JAN 6TH, 2016 | COMMENTS

PyGame - a Primer

PyGame (http://www.pygame.org/hifi.html) is a Python wrapper for the SDL library (https://www.libsdl.org/). SDL is a cross-platform library for accessing computer multimedia hardware components (sound, video, input, etc.). SDL is an extremely powerful tool for building all kinds of things, but it's written in C, and C is hard, so we use PyGame.

In this tutorial we'll go over the basic PyGame logic and collision detection as well as drawing to the screen and loading outside files into our game.

> **NOTE**: This tutorial assumes you have a basic understanding of the Python syntax, file structure, and OOP.

Setup

Navigate to the PyGame download page (http://www.pygame.org/download.shtml), and find the proper binary package for your operating system and version of Python.

Create a new file and input the following code:

```
1  import pygame
2  from pygame.locals import *
3
4  pygame.init()
```

As with all Python programs, we begin by importing the modules we want to use. In this case we will be importing `pygame` itself and `pygame.locals`, which we will use later for some of the constants. The last line initializes all the PyGame modules, it must be called before you do anything else with PyGame.

Building Blocks

Screen Objects

First things first: We need something to draw on, so we will create a "screen (https://www.pygame.org/docs/ref/display.html)" which will be our overall canvas. In order to create a screen to display on, we call the `set_mode()` method of `pygame.display` and then pass `set_mode()` a tuple with the width and height of the window we want (800x600 in this case):

```
1  import pygame
2  from pygame.locals import *
3
4  pygame.init()
5
6  screen = pygame.display.set_mode((800, 600))
```

If you run this now you'll see our window pop up briefly and then immediately disappear as the program exits. Not very impressive, is it? In the next section we will introduce our main game loop to ensure that our program only exits when we give it the correct input.

Game Loop

The main game/event loop (https://www.pygame.org/docs/ref/event.html) is where all the action happens. It runs continuously during gameplay, updating the game state, rendering the screen, and collecting input. When we create our loop we need to make sure that we have a way to get out of the loop and exit the application. To that end we will introduce some basic user input at the same time. All user input (and some other events we will get into later) go into the PyGame event queue, which you can access by calling `pygame.event.get()` . This will return a list of all the events in the queue, which we will loop through and respond to according to the type of event. For now all we care about are `KEYDOWN` and `QUIT` events:

```
1   # Variable to keep our main loop running
2   running = True
3
4   # Our main loop!
5   while running:
6       # for loop through the event queue
7       for event in pygame.event.get():
8           # Check for KEYDOWN event; KEYDOWN is a constant defined in pyga
    me.locals, which we imported earlier
9           if event.type == KEYDOWN:
10              # If the Esc key has been pressed set running to false to ex
    it the main loop
11              if event.key == K_ESCAPE:
12                  running = False
13          # Check for QUIT event; if QUIT, set running to false
14          elif event.type == QUIT:
15              running = False
```

Add these lines to the previous code and run it. You should see an empty window. It wont go away until you press the ESC key or trigger a QUIT event by closing the window.

Surfaces and Rects

Surfaces (https://www.pygame.org/docs/ref/surface.html) and Rects (https://www.pygame.org/docs/ref/rect.html) are basic building blocks in PyGame. Think of surfaces as a blank sheet of paper that you can draw whatever you want onto. Our screen object is also a Surface. They can hold images as well. Rects are a representation of a rectangular area that your Surface encompasses.

Let's create a basic Surface that's 50 pixels by 50 pixels, then let's fill in the Surface with a color. We'll use white because the default window background is black and we want it to be nice and visible. We'll then call the `get_rect()` method on our Surface to get the rectangular area and the x, y coordinates of our surface:

```
1  # Create the surface and pass in a tuple with its length and width
2  surf = pygame.Surface((50, 50))
3  # Give the surface a color to differentiate it from the background
4  surf.fill((255, 255, 255))
5  rect = surf.get_rect()
```

Blit and Flip

Just creating our Surface isn't actually enough to see it on the screen. To do that we need to Blit (https://www.pygame.org/docs/ref/surface.html#pygame.Surface.blit) the Surface onto another Surface. Blit is just a technical way to say draw. You can only Blit from one Surface object to another – but remember, our screen is just another Surface object. Here's how we'll draw our `surf` to the screen:

```
1  # This line says "Draw surf onto screen at coordinates x:400, y:300"
2  screen.blit(surf, (400, 300))
3  pygame.display.flip()
```

`blit()` takes two arguments: The Surface to draw and the location to draw it at on the source Surface. Here we use the exact center of the screen, but when you run the code you'll notice our `surf` does not end up centered on the screen. This is because `blit()` will draw surf starting at the top left position.

