# A Brief Introduction to Pygame

Dana Hughes

October 3, 2013

## 1 Installing Pygame

Pygame for Python3 is still in the experimental phase, but the latest version (1.9.1) should still work fine for simple games. Ubuntu (the OS running in the CU Virtual Machine) contains a pygame package which you may be able to install. If this doesn't work, it is possible to install for Python 3 from source.

### 1.1 Installing Pygame Package

The easiest way to use Pygame is to install it using the Ubuntu package manager, and use Python 2.7.3 instead of Python 3.2.3.

To install pygame using the Ubuntu package manager, open a terminal and enter the following command

```
$ sudo apt-get install python-pygame
```

This may ask you for your password, which will be *user* if you are using the CU Virtual Machine *and* you haven't changed the passowrd. Once installed, check that you can import the package in python

```

$ python
Python 2.7.3 (default, Oct 19 2012, 19:53:16)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pygame
>>>
```

If you get the following error,

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named pygame
```

then pygame isn't installed. Notice that we're using Python 2.7.3, *not* Python 3.2.3. This is easily accounted for by putting the following import statement at the top of your programs

```
from __future__ import division, print_function, absolute_import,
                        unicode_literals
```

The details of this line are unimportant. Suffice to say that it makes Python 2.7.3 behave like Python 3.2.3.

## 1.2 Installing From Source

If you insist on using Pygame with Python 3.2.3, you'll need to install from source. This isn't too hard, but is somewhat involved and takes more time than the above steps.

First, you'll need to create a temporary directory to put the installation files.

```
$ mkdir pygame_install
$ cd pygame_install
```

Second, you'll need subversion in order to get the latest version of Pygame. I've heard downloading the compressed source files from the pygame website doesn't work for Python 3, and I've tested this on the VM.

```
$ sudo apt-get install subversion
```

Now, you can download the Pygame source files with the following command

```
$ svn co svn://seul.org/svn/pygame/trunk pygame
```

Once that finishes downloading, cd to the newly created pygame folder

```
$ cd pygame
```

Before you run the configuration and setup scripts, you'll need to install a lot of dependencies with the following command

```
$ sudo apt-get install python3-dev libjpeg-dev libpng12-dev
                       libportmidi-dev libsdl-image1.2-dev libsdl-mixer1.2-dev
                       libsdl-ttf2.0-dev libsdl1.2-dev libsmpeg-dev libx11-dev
                       ttf-freefont libavformat-dev libswscale-dev
```

Once all those packages have installed (and if you still have the resolve after running that command), you can configure the setup using python3

```
$ python3 config.py
```

Notice that the command is **python3**, *not* **python**. This makes sure you're building this thing for Python 3.2.3.

Once the configuration is done, build the Pygame package with

```
$ python3 setup.py build
```

And the, install using

```
$ sudo python3 setup.py install
```

Assuming everything went according to plan, you should now have Pygame installed for Python 3.2.3. Test this out by trying to import in python3.

