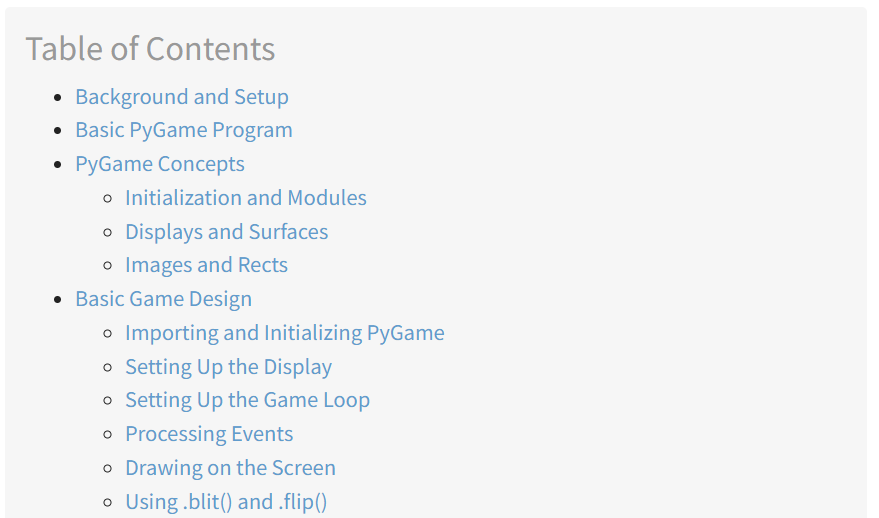
# Session 9

# Complete Game Development Using PyGame 1





## Background and Setup

## To install pygame on your platform, use the appropriate [pip](https://realpython.com/what-is-pip/) command:

## pip install pygame

## You can verify the install by loading one of the examples that comes with the library:

**python3 -m pygame.examples.aliens**

**If a game window appears, then pygame is installed properly!**

## Basic PyGame Program



Before getting down to specifics, let’s take a look at a basic pygame program. This program creates a window, fills the background with white, and draws a blue circle in the middle of it:

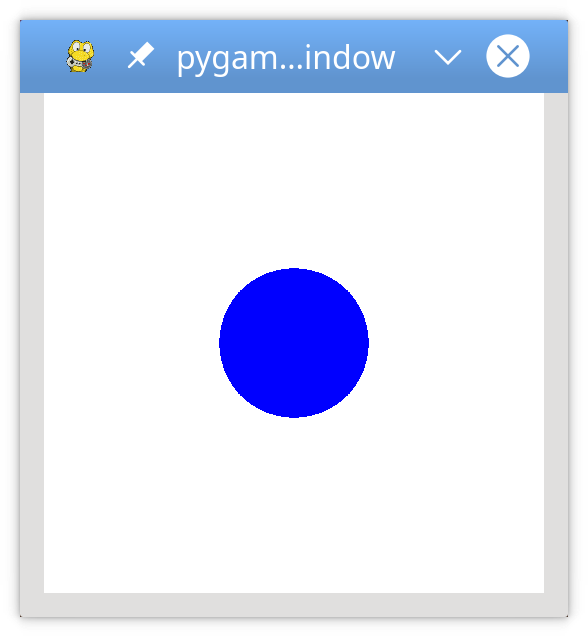
1. # Simple pygame program
2. # Import and initialize the pygame library
3. import pygame
4. pygame.init()
5. # Set up the drawing window
6. screen = pygame.display.set\_mode([500, 500])
7. # Run until the user asks to quit
8. running = True
9. while running:
10. # Did the user click the window close button?
11. for event in pygame.event.get():
12. if event.type == pygame.QUIT:
13. running = False
14. # Fill the background with white
15. screen.fill((255, 255, 255))
16. # Draw a solid blue circle in the center
17. pygame.draw.circle(screen, (0, 0, 255), (250, 250), 75)
18. # Flip the display
19. pygame.display.flip()
20. # Done! Time to quit.
21. pygame.quit()

When you run this program, you’ll see a window that looks like this:

Let’s break this code down, section by section:

* **Lines 4 and 5** import and initialize the pygame library. Without these lines, there is no pygame.
* **Line 8** sets up your program’s display window. You provide either a list or a tuple that specifies the width and height of the window to create. This program uses a list to create a square window with 500 pixels on each side.
* **Lines 11 and 12** set up a **game loop** to control when the program ends. You’ll cover game loops later on in this tutorial.
* **Lines 15 to 17** scan and handle **events** within the game loop. You’ll get to events a bit later as well. In this case, the only event handled is pygame.QUIT, which occurs when the user clicks the window close button.
* **Line 20** fills the window with a solid color. screen.fill() accepts either a list or tuple specifying the RGB values for the color. Since (255, 255, 255) was provided, the window is filled with white.
* **Line 23** draws a circle in the window, using the following parameters:
  + **screen:** the window on which to draw
  + **(0, 0, 255):** a tuple containing RGB color values
  + **(250, 250):** a tuple specifying the center coordinates of the circle
  + **75:** the radius of the circle to draw in pixels
* **Line 26** updates the contents of the display to the screen. Without this call, nothing appears in the window!
* **Line 29** exits pygame. This only happens once the loop finishes.

That’s the pygame version of “Hello, World.” Now let’s dig a little deeper into the concepts behind this code.



## PyGame Concepts

As pygame and the SDL library are portable across different platforms and devices, they both need to define and work with abstractions for various hardware realities. Understanding those concepts and abstractions will help you design and develop your games.

### Initialization and Modules

The pygame library is [composed of several Python constructs](https://www.pygame.org/docs/ref/pygame.html), which include several different **modules**. These modules provide abstract access to specific hardware on your system, as well as uniform methods to work with that hardware. For example, a display allows uniform access to your video display, while a joystick allows abstract control of your joystick.

After importing the pygame library in the example above, the first thing you did was [initialize PyGame](https://www.pygame.org/docs/tut/ImportInit.html) using pygame. init(). This function [calls the separate init() functions](https://www.pygame.org/docs/ref/pygame.html#pygame.init) of all the included pygame modules. Since these modules are abstractions for specific hardware, this initialization step is required so that you can work with the same code on Linux, Windows, and Mac.

Basic Game Design

Displays and Surfaces

In addition to the modules, pygame also includes several Python **classes**, which encapsulate non-hardware dependent concepts. One of these is the Surface which, at its most basic, defines a rectangular area on which you can draw. Surface objects are used in many contexts in pygame. Later you’ll see how to load an image into a Surface and display it on the screen.

In pygame, everything is viewed on a single user-created display, which can be a window or a full screen. The display is created using .set\_mode(), which returns a Surface representing the visible part of the window. It is this Surface that you pass into drawing functions like pygame.draw.circle(), and the contents of that Surface are pushed to the display when you call pygame.display.flip().

Images and Rects

Your basic pygame program draws a shape directly onto the display’s Surface, but you can also work with images on the disk. The image module allows you to load and save images in a variety of popular formats. Images are loaded into Surface objects, which can then be manipulated and displayed in numerous ways.

As mentioned above, Surface objects are represented by rectangles, as are many other objects in pygame, such as images and windows. Rectangles are so heavily used that there is a special Rect class just to handle them. You’ll be using Rect objects and images in your game to draw players and enemies, and to manage collisions between them.

## Basic Game Design

Before you start writing any code, it’s always a good idea to have some design in place. Since this is a session game, let’s design some basic gameplay for it as well:

* The goal of the game is to avoid incoming obstacles:
  + The player starts on the left side of the screen.
  + The obstacles enter randomly from the right and move left in a straight line.
* The player can move left, right, up, or down to avoid the obstacles.
* The player cannot move off the screen.
* The game ends either when the player is hit by an obstacle or when the user closes the window.

Here are some things that won’t be covered in this tutorial:

* No multiple lives
* No scorekeeping
* No player attack capabilities
* No advancing levels
* No boss characters

You’re free to try your hand at adding these and other features to your own program.