Notice the call to `pygame.display.flip()` after our Blit. Flip (https://www.pygame.org/docs/ref/display.html#pygame.display.flip) will update the entire screen with everything that has been drawn since the last flip. Without a call to `flip()`, nothing will show.

Sprites

What are Sprites? In programming terms a Sprite is a 2d representation of something on the screen. Essentially, a Sprite is a picture. Pygame provides a basic class called `Sprite`, which is meant to be extended and used to hold one or several graphical representations of an object that you want to display on the screen. We will extend the Sprite (https://www.pygame.org/docs/ref/sprite.html) class so that we can use its built in methods. We'll call this new object `Player`. `Player` will extend `Sprite` and have only two properties for now: `surf` and `rect`. We will also give `surf` a color (white in this case) just like the previous surface example except that now the Surface belongs to the `Player`:

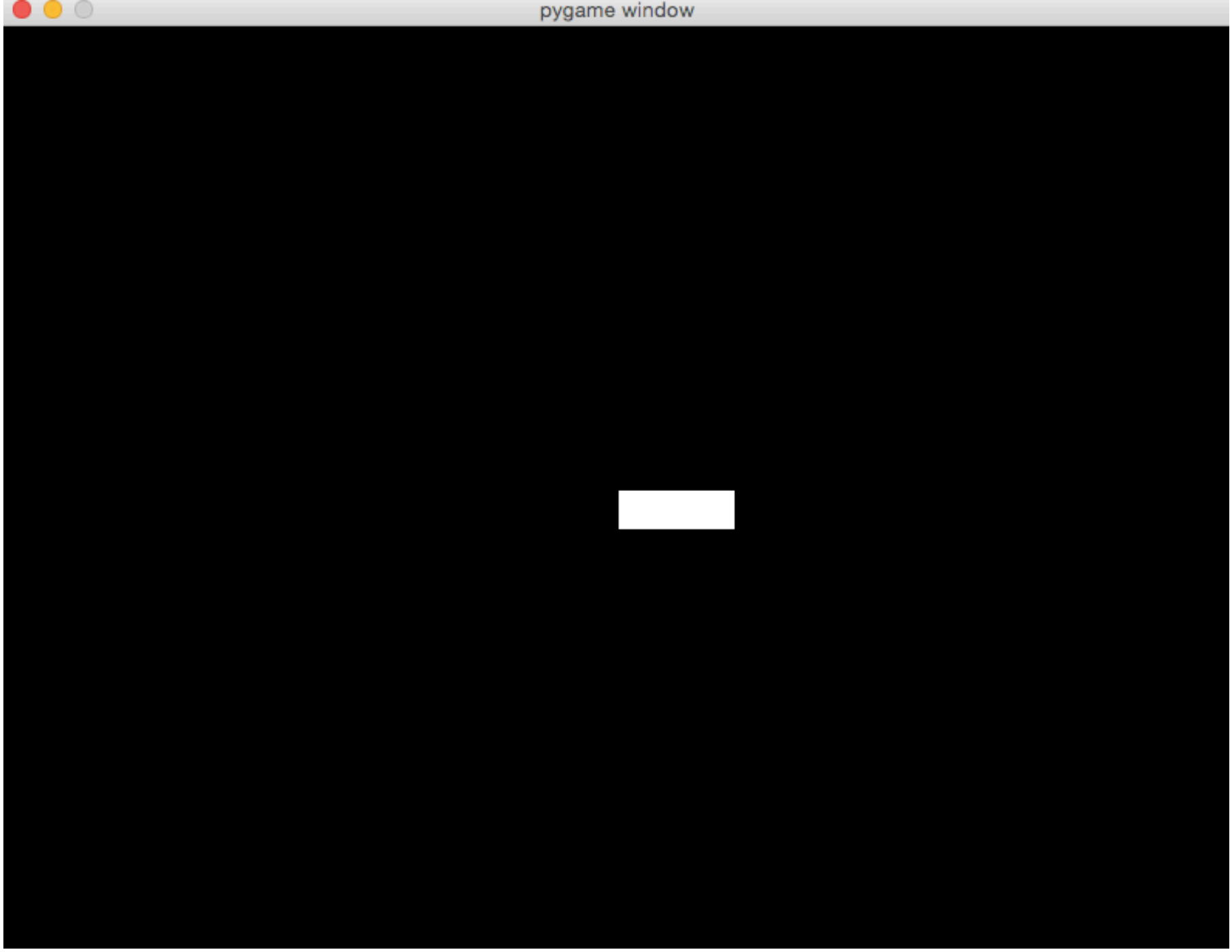
```
class Player(pygame.sprite.Sprite):
    def __init__(self):
        super(Player, self).__init__()
        self.surf = pygame.Surface((75, 25))
        self.surf.fill((255, 255, 255))
        self.rect = self.surf.get_rect()
```

