... :)

First Animation

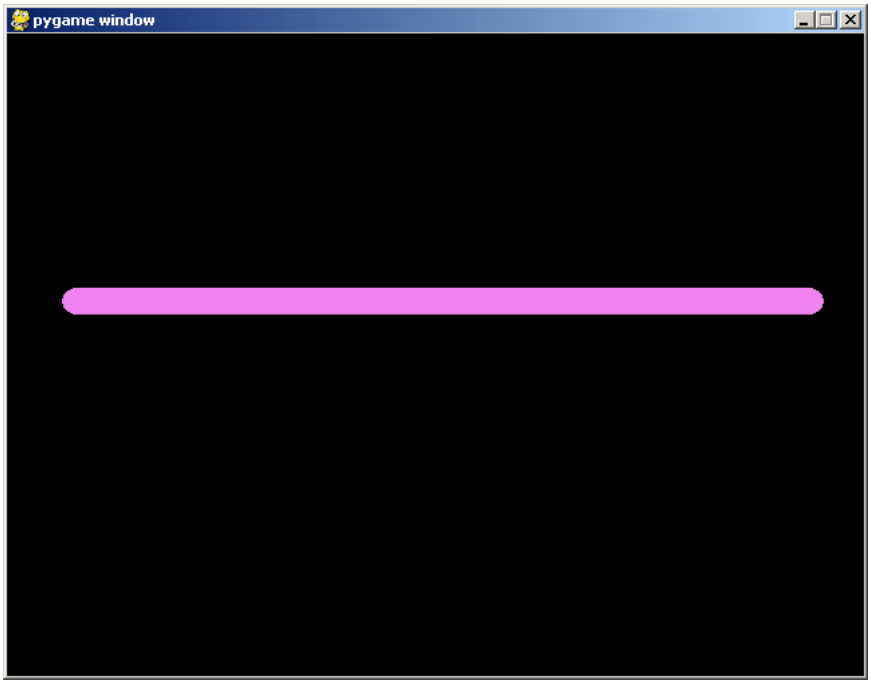
Alrighty ... it's time for the first animation program. Let's start small. I want to draw a ball moving slowing across the screen. Let's just say that it moves in a horizontal line. Try this

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
x = 50

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # Increase x by 1 if x is less than 600
    if x < 600:
        x = x + 1

    pygame.draw.circle(surface, violet, \
                       (x,200), 10)

    pygame.display.flip()
```

As you can see the center of the circle is `(x,200)` and `x` starts off with 50. Each time the body of the while-loop gets executed, the value of `x` increments by 1. `x` will stop incrementing when it reaches 600.

OK. Let's run it ...

Yikes! I got ***this*** ...



What's happening???

Oh ... since the program did not wipe out the old circle, you see all of them! The one with center at (50,200), and one at (51,200), the one at (52,200), ...

OK. We need to clear the screen before we draw the new one. Let's use black for our background color. Here's the program:

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)

x = 50

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

```
    if x < 600:
        x = x + 1

    surface.fill(black)
    pygame.draw.circle(surface, violet, \
                       (x,200), 10)

    pygame.display.flip()
```

Yippee! Our first animation!

Exercise. Now change your program so that you draw a moving ***pixel*** instead of a circle. To draw a single pixel you need to use 0 for the radius. Squint your eyes to see the running pixel.

Exercise. Can you make the circle grow as it moves? Start with a radius of 0 and grow the radius by 1 each time the body of the while-loop is executed. (It's gonna get really big!)

Second Animation

Now let's do an animation that involves deflection.

Take your moving pixel program from the previous section. Now suppose you let the pixel run into the wall. Remember that for a 640-by-480 surface, the x value is at most 639. This means that once x reaches 640, you need to change it to 639. Not only that you need to remember that once the pixel hits the wall, instead of incrementing the value of x by 1, you need to decrement it by one.

Just like I needed a variable `isgreen` to remember which color to use to draw our flickering screen program, I will need to remember if I should increment or decrement `x`. This is what I'm going to do ...

I will use a variable called `xspeed`. When the pixel is moving to the right `xspeed` will be 1. When the pixel hits the wall, I will set `xspeed` to -1. How's that?

[Note: For physics fans, I should really call this variable `xvelocity`.]

Not only that, instead of using "`x = x + 1`" when the pixel is moving right (i.e. xspeed is 1) and "`x = x - 1`" when the pixel is moving left, I can actually do

```
x = x + xspeed
```

Grrrrreat! One statement to handle both cases!

OK. Let's code that:

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)

x = 50
```

```
xspeed = 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x = x + xspeed

    if x > 639:
        x = 639
        xspeed = -1

    surface.fill(black)
    pygame.draw.circle(surface, violet, \
                       (x,200), 0)

    pygame.display.flip()
```

Ready? Squint your eyes ... and run the program.