```
$ python3
>>> import pygame
>>>
```

If you don't get the error message discussed above, then you've successfully installed Pygame for Python 3.2.3. Congratulations!

## 2 Pygame Overview

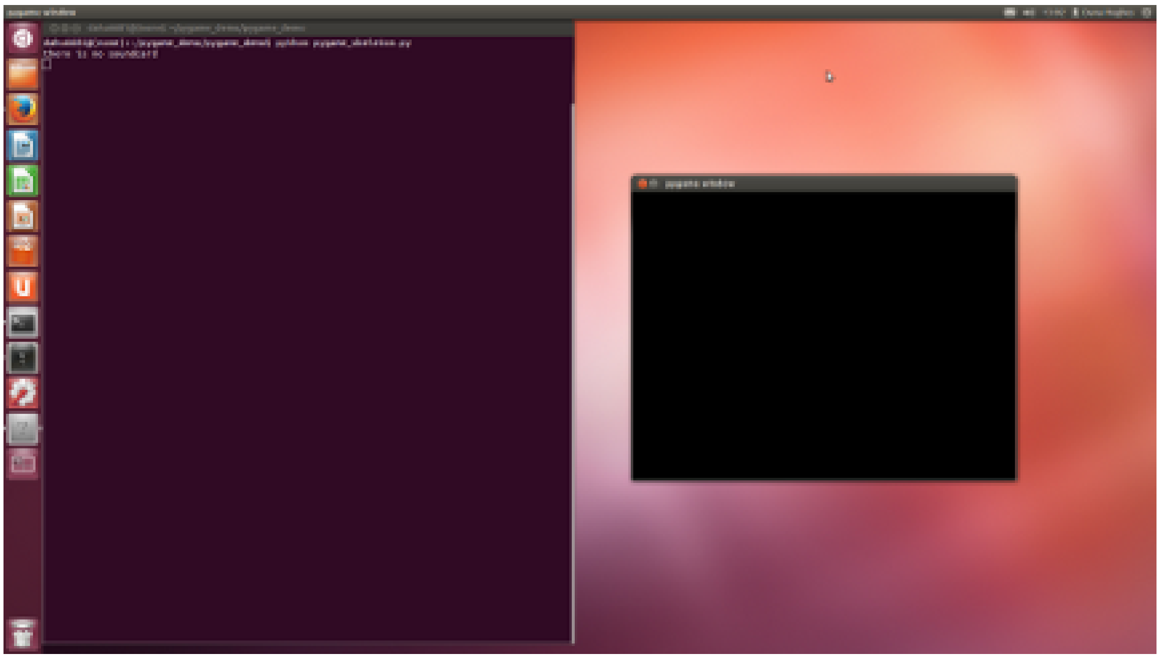
Pygame provides functionality for creating 2-D games. This functionality includes the ability to create a window for your game, to display images, play sounds and collect input from the keyboard or mouse. For those technically inclined, this is done using the SDL library (a fast way of directly accessing your computer's hardware), but this is an unimportant point.

A basic game made using Pygame will create a display and perform some initialization, and then enter into a game loop (a while loop with some condition for exiting the program). During each iteration of the loop, three major tasks typically occur:

1. The program checks for any user input – keystrokes, mouse clicks, etc.
2. The program updates variables for your game (the game *model*).
3. The program redraws the display, based on the current values of the game variables (the game *state*).

Each of these major components will be discussed in detail. A basic skeleton of this program is given below.

```
"""
pygame_skeleton.py

Dana Hughes
02-Oct-2013

A skeleton for a simply Pygame program
"""

# Import this to have Python 2.7 behave like Python 3.2
from __future__ import division, absolute_import, print_function,
                             unicode_literals

# Common modules to import
import pygame
from pygame.locals import *

# Some constants for use later
GAME_TITLE = "My Game Title" # What's the game called?
DISPLAY_SIZE = (640,480)      # Size of the screen for the game (in pixels)
DESIRED_FPS = 30              # Want the game to run at 30 FPS

# Initialize pygame
pygame.init()

# Create a display for the game (this creates a window)
screen = pygame.display.set_mode(DISPLAY_SIZE)
pygame.display.set_caption(GAME_TITLE)

# Create a clock to try to keep the game running at however many FPS
fps_clock = pygame.time.Clock()

# Initial game variables
game_running = True           # Is the player still playing?


##########################
### The main game loop ###
##########################
```



```
[illegible] python pygame_skeleton.py
[illegible]
```

Figure 1: The pygame_skeleton program showing a blank game window

```
while game_running:           # Keep going until the player quits

    ###########################
    ### Capture user input ###
    ###########################

    user_input = pygame.event.get()


    ##############################
    ### Update the game model ###
    ##############################


    ######################################
    ### Redraw the game on the screen ###
    ######################################

    pygame.display.flip()


    # Wait until this frame is finished (as per the FPS clock)
    fps_clock.tick(DESIRED_FPS)

# All done!  Cleanup time
pygame.quit()
```

Running this from the terminal creates a blank window, shown in 1. To close this window, you'll need to type ¡Ctrl¿-C in the terminal.

Looking at some specific lines gives a bit more insight into Pygame. First, we need to import some important pygame modules

```
# Common modules to import
import pygame
from pygame.locals import *
```

and initialize pygame

```
# Initialize pygame
pygame.init()
```

Once this is done, we can create a new screen for our game to reside in

```
# Create a display for the game (this creates a window)
screen = pygame.display.set_mode(DISPLAY_SIZE)
pygame.display.set_caption(GAME_TITLE)
```

This statement creates a new pygame display (a window) with size **DISPLAY_SIZE**, which we defined earlier to be $640x480$ (line 15). Finally, we begin a while loop which will terminate when **game_running** is set to False.

