Beginning Game Programming for Teens with Python

Julian Meyer on January 22, 2013

If you're new here, you may want to subscribe to my RSS feed or follow me on Twitter. Thanks for visiting!

This is a post by Tutorial Team Member Julian Meyer, a 13-year-old python developer. You can find him on Google+ and Twitter.

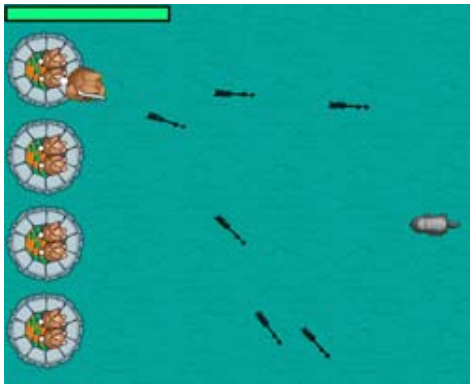
Have you ever wondered how video games are created? It's not as complicated as you might think!

In this tutorial, you'll create a simple game called Bunnies and Badgers, where the hero, the bunny, has to defend a castle against an attacking horde of badgers. :O

To write this game, you'll use Python. No, I'm not talking about a big snake! :]

Python is a computer programming language. We chose Python for this tutorial because it's a simple language to start out with, and is fun and easy to learn.

If you are new to Python, before you begin check out this book: Think Python: How to Think Like a Computer Scientist. That should get you up to speed.



Learn how to make a simple game with Python!

Then dive back here and get ready – there's a war coming on between the bunnies and the badgers. Keep reading to jump into the fray!

Getting Started: Installing Python

If you want to try this tutorial on a Windows PC, you need to install Python. Make sure you grab the 2.7.3 version and **NOT the 3.3.0 version**! After running the installer, you should have **IDLE** in your All Programs folder in your start menu. Launch IDLE to get started.

If you are on a Mac, you already have Python installed! Just open Terminal (/Applications/Utilities/Terminal.app), type in **python** and press Enter.

Note: If you install Python from python.org (and might have to if you want to get PyGame working), then you'll also have access to IDLE on the Mac as well. It should be in the "/Applications/Python 2.7" folder.

If you did it correctly, you should see something like this:

```
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012, 20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Note: If you want to exit the Python prompt (the triple angle brackets, >>>), you can either type **exit()** at the Python prompt and press Return or you can press Control+D.

Once you are at the Python prompt, to test if Python is correctly working, type in **print 1+1** and hit Enter/Return. It should print 2. You have just written your first Python program!

Now that you know Python is working correctly, you need to install PyGame in order to write a game using Python.

PyGame is a Python library that makes writing games a lot easier! It provides functionality such as image handling and sound playback that you can easily incorporate into your game.

Go here and download the PyGame installer appropriate for your system. Make sure you download a Python 2.7 version.

> **Note:** The PyGame installer from the link above will not work with the default Python from Apple that is installed on a Mac. You'll need to download Python from python.org and install it in order to use PyGame. Or, you can install both Python and PyGame via MacPorts.

To verify that you have PyGame installed properly, open IDLE or run Python via the Terminal and type in **import pygame** at the Python prompt. If this doesn't result in any output, then you're good.

If, on the other hand, it outputs an error like what's shown below, then PyGame is not installed.

