

# CHAPTER 2 – PYGAME BASICS

---

Just like how Python comes with several modules like `random`, `math`, or `time` that provide additional functions for your programs, the Pygame framework includes several modules with functions for drawing graphics, playing sounds, and handling mouse input.

This chapter will cover the basic modules and functions that Pygame provides and assumes you already know basic Python programming. If you have trouble with some of the programming concepts, you can read through the “Invent Your Own Computer Games with Python” book online at <http://inventwithpython.com>. This book is aimed at complete beginners to programming.

## GUI vs. CLI

The Python programs that you can write with Python’s built-in functions only deal with text through the `print()` and `input()` functions. Your program can display text on the screen and let the user type in text from the keyboard. This type of program has a **Command Line Interface**, or **CLI** (which sounds like the first part of “climb” and rhymes with “sky”). These programs are somewhat limited because they can’t display pictures, have colors, or use the mouse. These CLI programs only get input from the keyboard with the `input()` function and even then user must press enter before the program can respond to the input. This means **real-time** (that is, continuing to run code without waiting for the user) action games are impossible to make.

Pygame provides functions for creating programs with a **Graphical User Interface**, or **GUI** (pronounced, “goeey”). Instead of a text-based CLI, programs with a graphics-based GUI can show a window with images and colors.

## Hello World with Pygame

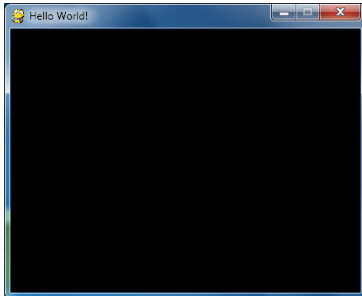
Our first program made with Pygame is a small program that makes a window that says “Hello World!” appear on the screen. Type in the following code into IDLE’s file editor and save it as *helloworld.py*. Then run the program by pressing **F5** or selecting **Run > Run Module** from the menu at the top of the file editor.

Remember, do not type the numbers or the periods at the beginning of each line (that’s just for reference in this book.)

```
1. import pygame, sys
2. from pygame.locals import *
3.
4. pygame.init()
5. WINDOWSURF = pygame.display.set_mode((400, 300))
6. pygame.display.set_caption('Hello World!')
7. while True: # main game loop
8.     for event in pygame.event.get():
9.         if event.type == QUIT:
```

```
10.         pygame.quit()
11.         sys.exit()
12.     pygame.display.update()
```

When you run this program, a black window like this will appear:



Yay! You’ve just made the world’s most boring video game! It’s just a blank window with “Hello World!” at the top of the window (in what is called the window’s **title bar** or **caption**). But creating a window is the first step to making graphical games. When you click on the X in the corner of the window, the program will end and the window will disappear.

Calling the `print()` function to make text appear in the window won’t work because `print()` is a function for CLI programs. The same goes for `input()` to get keyboard input from the user. Pygame uses other functions for input and output, which are explained later in this chapter. For now, let’s look at each line in our “Hello World” program in more detail.

```
1. import pygame, sys
```

Line 1 is a simple `import` statement that imports the `pygame` and `sys` modules so that our program can use the functions in them. All of the Pygame functions dealing with graphics, sound, and other features that Pygame provides are in the `pygame` module.

```
2. from pygame.locals import *
```

Line 2 is also an `import` statement. However, instead of the `import modulename` format, it uses the `from modulename import *` format. Normally if you want to call a function that is in a module, you must use the `modulename.functionname()` format after importing the module. However, with `from modulename import *`, you can skip the `modulename.` portion and simply use `functionname()` (just like Python’s built-in functions).

The reason we use this form of `import` statement for `pygame.locals` is because `pygame.locals` contains several constant variables that are easy to identify as being in the `pygame.locals` module without `pygame.locals.` in front of them. But for all other modules, you generally want to use the regular `import modulename` format. (There is more information about why you want to do this at <http://invpy.com/namespaces>.)

```
4. pygame.init()
```

Line 4 is the `pygame.init()` function, which always needs to be called after importing the `pygame` module and before calling any Pygame function. This function contains a lot of code that gets the Pygame library ready to use. You don't need to know what this function does, you just need to know that it needs to be called first in order for many Pygame functions to work. If you see an error message like, `pygame.error: font not initialized`, check to see if you forgot to call `pygame.init()`.

```
5. WINDOWSURF = pygame.display.set_mode((400, 300))
```

Line 5 is a call to the `pygame.display.set_mode()` function, which returns the `pygame.Surface` object for the window. (Surface objects are described later in this chapter.) Notice that we pass a tuple value of two integers to the function: `(400, 300)`. This tuple tells the `set_mode()` function how wide and how high to make the window in pixels. `(400, 300)` will make a window with a width of 400 pixels and height of 300 pixels. Pass a tuple of two integers to `set_mode()`, not just two integers themselves. The `pygame.Surface` object returned is stored in the `WINDOWSURF` variable. (We will just call them Surface objects for short.)

```
6. pygame.display.set_caption('Hello World!')
```

Line 6 sets the caption text that will appear at the top of the window by calling the `pygame.display.set_caption()` function. The string value `'Hello World!'` is passed in this function call to make that text appear as the caption.

## Game Loops and Game States

```
7. while True: # main game loop
8.     for event in pygame.event.get():
```

Line 7 is a `while` loop that has a condition of simply the value `True`. This means that it is an **infinite loop** that never exits due to its condition being the value `False`. The only way the program execution will ever exit out of an infinite loop is if a `break` statement is executed (which moves execution to the first line after the loop) or `sys.exit()` (which terminates the program). Also, if the infinite loop is inside a function, a `return` statement will also move execution out of the loop (and the function too).