```
##########################
### The main game loop ###
##########################

while game_running:            # Keep going until the player quits

    ##########################
    ### Capture user input ###
    ##########################

    user_input = pygame.event.get()


    #############################
    ### Update the game model ###
    #############################


    #####################################
    ### Redraw the game on the screen ###
    #####################################

    pygame.display.flip()


    # Wait until this frame is finished (as per the FPS clock)
    fps_clock.tick(DESIRED_FPS)
```

## 2.1 Adding a title to the display

When this program is run, the game screen just says **pygame window**. This is relatively non-descript, and we can change the title using the following command

```
pygame.display.set_caption("My Awesome Game")
```

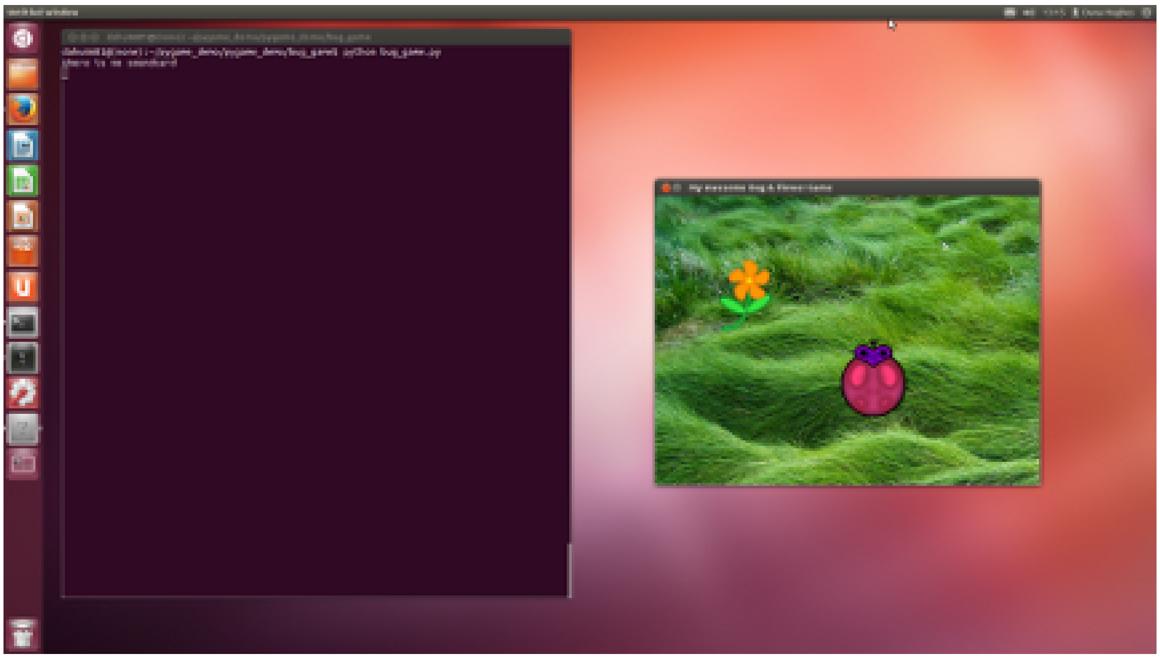


Figure 2: The finished bug game

## 3 Building a game

With the skeleton set up, let's create a simple game. For this "game", we'll have a little bug flying around, controlled by the arrow keys. The bug's goal is to land on a flower, which moves to a random spot when the bug gets close enough to touch it. When the player presses the Escape key, the game quits. Finally, we can change the color of the bug when it is clicked. The finished game is shown in 2

### 3.1 Initialization

For this game, we'll need a few variables. First, we'll need the position (x and y) of the bug, and the position of the flower (x and y). Also, we can maintain a score (1 point for each time the bug touches a flower. Set up these variables in the *Initial game variables* part of the program.

```
bug_x = 300            # Start position of the bug (center of the screen)
bug_y = 240

flower_x = 100         # Start position of the flower (100,100)
flower_y = 100

score = 0              # How many points so far?

bug_color = "red"      # What color is the bug right now?
```

In addition, we'll need some images, one for the background, one for the flower, and one for each color bug we want. To load the images, we use the pygame function **pygame.image.load**.