Let's put it all together!

```
# import the pygame module
import pygame

# import pygame.locals for easier access to key coordinates
from pygame.locals import *

# Define our player object and call super to give it all the properties
and methods of pygame.sprite.Sprite
# The surface we draw on the screen is now a property of 'player'
class Player(pygame.sprite.Sprite):
    def __init__(self):
        super(Player, self).__init__()
        self.surf = pygame.Surface((75, 25))
        self.surf.fill((255, 255, 255))
        self.rect = self.surf.get_rect()
```

```
# initialize pygame
pygame.init()

# create the screen object
# here we pass it a size of 800x600
screen = pygame.display.set_mode((800, 600))

# instantiate our player; right now he's just a rectangle
player = Player()


# Variable to keep our main loop running
running = True

# Our main loop!
while running:
    # for loop through the event queue
    for event in pygame.event.get():
        # Check for KEYDOWN event; KEYDOWN is a constant defined in pyga
me.locals, which we imported earlier
        if event.type == KEYDOWN:
            # If the Esc key has been pressed set running to false to ex
it the main loop
            if event.key == K_ESCAPE:
                running = False
        # Check for QUIT event; if QUIT, set running to false
        elif event.type == QUIT:
            running = False

    # Draw the player to the screen
    screen.blit(player.surf, (400, 300))
    # Update the display
    pygame.display.flip()
```

Run this code. You'll see a white rectangle at roughly the middle of the screen:



What do you think would happen if you changed `screen.blit(player.surf,(400,300))` to `screen.blit(player.surf, player.rect)`? Once changed, try printing `player.rect` to the console. The first two attributes of the `rect()` are x, y coordinates of the top left corner of the `rect()`. When you pass Blit a Rect, it will use those coordinates to draw the surface. We will use this later to make our player move!

User Input

Here's where the fun starts! Let's make our player controllable. We discussed earlier that the keydown event `pygame.event.get()` pulls the latest event off the top of the event stack. Well, Pygame has another event method (https://www.pygame.org/docs/ref/event.html) called `pygame.event.get_pressed()`. The `get_pressed()` method returns a dictionary with all the keydown events in the queue. We will put this in our main loop so we get the keys at every frame.

```
1  pressed_keys = pygame.event.get_pressed()
```

Now we'll write a method that will take that dictionary and define the behavior of the sprite based off the keys that are pressed. Here's what it might look like:

```
def update(self, pressed_keys):
    if pressed_keys[K_UP]:
        self.rect.move_ip(0, -5)
    if pressed_keys[K_DOWN]:
        self.rect.move_ip(0, 5)
    if pressed_keys[K_LEFT]:
        self.rect.move_ip(-5, 0)
    if pressed_keys[K_RIGHT]:
        self.rect.move_ip(5, 0)
```

`K_UP`, `K_DOWN`, `K_LEFT`, and `K_RIGHT` correspond to the arrow keys on the keyboard. So we check that key, and if it's set to `True`, then we move our `rect()` in the relevant direction. Rects have two built-in methods for moving; here we use "move in place (https://www.pygame.org/docs/ref/rect.html#pygame.Rect.move_ip)" – `move_ip()` – because we want to move the existing Rect without making a copy.

Add the above method to our `Player` class and put the `get_pressed()` call in the main loop. Our code should now look like this:

```
import pygame

from pygame.locals import *


class Player(pygame.sprite.Sprite):
    def __init__(self):
        super(Player, self).__init__()
        self.surf = pygame.Surface((75, 25))
        self.surf.fill((255, 255, 255))
        self.rect = self.surf.get_rect()

    def update(self, pressed_keys):
        if pressed_keys[K_UP]:
            self.rect.move_ip(0, -5)
        if pressed_keys[K_DOWN]:
            self.rect.move_ip(0, 5)
        if pressed_keys[K_LEFT]:
            self.rect.move_ip(-5, 0)
        if pressed_keys[K_RIGHT]:
            self.rect.move_ip(5, 0)

pygame.init()

screen = pygame.display.set_mode((800, 600))

player = Player()


running = True

while running:
    for event in pygame.event.get():
        if event.type == KEYDOWN:
            if event.key == K_ESCAPE:
                running = False
        elif event.type == QUIT:
            running = False

    pressed_keys = pygame.key.get_pressed()

    player.update(pressed_keys)

    screen.blit(player.surf, (400, 300))
    pygame.display.flip()
```