Ooops! The pixel disappeared when it went through the ***left*** wall!!! We forgot to deflect it off the left wall ... !!!

OK. When does that happen? Well ... when `x` is less than 0. Right? If that's the case we set `x` to 0 and `xspeed` to `1` again so that it moves right. Correct?

Here's the code:

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)
isgreen = True

violet = pygame.Color("violet")
black = (0,0,0)

x = 50
xspeed = 1

while 1:
```

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x = x + xspeed

    if x > 639:
        x = 639
        xspeed = -1
    elif x < 0:
        x = 0
        xspeed = 1

    surface.fill(black)
    pygame.draw.circle(surface, violet, \
                       (x,200), 0)

    pygame.display.flip()
```

Grrreat! Now it works perfectly.

Just so we see what happens inside the body of the while-loop we can print the value of x and at the same time let's slow down the program:

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)
isgreen = True

violet = pygame.Color("violet")
black = (0,0,0)

x = 50
xspeed = 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

```
    x = x + xspeed

    if x > 639:
        x = 639
        xspeed = -1
    elif x < 0:
        x = 0
        xspeed = 1

    print x, xspeed

    surface.fill(black)
    pygame.draw.circle(surface, violet, \
                       (x,200), 0)

    pygame.display.flip()
    pygame.time.delay(200)
```

But I hate squinting my eyes. Let's make the ball larger:

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)
isgreen = True

violet = pygame.Color("violet")
black = (0,0,0)

x = 50
xspeed = 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x = x + xspeed

    if x > 639:
        x = 639
        xspeed = -1
    elif x < 0:
        x = 0
```

```
        xspeed = 1

    print x, xspeed

    surface.fill(black)
    pygame.draw.circle(surface, violet, \
                       (x,200), 50)

    pygame.display.flip()
    pygame.delay(200)
```

Run it ...

... Huh??? It seems like part of the ball actually went into the walls before the ball deflects.

Exercise. How would you fix the above program so that the ball deflects immediately on contact with a wall?

Third Animation

Now take your ball moving program and change the radius to 0 again to get a pixel.

Let's make the pixel move up and down as well. It's really not that difficult. You just need to apply everything you have done to the y-value of the position. You need to have a `yspeed` and don't forget that the height of the surface is `480`.

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)
isgreen = True

violet = pygame.Color("violet")
black = (0,0,0)

x = 50
xspeed = 1
y = 60
yspeed = 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x = x + xspeed
    if x > 639:
        x = 639
        xspeed = -1
    elif x < 0:
        x = 0
        xspeed = 1

    y = y + yspeed
    if y > 479:
        y = 479
        yspeed = -1
    elif y < 0:
```

```
        y = 0
        yspeed = 1

    surface.fill(black)
    pygame.draw.circle(surface, violet, \
                       (x,y), 0)

    pygame.display.flip()
```

(If you hate squinting, just set the radius to a slightly larger value.)

Now let's have two moving pixels. I'm going to use `X` and `Y` for their positions, `Xspeed`, `Yspeed` for controlling their motion:

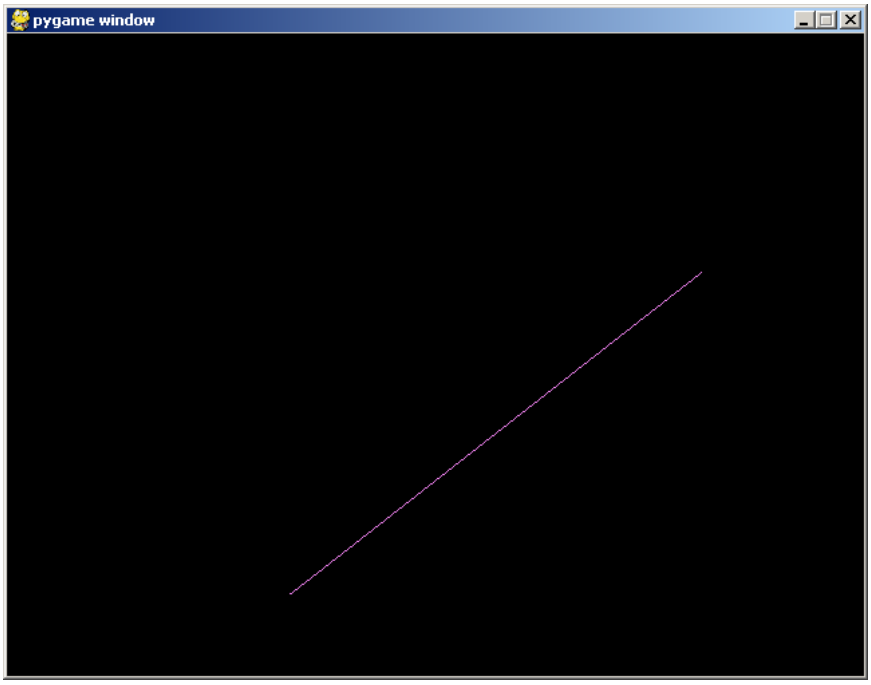
```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)

x = 50
xspeed = 1
y = 60
yspeed = 1

X = 600
Xspeed = 1
Y = 300
Yspeed = 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x = x + xspeed
    if x > 639:
        x = 639
        xspeed = -1
    elif x < 0:
```

```
        x = 0
        xspeed = 1

    y = y + yspeed
    if y > 479:
        y = 479
        yspeed = -1
    elif y < 0:
        y = 0
        yspeed = 1

    X = X + Xspeed
    if X > 639:
        X = 639
        Xspeed = -1
    elif X < 0:
        X = 0
        Xspeed = 1

    Y = Y + Yspeed
    if Y > 479:
        Y = 479
        Yspeed = -1
    elif Y < 0:
        Y = 0
        Yspeed = 1

    surface.fill(black)
    pygame.draw.circle(surface, violet, \
                       (x,y), 5)
    pygame.draw.circle(surface, violet, \
                        (X,Y), 5)

    pygame.display.flip()
```

(I've increased the radius to 5).

Run it and you do see two balls bouncing in the screen.

Exercise. Modify the above program so that instead of drawing two moving balls, use the positions x,y and X,Y as end points of a line segment to draw a line segment. You should see a line segment with its end points bouncing in the surface:



I want to make one change to the way we handle the speed after deflection.

Now think about this: Can we make a pixel move faster?

Right now, the x value changes by 1 (either plus 1 or minus 1). Can we change the x value by 2 each time we run the body of the while-loop? Of course. I can set the `xspeed` to 2. Right? I'm going to do that. At the same time, I'm going to print the x value and slow down the program again:

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)

x = 50
xspeed = 2
y = 60
yspeed = 1

X = 600
```

```
Xspeed = 1
Y = 300
Yspeed = 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x = x + xspeed
    if x > 639:
        x = 639
        xspeed = -1
    elif x < 0:
        x = 0
        xspeed = 1

    y = y + yspeed
    if y > 479:
        y = 479
        yspeed = -1
    elif y < 0:
        y = 0
        yspeed = 1

    print x, xspeed

    X = X + Xspeed
    if X > 639:
        X = 639
        Xspeed = -1
    elif X < 0:
        X = 0
        Xspeed = 1

    Y = Y + Yspeed
    if Y > 479:
        Y = 479
        Yspeed = -1
    elif Y < 0:
        Y = 0
        Yspeed = 1

    surface.fill(black)
```

```
    pygame.draw.circle(surface, violet, \
                       (x,y), 5)
    pygame.draw.circle(surface, violet, \
                        (X,Y), 5)

    pygame.display.flip()
    pygame.time.delay(200)
```

When you run the program, you realize something is wrong. The pixel does move by 2 pixels for each step. BUT ... once it collides with the wall, the moves backward by 1 pixel for each step. That's because of this:

```
    if x > 639:
        x = 639
        xspeed = -1
    elif x < 0:
        x = 0
        xspeed = 1
```

See that? `xspeed` is set to -1. The `x` value drops by 1 each time. We really want `xspeed` to be -2. Of course if `xspeed` started off with 3, we want `xspeed` to be -3 after collision with the right wall.

In other words the the `if-elif` should be this:

```
    if x > 639:
        x = 639
        xspeed = -xspeed
    elif x < 0:
        x = 0
        xspeed = -xspeed
```

Now do the same for `yspeed`, `Xspeed`, and `Yspeed`. Run your program and check that it's now correct even when the speeds are not 1 or -1.

Don't peek ahead if you have not tried to finish your modification.

The Third Animation Again

Here's the final program from the previous section:

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)

x = 50
xspeed = 2
y = 60
yspeed = 1

X = 600
Xspeed = 1
Y = 300
Yspeed = 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x = x + xspeed
    if x > 639:
        x = 639
        xspeed = -xspeed
    elif x < 0:
        x = 0
        xspeed = -xspeed

    y = y + yspeed
    if y > 479:
        y = 479
        yspeed = -yspeed
    elif y < 0:
        y = 0
        yspeed = -yspeed
```

```
    X = X + Xspeed
    if X > 639:
        X = 639
        Xspeed = -Xspeed
    elif X < 0:
        X = 0
        Xspeed = -Xspeed

    Y = Y + Yspeed
    if Y > 479:
        Y = 479
        Yspeed = -Yspeed
    elif Y < 0:
        Y = 0
        Yspeed = -Yspeed

    surface.fill(black)
    pygame.draw.line(surface, violet, \
                     (x,y), (X,Y))

    pygame.display.flip()
```

How to Prevent Big Headaches

As you can see the third animation program is longer than our average program. A real game is even longer.

Long programs are hard to write. Not only that they are harder to understand. You will find that in writing long programs, you frequently have to go back and modify what you wrote long time ago. So it's really important to write programs in a readable way. Be neat. Use meaningful variables. Put comments in your code. Etc.

Now, take a look at the third animation program. Do you see that parts of the code really look very similar?

```
x = x + xspeed
if x > 639:
   x = 639
   xspeed = -xspeed
elif x < 0:
    x = 0
    xspeed = xspeed
```

```
X = X + Xspeed
if X > 639:
   X = 639
   Xspeed = -Xspeed
elif X < 0:
   X = 0
   Xspeed = Xspeed
```

See that??? They both look like this extra code:

```
    d = d + v
    if d > 639:
        d = 639
        v = -v
    elif d < 0:
        d = 0
        v = v
```

Do you see that if `d` and `v` are replaced by `x` and `xspeed` you get the first piece of code? And that if you replace `d` and `v` by `X` and `Xspeed` you get the second?

You have to see it before you move on. It's important.

When you see code repetition, you should think of ...

functions ...

A Quickie on Functions

Let's do a simpler example. Suppose I want to do this:

```
x = 5
x = x + 1
print x
x = x + 1
print x
```

Note that `x = x + 1` occurs twice. Of course you know that this statement increments the value of x by 1.

I want to create a **function** that accepts the value of `x`, adds one to it, and passes the value back to the original x. Here's how you do it. I'm going to create a function called `blah`:

```
def blah(a):
    a = a + 1
    return a

print "running the main program ..."
x = 5
x = blah(x)
print "back in main program ... x =", x
```

I'm going to add some print statements to help you following the execution of the program:

```
def blah(a):
    print "yoohoo ... i'm in blah ..."
    print "a =", a
    a = a + 1
    print "a =", a
    print "i'm going back returning ", a
    return a

print "running the main program ..."
x = 5
x = blah(x)
print "back in main program ... x =", x
```

Exercise. What is the first statement executed?

The first thing you notice is that the first statement in the function `blah` is not the first statement to execute: look at your output screen.

The statements in the function executes only when you see `blah` in a statement, i.e., when you call the function:

```
x = blah(x)
```

So what happens when this statement is executed?

Python will take the value of x, which is 5 (look at the code), and use that as ***input*** into the function `blah`.

So let's jump to `blah` ... we'll come back to the statement `x = blah(x)` later.

Now in function `blah`, notice that there is a variable `a` that's between parentheses:

```
def blah(a):
```

This means that the value `5` is received by `a`; `a` is set to `5`.

Immediately after deciding the value of `a`, Python starts executing the statement in the body of the `blah`.

```
print "yoohoo ... i'm in blah ..."
print "a =", a
a = a + 1
print "a =", a
print "i'm going back returning ", a
return a
```

There is no mystery here ... except for the statement containing the word **`return`**. That's new.

Python will take what's after the word `return` and try to compute a value. The point of the word `return` is that this value is the ***output*** of the function. Not only that, it tells Python to go back to where it came from before entering the `blah` function. Anyway at this point the value of `a` is 6. So right now the output of the `blah` function is 6. Let's return to where we came from ...

Recall that we called the `blah` function from this statement in the main program:

```
x = blah(x)
```

This means that 6 (the output for this call of the `blah` function) will be given to `x`.

We are now out of the `blah` function and continue executing the statements in the main program.