```

# Load the images
background = pygame.image.load("images/background.png")

red_bug = pygame.image.load("images/red_bug.png")
blue_bug = pygame.image.load("images/blue_bug.png")
```

```
flower = pygame.image.load("images/flower.png")
```

In pygame terminology, the variables **background**, **red_bug**, **blue_bug** and **flower** refer to what are known as *Surfaces*.

Finally, we'll set a variable **bug** to **red_bug** to indicate that the bug is currently the red one. Why we're doing this will be made clear later.

```
bug = red_bug
```

## 3.2 Drawing the Display

When the program is run, the display still only shows a blank screen. We need to draw the images we have loaded to the display. This is the last step of the game loop, but the first we'll tackle (for simplicity later). Drawing an image onto the display is known as *blitting*.

We need to be very careful in what order we draw the images. If we draw the background *after* we draw the flower and bug, then the background will draw over the bug and flower, and we wont see them.

To draw something on the display, we use the command **screen.blit(image_name, (x,y) )**. Remember the variables we set up for the bug and flower position? This is where we use them. The position (x,y) refers to the upper-left corner of the images we loaded, not the center. We can load the background, flower and red bug using the following commands in the *Redraw the game on the screen* section of the game loop

```
# Redraw the game on the screen
screen.blit(background, (0,0))              # Draw the background first
screen.blit(flower, (flower_x, flower_y))   # Then the flower
screen.blit(bug, (bug_x, bug_y))        # Then the bug
```

Running this code will still give only a black screen. The last command that needs to be issued is **pygame.display.flip()**. The reasons for this is technical, but think of it as "I'm all done drawing, so go ahead and show it to the player".

```
# Redraw the game on the screen
...                                 # Lots of drawing here...
pygame.display.flip()               # Show the new screen
```

Now run the program. Yay! We have a bug and flower sitting in a field! You should see the image shown in 2.

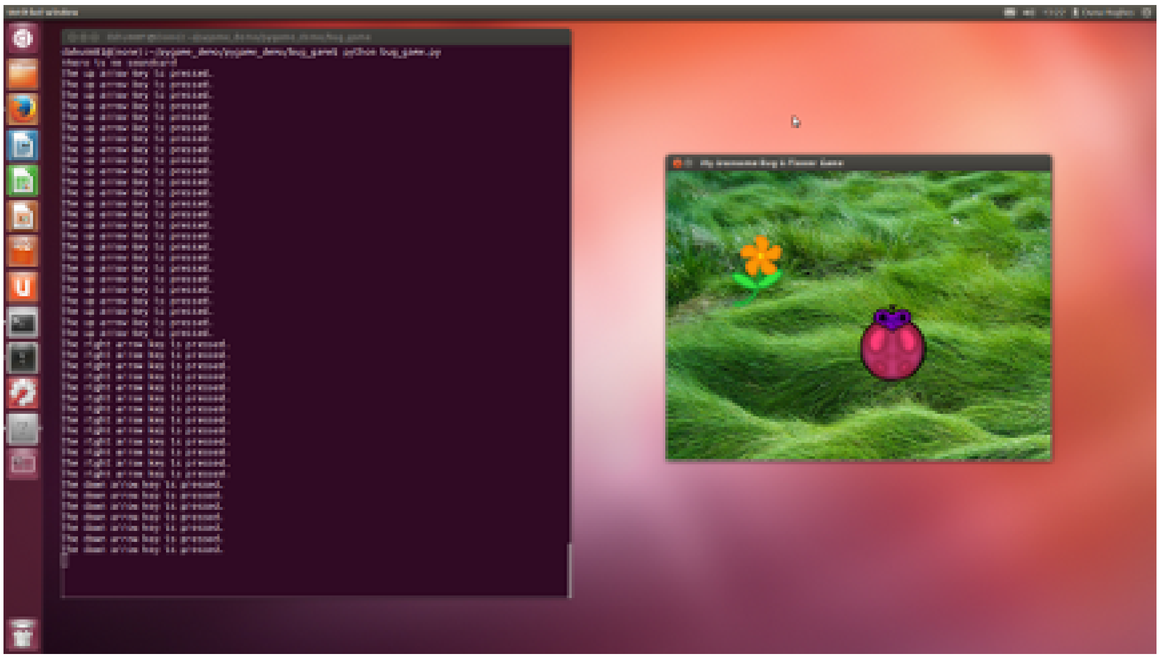
## 3.3 Collecting User Input

So far, we've drawn some stuff on the screen, but it's still not much of a game. To make this a game, we'll need to be able to collect user input. This can take the form of keyboard presses or mouse movement and mouse clicks.

Before we can check to see what the user has done, we must first ask pygame what has happened. Pygame stores user input in what is known as an *event queue*. The programmer must explicity ask pygame for this using the statement

```
# Capture user input
user_input = pygame.event.get()
```

Everything that has happened since last time we asked pygame is written into the variable **user_input**, which simply a list of events. For now, we won't use this list, but it's necessary to perform this step to ask about which keys are down, and which mouse buttons are clicked.