```
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import pygame
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named pygame
>>>
```

If you get an error like this, post on the forums and I will help you get it working.

Running Python Code from File

While you can run short bits of Python code at the Python prompt, if you want to work on a bigger program (like a game), you probably want to save your code to a file so that you don't have to type it in over and over again.

There are several ways to run a Python program as a file. One way is to use a plain text editor like Notepad (Windows), or TextEdit (Mac). Open a new text file, type in your Python code (like **print 1+1**). Then save it as **XXX.py** (The XXX can be any descriptive file name).

Then on Windows, double-click this file to run it. On Mac, open Terminal and type **python**, then drag the file that you saved onto the Terminal window and press Enter.

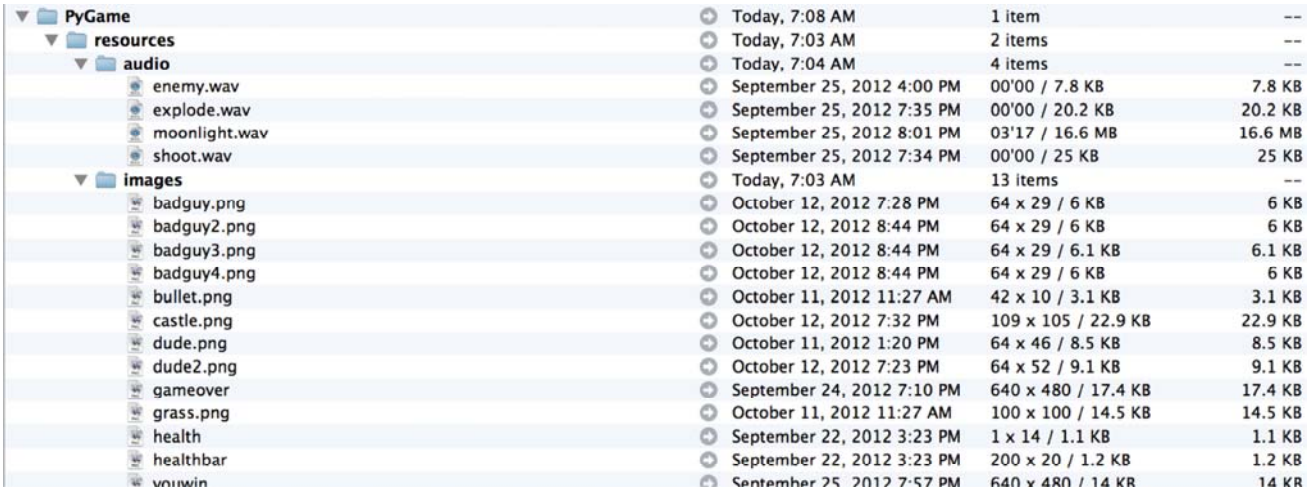
The other way is to type in your code using the IDLE editor, which is what you're going to do in this tutorial. To run idle, simply type **idle** from Terminal. Then choose File\New Window from the IDLE menu and you should have a text editor window where you can type in Python code. You can save your code changes via File\Save and even run the code via Run\Run Module (F5).

Do note that the Run menu is only available if you have a file open in an editor window.

Adding the Game Resources

You are almost ready to create your game. But what's a game without some great graphics and sound effects? I've put together all the graphics and sound effects you'll need for your game into a ZIP archive. You can download it here.

Once you've downloaded the file, create a folder for your game on your hard disk and extract the resources folder into that folder so that your game folder has a subfolder named **resources**, with the various resources grouped in additional folders inside it, like this:



PyGame	Today, 7:08 AM	1 item	--
resources	Today, 7:03 AM	2 items	--
audio	Today, 7:04 AM	4 items	--
enemy.wav	September 25, 2012 4:00 PM	00'00 / 7.8 KB	7.8 KB
explode.wav	September 25, 2012 7:35 PM	00'00 / 20.2 KB	20.2 KB
moonlight.wav	September 25, 2012 8:01 PM	03'17 / 16.6 MB	16.6 MB
shoot.wav	September 25, 2012 7:34 PM	00'00 / 25 KB	25 KB
images	Today, 7:03 AM	13 items	--
badguy.png	October 12, 2012 7:28 PM	64 x 29 / 6 KB	6 KB
badguy2.png	October 12, 2012 8:44 PM	64 x 29 / 6 KB	6 KB
badguy3.png	October 12, 2012 8:44 PM	64 x 29 / 6.1 KB	6.1 KB
badguy4.png	October 12, 2012 8:44 PM	64 x 29 / 6 KB	6 KB
bullet.png	October 11, 2012 11:27 AM	42 x 10 / 3.1 KB	3.1 KB
castle.png	October 12, 2012 7:32 PM	109 x 105 / 22.9 KB	22.9 KB
dude.png	October 11, 2012 1:20 PM	64 x 46 / 8.5 KB	8.5 KB
dude2.png	October 12, 2012 7:23 PM	64 x 52 / 9.1 KB	9.1 KB
gameover	September 24, 2012 7:10 PM	640 x 480 / 17.4 KB	17.4 KB
grass.png	October 11, 2012 11:27 AM	100 x 100 / 14.5 KB	14.5 KB
health	September 22, 2012 3:23 PM	1 x 14 / 1.1 KB	1.1 KB
healthbar	September 22, 2012 3:23 PM	200 x 20 / 1.2 KB	1.2 KB
youwin	September 25, 2012 7:57 PM	640 x 480 / 14 KB	14 KB

You are now ready to begin creating Bunnies and Badgers. :]

Step 1: Hello Bunny

Run IDLE and open a new text editor window, as mentioned in the previous section. Type the following code into the editor window:

```
# 1 - Import library
import pygame
from pygame.locals import *

# 2 - Initialize the game
pygame.init()
width, height = 640, 480
screen=pygame.display.set_mode((width, height))

# 3 - Load images
player = pygame.image.load("resources/images/dude.png")

# 4 - keep looping through
while 1:
    # 5 - clear the screen before drawing it again
    screen.fill(0)
    # 6 - draw the screen elements
    screen.blit(player, (100,100))
    # 7 - update the screen
    pygame.display.flip()
    # 8 - loop through the events
    for event in pygame.event.get():
        # check if the event is the X button
        if event.type==pygame.QUIT:
```

```
            # if it is quit the game
            pygame.quit()
            exit(0)
```

Save the file into your game folder (the one where the resources subfolder is) and name it **game.py**.

Let's go through the code section-by-section:

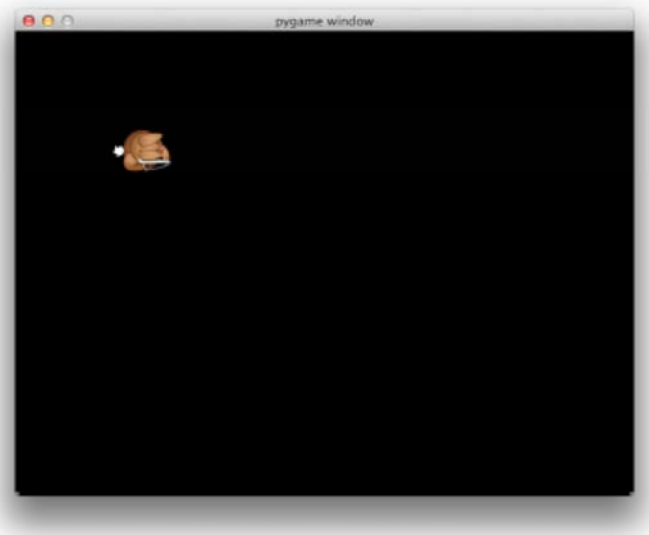
1. Import the PyGame library. This allows you to use functions from the library in your program.
2. Initialize PyGame and set up the display window.
3. Load the image that you will use for the bunny.
4. Keep looping over the following indented code.

   > **Note:** Where other languages like Objective-C, Java or PHP use curly braces to show a block of code to be executed within a while loop or an if statement, Python uses indenting to identify code blocks. So proper indentation is very important in Python – keep that in mind. :]

5. Fill the screen with black before you draw anything.
6. Add the bunny image that you loaded to the screen at x=100 and y=100.
7. Update the screen.
8. Check for any new events and if there is one, and it is a quit command, exit the program.

   > **Note:** According to the PyGame documentation, you shouldn't need to call **pygame.quit()** since the interpreter will automatically call it when the interpreter shuts down. However, at least on Mac OS, the game would hang on exit unless **pygame.quit()** was called.

If you run the code now (via Run\Run Module in the Idle menu), you should see a screen similar to the one below:



w00t the bunny is in the scene, and ready for action!

But the game looks scary and lonely with the bunny just standing there on a black background. Time to prettify things a little bit. :]

Step 2: Add Scenery

Let's start by adding a background to the game scene. This can be done with a couple more **screen.blit()** calls.

At the end of section #3, after loading the player image, add the following code:

```
grass = pygame.image.load("resources/images/grass.png")
castle = pygame.image.load("resources/images/castle.png")
```

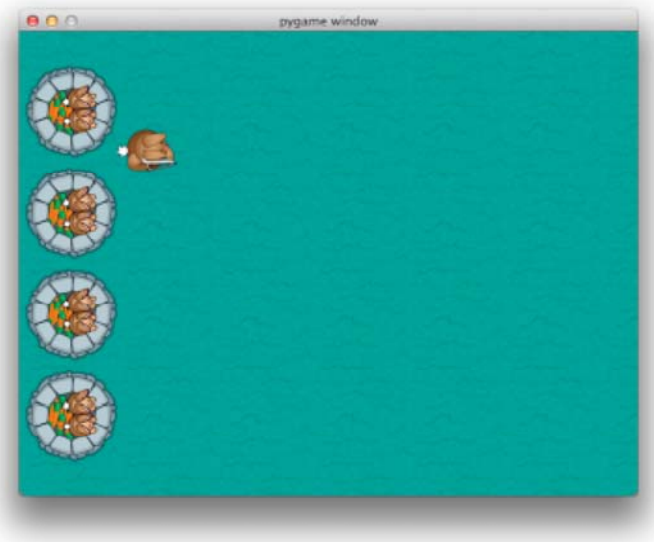
This loads the images and puts them into specific variables. Now they have to be drawn on screen. But if you check the grass image, you will notice that it won't cover the entire screen area, which is 640 x 480. This means you have to tile the grass over the screen area to cover it completely.

Add the following code to **game.py** at the beginning of section #6 (before the player is drawn on screen):

