Graduating from Scratch to Python: A Student and Teacher Guide

Seth Kenlon

Graduating from Scratch to Python: A Student and Teacher Guide

Seth Kenlon

Publication date 2017

Table of Contents

Chapter 1. Who should read this book

This book is intended for Scratch [https://www.raspberrypi.org/learning/getting-started-with-scratch/worksheet] users who are looking to graduate to a more advanced programming environment to make even more advanced games. It is written for *both* students and teachers, because everyone's a student at some point, and everyone can become a teacher.

Python [https://www.python.org/] is a popular open source language that is used in the majority of tech industries, including 3d modeling, filmmaking, desktop applications, IT, security, and gaming. It's an important language to learn, but it's also one of the easiest languages to learn because it's designed specifically to be clear and simple.

Are you ready for Python?

While Python is simpler than an advanced language like C++, it still is a professional-level language. It requires you to type out code, to keep files organised, and to debug.

Everyone has their own programming style, so there's no way to tell you exactly how you know when you're ready to graduate to Python. Here are some traits of a Scratch user who is probably ready to graduate to Python:

- Your games are getting too big or too complex for Scratch.
- You are comfortable using variables in Scratch.
- You find dragging and dropping blocks in Scratch slow and inefficient, and frequently look for ways to build your scripts faster.

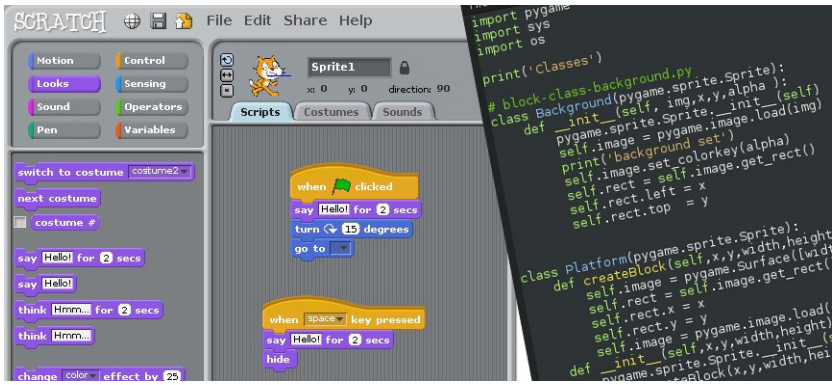
If any of those characteristics are familiar to you, then you are probably ready to tackle Python. However, you shouldn't feel that you *have to* start Python unless you want to. While it's true that you aren't going to get any jobs making games if all you know is Scratch, it's also true that Scratch is a lot of fun, and very powerful. Python is fun, too, and it does get easier, but it starts out hard. It takes work, and lots of practise. The most important thing is to keep programming fun, and you're free to move at your own pace.

If you are ready to learn something new, then this book is for you.

Chapter 2. Intro to Python

Python is an all-purpose programming language. It can be used to create desktop applications, 3d graphics, video games, and even websites. It's a good language to learn because it's simpler than complex languages like C, C++, or Java, but is almost as powerful, and is used in just about every industry that uses computers.

Unlike Scratch, Python doesn't have a GUI. You have to type code into a text editor, save the file, and then run it with the Python interpreter.



Starting Python

Since you've never used Python before, your first task is to try a few simple one-line programmes with it. You can talk directly to Python by running it in a terminal.

- On Linux or macOS, launch a terminal window and type **python**
- On Windows, launch Python from the Start menu.

  If there is no Python in the Start menu, launch PowerShell and type **Python.exe**

  If that doesn't work, make sure you have Python installed, or just use Linux [https://getfedora.org/en_GB/workstation/download/]. It's free and doesn't even require you to install it to use it.

Telling Python what to do

Just like Scratch has code blocks, Python has keywords that tell Python what you want it to do.

For instance, type this at your Python prompt and then press Return:

```
>>> print("Hello world.")
Hello world.
```

The keyword **print** tells Python to print out whatever text you give it in parentheses and quotes.

That's not very exciting, though. Unlike Scratch, which has probably 100 code blocks ready to use, Python only has access to some basic keywords, like **print**, **help**, and so on.

Use the **import** keyword to load more keywords.

```
>>> import turtle
```

Turtle is a fun module to use. It's exactly like the Pen code blocks in Scratch.

```
>>> turtle.begin_fill()
>>> turtle.forward(100)
>>> turtle.left(90)
>>> turtle.forward(100)
>>> turtle.left(90)
>>> turtle.forward(100)
>>> turtle.left(90)
>>> turtle.forward(100)
>>> turtle.end_fill()
```

Exercise

See what shapes you can draw with the turtle module.

To clear you turtle drawing area, use the **turtle.clear()** keyword.

What do you think the keyword **turtle.color("blue")** does?

Scripting Python

Now that you've had a chance to play around with Python in the terminal, it's time to write a Python script.

1. What is a script in Scratch?

It's a list of instructions that controls your programme.

A script in Python is a lot like a script in Scratch. The difference is that instead of using code blocks, Python uses keywords.

Open an empty text file and type this programme. In Python, the indentation matters, so type this exactly as you see it, using the Tab key to indent when necessary.

```
import turtle as t
import time

t.color("blue")
t.begin_fill()

n=0

while n < 4:
    t.forward(100)
    t.left(90)

t.end_fill()
time.sleep(5)
```

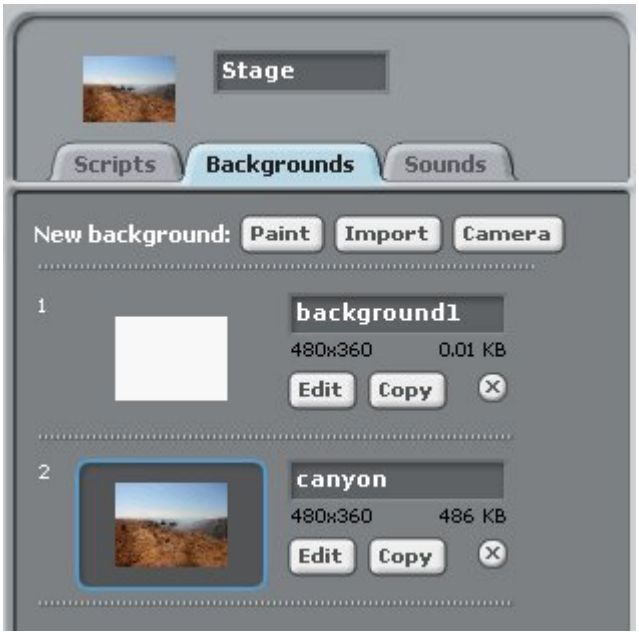
Save the text file as `turtle.py`.

To run your script, type **python $MOHE/turtle.py** on Linux or macOS. On Windows, type **Python.exe turtle.py**

Once you have run your script, it's time to explore an even better module. One popular module used to create video games with Python is called PyGame. It's got a lot more features than turtle, but is also a lot more complex, so be prepared.

Chapter 3. Creating the game world

A video game needs a *setting*, a world in which it takes place. In Scratch, you create a world by putting a costume on the stage.



In Python, you can create your setting in two different ways.

1. Set a background colour.

2. Set a background image.

Your background is only an image or colour. Your video game characters are not able to interact with things in the background, so don't put anything too important back there. It's just set dressing.