```
[illegible] python bug_game.py
There is no soundcard
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The up arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The right arrow key is pressed.
The down arrow key is pressed.
The down arrow key is pressed.
The down arrow key is pressed.
The down arrow key is pressed.
The down arrow key is pressed.
The down arrow key is pressed.
The down arrow key is pressed.
```

Figure 3: The bug game showing keyboard input in the terminal.

### 3.3.1 Key Presses

To see which keys are pressed, we can ask pygame using the statement

```
pressed_keys = pygame.key.get_pressed()
```

Let's say we want to see if the 'q' key is pressed. To do this, we simply check with the following *if* statement

```
if pressed_keys[K_q]:
    print("The 'q' key is pressed.")
```

Notice that we're checking if **K_q** is being pressed, not if the character **'q'** has been typed. This is because not all keys have convenient characters in python (e.g., Shift or Control), and to distinguish between lowercase and capital letters. To check for the arrow keys, we just need to replace **K_q** with **K_UP**, **K_DOWN**, **K_LEFT** and **K_RIGHT**. The terminal in 3 shows the output after pressing a few keys.

### 3.3.2 Mouse Clicks and Position

To find out where the mouse is (relative to the upper left corner of the screen), use the command

```
mouse_position = pygame.mouse.get_pos()
```

This will give back a position in the form $(x, y)$. To get the x and y positions, simply check the values at index 0 and 1, for example

```
print("The mouse is at position", mouse_position)
print("The x position is", mouse_position[0])
print("The y position is", mouse_position[1])
```

4 shows the game with the mouse position being printed to the terminal as the game runs.

To check if a mouse button is clicked, use the command

```
mouse_button = pygame.mouse.get_pressed()
```

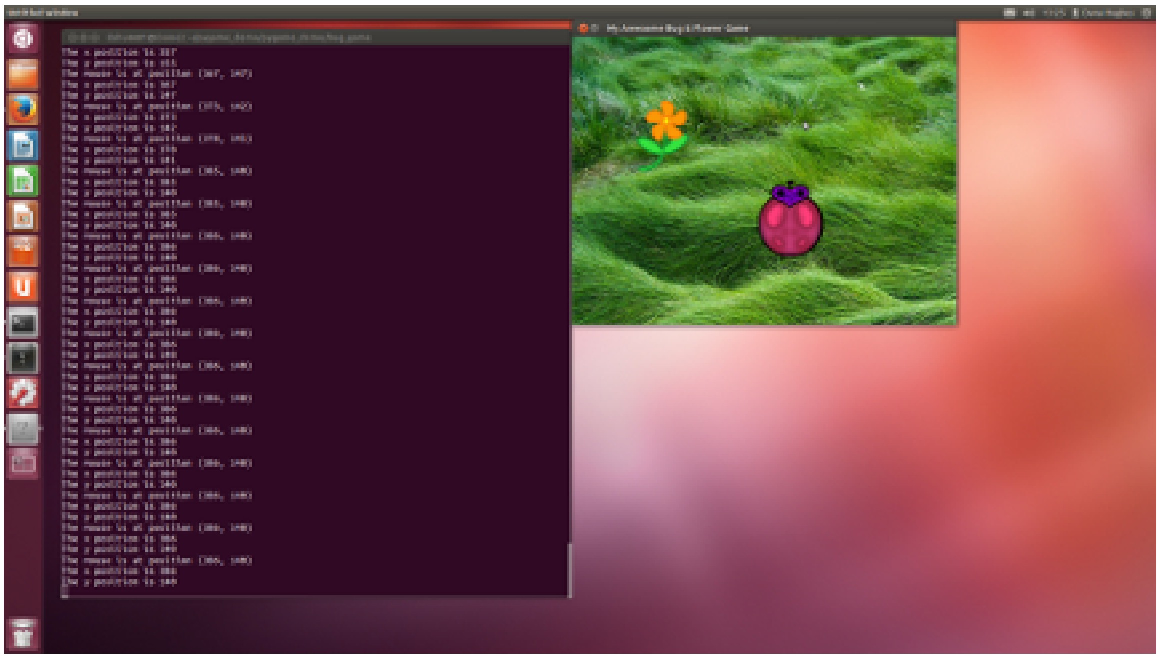


Figure 4: The bug game showing the mouse position in the terminal.

This gives back three values in the form $(b1, b2, b3)$, where b1, b2 and b3 are whether or not the left, center or right buttons are currently down. These take on the value 0 if the button isn't pressed, and 1 if it is. So, to check the status of the mouse buttons, we can print out the following