The games in this book all have infinite loops in them along with a comment calling it the “main game loop”. A **game loop** or **main loop** is a loop where most of the code that handles events, updates the game state, and draws the screen is done. Main loops are common in **event-driven programming** (also called **event-based programming**), which is where objects called “events” are created when the user does some event like moves the mouse, clicks a mouse button, or pushes down (or lets up on) a key on the keyboard. Inside the main loop is code that detects which events have happened, called **event detection**. (The event detection part of the loop is really just the single call to `pygame.event.get()`.) The main loop also has code that updates the game state based on which events have been detected, called **event handling**.

The **game state** is simply a way of referring to the values in all the variables in a game program. In many games, the game state includes the values in the variables that tracks the player's health and position (and the health and position of any enemies), which marks have been made on a board, or whose turn it is. Whenever something happens like the player taking damage (which lowers their health value), or an enemy moves somewhere, or it becomes someone else's turn we say that the game state has changed.

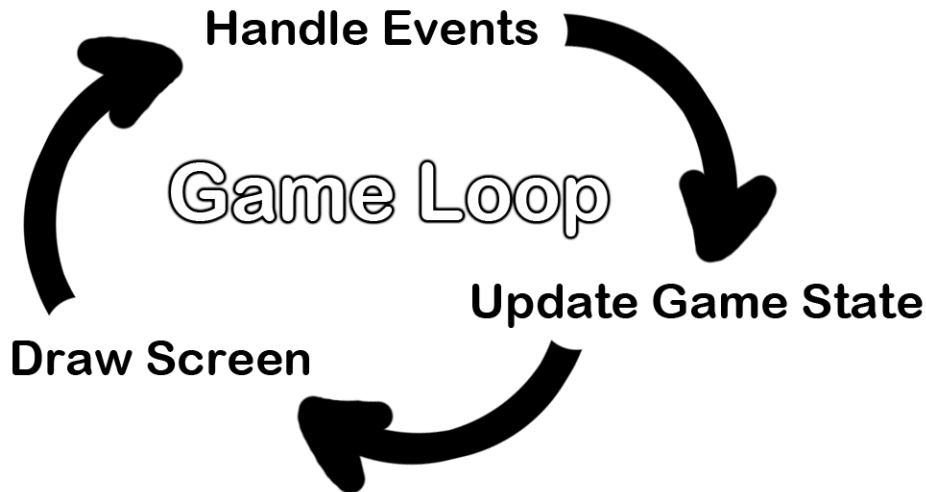
The text-based CLI games in “Invent Your Own Computer Games with Python” did not have these “game loops”. That's because these programs would make calculations or display text to the screen, and then program execution would hit an `input ()` function call and wait for the player to type something in. Until the player pressed the Enter key, the entire program execution stopped. These programs did not run in real-time, so nothing could really change until the player pressed Enter. The computer would either be quickly carrying out code in the program, or would be doing nothing while it waits for the player.

## Pygame.event.Event Objects

GUI programs are not like that. These programs can run in real-time and update the game state instead of stopping to wait for the user. Any time the user pressed a keyboard button or clicks the mouse on the program's window, a `pygame.event.Event` object is created by the Pygame library. (This is a type of object called `Event` that exists in the `event` module, which itself is in the `pygame` module.) We can find out which events have happened by calling the `pygame.event.get ()` function, which returns a list of `pygame.event.Event` objects (which we will just call `Event` objects for short.)

`Event` objects are created by the Pygame library when the user presses a keyboard button, moves the mouse, clicks a mouse button, quits the program by clicking the X in the corner of the window, or does a few other actions. The program can see all the `Event` objects that Pygame has created by calling the `pygame.event.get ()` function to return a list of `Event` objects for each event that has happened since the last time the `pygame.event.get ()` function was called. (Or, if `pygame.event.get ()` has never been called, the events that have happened since the start of the program.)

In general, the game loop in a game program does three things over and over again until the program terminates: it handles any events (such as user input), updates the game state (possibly because of something the player did, or just things about the game state that simply change over time), and then draws a representation of game state to the screen for the player to see.



Line 8 is a `for` loop that will iterate over the list of `Event` objects that were returned by `pygame.event.get()`. On each iteration through the `for` loop, a variable named `event` will be assigned the value of one of the event objects in this list. These assignments are done in order, so if the user clicked the mouse and then pressed a keyboard key, the event variable will have a mouse click event object on the first iteration and then a keyboard press event on the second iteration. If no event have been generated, then `pygame.event.get()` will return a blank list.

## The `QUIT` Event and `pygame.quit()` Function

```
9.         if event.type == QUIT:
10.             pygame.quit()
11.             sys.exit()
```

Event objects have a **member variable** (also called **attributes**) named `type` which tells us what kind of event the object represents. Pygame has a constant variable for each of possible types in the `pygame.locals` modules. Line 9 checks if the Event object's type is equal to the constant `QUIT`. (Remember that since we used the `from pygame.locals import *` form of the `import` statement, we only have to type `QUIT` instead of `pygame.locals.QUIT`.)

If the `Event` object is a quit event, then the block following the `if` statement executes. This block (which covers lines 10 and 11) has a call to the `pygame.quit()` and `sys.exit()` functions. The `pygame.quit()` function is sort of the opposite of the `pygame.init()` function: it runs code that deactivates the Pygame library. Your programs should always call `pygame.quit()` before they call `sys.exit()` to terminate the program. (Normally it doesn't really matter. But there is a bug in IDLE that causes IDLE to hang if a Pygame program terminates before `pygame.quit()` is called.)

Since we have no `if` statements that run code for other types of `Event` object, there is no event-handling code for when the user clicks the mouse, presses keyboard keys, or causes any other `Event` objects to be created. The user can do these things and it doesn't change anything in the program. After the `for` loop on line 8 is done handling all the `Event` objects that have been returned by `pygame.event.get()`, the program execution continues to line 12.