Setting up your Pygame script

A Python script starts with the filetype, your name, and the license you want to use. Use an open source license so that your friends can improve your game and share their changes with you:

```
#!/usr/bin/env python3
# by Seth Kenlon

## GPLv3
# This program is free software: you can redistribute it and/or
# modify it under the terms of the GNU General Public License as
# published by the Free Software Foundation, either version 3 of the
# License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Then, you tell Python what modules you want to use. Remember, modules are like sets of Scratch blocks.

```
import pygame # load pygame keywords
import sys    # let  python use your file system
import os     # help python identify your OS
```

Since you'll be working a lot with this script file, it helps to give yourself sections within the file so you know where to put stuff. You do this with block comments, which are comments that only you, the programmer, sees. Create three blocks in your code.

```
'''
Objects
'''

# put Python classes and functions here

'''
Setup
'''

# put run-once code here

'''
Main Loop
'''

# put game loop here
```

Next, set the window size for your game. Keep in mind that not everyone has a big computer screen, so it's best to use a screen size that will fit on most people's computers.

```
'''
Setup
'''
screenX = 960
screenY = 720
```

The Pygame engine requires some basic setup before you can use it in a script. You must set the frame rate, start its internal clock, and start (init) Pygame.

```
fps = 40   # frame rate
afps = 4   # animation cycles
clock = pygame.time.Clock()
pygame.init()
```

In Scratch, to activate a script, you use the *When green flag is clicked* block.

In Python, you don't have a green flag block, but you can use a keyword. A common keyword is `main`. If `main` is `True`, then the game is on. If `main` is not True, then the game has stopped.

```
main = True
```

Now you can set your background.

Setting the background

Scratch already knows what the *stage* is, but Python does not. Outside of Scratch, most video games just call the stage the "screen", so create a stage called screen for your game, and load a background imaage (`stage.png`) for it. Before you do, create a directory called `images` alongside of your Python script file and put a background image called `stage.png` into it.

```
screen   = pygame.display.set_mode([screenX,screenY])
backdrop = pygame.image.load(os.path.join('images','stage.png').convert()
backdropRect = screen.get_rect()
```

If you're just going to fill the background of your game screen with a colour, all you need is:

```
screen   = pygame.display.set_mode([screenX,screenY])
```

At this point, you could theoretically start your game. The problem is, it would only last for a millisecond.

To prove that this is the case, save your file as `your-name_game.py` (replace *`your-name`* with your actual name). Then launch your game with Python:

```
<prompt>&#36;</prompt> <command>python</command>
.&#47;your-name_game.py
```

If you're still using Windows, use this command:

```
<command>Python.exe</command> your-name_game.py
```

Don't expect much, because your game only lasts a millisecond right now. Fix that in the next section.

Looping

Unless told otherwise, a Python script runs once and only once. Computers are very fast these days, so your Python script runs in less than a second.

1. What code block in Scratch makes a script repeat something infinitely?

The forever loop.

To force your game to stay open and active long enough for someone to see it, let alone play it, use a *while* loop. While `main` is True, the game stays open.

During the main loop, use Pygame keywords to detect if keys on the keyboard have been pressed or released.

1. What would be a useful function for your game to have even though there's nothing in your game yet?

A key pressed to quit.

```
'''
Main loop
'''
while main == True:
```

```
for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit(); sys.exit()
            main = False

        if event.type == pygame.KEYUP:
            if event.key == ord('q'):
                pygame.quit()
                sys.exit()
                main = False
```

Also in your main loop, refresh your screen's background.

- If you are using an image:

```
screen.blit(backdrop, backdropRect)
```

- If you are just using a colour for the background:

```
screen.fill(black)
```

Finally, tell Pygame to refresh everything on the screen, and advance the game's clock.

```
pygame.display.flip()
clock.tick(fps)
```

Save your file, and run it again to see the most boring game ever created.

To quit the game, press q on your keyboard.

Chapter 4. Spawning a player

Every game needs a player, and every player needs a playable character.

Pygame uses sprites, too. In Scratch, you created a new sprite and then chose a costume for the sprite. You do something similar in Python. Before you do, create a directory called `images` alongside of your Python script file. Put the image you want to use for your sprite into the `images` folder.

In Python, when you create an object that you want to appear on screen, you create a *class*.

Near the top of your Python script, add the code to create a player. In the code sample below, the first 3 lines are already in the Python script that you're working on:

```
import pygame
import sys
import os # new code below

class Player(pygame.sprite.Sprite):
    '''
    Spawn a player
    '''
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.images = []
 img = pygame.image.load(os.path.join('images','hero.png')).convert()
 self.images.append(img)
 self.image = self.images[0]
 self.rect  = self.image.get_rect()
```

If you have a walk cycle for your playable character, save each drawing as an individual file called `hero1.png` to `hero8.png` in the `images` folder.

To tell Python to cycle through each file, you must use a loop.

4.1. What kind of loop does Scratch use to repeat something a specific number of times?

A repeat loop.

In Scratch, you used a repeat loop to repeat an action for a specific number of times. In Python, you use a for loop:

```
import pygame
import sys
import os # new code below

class Player(pygame.sprite.Sprite):
    '''
    Spawn a player
```

Spawning a player

```
    '''
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.images = []
        for i in range(1,9):
            img = pygame.image.load(os.path.join('images','hero' + str(i) + '.png')).co
            self.images.append(img)
            self.image = self.images[0]
            self.rect  = self.image.get_rect()
```

Bringing the player in the game world

In Scratch, when you want to show a sprite in your game, you can use a few Scratch blocks.

1. What are some of the Scratch blocks you might use to show a sprite on the stage?

Show, Go to X or Y, Change to costume.

In Python, you must call the Player sprite you created and add it to a sprite group. In this code sample, the first 3 lines are existing code, so add the lines afterwards:

```
screen = pygame.display.set_mode([screenX,screenY])
backdrop = pygame.image.load(os.path.join('images','stage.png')).convert()
backdropRect = screen.get_rect()

# new code below

player = Player()   # spawn player
player.rect.x = 0   # go to x
player.rect.y = 0   # go to y
movingsprites = pygame.sprite.Group()
movingsprites.add(player)
```

Try launching your game to see what happens. Warning: it won't do what you expect.

When you launch your project, the player sprite doesn't spawn. Actually, it spawns, but only for a millisecond.

1. How do you fix something that only happens for a millisecond?

Put it into the main loop.

To make the player spawn for longer than a millisecond, tell Python to draw it once per loop.

Change the bottom clause of your loop to look like this:

```
    screen.blit(backdrop, backdropRect)
    movingsprites.draw(screen) # draw player
    pygame.display.flip()
    clock.tick(fps)
```

Launch your game now. Your player spawns.

Setting the alpha channel

Your sprite probably has a coloured block around it. This is called the *alpha* channel. It's meant to be the colour of invisibility, but Python doesn't know to make it invisible yet.

You can tell Python what colour to make invisible by setting an alpha channel, using RGB values. If you don't know the RGB values your drawing uses as alpha, open your drawing in Krita or Inkscape and fill the empty space around your drawing with a unique colour. Take note of the colour's RGB values and use that in your Python script.

In this example code, *000* is used. That's black, so the sprite in the example code never uses true black for the parts that need to look black. RGB values are very strict, so you can use 000 for alpha and 111 for something very close to black in your actual drawing.

Using alpha requires the addition of two lines in your Sprite creation code. The first line is already in your code. Add the other 2 lines:

```
            img = pygame.image.load(os.path.join('images','hero' + str(i) + '.png')).co
            img.convert_alpha()     # optimise for alpha
            img.set_colorkey(alpha) # set alpha
```

Python doesn't know what to use as alpha, unless you tell it. In the setup area of your code, add some colour definitions. In the following code, the first two lines already exist in your script, so add the last 3 lines:

```
screenX = 960
screenY = 720
alpha = (0,0,0)
black = (1,1,1)
white = (255,255,255)
```

Launch your game to see the results.

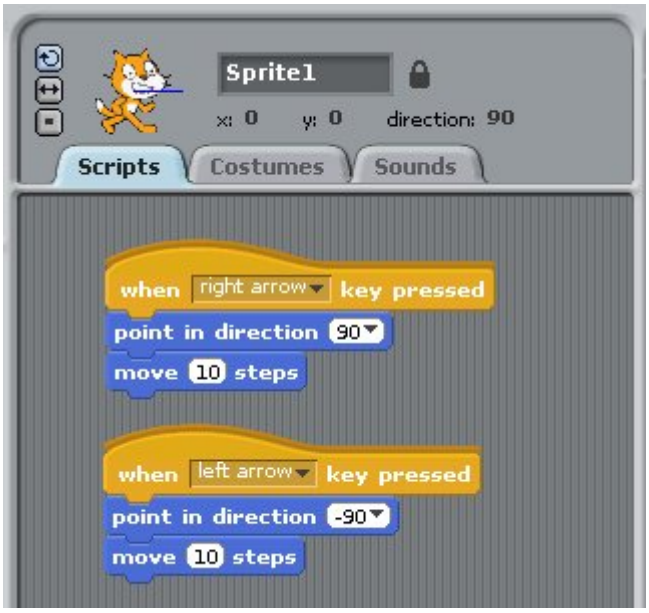
Chapter 5. Moving your sprite

A game isn't much fun if you can't move your character around.

5.1. Which code blocks does Scratch use to provide movement controls to a sprite?

When key is pressed, point in direction, move 10 steps.

Setting up the keys in Pygame to control your playable character is similar. In fact, you have already created a key to quit your game; the principle is the same for movement. However, getting your character to move is a little more complex.



Start with the easy part: setting up the controller keys.

Setting up keys for controlling your player sprite

Open your Python game script in a text editor.

1. Where should you put code that needs to run continuously throughout the game?

In the main loop.

To make Python listen for incoming key presses, add this code to the main loop. There's no code to make anything happen yet, so use `print` statements to signal success. This is a common debugging technique.

```
while main == True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit(); sys.exit()
            main = False

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT or event.key == ord('a'):
                print('left')
            if event.key == pygame.K_RIGHT or event.key == ord('d'):
```

```
                print('right')
            if event.key == pygame.K_UP or event.key == ord('w'):
         print('jump')

        if event.type == pygame.KEYUP:
            if event.key == pygame.K_LEFT or event.key == ord('a'):
                print('left stop')
            if event.key == pygame.K_RIGHT or event.key == ord('d'):
                print('right stop')
            if event.key == ord('q'):
                pygame.quit()
                sys.exit()
                main = False
```

Launch your game using Python, and watch the terminal window for output as you press the right, left, and up arrows, or the a, s, and w keys.

```
<prompt>&#36;</prompt> <command>python</command> <arg>./your-name_game.py</arg>
<computeroutput>left</computeroutput>
<computeroutput>left stop</computeroutput>
<computeroutput>right</computeroutput>
<computeroutput>right stop</computeroutput>
<computeroutput>jump</computeroutput>
```

This confirms that Pygame detects key presses correctly. Now it's time to do the hard work of making the sprite actually move.

Coding the player movement function

To make your sprite move, you must create a property for your sprite that represents movement. When your sprite is not moving, this variable is set to 0.

If you are animating your sprite, or decide to animate it in the future, you also must track frames to enable the walk cycle to stay on track.

Create the variables in the Player class. The first two lines are for context, so add the last three:

```
    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.momentumX = 0  # move along X
        self.momentumY = 0  # move along Y
        self.frame     = 0  # count frames
```

With those variables set, it's time to code the actual movement. Since the Player sprite doesn't have to respond to control all the time, its control functions only need to be a small part of what the Player sprite does.

When you want to make an object in Python do something independent of the rest of its code, you place your new code in a *function*. Python functions start with the keyword *def* for *define*.

Make a function in your Player sprite's code to add *some number* of pixels to your sprite's momentum. Don't worry about how many pixels get added yet. That gets decided in later code.

```
    def control(self,x,y):
        '''
        control player movement
        '''
```

```
        self.momentumX += x
        self.momentumY += y
```

It's built into Scratch, but when a sprite moves in Pygame, you have to tell Python to redraw the sprite in its new location, and where that new location is.

Since the Player sprite isn't always moving, the updates only need to be one function within the Player code. Add this function after the `control` function you created earlier.

First, create a variable for your sprite's current position. Set the value of this variable to the sprite's position before it gets moved.

```
    def update(self):
        '''
 Update sprite position
        '''

        currentX = self.rect.x
```

Then create a variable of where you want the sprite to go. Set the value of this variable to its current position plus *some number* of pixels. How many pixels gets set later.

Then set the new location of your sprite to the new calculated position.

```
 nextX = currentX+self.momentumX
        self.rect.x = nextX
```

Do the same thing for the Y position:

```
        currentY = self.rect.y
        nextY = currentY+self.momentumY
        self.rect.y = nextY
```

For animation, advance the animation frames as long as your sprite is moving, and use the corresponding animation frame:

```
        # moving left
        if self.momentumX < 0:
            self.frame += 1
            if self.frame > 3*afps:
                self.frame = 0
            self.image = self.images[self.frame//afps]

        # moving right
        if self.momentumX > 0:
            self.frame += 1
            if self.frame > 3*afps:
                self.frame = 0
            self.image = self.images[self.frame//afps+4]
```

All the code you've just written needs to know how many pixels to add to your sprite's current position. In Scratch, you used the a code block to tell your Sprite how fast to move. In Python, you just set a variable, and then you use that variable when triggering the functions of your Player sprite.

First, create the variable. In this code, the first two lines are for context, so just add the third line to your script:

```
movingsprites = pygame.sprite.Group()
movingsprites.add(player)
movesteps = 10      # how fast to move
```

Now that you have the function and the variable, use your key presses to trigger the function and send the variable to your Sprite.

Do this by replacing the `print` statements in your main loop with the Player sprite's name (player), the function (.control), and how many steps along the X axis and Y axis you want the Sprite to move with each loop.