Now you should be able to move your rectangle around the screen with the arrow keys. You may notice though that you can move off the screen, which is something we probably don't want. So let's add a bit of logic to the update method that tests if the rectangle's coordinates have moved beyond our 800 by 600 boundary; and if so, move it back to the edge:

```
def update(self, pressed_keys):
    if pressed_keys[K_UP]:
        self.rect.move_ip(0, -5)
    if pressed_keys[K_DOWN]:
        self.rect.move_ip(0, 5)
    if pressed_keys[K_LEFT]:
        self.rect.move_ip(-5, 0)
    if pressed_keys[K_RIGHT]:
        self.rect.move_ip(5, 0)

    #Keep player on the screen
    if self.rect.left < 0:
        self.rect.left = 0
    elif self.rect.right > 800:
        self.rect.right = 800
    if self.rect.top <= 0:
        self.rect.top = 0
    elif self.rect.bottom >= 600:
        self.rect.bottom = 600
```

Here instead of using a `move` method, we just alter the corresponding coordinates for top, bottom, left, or right.

Now let's add some enemies!

First let's create a new sprite class called 'Enemy'. We will follow the same formula we used for the player class:

```
class Enemy(pygame.sprite.Sprite):
    def __init__(self):
        super(Enemy, self).__init__()
        self.surf = pygame.Surface((20, 10))
        self.surf.fill((255, 255, 255))
        self.rect = self.surf.get_rect(center=(820, random.randint(0, 60
0)))
        self.speed = random.randint(5, 20)

    def update(self):
        self.rect.move_ip(-self.speed, 0)
        if self.rect.right < 0:
            self.kill()
```

There are a couple differences here that we should talk about. First off, when we call `get_rect()` on our surface, we are setting the center property x coordinate to 820, and our y coordinate to a random number generated by `random.randint()`.

Random (https://docs.python.org/3.5/library/random.html) is a python library that we will import at the beginning of our file in the complete code (`import random`). Why the random number? Simple: We want our incoming enemies to start past the right side of the screen (820), at a random place (0-600). We also use `random` to set a speed property for the enemies. This way we will have some enemies that are fast and some that are slow.

Our `update()` method for the enemies takes no arguments (we don't care about input for enemies) and simply moves the enemy toward the left side of the screen at a rate of speed. And the last `if` statement in the update method tests to see if the enemy has gone past the left side of the screen with the right side of its rectangle (so that they don't just disappear as soon as they touch the side of the screen). When they pass the side of the screen we call Sprites' built-in `kill()` method to delete them from their sprite group thereby preventing them from being rendered. Kill does not release the memory taken by the enemy and relies on you no longer having a reference to it so the Python garbage collector will take care of it.

Groups

Another super useful object that PyGame provides are Sprite groups (https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.Group). They are exactly what they sound like – Groups of Sprites. So why do we use Sprite Groups instead of a list? Well, sprite groups have several methods built into them that will help us later with collisions and updating. Let's make a Group right now that will hold all the Sprites in our game. After we create it, we will add the Player to the Group since that's our only Sprite so far. We can create another group for enemies as well. When we call a Sprite's `kill()` method, the sprite will be removed from all groups that it is a part of.

```
1  enemies = pygame.sprite.Group()
2  all_sprites = pygame.sprite.Group()
3  all_sprites.add(player)
```

Now that we have this `all_sprites` group, let's change how we are rendering our objects so that we render all objects in this group.

```
1  for entity in all_sprites:
2      screen.blit(entity.surf, entity.rect)
```

Now anything we put into `all_sprites` will be rendered.

Custom Events

Now we have a Sprite Group for our enemies but no actual enemies. So how do we get some enemies on the screen? We could just create a bunch of them at the beginning, but then our game wouldn't last more than a few seconds. So we will create a custom event that will fire off

every few seconds and trigger the creation of a new enemy. We listen for this event in the same way that we listened for key presses or quit events. Creating a custom event is as easy as naming it:

```
1  ADDENEMY = pygame.USEREVENT + 1
```

That's it! Now we have an event called `ADDENEMY` that we can listen for in our main loop. The only gotcha to keep in mind here is that we need our custom event to have a unique value that is greater than the value of `USEREVENT` . That's why we set our new event to equal `USEREVENT + 1` . One small note for anyone curious about about these events: They are at their core just integer constants. `USEREVENT` has a numeric value and any custom event we create needs to be an integer value that is greater than `USEREVENT` (because all the values less than `USEREVENT` are already taken by built-ins).

Now that we've defined our event, we need to insert it into the event queue. Since we need to keep creating them over the course of the game, we will set a timer. To do this we use PyGame's `time()` object.

```
1  pygame.time.set_timer(ADDENEMY, 250)
```

This tells PyGame to fire our `ADDENEMY` event every 250 milliseconds (every quarter second). This goes outside of our game loop, but will still fire throughout the entire game. Now let's add some code to listen for our event:

```
1  while running:
2      for event in pygame.event.get():
3          if event.type == KEYDOWN:
4              if event.key == K_ESCAPE:
5                  running = False
6          elif event.type == QUIT:
7              running = False
8          elif(event.type == ADDENEMY):
9              new_enemy = Enemy()
10             enemies.add(new_enemy)
11             all_sprites.add(new_enemy)
```

> Keep in mind that `set_timer()` is exclusively used for inserting events into the PyGame event queue – it doesn't do anything else.