### Importing and Initializing PyGame

After you import pygame, you’ll also need to initialize it. This allows pygame to connect its abstractions to your specific hardware:

1# Import the pygame module

2import pygame

3

4# Import pygame.locals for easier access to key coordinates

5# Updated to conform to flake8 and black standards

6from pygame.locals import (

7 K\_UP,

8 K\_DOWN,

9 K\_LEFT,

10 K\_RIGHT,

11 K\_ESCAPE,

12 KEYDOWN,

13 QUIT,

14)

15

16# Initialize pygame

17pygame.init()

### The Pygame library defines many things besides modules and classes. It also defines some local constants for things like keystrokes, mouse movements, and display attributes. You reference these constants using the syntax pygame.<CONSTANT>. By importing specific constants from pygame.locals, you can use the syntax <CONSTANT> instead. This will save you some keystrokes and improve overall readability.

### Setting Up the Display

Create a [screen](https://www.pygame.org/docs/ref/display.html) to be the overall canvas.

1# Import the pygame module

2import pygame

3

4# Import pygame.locals for easier access to key coordinates

5# Updated to conform to flake8 and black standards

6from pygame.locals import (

7 K\_UP,

8 K\_DOWN,

9 K\_LEFT,

10 K\_RIGHT,

11 K\_ESCAPE,

12 KEYDOWN,

13 QUIT,

14)

15

16# Initialize pygame

17pygame.init()

18

19# Define constants for the screen width and height

20SCREEN\_WIDTH = 800

21SCREEN\_HEIGHT = 600

22

23# Create the screen object

24# The size is determined by the constant SCREEN\_WIDTH and SCREEN\_HEIGHT

25screen = pygame.display.set\_mode((SCREEN\_WIDTH, SCREEN\_HEIGHT))

You create the screen to use by calling pygame.display.set\_mode() and passing a tuple or list with the desired width and height. In this case, the window is 800x600, as defined by the constants SCREEN\_WIDTH and SCREEN\_HEIGHT on lines 20 and 21. This returns a Surface which represents the inside dimensions of the window. This is the portion of the window you can control, while the OS controls the window borders and title bar.

If you run this program now, then you’ll see a window pop up briefly and then immediately disappear as the program exits. Don’t blink or you might miss it! In the next section, you’ll focus on the main game loop to ensure that your program exits only when given the correct input.

### Setting Up the Game Loop

Every game from Pong to Fortnite uses a [game loop](http://www.informit.com/articles/article.aspx?p=2167437&seqNum=2) to control gameplay. The game loop does four very important things:

1. Processes user input
2. Updates the state of all game objects
3. Updates the display and audio output
4. Maintains the speed of the game

Every cycle of the game loop is called a **frame**, and the quicker you can do things each cycle, the faster your game will run. Frames continue to occur until some condition to exit the game is met. In your design, two conditions can end the game loop:

1. The player collides with an obstacle. (You’ll cover collision detection later.)
2. The player closes the window.

The first thing the game loop does is process user input to allow the player to move around the screen. Therefore, you need some way to capture and process a variety of input. You do this using the pygame event system.

### Processing Events

Key presses, mouse movements, and even joystick movements are some of the ways in which a user can provide input. All user input results in an event being generated. Events can happen at any time and often (but not always) originate outside the program. All events in pygame are placed in the event queue, which can then be accessed and manipulated. Dealing with events is referred to as handling them, and the code to do so is called an event handler.

Every event in pygame has an event type associated with it. For your game, the event types you’ll focus on are keypresses and window closure. Keypress events have the event type KEYDOWN, and the window closure event has the type QUIT. Different event types may also have other data associated with them. For example, the KEYDOWN event type also has a variable called key to indicate which key was pressed.

You access the list of all active events in the queue by calling pygame.event.get(). You then loop through this list, inspect each event type, and respond accordingly:

27# Variable to keep the main loop running

28running = True

29

30# Main loop

31while running:

32 # Look at every event in the queue

33 for event in pygame.event.get():

34 # Did the user hit a key?

35 if event.type == KEYDOWN:

36 # Was it the Escape key? If so, stop the loop.

37 if event.key == K\_ESCAPE:

38 running = False

39

40 # Did the user click the window close button? If so, stop the loop.