```
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT or event.key == ord('a'):
                player.control(-movesteps,0)
            if event.key == pygame.K_RIGHT or event.key == ord('d'):
                player.control(movesteps,0)
            if event.key == pygame.K_UP or event.key == ord('w'):
                print('jump')

        if event.type == pygame.KEYUP:
            if event.key == pygame.K_LEFT or event.key == ord('a'):
                player.control(movesteps,0)
            if event.key == pygame.K_RIGHT or event.key == ord('d'):
                player.control(-movesteps,0)
            if event.key == ord('q'):
                pygame.quit()
                sys.exit()
                main = False
```

Try your game now. Warning: it won't do what you expect, yet!

1. Why doesn't your sprite move yet?

The main loop doesn't call the `update` function.

Add code to your main loop to tell Python to update the position of your Sprite. Add the line with the comment:

```
    player.update()  # update player position
    movingsprites.draw(screen)
    pygame.display.flip()
    clock.tick(fps)
```

Chapter 6. Creating platforms

Pygame is excellent at making side-scrolling platformers. It can do more than just side-scrollers, but anything more complex than a side-scroller takes a lot more coding, so it's best to start with a good old fashioned platformer.

A platformer game needs platforms.

In Scratch, if you want to make a platformer, you might just paint platforms onto your stage. But in Python, the platforms themselves are sprites, just like your playable sprite.

There are two major steps in creating the platforms. First, you must code the objects, and then you must map out where you want the objects to appear.

Coding platform objects

1. What Python keyword is used to create an object in Pygame?

A class.

To build a platform object, you create a class called `Platform`. It's a sprite, just like your Player sprite, with many of the same properties.

Your platforms need to know a lot of information about what kind of platform you want, and where it should be in the game world, and what image it should contain. A lot of that information might not even exist yet, depending on how much you have planned out your game, but that's all right. Just as you didn't tell your Player sprite how fast to move until the end of the Movement chapter, you don't actually have to tell the Platforms everything up front.

Near the top of your script, create a new class. The first three lines in the code are for context:

```
import pygame
import sys
import os
## new code below:

class Platform(pygame.sprite.Sprite):
    # x location, y location, img width, img height, img file
    def __init__(self,xloc,yloc,imgw,imgh,img):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.Surface([imgw,imgh])
        self.image.convert_alpha()
        self.image.set_colorkey(alpha)
        self.blockpic = pygame.image.load(img).convert()
        self.rect = self.image.get_rect()
        self.rect.y = yloc
        self.rect.x = xloc

        # paint the image into the blocks
        self.image.blit(self.blockpic,(0,0),(0,0,imgw,imgh))
```

This class creates an object on your screen in *some* X and Y location, with *some* width and height, using *some* image file for texture.

1. If you were to launch your game right now, would you see any platforms? Why or why not?

No, because nothing calls the Platform class yet, and the platforms are not used in the main loop.

The next step is to map out where all of your platforms need to appear.

Mapping your game world

Mapping out your game world is a vital part of level design, and of game programming in general. It involves a lot of maths, too, but nothing too difficult, and Python can help.

Paint your world

The first thing you need to do is create your game world. You can use GIMP, Krita, Inkscape, or any other variety of open source graphics applications to draw your world.

Cut the world into tiles

Once you've done that, you need to cut it up into tiles. You can do that by using your graphic application's *crop* tool. You need at least two tilse:

- Ground: this is the ground level of your game world. It's the part of the screen that keeps your player from falling off the world.
- Platforms: these are elevated objects that your character can climb or jump onto.

Save each tile in your `images` directory, using names that make sense. For instance, call the ground tile `block0.png`, the first platform `block1.png`, the next platform `block2.png`, and so on.

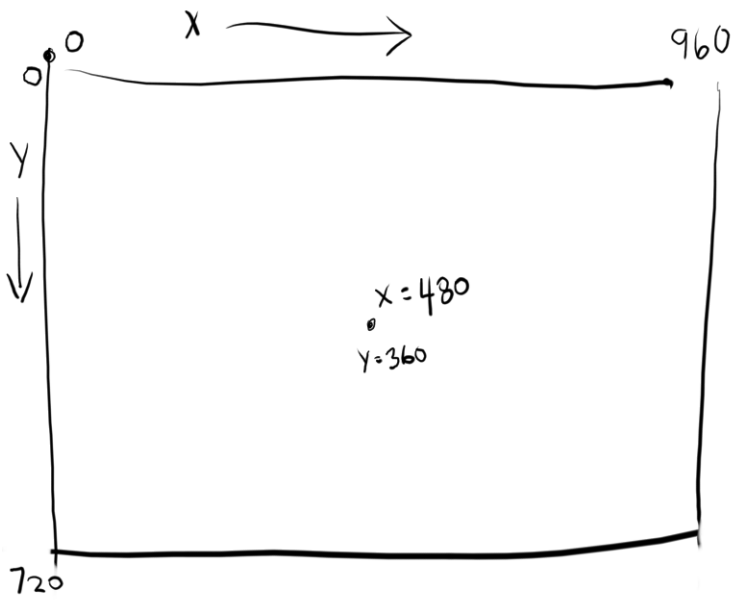
Do maths

Now it's time for maths.

To calculate where everything needs to be placed in your game world, you need to know:

- The size of your screen.
- The size of each tile.
- Where on the screen you want a tile to appear.

In Scratch, your stage had coordinates with 0,0 in the very center of the screen. In Pygame, your game world has coordinates, but 0,0 is in the top left corner of your screen.



To calculate where your ground tile is, for instance, you must fill in the variables listed above. The first one doesn't change much, since it's the size of your game world screen. These are the values you entered for `screenX` and `screenY` in your Python script.

- The size of your screen. Assume, for this example, that your screen is 960 pixels wide and 720 pixels tall.

Next, you must find the dimensions of your tile. Assume that your ground tile is named `block0.png`. To find its dimensions, you can open it in Krita. Click on the Image menu and select Properties. The dimensions are provided at the very top of the Properties window.

Alternately, you can create a simple Python script to tell you the dimensions of an image. Open a new text file and type this code into it:

```
#!/usr/bin/env python3

from PIL import Image
import os.path
import sys

if len(sys.argv) > 1:
    print(sys.argv[1])
else:
    sys.exit('Syntax: identify.py [filename]')

pic = sys.argv[1]
dim = Image.open(pic)
X   = dim.size[0]
Y   = dim.size[1]

print(X,Y)
```

Save the text file as `identify.py`.

To set up this script, you must install an extra set of Python keywords:

```
  <prompt>&#36;</prompt> <command>pip3</command> install Pillow --user
```

Once that has installed, run your script:

```
  <prompt>&#36;</prompt> <command>python ./identify.py</command>
  topics/images/block0.png
  <computeroutput>(1920, 129)</computeroutput>
```

- The size of each tile. In this example, the image size of the ground tile is 1920 pixels wide and 129 high.

Finally, you must calculate where you want to place the tile on your game screen. In the case of the ground, you want the tile to be anchored at the very bottom of the screen.

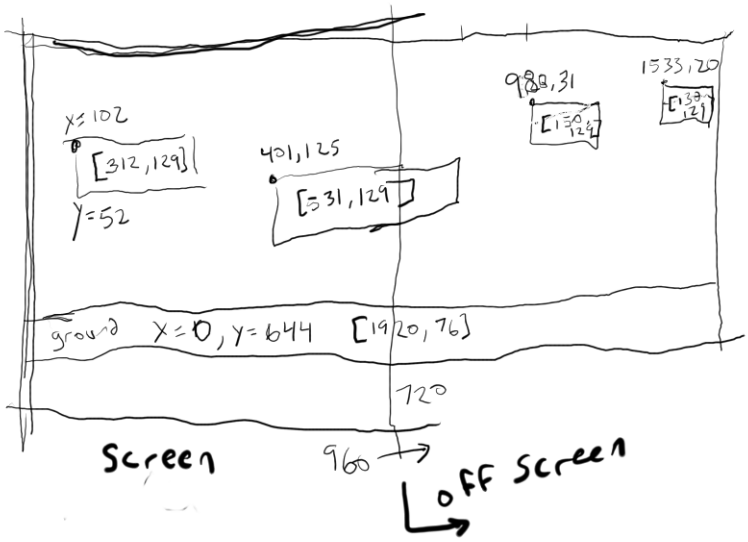
Along the X axis, you want your ground to start at 0. That is, you want the ground to be anchored all the way to the left, and extend from there.

Along the Y axis, you have to do some math. Since you know that the screen is 720 pixels tall, and your ground is 129 pixels tall: subtract 129 from 720 to get the pixel at with you want the ground to *start*. Objects are always "pinned" in Pygame by their top left corner, so if you place your ground tile at 591 pixels, then the ground tile extends downward along the Y axis from there.

- Where on the screen you want a tile to appear. From your calculation, you know that the position you need is 0,591.

You must do this calculation for each tile in your game world. Not every tile will fit into your screen, of course, but that's why they call it a "side-scroller". Anything that doesn't fit in your game screen will be *scrolled* into view as the player moves toward it. So for now, assume that your world is infinitely wide, and jot down the coordinates for each tile.

It might help to make a simple sketch of your layout so that you have all the information you need on one reference sheet.

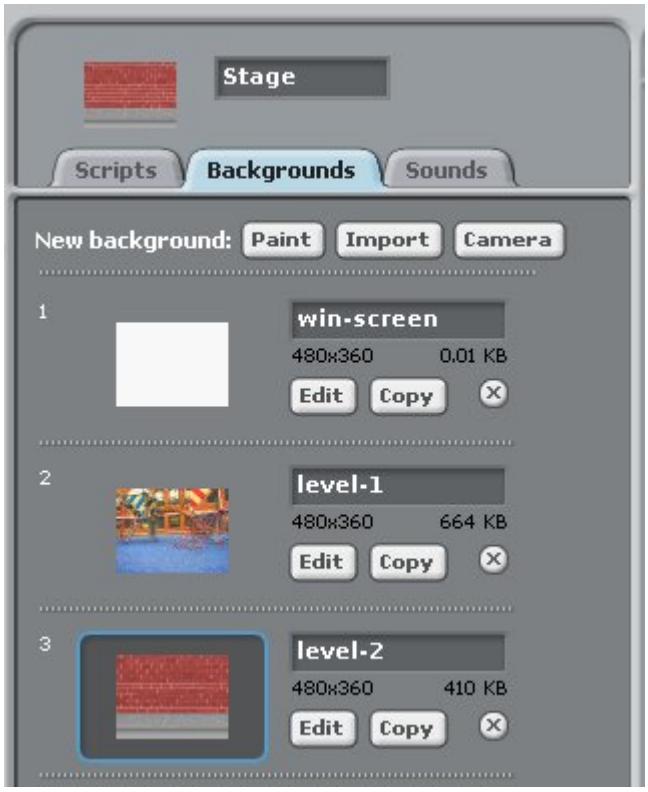


Spawning platforms

Now that you know where you want your Platforms to be, you must tell Python where to place them, and add them to your main loop so they get drawn to your screen.

1. In Scratch, how did you create a new level for a game?

You created a new costume for the stage.



In Pygame, a level is just a collection of assets that get drawn on the screen. Since this is a small game, you can create a Python function to represent Level 1 of your game. In complex games, each Level might be its own script file that gets called by your main file.

Under your classes, but above your setup, create your Level 1 function. In this code, the final 5 lines are for context, so just add the function and its contents, but don't type exactly what you see here. Instead, use the tiles and values that you have obtained to create your own map.
<pragramlisting> def level1(): platform_list = pygame.sprite.Group() block = Platform(0,591,1920,129, os.path.join('images','block0.png')) platform_list.add(block) block = Platform(20,427,173,72, os.path.join('images','block1.png')) platform_list.add(block) block = Platform(60,227,337,72, os.path.join('images','block2.png')) platform_list.add(block) return platform_list ''' Setup ''' alpha = (0,0,0) black = (1,1,1) </pragramlisting>

This function, whatever yours may look like, really only does a few things. First, you creates a sprite group called `platform_list`. Then you create a platform called as a variable called `block`, using your Platform class. Then you add the platform you just created to the group.

You repeat this process as many times as you need in order to build your level.

Finally, you output the `platform_list` into whatever variable called the function.

That means that whenever you need to create a new level, you can make a new function with a whole new set of platforms.

In the setup area of your script file, generate a platform list for Level 1 by calling the function. In this code, only the middle line is new:

```
backdropRect = screen.get_rect()
platform_list = level1()  # set stage to Level 1
player = Player()
```

1. How do you think you would control changing Levels in a complex game?

Use an if statement to determine if a player has achieved a certain goal. If they have, set the platform_list to the next level.

Is your game ready to launch yet?

1. How do we make sure assets get drawn to the screen during a game?

Add it to the main loop.

To add your platform_list to the main loop, add one line to the bottom of your script, within the loop code. Add the middle line:

```
screen.blit(backdrop, backdropRect)
platform_list.draw(screen)  # draw platforms on screen
player.update()
```

Launch your game, and adjust the placement of your platforms as needed. Don't worry that you can't see the platforms that are spawned off-screen; you'll fix that soon.

Chapter 7. Simulating gravity

The real world is full of movement and life. The thing that makes the real world so busy and dynamic is *physics*. Physics is the way matter moves through space. Since a video game world has no matter, it also has no physics, so game programmers have to *simulate* physics.

In terms of a video game, there are basically only two aspects of physics that are important: gravity and collision.

7.1. What is gravity?

Gravity is the tendency for objects with mass to be drawn toward one another. The larger the object, the more gravitational influence it exerts.

For your side-scroller to work, you must simulate gravity so that your player falls toward your game world's Earth.

Adding a gravity function

Remember that your player all ready has a property to determine motion. Use this property to pull the player sprite toward the bottom of the screen.

1. Is the bottom of the screen a low number, like 0, or a high number, like 720?

In Pygame, higher numbers are closer to the bottom edge of the screen.

In the real world, gravity affects everything. In platformers, however, gravity is selective because if you add gravity to your entire game world, all of your platforms would fall to the ground. Instead, add gravity just to your player sprite. Specifically, add a `gravity` function in your Player class.

```
    def gravity(self):
        self.momentumY += 3.2   # how fast player falls

if self.rect.y > 960 and self.momentumY >= 0:
            self.momentumY = 0
```

This is a simple function. First, you set your player in vertical motion whether your player wants to be in motion or not. In other words, you have programmed your player to always be falling; that's gravity.

1. If you launch your game now, will gravity work?

No, because the function hasn't been called in the main loop.

For the gravity function to have an effect, you must call it in your main loop. This way, Python will apply the falling motion to your player once every clock tick.

In this code, add the middle line to your loop:

```
    platform_list.draw(screen)
    player.gravity()   # check gravity
    player.update()
```

Launch your game now to see what happens. Look sharp, because it happens fast: your player falls out of the sky, right off of your game screen.

Your gravity simulation is working, but maybe too well.

Note

As an experiment, you can try changing the rate at which your player falls, but return the gravity to its original value before continuing.

Adding a floor to gravity

The problem with your character falling off the world is that there's no way for your game to detect it. In some games, if a player falls off the world, the sprite is deleted and respawned somewhere new. In other games, the player loses points, or a life. Whatever you want to have happen when a player falls off the world, you have to be able to detect when the player has disappeared off screen.

1. What code block do you use to check for a condition in Scratch?

Forever if, if, or if/then.

In Python, to check for a condition, you can use an *if* statement.

You must check to see if your player is falling, and how far your player has fallen. If your player has fallen so far that it has reached the bottom of the screen, then you can do *something*. To keep things simple, just set the position of the player sprite to 20 pixels above the bottom edge.

Make your gravity function look like this:

```
    def gravity(self):
        self.momentumY += 3.2    # how fast player falls

        if self.rect.y > screenY and self.momentumY >= 0:
            self.momentumY     = 0
            self.rect.y        = screenY-20
```

And then launch your game. Your sprite should bounce at the bottom of the screen.

What your player really needs is a way to fight gravity. The problem with gravity is, you can fight it unless you have something to push off of. In the next chapter, you will add collisions, and the ability to jump, to your game.

Chapter 8. Simulating collisions

In a simple video game, the two most important aspects of physics are gravity and collision. You have all ready implemented gravity in your game, so now it's time to simulate collision.

8.1. What is a collision? Can you think of some everyday collisions that happen?

A collision is when two objects touch. A collision happens when you jump on a trampoline, or when a cricket bat hits a cricket ball, or even when you're just walking to school.

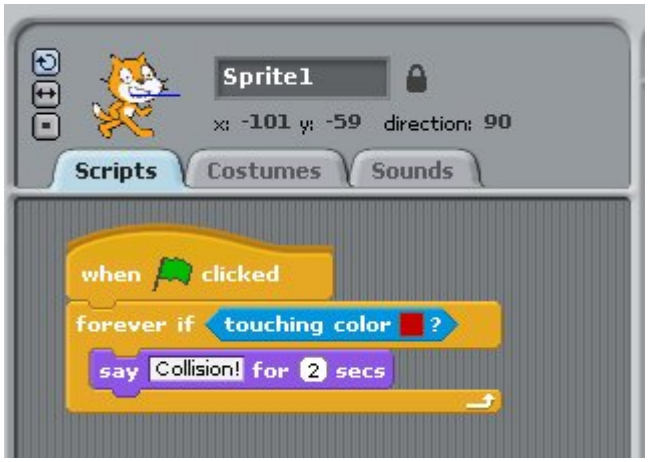
For your side-scroller to work, you simulated gravity so that your player falls toward your game world's Earth just as objects do in real life, but now you must simulate collisions so that your player doesn't fall *through* solid objects.

Making solid objects

Collisions are used for many things. In movement, a collision usually signals a *stop*. That's what a collision against gravity does: if your player hits the ground, then your player stops falling.

1. How does Scratch check for collisions?

Forever if > touching



In Pygame, collisions are detected with the special keyword `spritecollide`. You tell Python to check for any collisions, and if there are collisions, then to react in some way.

In order for Python to be able to check for a collision between two objects, it needs to know what two objects (or group of objects) to check.

In your player's `update` function, add this code:

```
        # collisions
        block_hit_list = pygame.sprite.spritecollide(self, blocker, False)
        if self.momentumX > 0:
            for block in block_hit_list:
                self.rect.y = currentY
                self.rect.x = currentX+9
                self.momentumY     = 0

        if self.momentumY > 0:
            for block in block_hit_list:
                self.rect.y = currentY
```

```
                    self.momentumY     = 0
```

This code checks for collisions between your player sprite and any object in the `platform_list` group. That means that any time your player touches a platform, a collision happens.

When a collision between these objects is detected, you tell Python to set your player's Y position to where ever it is now, and to change it momentum to 0. With these settings, your player is no longer subject to gravity; it is no longer falling. But the moment your player stops touching a platform, gravity kicks back in and pulls the player sprite down.

If you try to launch your game now, it will crash.

```
<prompt>&#36;</prompt> <command>python</command> ./your-name_game.py
<computeroutput>
  Traceback (most recent call last):
File "./your-name_game.py", line 183, in &#60;module&#62;
  player.update() # refresh player
  TypeError: update() takes exactly 2 arguments (1 given)
</computeroutput>
```

Your player sprite's new update function needs to know what the current list of platforms is. Remember, in complex games you might have a different platform list for each level, so you have to feed the current list to the `player.update` function.

Near the bottom of your script, change the `player.update` call to this (change the middle line):

```
    player.gravity()
    player.update(platform_list) # update and collision
    movingsprites.draw(screen)
```

Now launch your game. Your player successfully resists the pull of gravity by colliding with the ground.

Chapter 9. Fighting gravity with jumping

Now that you've worked so hard to simulate gravity, you need to give your player a way to fight against gravity by jumping.

A jump is a temporary reprieve from gravity. For a few moments, you jump *up* instead of fall down, the way gravity is pulling you. But once you hit the peak of your jump, gravity kicks in again and pulls you back down to Earth.

In code, this translates to variables. First, you must establish variables for the player sprite so that Python can track whether or not the sprite is jumping or not. Once the player sprite is jumping, then gravity gets applied to the player sprite again, pulling it back down to the nearest object.

Setting jump state variables

You must add two new variables to your Player class:

- One to track whether your player is jumping or not, determined by whether or not your player sprite is standing on solid ground.
- One to bring the player back down to the ground.

Add these variables to your player class. In the following code, the first 3 lines are for context, so just add the final two:

```
self.momentumX = 0
self.momentumY = 0
self.frame     = 0
# gravity variables here
self.collide_delta = 0
self.jump_delta    = 6
```

The first variable is set to 0 because in its natural state, the player sprite is not in a mid-jump. The other variable is set to 6 to prevent the sprite from bouncing (actually, jumping) when it first lands in the game world. When you've finished this chapter, you can try setting it to 0 to see what happens.

Colliding mid-jump

If you jump across on a trampoline, your jumps are pretty impressive. But what would happen if you jumped into a wall (don't try it to find out)? Your jump, no matter how impressive it started out to be, would end very quickly because you collided with something much larger and much more solid than you.

To mimic that in your video game, you must set the `self.collide_delta` variable to 0 whenever your player sprite collides with something. If `self.collide_delta` is anything other than 0, then your player is jumping, and your player can't jump when your player has hit a wall or the Earth.

In the `update` function of your Player class, modify the code to look like this (you only need to add the two lines with commends by them):

```
            self.rect.x = currentX+9
            # gravity
            self.momentumY     = 0
            self.collide_delta = 0 # stop jumping
```

```
        if self.momentumY > 0:
            for block in block_hit_list:
                self.rect.y = currentY
                # gravity
                self.momentumY     = 0
                self.collide_delta = 0 # stop jumping
```

Jumping

Your simulated gravity always wants your player's Y axis movement to be 0 or more. To create a jump, you write code that sends your player sprite off of solid ground, into the air.

In the `update` function of your Player class, add a temporary reprieve from gravity:

```
        # gravity
        if self.collide_delta < 6 and self.jump_delta < 6:
            self.jump_delta     = 6*2
            self.momentumY     -= 33  # how high to jump

            self.collide_delta += 6
            self.jump_delta    += 6
```

According to this code, a jump sends the player sprite 33 pixels into the air. It's a negative 33 because, remember, a lower number in Pygame means closer to the top of the screen.

The player sprite is prevented from jumping again until it collides with a platform; this prevents mid-air jumps.

Note

Try setting `self.collide_delta` and `self.jump_delta` to 0 for a 100% chance to jump in mid-air.

All that code simulates a jump, but it never gets triggered because you player has never jumped. Your player sprite's `self.jump_delta` was set to 6 initially, and your jump update code only gets triggered when it's less than 6.

To trigger a new setting for the jumping varable, create a `jump` function in your Player class that sets the `self.jump_delta` to less than 6, causing gravity to be temporarily reprieved by sending your player sprite 33 pixels into the air.