Now we are listening for our `ADDENEMY` event, and when it fires, we create a new instance of the `Enemy` class. Then we add that instance to the `enemies` Sprite Group (which we will later use to test for collision) and to the `all_sprites` Group (so that it gets rendered along with everything else).

Collision

This is why you will love PyGame! Writing collision code is hard, but PyGame has a LOT of collision detection methods, some of which you can find here (https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.collide_rect). For this tutorial we will be using spritecollideany (https://www.pygame.org/docs/ref/sprite.html#pygame.sprite.spritecollideany). The `spritecollideany()` method takes a Sprite object and a Sprite Group and tests if the Sprite object intersects with any of the Sprites in the Sprite group. So we will take our player Sprite and our enemies Sprite Group and test if our player has been hit by an enemy. Here's what it looks like in code:

```
if pygame.sprite.spritecollideany(player, enemies):
    player.kill()
```

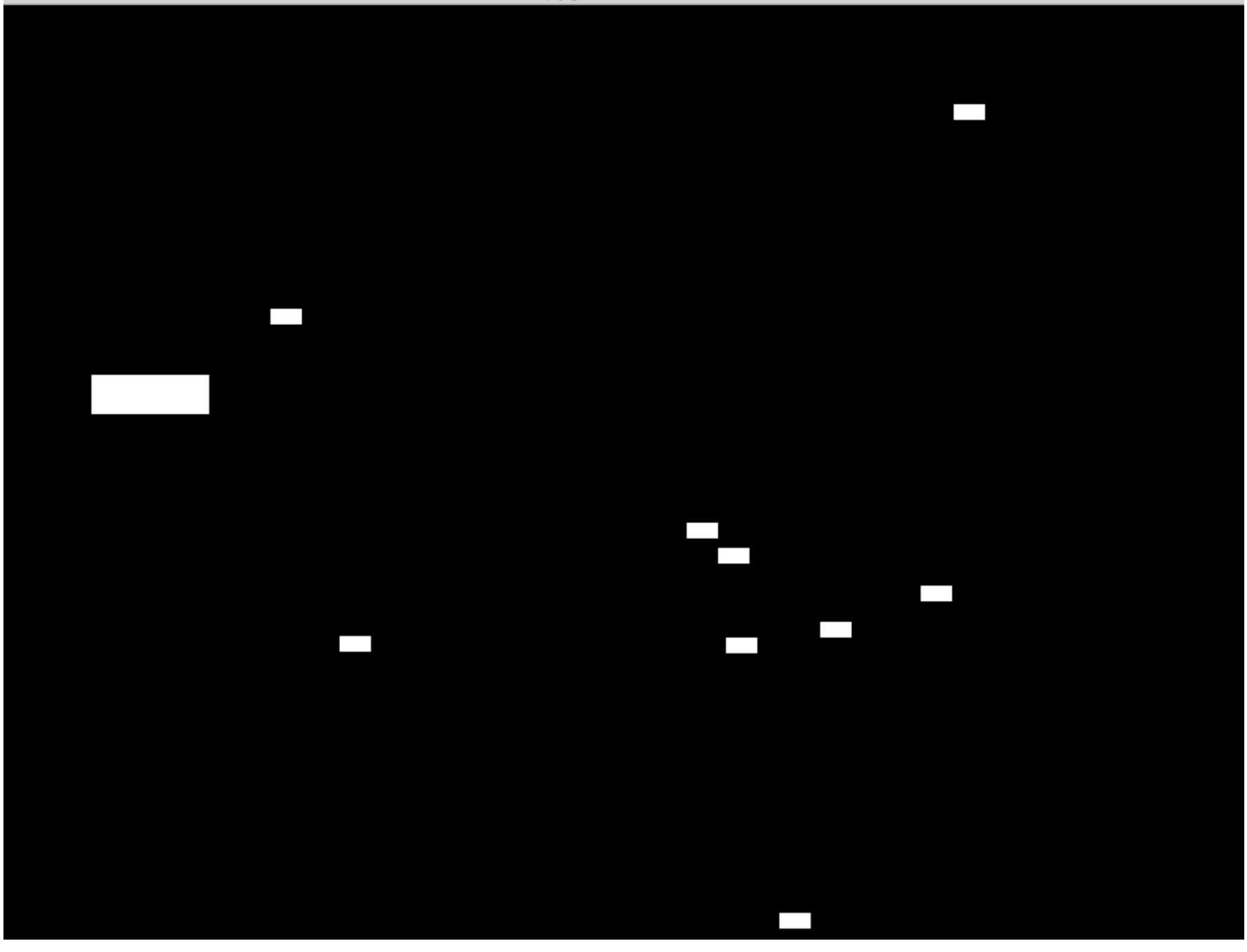
We test if our `player` Sprite object collides with any Sprites in the `enemies` Sprite Group, and if it does, we call the `kill()` method on the `player` Sprite. Because we are only rendering sprites in the `all_sprites` Group, and the `kill()` method removes a Sprite from all its Groups, our `player` will no longer be rendered, thus 'killing' it. Let's put it all together:

```
# import the pygame module
import pygame

# import random for random numbers!
import random

# import pygame.locals for easier access to key coordinates
from pygame.locals import *


class Player(pygame.sprite.Sprite):
    def __init__(self):
        super(Player, self).__init__()
        self.surf = pygame.Surface((75, 25))
        self.surf.fill((255, 255, 255))
        self.rect = self.surf.get_rect()

    def update(self, pressed_keys):
        if pressed_keys[K_UP]:
            self.rect.move_ip(0, -5)
        if pressed_keys[K_DOWN]:
            self.rect.move_ip(0, 5)
        if pressed_keys[K_LEFT]:
            self.rect.move_ip(-5, 0)
        if pressed_keys[K_RIGHT]:
            self.rect.move_ip(5, 0)

        # Keep player on the screen
        if self.rect.left < 0:
            self.rect.left = 0
```

```
        elif self.rect.right > 800:
            self.rect.right = 800
        if self.rect.top <= 0:
            self.rect.top = 0
        elif self.rect.bottom >= 600:
            self.rect.bottom = 600


class Enemy(pygame.sprite.Sprite):
    def __init__(self):
        super(Enemy, self).__init__()
        self.surf = pygame.Surface((20, 10))
        self.surf.fill((255,255,255))
        self.rect = self.surf.get_rect(
            center=(random.randint(820, 900), random.randint(0, 600))
        )
        self.speed = random.randint(5, 20)

    def update(self):
        self.rect.move_ip(-self.speed, 0)
        if self.rect.right < 0:
            self.kill()


# initialize pygame
pygame.init()

# create the screen object
# here we pass it a size of 800x600
screen = pygame.display.set_mode((800, 600))

# Create a custom event for adding a new enemy.
ADDENEMY = pygame.USEREVENT + 1
pygame.time.set_timer(ADDENEMY, 250)

# create our 'player'; right now he's just a rectangle
player = Player()

background = pygame.Surface(screen.get_size())
background.fill((0, 0, 0))

enemies = pygame.sprite.Group()
all_sprites = pygame.sprite.Group()
all_sprites.add(player)

running = True

while running:
    for event in pygame.event.get():
```

```
80          if event.type == KEYDOWN:
81              if event.key == K_ESCAPE:
82                  running = False
83          elif event.type == QUIT:
84              running = False
85          elif(event.type == ADDENEMY):
86              new_enemy = Enemy()
87              enemies.add(new_enemy)
88              all_sprites.add(new_enemy)
89      screen.blit(background, (0, 0))
90      pressed_keys = pygame.key.get_pressed()
91      player.update(pressed_keys)
92      enemies.update()
93      for entity in all_sprites:
94          screen.blit(entity.surf, entity.rect)
95
96      if pygame.sprite.spritecollideany(player, enemies):
97          player.kill()
98
99      pygame.display.flip()
```