```
print("The mouse buttons are", mouse_buttons)
if mouse_buttons[0]:
    print("The left button is pressed.")
if mouse_buttons[1]:
    print("The center button is pressed.")
if mouse_buttons[2]:
    print("The right button is pressed.")
```

5 shows the game with the mouse button states being printed to the terminal as the game runs.

## 3.4 Updating the Game Model

### 3.4.1 Moving the Bug

Now we have the user input, let's use this to update our game model. First, we want to see if the user wants to quit. If so, we'll set **game_running** to False, which let's the game exit the while loop. The user presses 'q' to quit, which we've seen how to check for.

```
if pressed_keys[K_q]:
    game_running = False
```

Now run the game and press 'q'. Great! No more Control-C!

Moving the bug should be done in a similar manner. If the user presses up, we want the bug to move up, etc. Remember that we have two variables for where the bug is, **bug_x** and **bug_y**. If we want the bug to move up, we need to decrease **bug_y**. If we want the bug to move down, we increase **bug_y**. To move left, decrease **bug_x**, and to move right, increase **bug_x**. For now, let's just add or subtract 1 to each of these values, based on what key is pressed

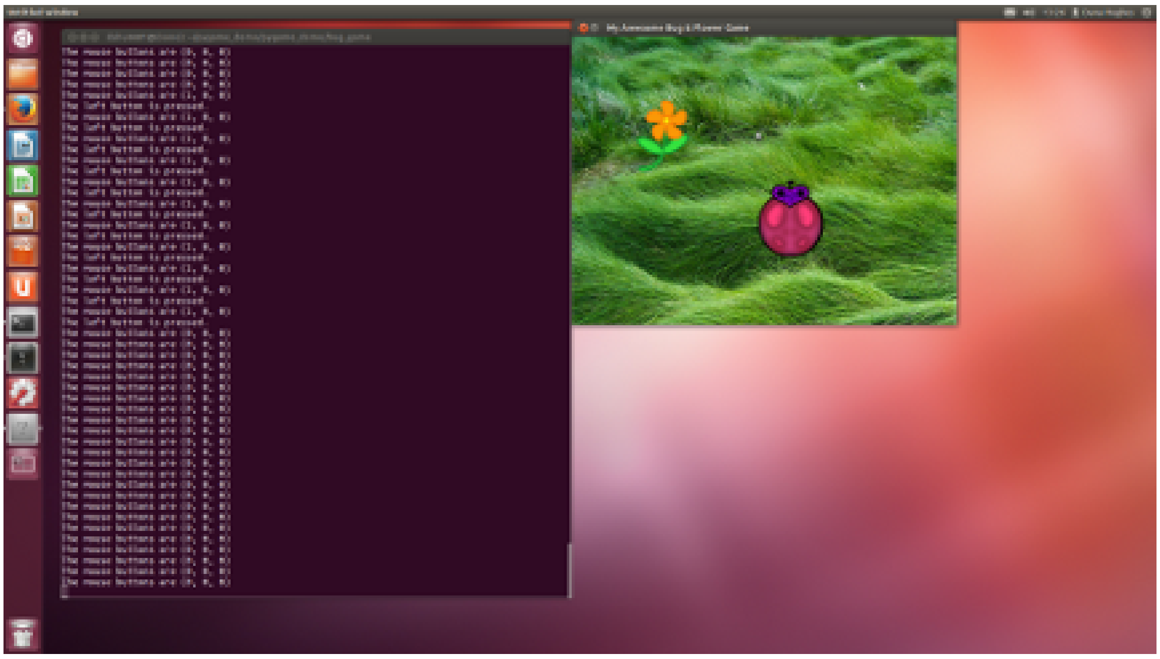


Figure 5: The bug game showing the mouse button states in the terminal.