```
    def jump(self,platform_list):
        self.jump_delta = 0
```

That's all the `jump` function requires, believe it or not. The rest happens in the `update` function.

There's one final thing to do before jumping works in your game. If you can't think of what it is, try playing your game now to see how jumping works for you.

You probably realise now that nothing in your main loop is actually calling the `jump` function. You made a placeholder key press for it very early on, but right now all the jump key does is print `jump` to the terminal.

Calling the jump function

In your main loop, change the result of the up arrow from printing a debug statement to calling the `jump` function.

Notice that the `jump` function, like the `update` function, needs to know about collisions, so you have to tell it which `platform_list` to use.

```
            if event.key == pygame.K_UP or event.key == ord('w'):
                player.jump(platform_list)
```

If you would rather use the Spacebar for jumping, set the key to `pygame.K_SPACE` instead of pygame.K_UP.

Try your game now. Next up, the world needs to scroll.

Chapter 10. Putting the scroll in side-scroller

Back when you designed your level design layout, you probably had some portion of your level extend past your viewable screen. The ubiquitous solution to that problem is, as the term "side-scroller" suggests, scrolling.

The key to scrolling is to make the platforms *around* the player sprite move when the player sprite gets close to the edge of the screen. This provides the illusion that the screen is a "camera" panning across the game world.

This scrolling trick requires two deadzones at which point your avatar stands still while the world scrolls by. You need a trigger point to go forward, and another if you want to enable your player to go backward.

These two points are just two variables. Set them each about 100 or 200 pixels from each screen edge. Create these variables in your setup section. In the following code, the first two lines are for context, so just add the last two lines:

```
movingsprites.add(player)
movesteps = 10
forwardX  = 600     # when to scroll
backwardX = 230     # when to scroll
```

In the main loop, check to see if your player sprite is at the `forwardX` or `backwardX` scroll point. If so, move all platforms either left or right, depending on whether the world is moving forward or backwards.

In the following code, the final 4 lines of code are for reference:

```
    # scroll the world forward
    if player.rect.x >= forwardX:
        scroll = player.rect.x - forwardX
        player.rect.x = forwardX
        for platform in platform_list:
            platform.rect.x -= scroll

    # scroll the world backward
    if player.rect.x <= backwardX:
        scroll = min(1,(backwardX - player.rect.x))
        player.rect.x = backwardX
        for platform in platform_list:
            platform.rect.x += scroll

    ## scrolling code above
    screen.blit(backdrop, backdropRect)
    platform_list.draw(screen)
    player.gravity()
    player.update(platform_list)
```

In this code, scrolling backwards is enabled, but at a logarithmic rate. In other words, the player sprite can go back, but the scrolling happens slower than going forward. If you don't want that effect, don't use the `min` keyword.

Launch your game and try it out.

Chapter 11. Looting

At this point, you now know all the basics to program video game mechanics. You can build upon these basics to create a fully-functional video game all your own. As an example of how to leverage what you already know for new purposes, this chapter covers how to implement a looting system using what you already know about platforms.

In most video games, you have the opportunity to "loot", or collect treasures and other items within the game world. Loot usually increases your score, or your health, or they provide information leading you to your next quest.

Including loot in your game is similar to programming platforms. Like platforms, loot has no user controls, loot scrolls with the game world, and must check for collisions with the player sprite.

Creating the loot function

Loot is so similar to platforms that you don't even need a Loot class. You can just reuse the Platform class and call the results loot.

Since loot type and placement probably differs from level to level, create a new function called `loot1` just under your `level1` function.

Since loot items are not actual platforms, you must also create a new `loot_list` group, and add loot objects to it. You'll use this group when checking for collisions.

```
def loot1():
    loot_list = pygame.sprite.Group()
    loot = Platform(666,355,92,99, os.path.join('images','loot.png'))
    loot_list.add(loot)
    return loot_list
```

You can add as many loot objects as you like, just remember to add them each to the group. Placement of loot can be just as complex as mapping platforms, so use your level design you used to map out your platforms.

Call the function in the Setup section of your script. In the following code, the first two lines are for context, so add the third:

```
backdropRect  = screen.get_rect()
platform_list = level1()
loot_list     = loot1()  # spawn loot
```

As you know by now, the loot won't get drawn to the screen unless you include it in your main loop. Add the middle line from the following code sample to your loop:

```
    platform_list.draw(screen)
    loot_list.draw(screen) # refresh loot
    player.gravity()
```

Launch your game to see what happens.

Your loot objects are spawned, but they don't do anything when your player runs into them, nor do they scroll when your player runs past them. Fix these issues next.

Scrolling loot

Like platforms, loot has to scroll when the player moves through the game world. The logic is identical to platform scrolling. To scroll the forward, add the last two lines:

```
        for platform in platform_list:
            platform.rect.x -= scroll
        for loot in loot_list:    # scroll loot
            loot.rect.x -= scroll # scroll loot
```

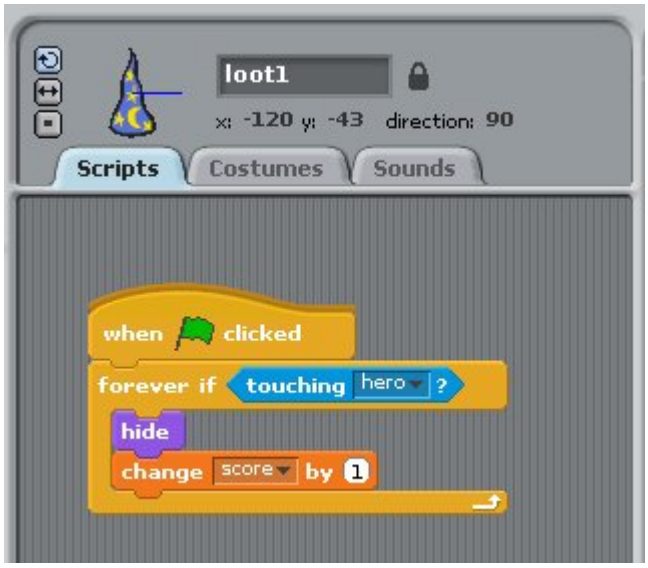
To scroll backward, add the last two lines:

```
        for platform in platform_list:
            platform.rect.x += scroll
        for loot in loot_list:
    loot.rect.x += scroll
```

Launch your game again to see that your loot objects now act like they're *in* the game world instead of just painted on top of it.

Detecting collisions

Like platforms, you can check for collisions between loot and your player. The logic is the same as platform collisions, except that a hit doesn't affect gravity. Instead, a hit causes the loot to disappear and increment the player's score.



When your player touches a loot object, you can remove that object from the `loot_list`. That means that when your main loop redraws all loot items in `loot_list`, it won't redraw that particular object, so it looks like the player has picked up the loot.

Add the following code above the platform collision detection. The last line is for context:

```
        loot_hit_list = pygame.sprite.spritecollide(self, loot_list, False)
        for loot in loot_hit_list:
            loot_list.remove(loot)
            self.score += 1
    print(self.score)

block_hit_list = pygame.sprite.spritecollide(self, platform_list, False)
```

Not only do you remove the loot object from its group when a collision happens, you award your player with a bump in score. You haven't created a score variable yet, so add that to your player's properties. In the following code, first two lines are for context, so just add the score variable:

```
        self.momentumY = 0
        self.frame     = 0
 self.score     = 0
```

As with platforms, you have to tell the `update` function which list to use when detecting collisions.