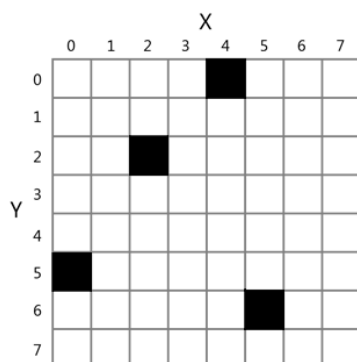
```
12. pygame.display.update()
```

Line 12 calls the `pygame.display.update()` function, which draws the Surface object returned by `pygame.display.set_mode()` to the screen (remember we stored this object in the `WINDOWSURF` variable). Since the Surface object hasn't changed (by, say, some of the drawing functions that are explained later in this chapter), the same black image is redrawn to the screen each time `pygame.display.update()` is called.

That is the entire program. After line 12 is done, the infinite while loop on line 7 starts again. This program does nothing besides make a black window appear on the screen, constantly check for a `QUIT` event, and then redraws the unchanged black window to the screen over and over again. Let's learn how to make interesting things appear on this window instead of just blackness by learning about pixels, Surface objects, and the Pygame drawing functions.

## Pixels and Screen Coordinates

The window that the "Hello World" program creates is just composed of little square dots on your screen called pixels. Each pixel starts off as black but can be set to a different color. Imagine that instead of a Surface object that is 400 pixels wide and 300 pixels tall, we just had a Surface object that was 8 pixels by 8 pixels. If that tiny 8x8 Surface was enlarged so that each pixel looks like a square in a grid, and we added numbers for the X and Y axis, then a good representation of it could look something like this:



We can refer to a specific pixel by using a Cartesian Coordinate system. Each column (called the **X-axis**) and each row (called the **Y-axis**) will have an "address" of an integer from 0 to 7 so that we can locate any pixel by specifying the X and Y axis.

For example, in the above 8x8 image, we can see that the pixels at the XY coordinates (4, 0), (2, 2), (0, 5), and (5, 6) have been painted black, while all the other pixels are painted white. XY coordinates are also called **points**. If you've taken a math class and learned about Cartesian Coordinates, you might notice that the Y-axis starts at 0 at the *top* and then increases going *down*, rather increasing as it goes up. This is just how Cartesian Coordinates work in Pygame (and almost every programming language.)

The Pygame framework often represents Cartesian Coordinates as a **tuple** of two integers, such as (4, 0) or (2, 2). The first integer is the X coordinate and the second is the Y coordinate. (Cartesian Coordinates

are covered in more detail in chapter 12 of “Invent Your Own Computer Games with Python” at <http://invpy.com/chap12>)

## Surface Objects and The Window

Surface objects are rectangular representations of a 2D image. You can program the computer to draw on them to change the color of their pixels, and then display a Surface object on the screen. The Surface object returned by `pygame.display.set_mode()` is called the **display Surface**. The window border and the title bar and buttons are not part of the display Surface object.

Anything that is drawn on the display Surface object will be displayed on the window when the `pygame.display.update()` function is called. It is a lot faster to draw on a Surface object (which only exists in the computer’s memory) than it is to draw a Surface object to the computer screen. Computer memory is much faster to change than pixels on a monitor. Often, your program will draw several different things to a Surface object. Once you are done drawing this image (called a **frame**, just like a still image on a paused DVD is called) on a Surface object, it can be drawn to the screen. The computer can draw frames very quickly, and our programs will often run around 30 frames per second. (This is called the “framerate” and is explained later in this chapter.)

Drawing on Surface objects will be covered in the “Primitive Drawing Functions” and “Drawing Images” sections later this chapter.

## Colors

There are three primary colors of light: red, green and blue. (Red, blue, and yellow are the primary colors for paints and pigments, but the computer monitor uses light, not paint.) By combining different amounts of these three colors you can form any other color. In Python, we represent colors with tuples of three integers. The first value in the tuple is how much red is in the color. An integer value of 0 means there is no red in this color, and a value of 255 means there is a maximum amount of red in the color. The second value is for green and the third value is for blue. These tuples of three integers used to represent a color are often called **RGB values**.

Because you can use any combination of 0 to 255 for each of the three primary colors, this means Pygame can draw 16,581,375 different colors (that is,  $255 \times 255 \times 255$  colors). However, if try to use a number larger than 255 or a negative number, you will get an error that looks like “ValueError: invalid color argument”.

For example, we will create the tuple `(0, 0, 0)` and store it in a variable named `BLACK`. With no amount of red, green, or blue, the resulting color is completely black. The color black is the absence of any color.

The tuple `(255, 255, 255)` for a maximum amount of red, green, and blue to result in white. The color white is the full combination of red, green, and blue. The tuple `(255, 0, 0)` represents the maximum amount of red but no amount of green and blue, so the resulting color is red. Similarly, `(0, 255, 0)` is green and `(0, 0, 255)` is blue.

You can mix the amount of red, green, and blue to form other colors. For example, yellow is (255, 255, 0), a mixture of red and green. Combining a little bit of red and green as (128, 128, 0) will give you purple. RGB values are used not only in Pygame, but in many other libraries and pieces of software.

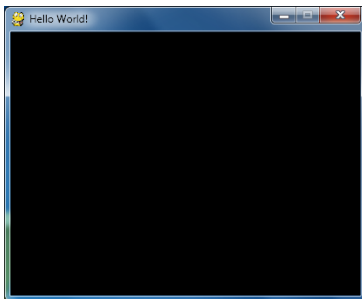
## Transparent Colors

When you look through a glass window that has a deep red tint, all of the colors behind it have a red shade added to them. You can mimic this effect by adding a fourth 0 to 255 integer value to your color values.