```
    for x in range(width/grass.get_width()+1):
        for y in range(height/grass.get_height()+1):
            screen.blit(grass,(x*100,y*100))
    screen.blit(castle,(0,30))
    screen.blit(castle,(0,135))
    screen.blit(castle,(0,240))
    screen.blit(castle,(0,345 ))
```

As you can see, the for statement loops through x first. Then, within that for loop, it loops through y and draws the grass at the x and y values generated by the for loops. The next couple of lines just draw the castles on the screen.

If you run the program now, you should get something like this:



Much better – this is starting to look good! :]

Step 3: Make the Bunny Move

Next you need to add some actual gameplay elements, like making the bunny respond to key presses.

To do that, first you'll implement a good method of keeping track of which keys are being pressed at a given moment. You can do this simply by making an array of key states that holds the state of each key you want to use for the game.

Add the following code to **game.py** at the end of section #2 (after you set the screen height and width):

```
keys = [False, False, False, False]
playerpos=[100,100]
```

This code is pretty self-explanatory. The keys array keeps track of the keys being pressed in the following order: WASD. Each item in the array corresponds to one key – the first to W, the second to A and so on.

The playerpos variable is where the program draws the player. Since the game will move the player to different positions, it's easier to have a variable that contains the player position and then simply draw the player at that position.

Now you need to modify the existing code for drawing the player to use the new **playerpos** variable. Change the following line in section #6:

```
    screen.blit(player, (100,100))
```

To:

```
    screen.blit(player, playerpos)
```

Next, update the keys array based on which keys are being pressed. PyGame makes detecting key presses easy by adding event.key functions.

At the end of section #8, right after the block checking for event.type==pygame.QUIT, put this code (at the same indentation level as the pygame.QUIT if block):

```
        if event.type == pygame.KEYDOWN:
            if event.key==K_w:
                keys[0]=True
            elif event.key==K_a:
                keys[1]=True
            elif event.key==K_s:
                keys[2]=True
            elif event.key==K_d:
                keys[3]=True
        if event.type == pygame.KEYUP:
            if event.key==pygame.K_w:
                keys[0]=False
            elif event.key==pygame.K_a:
                keys[1]=False
            elif event.key==pygame.K_s:
                keys[2]=False
            elif event.key==pygame.K_d:
                keys[3]=False
```

Wow! Those are a lot of lines of code. If you break it down into the if statements though, it's not that complicated.

First you check to see if a key is being pressed down or released. Then you check which key is being pressed or released, and if the key being pressed or released is one of the keys you're using, you update the keys variable accordingly.

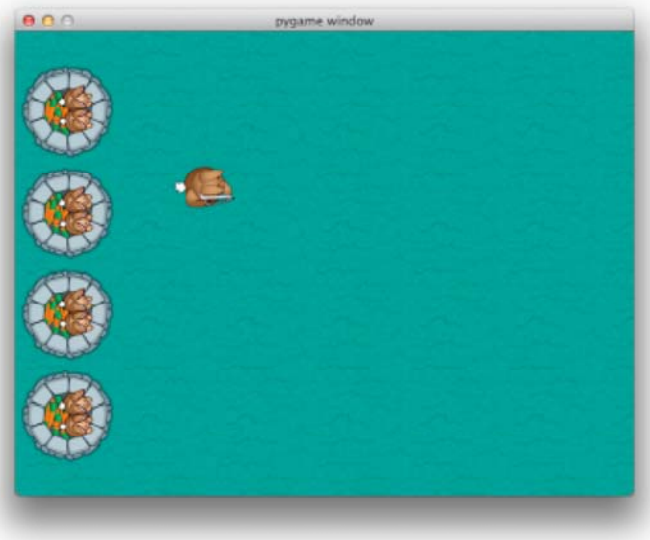
Finally, you need to update the **playerpos** variable in response to the key presses. This is actually very simple.

Add the following code to the end of game.py (with one indentation level, putting it at the same level as the for loop):