Test this out!



Images

Now we have a game, but kind of an ugly game. Next we will replace all the boring white rectangles with cool images that will make the game feel like an actual game.

In the previous code examples we used a Surface object filled with a color to represent everything in our game. While this is a good way to get a handle on what a surface is and how they work, it makes for an ugly game. We're going to add some pictures for the enemies and the player. I like to draw my own images, so I made a little jet for the player and some missiles for the enemies, which you can download from the repo (https://github.com/realpython/pygame-primer). You're welcome to use my art, draw your own, or download some free game art assets (http://www.gameart2d.com/) to use.

Altering the Object Constructors

Our current player constructor looks like this:

```
1  class Player(pygame.sprite.Sprite):
2      def __init__(self):
3          super(Player, self).__init__()
4          self.surf = pygame.Surface((75, 25))
5          self.surf.fill((255, 255, 255))
6          self.rect = self.surf.get_rect()
```

Our new constructor will look like this:

```
1  class Player(pygame.sprite.Sprite):
2      def __init__(self):
3          super(Player, self).__init__()
4          self.image = pygame.image.load('jet.png').convert()
5          self.image.set_colorkey((255, 255, 255), RLEACCEL)
6          self.rect = self.image.get_rect()
```

We want to replace our Surface object with an image. We will use `pygame.image.load()` by passing it a path to a file. The `load()` method will actually return a Surface object. We then call `convert()` on that Surface object to create a copy that will draw more quickly on the screen.