```
    def update(self,platform_list,loot_list):
```

When calling the `update` function in your main loop, include the `loot_list`:

```
    player.gravity()
    player.update(platform_list,loot_list) # refresh player
```

As you can see, you've learned got all the basics. All you have to do now is use what you know in new ways.

Chapter 12. Going behind enemy lines

Just as you technically know how to create a looting system as long as you know how to do platforms, you also technically all ready know how to implement enemies.

12.1. How do you create a moving enemy in Scratch?

> Forever loops and movement blocks, along with collision blocks so that the enemy moves back and forth within a certain region of the game world.

In Scratch, you probably used a forever-loop and collisions to make an enemy sprite roam some section of the game world, hoping to collide with the hero.

Creating the enemy sprite

Your enemy sprite shares a lot in common with you player sprite. You have to make a class so that enemies can spawn, you have to create an update function so that enemies detect collisions, and you have to make a movement function so your enemy can roam around.

Start with the class. Conceptually, it's mostly the same as your Player class. You set an image or series of images to it and set its starting position.

At the top of the Objects section of your code, create a class called `Enemy` with this code:

```
class Enemy(pygame.sprite.Sprite):
    '''
    Spawn an enemy
    '''
    def __init__(self,x,y,img):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(os.path.join('images',img))
```

```
        self.image.convert_alpha()
        self.image.set_colorkey(alpha)
        self.rect = self.image.get_rect()
        self.rect.x = x
        self.rect.y = y
```

Spawning an enemy

You can make the class useful for spawning more than just one enemy by allowing yourself to tell the class which image to use for the sprite, and where in the world the sprite should appear. This means that you can use this same enemy class to generate any number of enemy sprites anywhere in the game world. All you have to do is make a call to the class, tell it which image to use, along with the X and Y coordinates.

Again, this is similar in principle to spawning a player sprite. In the Setup section of your script, add this code. The first two lines are for context, so just add the final three:

```
movingsprites = pygame.sprite.Group()
movingsprites.add(player)
# enemy code below
enemy   = Enemy(777,531,'owl.png') # spawn enemy
enemies = pygame.sprite.Group()    # create enemy group
enemies.add(enemy)                 # add enemy to group
```

All the same rules apply to the enemy that apply to platforms or loot objects. They need to scroll when the player moves, and they need to be updated in the main loop.

In your main loop, add these blocks of code. First, use the same rules for scrolling platforms and loot. The first two lines are for context, so just add the final two:

```
        for loot in loot_list:
            loot.rect.x -= scroll
        for enemy in enemies:
            enemy.rect.x -= scroll
```

To scroll in the other direction:

```
        for loot in loot_list:
            loot.rect.x += scroll
        for enemy in enemies:
            enemy.rect.x += scroll
```

And then draw all enemies in the enemy group to the screen. Right now you only have one enemy, but you can add more later if you want. As long as you add an enemy to the enemies group, it will get drawn to the screen during the main loop. The middle line in the new line you need to add:

```
    movingsprites.draw(screen)
    enemies.draw(screen)  # refresh enemies
    pygame.display.flip()
```

Launch your game. Your enemy appears in the game world at whatever X and Y coordinate you chose.

Hitting the enemy

An enemy isn't much of an enemy if they have no effect on the player. It's common for enemies to cause damage when a player collides with them. The logic is exactly the same as loot.

Going behind enemy lines

Since you probably want to track the player's health, the collision check actually happens in the Player class rather than in the Enemy class. You can track the enemy's health, too, if you want. The logic and code is pretty much the same, but for now just track the player's health.

To track player health, you must first establish a variable for the player's health. Add the third line to you Player class:

```
        self.frame     = 0
        self.score     = 0
        self.health    = 10  # track health
```

In the `update` function of your Player class, add this code block in the same area as your loot and platform collision checks:

```
        hit_list = pygame.sprite.spritecollide(self, enemies, False)
        for enemy in hit_list:
            self.health -= 1
            print(self.health)
```

Moving the enemy

An enemy that stands still is useful if you want, for instance, spikes or traps that can harm your player, but it's more of a challenge if the enemies move around a little.

Unlike a player sprite, the enemy sprite is not controlled by the user. Its movements must be automated.

How do you get an enemy to move back and forth within the game world when the game world scrolls whenever the player moves?

You tell your enemy sprite to take, for example, 10 paces to the right, and then 10 paces to the left. An enemy sprite can't count, so you have to create a variable to keep track of how many paces your enemy has moved, and then program your enemy to move either right or left depending on the value of your counting variable.

First, create the counter variable in your Enemy class. Add the last line in this code sample:

```
        self.rect = self.image.get_rect()
        self.rect.x = x
        self.rect.y = y
        self.counter = 0  # counter variabla
```

Next, create a `move` function in your Enemy class. Use an if-else loop to create what is called an *infinite loop*:

- Move right if the counter is 0 to 100.
- Move left if the counter is 100 to 200.
- Reset the counter back to 0 if the counter is greater than 200.

A infinite loop has no ending; it loops forever because nothing in the loop is ever untrue. The counter, in this case, will always be either between 0 and 100 or 100 and 200, so the enemy sprite will walk right to left and right to left forever.

```
    def move(self):
        '''
        enemy movement
        '''
```

```
        if self.counter >= 0 and self.counter <= 100:
            self.rect.x  += 10
        elif self.counter >= 100 and self.counter < 150:
            self.rect.x  -= 10
        else:
            self.counter = 0
            print('reset')

        self.counter += 1
```

Will this code work if you launch your game now?

Of course not. You must call the `move` function from your main loop. Add the middle line from this sample code to your loop:

```
player.update(platform_list,loot_list)
enemy.move()  # move enemy sprite
movingsprites.draw(screen)
```

Launch your game and see if you can avoid your enemy. Then try adding some more enemies. As an exercise, see if you can think of how you can change how far different enemy sprites move.

Chapter 13. Colophon

You now know how to make a complete game in Python with the Pygame libraries. If you want to go farther, then here are the next steps:

1. Start learning Linux. Especially in terms of programming, Linux is the most powerful tool available. Not only can you see all the code making your computer run (because Linux and all of its tools itself is open source), but you have easy access to all the latest developments for graphics and device drivers. Valve (creators of Steam) and Vulkan (formerly OpenGL) use Linux as a base platform for a reason.

   If you're not sure how to get started, buy a book about getting started with Fedora Linux, and step through it. As a quick introduction, see the website Switch to Linux [http://klaatu.multics.org/switch]

2. Practise makes perfect. Go make another platformer. After that, make yet another, but this time challenge yourself to come up with something you don't know how to do yet.

3. When in doubt, read the Python 3 docs [https://docs.python.org/3/] and the Pygame docs [https://www.pygame.org/docs]. They aren't always easy to understand right away, but between reading them and looking online or within your Linux OS for sample code, you'll pick up all kinds of new tricks.

4. When you're comfortable with Pygame, look for what's next. When you're a programmer, you're also a student, always learning new things. You can safely stay with Python, because Python is used by nearly every industry out there, but there's a lot more to Python than just Pygame, so go learn it. Write your own Python libraries. Learn PyQt, and Numpy, and Cython. Move forward!

About this book

This book was written in Docbook 5 [http://docbook.org], an XML schema. It was written on Slackware [http://slackware.com] Linux in the Emacs text editor.