This value is known as the **alpha value**. It is a measure of how transparent a color is. Normally when you draw a pixel onto a surface object, the new color completely replaces whatever color was already there. But with colors that have an alpha value, you can instead just add a colored tint to the color that is already there.

For example, this tuple of three integers is for the color green: (0, 255, 0). But if we add a fourth integer for the alpha value, we can make this a half transparent green color: (0, 255, 0, 128). An alpha value of 255 is completely opaque, that is, no transparency at all (the colors (0, 255, 0) and (0, 255, 0, 255) look exactly the same. An alpha value of 0 means the color is completely transparent. If you draw any color that has an alpha value of 0 to a surface object, it will have no effect, because this color is completely transparent and invisible.

If we were to create a color tuple to draw the legendary Invisible Pink Unicorn, we would use (255, 192, 192, 0), which ends up looking completely invisible just like any other color that has a 0 for its alpha value. (How people can know what color an invisible unicorn is, I have no idea.)



(Above is a screenshot of a Pygame program that has a drawing of the Invisible Pink Unicorn.)

## pygame.Color Objects

You need to know how to represent a color because Pygame's drawing functions need a way to know what color you want to draw with. A tuple of three or four integers is one way. Another way is as a `pygame.Color` object. You can create `Color` objects by calling the `pygame.Color()` constructor function and passing either three or four integers. You can store this `Color` object in variables just like you can store tuples in variables. Try typing the following into the interactive shell:

```
>>> import pygame
>>> pygame.Color(255, 0, 0)
(255, 0, 0, 255)
>>> myColor = pygame.Color(255, 0, 0, 128)
>>> myColor == (255, 0, 0, 128)
True
>>>
```

Any drawing function in Pygame (which we will learn about in a bit) that has a color parameter can have either the tuple form or `Color` object form of a color passed for it.

Now that you know how to represent colors (as a `pygame.Color` object or a tuple of three or four integers for red, green, blue, and optionally alpha) and coordinates (as a tuple of two integers for X and Y), let's learn about `pygame.Rect` objects so we can start using Pygame's drawing functions.

## Rect Objects

Remember that in Pygame we can represent points as a tuple of two integers for the X and Y coordinate. Pygame also has two ways to represent rectangular areas. The first is a tuple of four integers for the X coordinate of the top left corner, the Y coordinate of the top left corner, and the width and then height of the rectangle. The second way is as a `pygame.Rect` object, which we will call Rect objects for short. (This is just like how colors can be represented as a tuple or a `Color` object.)

For example, this creates a Rect object with a top left corner at (10, 20) that is 200 pixels wide and 300 pixels tall:

```
>>> import pygame
>>> spamRect = pygame.Rect(10, 20, 200, 300)
```

The handy thing about this is that the Rect object automatically calculates the coordinates for other features of the rectangle. For example, if you need to know the x coordinate of the right edge of the `pygame.Rect` object we stored in the `spamRect` variable, you can just access the Rect object's `right` attribute:

```
>>> spamRect.right
210
```

The Pygame code for the Rect object automatically calculated that if the left edge is at the X coordinate 10 and the rectangle is 200 pixels wide, then the right edge must be at the X coordinate 210. If you reassign the right attribute, all the other attributes are automatically recalculated:

```
>>> spam.right = 350
>>> spam.left
150
```

Here's a list of all the attributes that `pygame.Rect` objects provide (in our example, the variable where the Rect object is stored is named `myRect`):

Attribute Name	Description
<code>myRect.left</code>	The int value of the X-coordinate of the left side of the rectangle.
<code>myRect.right</code>	The int value of the X-coordinate of the right side of the rectangle.
<code>myRect.top</code>	The int value of the Y-coordinate of the top side of the rectangle.
<code>myRect.bottom</code>	The int value of the Y-coordinate of the bottom side.
<code>myRect.centerx</code>	The int value of the X-coordinate of the center of the rectangle.
<code>myRect.centery</code>	The int value of the Y-coordinate of the center of the rectangle.
<code>myRect.width</code>	The int value of the width of the rectangle.
<code>myRect.height</code>	The int value of the height of the rectangle.
<code>myRect.size</code>	A tuple of two ints: (width, height)
<code>myRect.topleft</code>	A tuple of two ints: (left, top)
<code>myRect.topright</code>	A tuple of two ints: (right, top)
<code>myRect.bottomleft</code>	A tuple of two ints: (left, bottom)
<code>myRect.bottomright</code>	A tuple of two ints: (right, bottom)
<code>myRect.midleft</code>	A tuple of two ints: (left, centery)
<code>myRect.midright</code>	A tuple of two ints: (right, centery)
<code>myRect.midtop</code>	A tuple of two ints: (centerx, top)
<code>myRect.midbottom</code>	A tuple of two ints: (centerx, bottom)