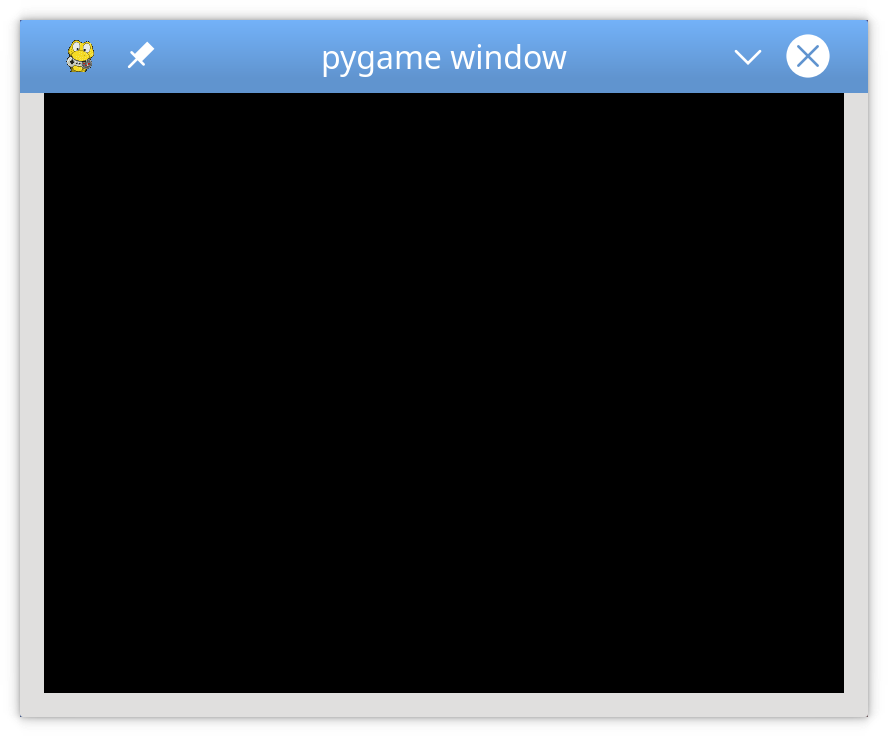
41 elif event.type == QUIT:

42 running = False

Let’s take a closer look at this game loop:

* **Line 28** sets up a control variable for the game loop. To exit the loop and the game, you set running = False. The game loop starts on line 29.
* **Line 31** starts the event handler, walking through every event currently in the event queue. If there are no events, then the list is empty, and the handler won’t do anything.
* **Lines 35 to 38** check the current event. type is a KEYDOWN event. If it is, then the program checks which key was pressed by looking at the event.key attribute. If the key is the Esc key, indicated by K\_ESCAPE, then it exits the game loop by setting running = False.
* **Lines 41 and 42** do a similar check for the event type called QUIT. This event only occurs when the user clicks the window close button. The user may also use any other operating system action to close the window.

When you add these lines to the previous code and run it, you’ll see a window with a blank or black screen:



The window won’t disappear until you press the Esc key, or otherwise trigger a QUIT event by closing the window.

### Drawing on the Screen

In the sample program, you drew on the screen using two commands:

1. **screen.fill()** to fill the background
2. **pygame.draw.circle()** to draw a circle

Now you’ll learn about a third way to draw to the screen: using a Surface.

Recall that a [Surface](https://www.pygame.org/docs/ref/surface.html) is a rectangular object on which you can draw, like a blank sheet of paper. The screen object is a Surface, and you can create your Surface objects separate from the display screen. Let’s see how that works:

44# Fill the screen with white

45screen.fill((255, 255, 255))

46

47# Create a surface and pass in a tuple containing its length and width

48surf = pygame.Surface((50, 50))

49

50# Give the surface a color to separate it from the background

51surf.fill((0, 0, 0))

52rect = surf.get\_rect()

After the screen is filled with white on line 45, a new Surface is created on line 48. This Surface is 50 pixels wide, 50 pixels tall, and assigned to surf. At this point, you treat it just like the screen. So on line, 51 you fill it with black. You can also access its underlying Rect using .get\_rect(). This is stored as rect for later use.