```
if pressed_keys[K_UP]:
    bug_y = bug_y - 1
if pressed_keys[K_DOWN]:
    bug_y = bug_y + 1
if pressed_keys[K_LEFT]:
    bug_x = bug_x - 1
if pressed_keys[K_RIGHT]:
    bug_x = bug_x + 1
```

Now run the program. You should be able to move the bug, and quit the game. This is all the keyboard tasks we need to handle! If you want your bug to move faster, simply add or subtract a bigger number.

### 3.4.2 Touching the flower

Now we need to detect if the bug is touching the flower. To do this, we could either use a bit of math with the flower and bug position variables, or we can ask pygame for the *bounding boxes* of the bug and flower, which is just an imaginary rectangle surrounding each. To get this, we create two **Rect** values by using the command **Rect((x, y), image.get_size())**. So, for our bug and flower, we can make these with

```
bug_box = Rect( (bug_x, bug_y), bug.get_size())
flower_box = Rect( (flower_x, flower_y) flower.get_size())
```

Now, to see if they overlap, we simply need to use the **bug_box.colliderect(flower_box)** command. This will return **True** if the two rectangles overlap, and **False** otherwise. We could exhange **bug_box** and **flower_box** and get the same effect, or check if **bug_box** collides with other bounding boxes.

So, let's see if the bug has touched the flower. If it has, move the flower to a random spot, and increase the score by 1. Note that we'll have to import the **random** module for this.

```
if bug_box.colliderect(flower_box):
    score = score + 1
    print ("Current Score is", score)
    flower_x = random.randint(0, DISPLAY_SIZE[0] - flower.get_width())
    flower_y = random.randint(0, DISPLAY_SIZE[1] - flower.get_height())
```

Notice where the flower is moved to. We want it's x position to be somewhere between 0 and the right edge of the display (the DISPLAY_SIZE width), but this position is a reference to the upper-left corner of the flower, so to keep it from being moved off the screen, we subtract the width of the flower image from the highest possible position. This makes sure that the furthest right the flower can be placed puts the right edge of the flower on the right edge of the display. The y component is similar.

Now run the program, you should have something resembling a game at this point.

### 3.4.3 Changing the Bug's Color

Another fun thing we want to do is to be able to change the bug's color by clicking on it. This isn't too hard to do, since we already have the mouse's position (**mouse_position**), whether or not the left button is clicked (**mouse_buttons**

0

), and a bounding box for the bug (**bug_box**). Pygame has a command for checking if a point is within a box, called **collidepoint**, which is similar to **colliderect**. So, to change the color of the bug, first we check if the left mouse button is down. If it is, check if the mouse's position is within the **bug_box**. If it is, then change the bug's color.

```
if mouse_buttons[0]:                    # Is the left mouse button pressed?
    if bug_box.collidepoint(mouse_position):        # Mouse is on the bug?
        # Swap to the other color (either red or blue)
        if bug_color == "red":
            bug = blue_bug
            bug_color = "blue"
        else:
            bug = red_bug
            bug_color = "red"
```

There are better ways to do this in Pygame, but this is a sufficient for our example.

## 3.5 Maintaining Timing

One thing that hasn't been mention yet is how to ensure that the game runs consistently among all computers. Some computers are faster than others, so if we run this on a very fast computer, it may be running too fast for the user to play. To set a limit on how fast the game runs, we need to set up a clock which ensures that the program runs at a certain framerate (typically 30 frames per second). First, create a few variables during the initialization phase

```
DESIRED_FPS = 30                # Want the game to run at 30 FPS

# Create a clock to try to keep the game running at however many FPS
fps_clock = pygame.time.Clock()
```

Now, at the end of the game loop, have the clock tick the appropriate number of times to achieve our desired framerate

```
fps_clock.tick(DESIRED_FPS)
```

The game should now run at a *maximum* of 30 frames per second. If the game is really complex, or running on a very slow computer, it may run at a slower rate than this.

## 3.6 Cleaning Things Up

When the player quits the game, there's a lot of stuff that needs to be cleaned up. This is similar to closing a file after reading or writing to it. To do this, we use the function **pygame.quit()**

```
pygame.quit()
```

At the end of your program (after you exit the main game loop). Then, you're all done!