## Primitive Drawing Functions

Pygame provides several different functions for drawing different shapes onto a surface object. These shapes such as rectangles, circles, ellipses, lines, or individual pixels are often called **drawing primitives**. Open IDLE's file editor and type in the following program, and save it as *drawing.py*.

```

1. import pygame, sys
2. from pygame.locals import *
3.
4. pygame.init()
5.
6. # set up the window
7. WINDOWSURF = pygame.display.set_mode((500, 400), 0, 32)
8. pygame.display.set_caption('Drawing')
9.
10. # set up the colors
11. BLACK = ( 0,  0,  0)
12. WHITE = (255, 255, 255)
13. RED   = (255,  0,  0)
14. GREEN = ( 0, 255,  0)
15. BLUE  = ( 0,  0, 255)
16.
17. # draw on the surface object
18. WINDOWSURF.fill(WHITE)
19. pygame.draw.polygon(WINDOWSURF, GREEN, ((146, 0), (291, 106), (236, 277), (56, 277), (0, 106)))

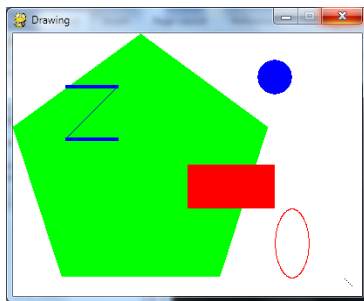
```

```

20. pygame.draw.line(WINDOWSURF, BLUE, (60, 60), (120, 60), 4)
21. pygame.draw.line(WINDOWSURF, BLUE, (120, 60), (60, 120))
22. pygame.draw.line(WINDOWSURF, BLUE, (60, 120), (120, 120), 4)
23. pygame.draw.circle(WINDOWSURF, BLUE, (300, 50), 20, 0)
24. pygame.draw.ellipse(WINDOWSURF, RED, (300, 250, 40, 80), 1)
25. pygame.draw.rect(WINDOWSURF, RED, (200, 150, 100, 50))
26.
27. pixObj = pygame.PixelArray(WINDOWSURF)
28. pixObj[480][380] = BLACK
29. pixObj[482][382] = BLACK
30. pixObj[484][384] = BLACK
31. pixObj[486][386] = BLACK
32. pixObj[488][388] = BLACK
33. del pixObj
34.
35. # run the game loop
36. while True:
37.     for event in pygame.event.get():
38.         if event.type == QUIT:
39.             pygame.quit()
40.             sys.exit()
41.     pygame.display.update()

```

When this program is run, the following window is displayed until the user closes the window:



Notice how we make constant variables for each of the colors. Doing this makes our code more readable, because seeing `GREEN` in the source code is much easier to understand as representing the color green than `(0, 255, 0)` is.

The drawing functions are named after the shapes they draw. The parameters you pass these functions tell them which surface to draw on, where to draw the shape (and what size), in what color, and how wide to make the lines. You can see how these functions are called in the *drawing.py* program, but here is a short description of each function:

- **fill(color)** – The `fill()` method is not a function but a method of `pygame.Surface` objects. It will completely fill in the entire Surface object with whatever color value you pass as for the `color` parameter. It is often called on the display Surface to start drawing each frame from scratch.
- **pygame.draw.polygon(surface, color, pointlist, width)** – A polygon is shape made up of only flat sides. Triangles, squares, and pentagons are all examples of polygons. Circles are not polygons

because they do not have flat sides. The `surface` and `color` parameters tell the function on what surface to draw the polygon, and what color to make it.

The third parameter is a tuple or list of points (that is, tuple or list of two-integer tuples for XY coordinates.) The polygon is drawn by drawing lines between each point and the point that comes after it in the tuple. Then a line is drawn from the last point to the first point. You can also pass a list of points instead of a tuple of points.

The `width` parameter is optional. If you leave it out, the polygon that is drawn will be filled in, just like our green polygon on the screen is filled in with the green color. If you do pass an integer value for the `width` parameter, only the outline of the polygon will be drawn. The integer represents how many pixels width the polygon's outline will be. Passing 1 for the `width` parameter will make a skinny polygon, while passing 4 or 10 or 20 will make thicker polygons. If you pass the integer 0 for the `width` parameter, the polygon will be filled in (just like if you left the `width` parameter out entirely.)

All of the `pygame.draw` drawing functions have optional `width` parameters at the end, and they work the same way as `pygame.draw.polygon()`'s `width` parameter. Probably a better name for the `width` parameter would have been "thickness", since that parameter controls how thick the shapes you draw are.

- **`pygame.draw.line(surface, color, start_point, end_point, width)`** – This function draws a line between the `start_point` and `end_point` parameters. You can also set what color and the width of the line. (Pygame draws really, really thick lines kind of funny. Most of the time this doesn't matter, but you can read <http://invpy.com/thicklines> for more information.)
- **`pygame.draw.lines(surface, color, closed, pointlist, width)`** – This function draws a series of lines from one point to the next, much like `pygame.draw.polygon()`. The only difference is that if you pass `False` for the `closed` parameter, there will not be a line from the last point in the `pointlist` parameter to the first point. (If you pass `True`, then it will draw a line from the last point to the first.)
- **`pygame.draw.circle(surface, color, center_point, radius, width)`** – This function draws a circle. The center of the circle is at the `center_point` parameter. The size of the circle is determined by the integer passed for the `radius` parameter.

The radius of a circle is the distance from the center to the edge. (The radius of a circle is always half of the diameter.) Passing 20 for the `radius` parameter will draw a circle that has a radius of 20 pixels.

- **`pygame.draw.ellipse(surface, color, bounding_rectangle, width)`** – This function draws an ellipse, which is like a squashed or stretched circle. This function has all the usual parameters, but in order to tell the function how large and where to draw the ellipse, you must specify the bounding rectangle of the ellipse. A **bounding rectangle** is the smallest rectangle that can be drawn around a shape. Here's an example of an ellipse and its bounding rectangle:



Remember that instead of passing a tuple of four integers for the `bounding_rectangle` parameter, you can pass a `pygame.Rect` object. Note that you do not specify the center point for the ellipse like you do for the `pygame.draw.circle()` function, just the bounding rectangle.

- **`pygame.draw.rect(surface, color, rectangle_tuple, width)`** – This function draws a rectangle. The `rectangle_tuple` is either a tuple of four integers (for the XY coordinates of the top left corner,

and the width and height) or a `pygame.Rect` object can be passed instead. If the `rectangle_tuple` has the same size for the width and height, a square will be drawn.