Here's another program to help you understand functions. Make sure you run it. Try to understand how it works:

```
def bleh(b):
    print "i'm in bleh ... b =", b
    print "about to return with", b + 3
    return b + 3

y = 5
y = bleh(y)
print y

z = bleh(y)
print z

x = bleh(y + 2)
print x

a = bleh(1)
print a
```

Exercise. Complete the following program. The `foo` function accepts a value and returns twice the value passed in.

```
def foo(b):
    return ________________

x = 5
y = foo(x)
print y
```

You should see 10.

Exercise. Complete the following program. The `square` function accepts a value and returns the square of that

value:

```
def ___________(_____):
    return ___________________

x = 5
y = square(x)
print y
```

You should see 25.

Exercise. Compute the following program. The `cube` function accepts a value and returns the cube of that value:

```
# write your cube function here

y = cube(5)
print y
```

You should see 125.

You can also pass in two values and return two values. Here's an example:

```
def bleh(a, b):
    print "yoohoo ... i'm in bleh ..."
    print "a =", a
    print "b =", b
    a = a + 1
    b = b - 1
    print "i'm going back with ", a, b
    return a, b

print "running the main program ..."
x = 5
y = 7
x, y = bleh(x, y)
print "back in main program ... ", x, y
```

Run the program and read the following carefully ...

When you call `bleh(x,y)` the value of `x` is passed to `a` and the value of `y` is passed to `b`. After some computation, the statement

```
return a, b
```

will pass the value of `a` to `x` and the value of `b` to `y` because

of the statement

```
x, y = bleh(a, b)
```

In fact you can pass in as many values as you like and you can return as many values as you like. See if you can understand this program. Ask questions if you need help.

```
def bar(a, b, c):
    return a + b + c

x = 1
y = 2
z = 3
w = bar(x, y, z)
print w

w = bar(-1, 0, 1)
print w

w = bar(x + 1, y, z + 2)
print w
```

That's enough for now. Let's get back to the third animation program. This is only a quick introduction to functions. We'll come back to functions later.

Functions for Third Animation Program

Quickly review the last section where we talked about the third animation program.

I'm going to create a function. I'll call it `move`.

```
def move(d, v):
    d = d + v
    if d > 639:
        d = 639
        v = -v
    elif d < 0:
        d = 0
        v = v
    return d, v
```

I'm going to put that into my third animation program and call it when I need to change `x`, `xspeed` and `X`, `Xspeed`:

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)

def move(d, v):
    d = d + v
    if d > 639:
        d = 639
        v = -v
    elif d < 0:
        d = 0
        v = v
    return d, v

x = 50
xspeed = 2
y = 60
yspeed = 1
```

```
X = 600
Xspeed = 1
Y = 300
Yspeed = 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x, xspeed = move(x, xspeed)

    y = y + yspeed
    if y > 479:
        y = 479
        yspeed = -yspeed
    elif y < 0:
        y = 0
        yspeed = -yspeed

    X, Xspeed = move(X, Xspeed)

    Y = Y + Yspeed
    if Y > 479:
        Y = 479
        Yspeed = -Yspeed
    elif Y < 0:
        Y = 0
        Yspeed = Yspeed

    surface.fill(black)
    pygame.draw.line(surface, violet, \
                     (x,y), (X,Y))

    pygame.display.flip()
```

Run it to make sure it still works.

But I now notice immediately that it's the same for `y`, `yspeed` and `Y`, `Yspeed` as well!!! So I'm going to do the same.

Hmmm ... I think I will rename my first function as `xmove` since it moves along the x direction and I'll call the `y`

direction version `ymove`. How's that?

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)

def xmove(d, v):
    d = d + v
    if d > 639:
        d = 639
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

def ymove(d, v):
    d = d + v
    if d > 479:
        d = 479
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

x = 50
xspeed = 2
y = 60
yspeed = 1

X = 600
Xspeed = 1
Y = 300
Yspeed = 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
```

```
            sys.exit()

    x, xspeed = xmove(x, xspeed)
    y, yspeed = ymove(y, yspeed)
    X, Xspeed = xmove(X, Xspeed)
    Y, Yspeed = ymove(Y, Yspeed)

    surface.fill(black)
    pygame.draw.line(surface, violet, \
                     (x,y), (X,Y))

    pygame.display.flip()
```

Now if you stare ***really*** hard at the two functions `xmove` and `ymove`, you realize that they are actually very similar.

```
def xmove(d, v):
    d = d + v
    if d > 639:
        d = 639
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v
```

```
def ymove(d, v):
    d = d + v
    if d > 479:
        d = 479
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v
```

The only difference is that `xmove` has the number `639` and `ymove` has `479`. Furthermore these numbers appear at the same spot ... hmmm ...