Next we call the `set_colorkey()` method on our image. The `set_colorkey` method sets the color in the image that PyGame will render as transparent. In this case I chose white, because that's the background of my jet image. RLEACCEL (https://www.pygame.org/docs/ref/surface.html#pygame.Surface.set_colorkey) is a optional parameter that will help PyGame render faster on non-accelerated displays.

Lastly, we get our `rect` object in the same way as before: By calling `get_rect()` on our image.

> Remember the image is still a surface object; it now just has a picture painted on it.

Let's do the same thing with the enemy constructor:

```
class Enemy(pygame.sprite.Sprite):
    def __init__(self):
        super(Enemy, self).__init__()
        self.image = pygame.image.load('missile.png').convert()
        self.image.set_colorkey((255, 255, 255), RLEACCEL)
        self.rect = self.image.get_rect(
            center=(random.randint(820, 900), random.randint(0, 600))
        )
        self.speed = random.randint(5,20)
```

Now we have the same game we had before but nicely skinned with some cool images. I think it's missing something, though. Let's add a few clouds going past to give the impression of a jet flying through the sky. To do this we are going to use the exact same principles we used before. First, we will create the `Cloud` object with an image of a cloud and an `update()` method that continuously moves the cloud toward the left side of the screen. Then we will create a custom event to spawn our clouds at a set interval (we will also add the spawned clouds to the `all_sprites` group). Here's what our cloud object will look like:

```
class Cloud(pygame.sprite.Sprite):
    def __init__(self):
        super(Cloud, self).__init__()
        self.image = pygame.image.load('cloud.png').convert()
        self.image.set_colorkey((0, 0, 0), RLEACCEL)
        self.rect = self.image.get_rect(
            center=(random.randint(820, 900), random.randint(0, 600))
        )

    def update(self):
        self.rect.move_ip(-5, 0)
        if self.rect.right < 0:
            self.kill()
```

That should all look familiar, as should this event creation code, which we will put right below our enemy creation event:

```
ADDCLOUD = pygame.USEREVENT + 2
pygame.time.set_timer(ADDCLOUD, 1000)
```

And let's create a new Sprite Group for them:

```
clouds = pygame.sprite.Group()
```

Now in our main game loop, where we step through our event queue, we need to start listening for our `ADDCLOUD` event.

This:

```
for event in pygame.event.get():
    if event.type == KEYDOWN:
        if event.key == K_ESCAPE:
            running = False
    elif event.type == QUIT:
        running = False
    elif event.type == ADDENEMY:
        new_enemy = Enemy()
        enemies.add(new_enemy)
        all_sprites.add(new_enemy)
```