### Using .blit() and .flip()

Just creating a new Surface isn’t enough to see it on the screen. To do that, you need to [blit](https://www.pygame.org/docs/ref/surface.html#pygame.Surface.blit) the Surface onto another Surface. The term blit stands for **Block Transfer** and .blit() is how you copy the contents of one Surface to another. You can only .blit() from one Surface to another, but since the screen is just another Surface, that’s not a problem. Here’s how you draw surf on the screen:

54# This line says "Draw surf onto the screen at the center"

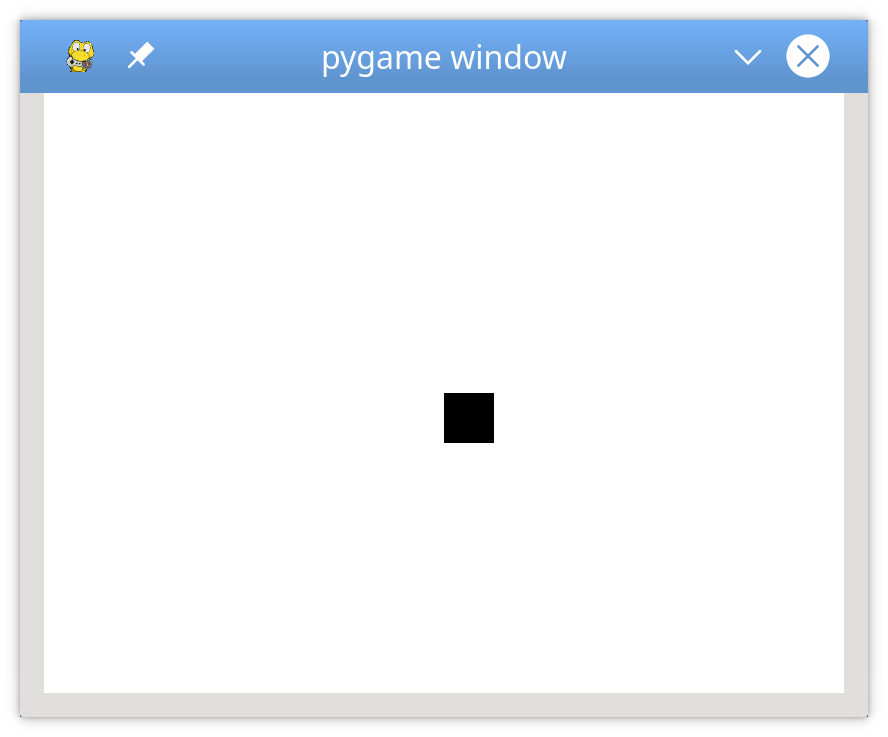
55screen.blit(surf, (SCREEN\_WIDTH/2, SCREEN\_HEIGHT/2))

56pygame.display.flip()

The .blit() call on line 55 takes two arguments:

1. The Surface to draw
2. The location at which to draw it on the source Surface

The coordinates (SCREEN\_WIDTH/2, SCREEN\_HEIGHT/2) tell your program to place surf in the exact center of the screen, but it doesn’t quite look that way:



The reason why the image looks off-center is that .blit() puts the **top-left corner** of the surf at the location given. If you want the surf to be centered, then you’ll have to do some math to shift it up and to the left. You can do this by subtracting the width and height of the surf from the width and height of the screen, dividing each by 2 to locate the center, and then passing those numbers as arguments to the screen.blit():

54# Put the center of surf at the center of the display

55surf\_center = (

56 (SCREEN\_WIDTH-surf.get\_width())/2,

57 (SCREEN\_HEIGHT-surf.get\_height())/2

58)

59

60# Draw surf at the new coordinates

61screen.blit(surf, surf\_center)

62pygame.display.flip()

Notice the call to [pygame.display.flip()](https://www.pygame.org/docs/ref/display.html#pygame.display.flip) after the call to blit(). This updates the entire screen with everything that’s been drawn since the last flip. Without the call to .flip(), nothing is shown.

SUMMARY:

This Game Development exercise helps the programmer to perform the below Game Development using PyGame

* Draw items on your screen
* Play sound effects and music
* Handle user input
* Implement event loops
* Describe how game programming differs from standard procedural Python programming