AHA!!! What about this ...

```
import pygame
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)

def move(d, v, m):
    d = d + v
    if d > m:
        d = m
```

```
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

x = 50
xspeed = 2
y = 60
yspeed = 1

X = 600
Xspeed = 1
Y = 300
Yspeed = 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x, xspeed = move(x, xspeed, 639)
    y, yspeed = move(y, yspeed, 479)
    X, Xspeed = move(X, Xspeed, 639)
    Y, Yspeed = move(Y, Yspeed, 479)

    surface.fill(black)
    pygame.draw.line(surface, violet, \
                     (x,y), (X,Y))

    pygame.display.flip()
```

Read this new version very slowly and study it carefully. Compare it with the previous version. If you don't see how I got to this step, please ask for help!

See how small the program is now?

Unpacking

By the way instead of writing two assignments:

```
a = 1
b = 3
print a, b
```

you can bunch them up into one like so:

```
a, b = 1, 3
print a, b
```

This is called **unpacking**.

What happens is that the `1,3` is really the same as the tuple `(1,3)`. The assignment

```
a, b = 1, 3
```

basically unpacks the tuple `(1,3)` to get `1` and `3`, and then distributes the values to `a` and `b`.

Incidentally if you ever need to **swap** the values of two variables you can do this:

```
a = 0
b = 1
print a, b
a, b = b, a # swap
print a, b
```

Now we're going to use unpacking to clean up the code a little bit more.

Since we're used to viewing `x` and `y` together as a position of a point I would like to do this (make sure you see where I've modified the program):

```
import pygame
WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)
```

```
def move(d, v, m):
    d = d + v
    if d > m:
        d = m
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

x, y = 50, 60
xspeed, yspeed = 2, 1

X, Y = 600, 300
Xspeed, Yspeed = 1, 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x, xspeed = move(x, xspeed, 639)
    y, yspeed = move(y, yspeed, 479)
    X, Xspeed = move(X, Xspeed, 639)
    Y, Yspeed = move(Y, Yspeed, 479)

    surface.fill(black)
    pygame.draw.line(surface, violet, \
                     (x,y), (X,Y))

    pygame.display.flip()
```

Try to read this program and see how much easier it is to say “the first point starts at 50, 60” when compared to the earlier version.

One Last Change

One last change and I'm done. Go ahead and take a quick glance at the latest version of the third animation program.

In the first and third call to the `move` function the number `639` comes from collision with the right side of the screen since the width of the screen is `640`.

```
...
    x, xspeed = move(x, xspeed, 639)
    y, yspeed = move(y, yspeed, 479)
    X, Xspeed = move(X, Xspeed, 639)
    Y, Yspeed = move(Y, Yspeed, 479)
...
```

I already have a constant `WIDTH` with value `640`. I prefer to use `WIDTH - 1` instead of the value `639`:

```
...
    x, xspeed = move(x, xspeed, WIDTH - 1)
    y, yspeed = move(y, yspeed, 479)
    X, Xspeed = move(X, Xspeed, WIDTH - 1)
    Y, Yspeed = move(Y, Yspeed, 479)
...
```

Why is that the case? Well if I need to change my game window from a width of 640 to say a width of 600, then I only need to change it in the value assigned to `WIDTH`. With my old program, I would need to change the third value passed into two calls to the `move` function as well.

In general the practice in writing programs is that **if a value has a special meaning, you should use a constant variable rather than that value directly**.

Of course I also need to change the `479` to `HEIGHT - 1`. So altogether now I have this version:

```
import pygame
WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

violet = pygame.Color("violet")
black = (0,0,0)

def move(d, v, m):
    d = d + v
    if d > m:
        d = m
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

x, y = 50, 60
xspeed, yspeed = 2, 1

X, Y = 600, 300
Xspeed, Yspeed = 1, 1

while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    x, xspeed = move(x, xspeed, WIDTH - 1)
    y, yspeed = move(y, yspeed, HEIGHT - 1)
    X, Xspeed = move(X, Xspeed, WIDTH - 1)
    Y, Yspeed = move(Y, Yspeed, HEIGHT - 1)

    surface.fill(black)
    pygame.draw.line(surface, violet, \
                     (x,y), (X,Y))
    pygame.display.flip()
```

Now I'm done.

Exercise. Modify the program so that the line has randomly chosen starting/end points, speeds, and color. Try different values for `WIDTH` and `HEIGHT`.