## pygame.PixelArray Objects

Unfortunately, there isn't a single function you can call that will set a single pixel to a color (unless you call `pygame.draw.rect()` with a 1x1 rectangle or call `pygame.draw.line()` with the same start and end point). The Pygame framework needs to run some code behind the scenes before and after drawing to a Surface object. If it had to do this for every single pixel you wanted to set, your program would run much slower. (By my quick testing, drawing pixels becomes two or three times slower.)

Instead, you should tell Pygame to do the set up just once. This is what line 27 does when it creates a `pygame.PixelArray` object (we'll call them PixelArray objects for short). The Surface object that we want to draw pixels on is passed as an argument. Creating a PixelArray object of a Surface object will "lock" the Surface object. While a Surface object is locked, the drawing functions can still be called on it, but it cannot have images like PNG or JPG images drawn on it with the `blit()` method. (This method is explained later in this chapter.)

The PixelArray object that is returned can have individual pixels set by accessing them with two indexes. For example, line 28's `pixObj[480][380] = BLACK` will set the pixel at X coordinate 480 and Y coordinate 380 to be black (remember that the `BLACK` variable stores the color tuple `(0, 0, 0)`).

To tell Pygame that you are finished drawing individual pixels, delete the PixelArray object with a `del` statement. This is what line 33 does. Deleting the PixelArray object will "unlock" the Surface object so that you can once again draw images on it. If you forget to delete the PixelArray object, the next time you try to blit (that is, draw) an image to the Surface, the program will raise an error that says, `"pygame.error: Surfaces must not be locked during blit"`.

## The pygame.display.update() Function

After you are done calling the drawing functions to make the display Surface object look the way you want, you must call `pygame.display.update()` to make the display Surface actually appear on the user's monitor.

The one thing that you must remember is that `pygame.display.update()` will only make the display Surface (that is, the `pygame.Surface` object that was returned from the call to `pygame.display.set_mode()`) appear on the screen. If you want the images on other Surface objects to appear on the screen, you must "blit" them to the display Surface object first with the `blit()` method (which is explained next in the "Drawing Images" section.)

## Drawing Images

Making good graphics by just using the drawing functions can be hard. It's usually much easier to use graphics software to draw images and then save them to an image file. Or you can just go on the Internet and download image files that you want use in your games. Pygame has functions for loading and

displaying images stored in these files on the screen. The differences between these image file formats is explained at <http://invpy.com/formats>, but in general I recommend always using PNG images.

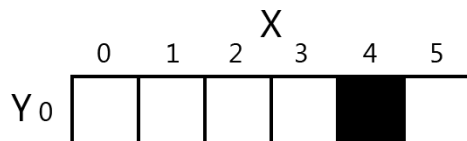
You can load an image file with the `pygame.image.load()` function. It has one parameter, which is a string of the filename of the image file. The function then returns a Surface object that has the image drawn on it. This function stores a Surface object of some cat picture in the `catImg` variable:

```
catImg = pygame.image.load('cat.png')
```

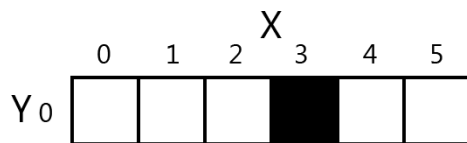
Of course, only the display Surface object can be drawn to the screen when `pygame.display.update()` is called. So we have to copy the `catImg` Surface object to the display Surface object by “blitting” it to the display Surface. Blitting is done with the `blit()` method, which is a method of Surface objects. Here is a `blit()` method call that will copy

## Animation

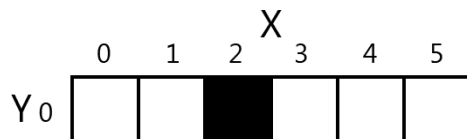
Now that we know how to get the Pygame framework to draw to the screen, let’s learn how to make animated pictures. A game with only still, unmoving images would be fairly dull. (Sales of my games “Look At This Rock” and “Look At This Rock 2: A Different Rock” have been disappointing. <http://invpy.com/rock>) Animated images are the result of drawing an image on the screen, then a split second later drawing a slightly different image on the screen. Imagine the program’s window was 6 pixels wide and 1 pixel tall, with all the pixels white except for a black pixel at 4, 0. It would look like this:



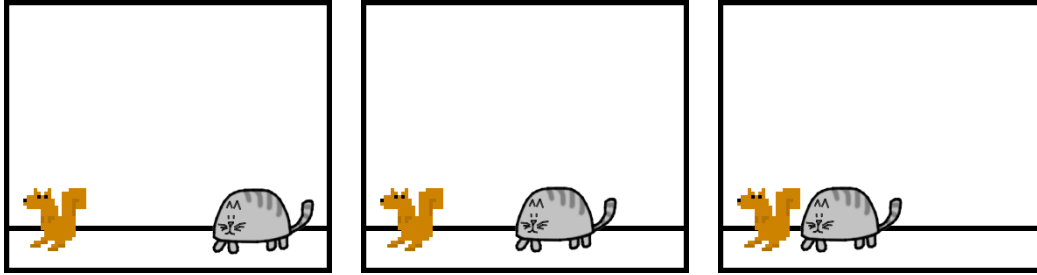
If you changed the window so that 3, 0 was black and 4,0 was white, it would look like this:



To the user, it looks like the black pixel has “moved” over to the left. If you redrew the window to have the black pixel at 2, 0, it would continue to look like the black pixel is moving left:



It may look like the black pixel is moving, but this is just an illusion. To the computer, it is just showing three different images that each have just one black pixel. Consider if the three following images were rapidly shown on the screen:



To the user, it would look like the cat is moving towards the squirrel. But to the computer, they're just a bunch of pixels. The trick to making believable looking animation is to have your program draw a picture to the window, wait a fraction of a second, and then draw a slightly different picture.