```
    # 9 - Move player
    if keys[0]:
        playerpos[1]-=5
    elif keys[2]:
        playerpos[1]+=5
    if keys[1]:
        playerpos[0]-=5
    elif keys[3]:
        playerpos[0]+=5
```

This code simply checks which of the keys are being pressed and adds or subtracts from the player's x or y position (depending on the key pressed) to move the player.

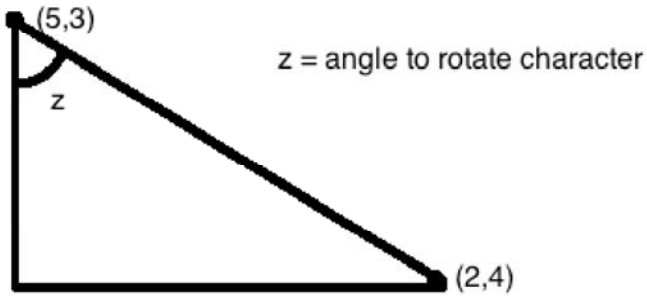
Run the game and you should get a player just like before. Try pressing WASD. Yay! It works.



Step 4: Turning the Bunny

Yes, your bunny now moves when you press keys but wouldn't it be even cooler if you could use your mouse to rotate the bunny to face a direction of your choosing, so he's not facing the same way all the time? It's simple enough to implement using trigonometry.

Take a look at the following illustration:



atan2(diff x, diff y) = z
atan2(5-2, 3-4) = z
atan2(3, -1) = z

In the above image, if (5,3) is the position of the bunny and (2,4) is the current position of the mouse, you can find the rotation angle (z) by applying the atan2 trigonometric function to the difference in distances between the two points. Of course, once you know the rotation angle, you can simply rotate the bunny accordingly. :]

If you're a bit confused about this part, don't worry – you can continue on anyway. But this is why you should pay attention in Math class! :] You'll use this stuff all the time in game programming.

Now you need to apply this concept to your game. To do this, you can use the PyGame **Surface.rotate(degrees)** function. Incidentally, keep in mind that the Z value is in radians. :[

The atan2 function comes from the Python math library. So add this to the end of section #1 first:

```
import math
```

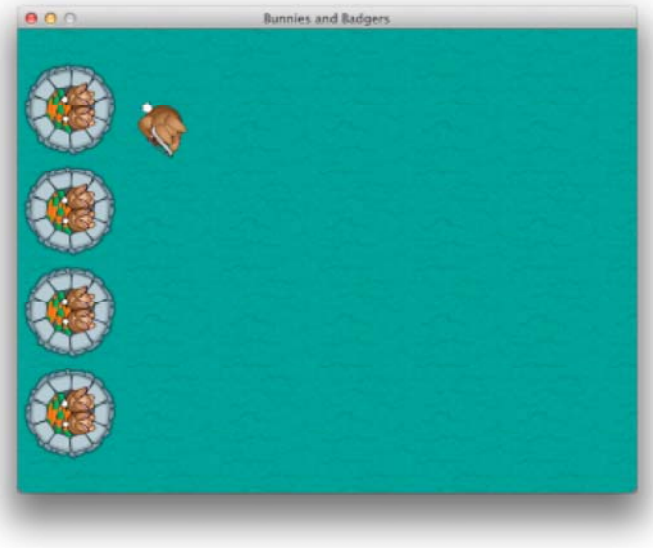
Then, replace the last line in section #6 (the player.blit line) with the following code:

```
    # 6.1 - Set player position and rotation
    position = pygame.mouse.get_pos()
    angle = math.atan2(position[1]-(playerpos[1]+32),position[0]-(playerpos[0]+26))
    playerrot = pygame.transform.rotate(player, 360-angle*57.29)
    playerpos1 = (playerpos[0]-playerrot.get_rect().width/2, playerpos[1]-playerrot.get_rect
().height/2)
    screen.blit(playerrot, playerpos1)
```

Let's go through the basic structure of the above code. First you get the mouse and player positions. Then you feed those into the atan2 function. After that, you convert the angle received from the atan2 function from radians to degrees (multiply radians by approximately 57.29 or 360/2π).

Since the bunny will be rotated, its position will change. So now you calculate the new bunny position and display the bunny on screen.

Run the game again. If you use just the WASD keys, then the game should behave exactly like before. But if you move your mouse, the bunny rotates too. Cool!



Step 5: Shoot, Bunny, Shoot!

Now that your bunny's moving around, it's time to add a little more action. :] How about letting the bunny shoot enemies using arrows? This is no mild-mannered rabbit!

This step is a bit more complicated because you have to keep track of all the arrows, update them, rotate them, and delete them when they go off-screen.

First of all, add the necessary variables to the end of the initialization section, section #2:

```
acc=[0,0]
arrows=[]
```

The first variable keeps track of the player's accuracy and the second array tracks all the arrows. The accuracy variable is essentially a list of the number of shots fired and the number of badgers hit. Later, we will be using this information to calculate an accuracy percentage.

Next, load the arrow image at the end of section #3:

```
arrow = pygame.image.load("resources/images/bullet.png")
```

Now when a user clicks the mouse, an arrow needs to fire. Add the following to the end of section #8 as a new event handler:

```
        if event.type==pygame.MOUSEBUTTONDOWN:
            position=pygame.mouse.get_pos()
            acc[1]+=1
            arrows.append([math.atan2(position[1]-(playerpos1[1]+32),position[0]-(playerpos1
[0]+26)),playerpos1[0]+32,playerpos1[1]+32])
```

This code checks if the mouse was clicked and if it was, it gets the mouse position and calculates the arrow rotation based on the rotated player position and the cursor position. This rotation value is stored in the **arrows** array.

Next, you have to actually draw the arrows on screen. Add the following code right after section #6.1:

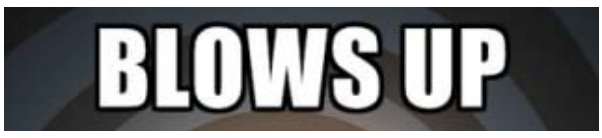
```
    # 6.2 - Draw arrows
    for bullet in arrows:
        index=0
        velx=math.cos(bullet[0])*10
        vely=math.sin(bullet[0])*10
        bullet[1]+=velx
        bullet[2]+=vely
        if bullet[1]<-64 or bullet[1]>640 or bullet[2]<-64 or bullet[2]>480:
            arrows.pop(index)
        index+=1
        for projectile in arrows:
            arrow1 = pygame.transform.rotate(arrow, 360-projectile[0]*57.29)
            screen.blit(arrow1, (projectile[1], projectile[2]))
```

The **vely** and **velx** values are calculated using basic trigonometry. 10 is the speed of the arrows. The if statement just checks if the bullet is out of bounds and if it is, it deletes the arrow. The second for statement loops through the arrows and draws them with the correct rotation.

Try and run the program. You should have a bunny that shoots arrows when you click the mouse! :D

Step 6: Take Up Arms! Badgers!

OK, you have a castle and you have a hero who can move and shoot. So what's missing? Enemies who attack the castle that the hero can shoot at!



In this step, you'll create randomly generated badgers that run at the castle. There will be more and more badgers as the game progresses. So, let's make a list of what you'll need it to do.

1. Add bad guys to a list an array.
2. Update the bad guy array each frame and check if they are off screen.
3. Show the bad guys.

Easy, right? :]

First, add the following code to the end of section #2:

```
badtimer=100
badtimer1=0
badguys=[[640,100]]
healthvalue=194
```

The above sets up a timer (as well as a few other values) so that the game adds a new badger after some time has elapsed. You decrease the badtimer every frame until it is zero and then you spawn a new badger.

Now add the following to the end of section #3:

```
badguyimg1 = pygame.image.load("resources/images/badguy.png")
badguyimg=badguyimg1
```

The first line above is similar to all the previous image-loading code. The second line sets up a copy of the image so that you can animate the bad guy much more easily.

Next, you have to update and show the bad guys. Add this code right after section #6.2:

```
    # 6.3 - Draw badgers
    if badtimer==0:
        badguys.append([640, random.randint(50,430)])
        badtimer=100-(badtimer1*2)
        if badtimer1>=35:
            badtimer1=35
        else:
            badtimer1+=5
    index=0
    for badguy in badguys:
        if badguy[0]<-64:
            badguys.pop(index)
        badguy[0]-=7
        index+=1
```

```
    for badguy in badguys:
        screen.blit(badguyimg, badguy)
```

Lots of code to go over. :] The first line checks if **badtimer** is zero and if it is, creates a badger and sets **badtimer** up again based on how many times **badtimer** has run so far. The first for loop updates the x position of the badger, checks if the badger is off the screen, and removes the badger if it is offscreen. The second for loop draws all of the badgers.

In order to use the random function in the above code, you also need to import the random library. So add the following to the end of section #1:

```
import random
```

Finally, add this line right after the while statement (section #4) to decrement the value of **badtimer** for each frame:

```
badtimer-=1
```

Try all of this code out by running the game. Now you should start seeing some real gameplay – you can shoot, move, turn, and badgers try to run at you.

But wait! Why aren't the badgers blowing up the castle? Let's quickly add that in…

Add this code right before the **index+=1** on the first for loop in section #6.3:

```
        # 6.3.1 - Attack castle
        badrect=pygame.Rect(badguyimg.get_rect())
        badrect.top=badguy[1]
        badrect.left=badguy[0]
        if badrect.left<64:
            healthvalue -= random.randint(5,20)
            badguys.pop(index)
        # 6.3.3 - Next bad guy
```

This code is fairly simple. If the badger's x value is less than 64 to the right, then delete that bad guy and decrease the game health value by a random value between 5 and 20. (Later, you will display the current health value on screen.)

If you build and run the program, you should get a bunch of attacking badgers who vanish when they hit the castle. Although you cannot see it, the badgers are actually lowering your health.

Step 7: Collisions with Badgers and Arrows

The badgers are attacking your castle but your arrows have no effect on them! How's a bunny supposed to defend his home?

Time to set the arrows to kill the badgers so you can safeguard your castle and win the game! Basically, you have to loop through all of the bad guys and inside each of those loops, you have to loop through all of the arrows and check if they collide. If they do, then delete the badger, delete the arrow, and add one to your accuracy ratio.

Right after section #6.3.1, add this:

```
        #6.3.2 - Check for collisions
        index1=0
        for bullet in arrows:
            bullrect=pygame.Rect(arrow.get_rect())
            bullrect.left=bullet[1]
            bullrect.top=bullet[2]
            if badrect.colliderect(bullrect):
                acc[0]+=1
                badguys.pop(index)
                arrows.pop(index1)
            index1+=1
```

There's only one important thing to note in this code. The if statement is a built-in PyGame function that checks if two rectangles intersect. The other couple of lines just do what I explained above.

If you run the program, you should finally be able to shoot and kill the badgers.

Step 8: Add a HUD with Health Meter and Clock

The game's progressing pretty well. You have your attackers, and you have your defender. Now all you need is a way to keep score and to show how well the bunny is doing.

The easiest way to do this is to add a HUD (Heads Up Display) that shows the current health level of the castle. You can also add a clock to show how long the castle has survived.

First add the clock. Add the following code right before the beginning of section #7:

```
    # 6.4 - Draw clock
    font = pygame.font.Font(None, 24)
    survivedtext = font.render(str((90000-pygame.time.get_ticks())/60000)+":"+str
((90000-pygame.time.get_ticks())/1000%60).zfill(2), True, (0,0,0))
    textRect = survivedtext.get_rect()
    textRect.topright=[635,5]
    screen.blit(survivedtext, textRect)
```

The above code simply creates a new font using the default PyGame font set to size 24. Then that font is used to render the text of the time onto a surface. After that, the text is positioned and drawn onscreen.

Next add the health bar. But before drawing the health bar, you need to load the images for the bar. Add the following code to the end of section #3:

```
healthbar = pygame.image.load("resources/images/healthbar.png")
health = pygame.image.load("resources/images/health.png")
```

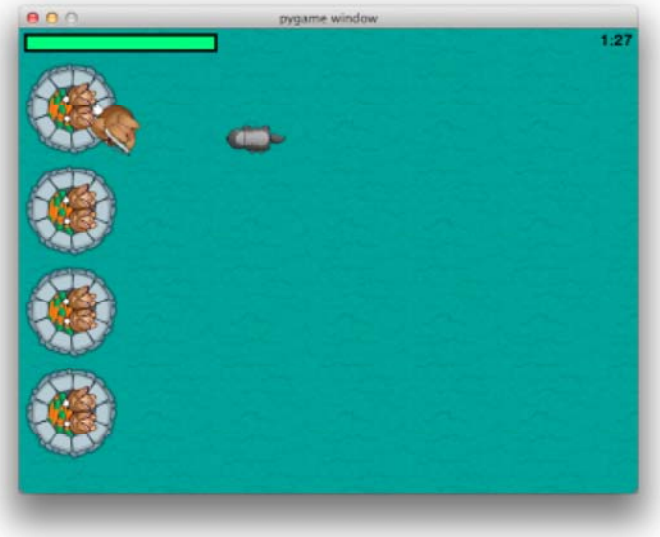
The first is the red image used for the full health bar. The second is the green image used to show the current health level.

Now add the following code right after section #6.4 (which you just added) to draw the health bar on screen:

```
    # 6.5 - Draw health bar
    screen.blit(healthbar, (5,5))
    for health1 in range(healthvalue):
        screen.blit(health, (health1+8,8))
```

The code first draws the all-red health bar. Then it draws a certain amount of green over the bar, according to how much life the castle has remaining.

If you build and run the program, you should have a timer and a health bar.



Step 9: Win or Lose

But what's this? If you play for long enough, even if your health goes down to zero, the game still continues! Not just that, you can continue to shoot at the badgers, too. That isn't going to work, now is it? You need some sort of a win/lose scenario to make the game worth playing.

So let's add the win and lose condition as well as the win or lose screen. :] You do this by exiting out of the main loop and going into a win/lose loop. In the win/lose loop, you have to figure out if the user won or lost and display the screen accordingly.

Here is a basic outline of the win/lose scenarios:

If time is up (90000 ms or 90 seconds) then:

- Stop running the game
- Set outcome of game to 1 or win

If the castle is destroyed then:

- Stop running game
- Set outcome of game to 1 or win

Calculate the accuracy either way.

> **Note:** The acc[0]*1.0 is just converting acc[0] to a float. If you do not do this, the division operand will return an integer like 1 or 2 instead of a float like 1.5

Add these lines to the end of **game.py**:

```
    #10 - Win/Lose check
    if pygame.time.get_ticks()>=90000:
        running=0
        exitcode=1
    if healthvalue<=0:
        running=0
        exitcode=0
    if acc[1]!=0:
        accuracy=acc[0]*1.0/acc[1]*100
    else:
        accuracy=0
# 11 - Win/lose display
if exitcode==0:
    pygame.font.init()
    font = pygame.font.Font(None, 24)
    text = font.render("Accuracy: "+str(accuracy)+"%", True, (255,0,0))
    textRect = text.get_rect()
    textRect.centerx = screen.get_rect().centerx
    textRect.centery = screen.get_rect().centery+24
    screen.blit(gameover, (0,0))
    screen.blit(text, textRect)
else:
    pygame.font.init()
    font = pygame.font.Font(None, 24)
    text = font.render("Accuracy: "+str(accuracy)+"%", True, (0,255,0))
    textRect = text.get_rect()
    textRect.centerx = screen.get_rect().centerx
    textRect.centery = screen.get_rect().centery+24
    screen.blit(youwin, (0,0))
    screen.blit(text, textRect)
while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            exit(0)
    pygame.display.flip()
```

This is the longest bit of code yet! But it's not that complicated.

The first if statement checks if the time is up. The second one checks if the castle is destroyed. The third one calculates your accuracy ratio. After that, a quick if statement checks if you won or lost and displays the correct image.

Of course, if you want to display images for the win and lose screens, then those images have to be loaded first. So add the following code to the end of section #3:

```
gameover = pygame.image.load("resources/images/gameover.png")
youwin = pygame.image.load("resources/images/youwin.png")
```

One more quick thing. Change section #4 from:

```
# 4 - keep looping through
while 1:
    badtimer-=1
```

To:

```
# 4 - keep looping through
running = 1
exitcode = 0
while running:
    badtimer-=1
```

The running variable keeps track of if the game is over and the exit code variable keeps track of whether the player won or lost.

Run the game again and now you can either triumph or die trying! Cool. :]

Step 10: Gratuitous Music and Sound Effects!

The game's looking pretty good but how does it sound? It's a bit quiet, isn't it? Adding some sound effects can change the whole feel of a game.

PyGame makes loading and playing sounds super simple. First, you have to initialize the mixer by adding this at the end of section #2:

```
pygame.mixer.init()
```

Then load the sounds and set the volume level at the end of section #3:

```
# 3.1 - Load audio
hit = pygame.mixer.Sound("resources/audio/explode.wav")
enemy = pygame.mixer.Sound("resources/audio/enemy.wav")
shoot = pygame.mixer.Sound("resources/audio/shoot.wav")
hit.set_volume(0.05)
enemy.set_volume(0.05)
shoot.set_volume(0.05)
pygame.mixer.music.load('resources/audio/moonlight.wav')
pygame.mixer.music.play(-1, 0.0)
pygame.mixer.music.set_volume(0.25)
```

Most of the above code is simply loading the audio files and configuring the audio volume levels. But pay attention to the pygame.mixer.music.load line – that line loads the background music for the game and the next line sets the background music to repeat forever.

That takes care of the audio configuration. :] Now all you need to do is to play the various sound effects as needed. Do that as per the directions in the comments for the code below:

```
# section 6.3.1 after if badrect.left<64:
hit.play()
# section 6.3.2 after if badrect.colliderect(bullrect):
enemy.play()
# section 8, after if event.type==pygame.MOUSEBUTTONDOWN:
shoot.play()
```

Run the game one more time and you'll notice that you now have background music and sound effects for collisions and shooting. The game feels that much more alive! :]

Where To Go From Here?

You should be proud of yourself: you have just finished creating a fun game packed with music, sound effects, a killer rabbit, kamikaze badgers and more. I told you you could do it! :]

You can download the final source code for the game here.

At this point, feel free to extend this game into your own creation! Maybe try replacing the art with your own drawings, or adding different weapons or types of monsters into the game!

Do you have any questions or comments about what you've done in this tutorial? If so, join the discussion in the forums! I would love to hear from you.

This is a post by Tutorial Team Member Julian Meyer, a 13-year-old python developer.