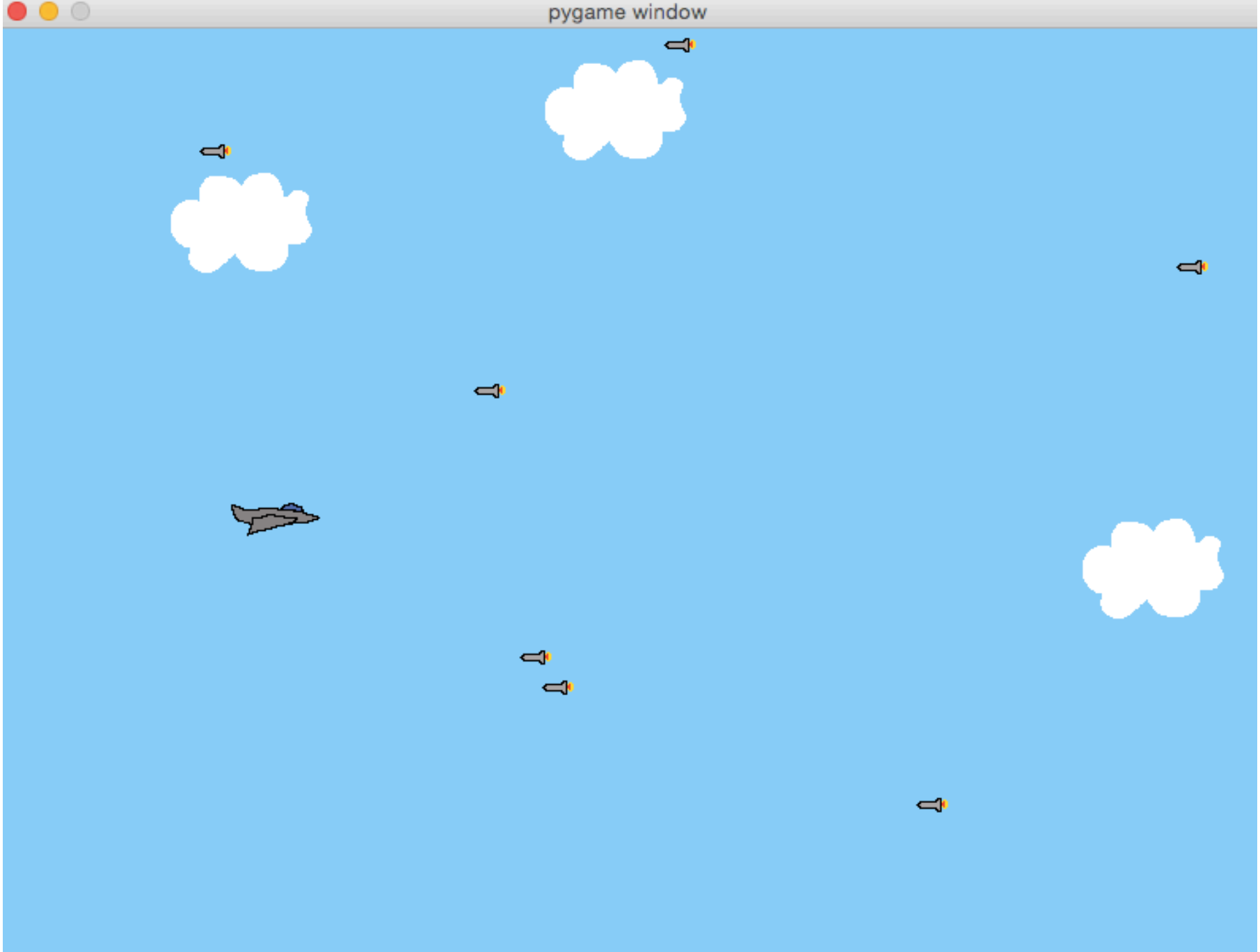
Will become this:

```
for event in pygame.event.get():
    if event.type == KEYDOWN:
        if event.key == K_ESCAPE:
            running = False
    elif event.type == QUIT:
        running = False
    elif event.type == ADDENEMY:
        new_enemy = Enemy()
        enemies.add(new_enemy)
        all_sprites.add(new_enemy)
    elif event.type == ADDCLOUD:
        new_cloud = Cloud()
        all_sprites.add(new_cloud)
        clouds.add(new_cloud)
```

We're going to add the clouds to the `all_sprites` Group as well as the new clouds Group. We add them to both because we're using `all_sprites` to render and `clouds` to call their update function. You might ask why we don't add them to the existing `enemies` Group; after all, we're calling nearly identical update functions on them. The reason is, we don't want to test the player for collisions with the clouds. Our jet needs to pass cleanly through all the clouds. Now all that's left is calling our clouds Group `update()` method.

Conclusion

That's it! Test it again, and you should see something like:



The complete code is available on the GitHub repo (https://github.com/realpython/pygame-primer). I hope you enjoyed the tutorial and found it helpful.

Posted by Real Python Jan 6th, 2016 python (/blog/categories/python/)

See an error in this post? Please submit a pull request on Github (https://github.com/realpython/realpython-blog).

Want to learn more? Download the Real Python course.

Download Now » $60 (https://app.simplegoods.co/i/IQCZADOY)

Or, click here (http://www.realpython.com/) to learn more about the course.

« Development and Deployment of Cookiecutter-Django via Docker (/blog/python/development-and-deployment-of-cookiecutter-django-via-docker/)

Development and Deployment of Django on Fedora » (/blog/python/development-and-deployment-of-cookiecutter-django-on-fedora/)

Comments

Categories

- analytics (/blog/categories/analytics/) [4]
- api (/blog/categories/api/) [6]
- bottle (/blog/categories/bottle/) [2]
- data science (/blog/categories/data-science/) [8]
- devops (/blog/categories/devops/) [16]
- django (/blog/categories/django/) [31]
- docker (/blog/categories/docker/) [6]
- editors (/blog/categories/editors/) [3]
- flask (/blog/categories/flask/) [32]
- front-end (/blog/categories/front-end/) [11]
- fundamentals (/blog/categories/fundamentals/) [18]
- migrations (/blog/categories/migrations/) [3]
- nosql (/blog/categories/nosql/) [5]
- opencv (/blog/categories/opencv/) [3]
- pyramid (/blog/categories/pyramid/) [1]
- redis (/blog/categories/redis/) [4]
- scraping (/blog/categories/scraping/) [2]
- sql (/blog/categories/sql/) [1]
- testing (/blog/categories/testing/) [10]
- web2py (/blog/categories/web2py/) [1]

Questions? info@realpython.com (mailto:info@realpython.com).

Back to top