Here is an example program demonstrating a simple animation. Type this code into IDLE's file editor and save it as *animation.py*. It will also require the image file *cat.png* to be in the same folder as the *animation.py* file. You can download this image from <http://invpy.com/cat.png>. This code is available at

```
1. import pygame, sys
2. from pygame.locals import *
3.
4. pygame.init()
5.
6. FPS = 30 # frames per second setting
7. fpsClock = pygame.time.Clock()
8.
9. # set up the window
10. WINDOWSURF = pygame.display.set_mode((400, 300), 0, 32)
11. pygame.display.set_caption('Animation')
12.
13. WHITE = (255, 255, 255)
14. catImg = pygame.image.load('cat.png')
15. catx = 10
16. caty = 10
17. direction = 'right'
18.
19. while True: # the main game loop
20.     WINDOWSURF.fill(WHITE)
21.
22.     if direction == 'right':
23.         catx += 5
24.         if catx == 280:
25.             direction = 'down'
26.     elif direction == 'down':
27.         caty += 5
28.         if caty == 220:
29.             direction = 'left'
30.     elif direction == 'left':
31.         catx -= 5
32.         if catx == 10:
33.             direction = 'up'
34.     elif direction == 'up':
```

```
35.         caty -= 5
36.         if caty == 10:
37.             direction = 'right'
38.
39.     WINDOWSURF.blit(catImg, (catx, caty))
40.
41.     for event in pygame.event.get():
42.         if event.type == QUIT:
43.             pygame.quit()
44.             sys.exit()
45.
46.     pygame.display.update()
47.     fpsClock.tick(FPS)
```

## Frames Per Second and `pygame.time.Clock` Objects

The number of pictures that the program draws per second is commonly called the **framerate**, and is measured in **FPS** or **frames** per second. (On computer monitors, the common name for FPS is hertz. Many monitors have a framerate of 60 hertz, or 60 frames per second.) A low frame rate in video games can make the game look choppy or jumpy. This is usually caused when the program has to compute a large amount of code in between drawing frames to the screen. If the program cannot run fast enough to draw to the screen frequently, then the FPS goes down. (But the games in this book are simple enough that this won't be issue even on old computers.)

A `pygame.time.Clock` object can help us make sure our program runs at a certain FPS. This `Clock` object will set the general speed that our programs run at by putting in small pauses on each iteration of the game loop. If we didn't have these pauses, our game program would run as fast as the computer could run it. This is often too fast for the player, and as computers get faster they would run the game faster too. A `Clock` object makes sure the game runs at the same speed no matter how fast of a computer it runs on. The `Clock` object is created on line 7 of the *animation.py* program.

```
7. fpsClock = pygame.time.Clock()
```

The `Clock` object's `tick()` method is called to put a pause in the program. This method should be called at the very end of the game loop and after the call to `pygame.display.update()`. The length of the pause is calculated based on how long it took to execute the code in the game loop. In the *animation* program, is it run on line 47 right before the end of the game loop.

```
47.     fpsClock.tick(FPS)
```

Try modifying the FPS constant variable to run the same program at different frame rates. Setting it to a lower value would make the program run slower. Setting it to a higher value would make the program run faster.

## Drawing Images

The drawing functions are fine if you want to draw simple shapes on the screen, but many games have images (also called **sprites**) as the main part of their graphics. Pygame is able to draw images to Surface objects from PNG, JPG, GIF, and BMP files. (The differences between these image file formats is described at <http://invpy.com/formats>.)

The string with the image file's filename is passed to the `pygame.image.load()` function, which then returns a Surface object that has the image drawn on it. (If you get an error message like “`pygame.error: Couldn't open cat.png`”, then make sure the `cat.png` file is in the same folder as the `animation.py` file before you run the program.) This Surface object will be a separate Surface from the display Surface object, so we must blit (that is, copy) the image's Surface object to the display Surface object. **Blitting** is a name for drawing the contents of one Surface onto another and in Pygame it is done with the `blit()` method of Surface objects.

```
39.     WINDOWSURF.blit(catImg, (catx, caty))
```


Line 39 of the animation program calls the `blit()` method on the Surface object stored in `WINDOWSURF`. There are two parameters for `blit()`. The first is the source Surface object, which is what will be copied onto the `WINDOWSURF` Surface object. The second parameter is a two-integer tuple for the x and y values of the topleft corner where the image should be blitted to.

If `catx` and `caty` were set to 100 and 200 and the width of `catImg` was 25 and the height was 30, this `blit()` call would copy this image onto `WINDOWSURF` so that the topleft corner of the `catImg` was at the XY coordinate (100, 200) and the bottomright XY coordinate was (125, 230).

The rest of the game loop is just changing the `catx`, `caty`, and `direction` variables so that the cat moves around the window. There is also a call to `pygame.event.get()` to handle the `QUIT` event.

## Fonts

If you want to draw text to the screen, you *could* write several calls to `pygame.draw.line()` to draw out lines of each letter. This would be a headache to type out all those `pygame.draw.line()` calls and figure out all the XY coordinates, and probably wouldn't look very good.



Hand-drawn text "Hello Ugly World" in a jagged, pixelated font.

The above message would take forty one calls to the `pygame.draw.line()` function to make. Instead, Pygame provides some much simpler functions for fonts and creating text. Here is a small Hello World program using Pygame's font functions. Type it into IDLE's file editor and save it as `fonttext.py`:

```
1. import pygame, sys
2. from pygame.locals import *
3.
4. pygame.init()
5. WINDOWSURF = pygame.display.set_mode((400, 300))
```

```

6. pygame.display.set_caption('Hello World!')
7.
8. WHITE = (255, 255, 255)
9. GREEN = (0, 255, 0)
10. BLUE = (0, 0, 128)
11.
12. fontObj = pygame.font.Font('freesansbold.ttf', 32)
13. textSurfaceObj = fontObj.render('Hello world!', True, GREEN, BLUE)
14. textRectObj = textSurfaceObj.get_rect()
15. textRectObj.center = (200, 150)
16.
17. while True: # main game loop
18.     WINDOWSURF.fill(WHITE)
19.     WINDOWSURF.blit(textSurfaceObj, textRectObj)
20.     for event in pygame.event.get():
21.         if event.type == QUIT:
22.             pygame.quit()
23.             sys.exit()
24.     pygame.display.update()

```

There are six steps to making text appear on the screen:

1. Create a `pygame.font.Font` object. (Like on line 12)
2. Create a Surface object with the text drawn on it by calling the Font object's `render()` method. (Line 13)
3. Create a Rect object from the Surface object by calling the Surface object's `get_rect()` method. (Line 14)
4. Set the position of the Rect object by changing one of its attributes. On line 15, we set the center of the Rect object to be at 200, 150.
5. Blit the Surface object with the text onto the Surface object returned by `pygame.display.set_mode()`. (Line 19)
6. Call `pygame.display.update()` to make the display Surface appear on the screen. (Line 24)

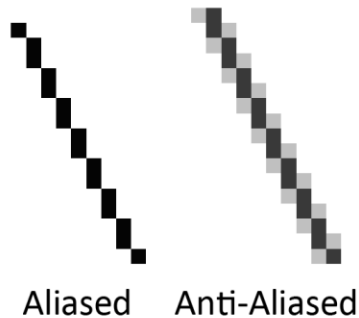
The parameters to the `pygame.font.Font()` constructor function is a string of the font file to use, and an integer of the size of the font (in points, like how word processors measure font size). On line 12, we pass `'freesansbold.ttf'` (this is a font that comes with Pygame) and the integer 32 (for a 32-point sized font.)

The parameters to the `render()` method call are a string of the text to render, a Boolean value to specify if we want anti-aliasing (explained later in this chapter), the color of the text, and the color of the background. If you want a transparent background, then simply leave off the background color parameter in the method call.

## Anti-Aliasing

**Anti-aliasing** is a graphics technique for making text and shapes look less blocky by adding a little bit of blur to their edges. It takes a little more computation time to do anti-aliasing, so although the graphics may look better, your program may run slower (but only just a little).

If you zoom in on an aliased line and an anti-aliased line, they look like this:



To make Pygame's text use anti-aliasing, just pass `True` for the second parameter of the `render()` method. There are also two other primitive drawing functions. The `pygame.draw.aaline()` and `pygame.draw.aalines()` functions have the same parameters as `pygame.draw.line()` and `pygame.draw.lines()`, except they will draw anti-aliased (smooth) lines instead of aliased (blocky) lines.

## Playing Sounds

Playing sounds that are stored in sound files is even simpler than displaying images from image files. First, you must create a `pygame.mixer.Sound` object (which we will call `Sound` objects for short) by calling the `pygame.mixer.Sound()` constructor function. It takes one string parameter, which is the filename of the sound file. Pygame can load WAV, MP3, or OGG files. (The difference between these audio file formats is explained at <http://invpy.com/formats>.) To play this sound, call the `Sound` object's `play()` method. If you want to stop the `Sound` object from playing immediately, just call the `stop()` method. The `stop()` method has no arguments. Here is some sample code:

```
soundObj = pygame.mixer.Sound('beeps.wav')
soundObj.play()
import time
time.sleep(1) # wait a second and let the sound play
soundObj.stop()
```

The `play()` method call will immediately return and keep playing the sound while the program execution continues on.

The `Sound` objects are good for sound effects to play when the player takes damage, slashes a sword, or collects a coin. But your games might also be better if they had background music playing. To load a background music file, call the `pygame.mixer.music.load()` function and pass it a string argument of the sound file to load. This file can be WAV, MP3, or MID format. To begin playing the

sound file as the background music, call the `pygame.mixer.music.play(-1, 0.0)` function. The `-1` argument makes the background music loop when it reaches the end of the sound file. (If you set it to an integer `0` or larger, then the music will only loop that number of times instead of looping forever.) The `0.0` means to start playing the sound file from the beginning. (If you set it to a larger integer, the music will begin playing that many seconds into the sound file. For example, if you pass `13.5` for the second parameter, the sound file will begin playing at the point `13.5` seconds from the beginning.)

To stop playing the background music immediately, call the `pygame.mixer.music.stop()` function. This function has no arguments.

Here is some example code of the sound methods and functions:

```
# Loading and playing a sound effect:
soundObj = pygame.mixer.Sound('beepingsound.wav')
soundObj.play()
soundObj

# Loading and playing background music:
pygame.mixer.music.load(backgroundmusic.mp3')
pygame.mixer.music.play(-1, 0.0)
...some more code...
pygame.mixer.music.stop()
```

## Summary

This covers the basics of making graphical games with the Pygame framework. Of course, just reading about these functions probably isn't enough to help you learn how to make games using these functions. The rest of the chapters in this book each focus on the source code for a small, complete game. This will give you an idea of what complete game programs "look like", so you can then apply those same ideas to your own game programs.

Unlike the "Invent Your Own Computer Games with Python" book, this book assumes that you know the basics of Python programming. If you have trouble remembering how variables, functions, loops, if-else statements, and conditions work, you can probably figure it out just by seeing what's in the code and how the program behaves. But if you are still stuck, you can read the "Invent with Python" book (it's for people who are completely new to programming) for free online at <http://inventwithpython.com>.