Graduating from Scratch to Python: A Student and Teacher Guide

Seth Kenlon

Jess Weichler

Graduating from Scratch to Python: A Student and Teacher Guide

Seth Kenlon
Jess Weichler

Publication date 2017

Copyright © 2017 Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

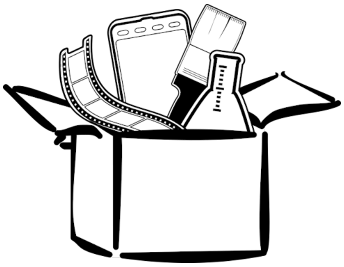


Table of Contents

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, web development, security, video games, and many non-tech industries [https://wiki.python.org/moin/OrganizationsUsingPython] including financial, science, education, and government. 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's also a serious, 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.

The good news is that once you get used to it, it's easy, fun, and it'll make it easier for you to get a job in the industry of your choice. Even if the company you want to work for doesn't use Python, knowing Python makes it a breeze to learn something knew like Lua, C++, and other languages.

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

For teachers

This book can be used as a lesson plan, with each chapter representing one full class session. The principles in the book build upon one another, so start at the beginning, don't rush, and by the end, your students should be able to create code with no further instruction from you.

This course has been used as the lesson plan in a class for children in year 5 and 6 (ages 12-14) by Makerbox [http://makerbox.org.nz] at Rata Studios in New Zealand, with great success. The feedback from its initial run has been incorporated in this edition.

If you don't know Python yourself, provide twice the amount of time for each lesson. Teach the lesson as written, and learn Python along with your students. Tell them that you are also learning, and encourage collaboration in the classroom. Working together on a problem is what open source [https://opensource.com/resources/what-open-source] is all about.

For students

Learning new stuff is hard, especially when you're already really good at something. If you're looking at this book, then you probably know Scratch like the back of your hand. When you start learning Python, you're going to feel like you've taken a huge step backward.

Take a moment to think back to when you were first learning Scratch. You didn't know you had to start with a specific block so that the game would start when the green flag was clicked, you didn't know the difference between an if/then and an forever-if block, or even how to get the cat to move 10 steps.

In learning Python, you're returning to that same place: the very beginning. At first, it will be frustrating that you don't know how to do even simple tasks, and even once you start making things happen, it will take a long time. You don't even get a moving hero sprite until *Moving your sprite* .

So why bother learning Python?

The thing about Python is that once you learn how it works, you can do anything with it. You can make games, you can script 3d games, you can write desktop applications, web sites, and much more. Later in life, when you're looking for a job, knowing Python just might be the reason you get hired, and even if a company doesn't use Python, you can learn new languages easily because you'll already know how programming languages work.

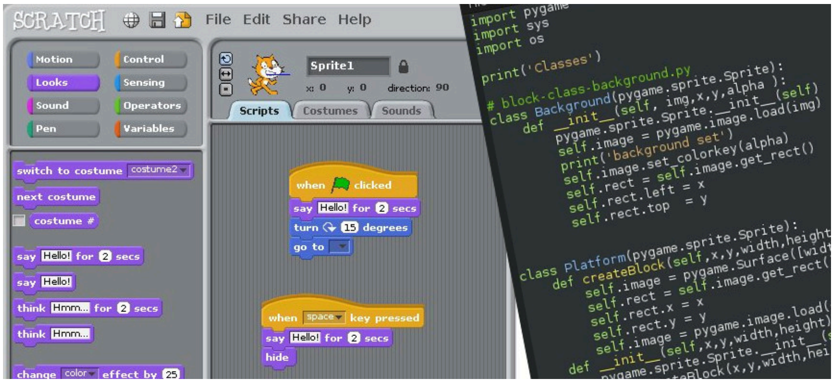
In the video game industry, you hear a lot about applications like Unity and Unreal Engine, and you probably want to learn those as soon as possible. While those are great applications to know, don't rush into them. These kinds of game engines are important, but they require a basic knowledge of programming first. You don't have to learn Python before using a big game engine, but you do have to learn programming.

Python is an easy way to learn programming, and a gateway to more advanced languages.

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 powerful and robust, 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.



```
import pygame
import sys
import os

print('Classes')

# block-class-background.py
class Background(pygame.sprite.Sprite):
    def __init__(self, img,x,y,alpha ):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(img)
        print('background set')
        self.image.set_colorkey(alpha)
        self.rect = self.image.get_rect()
        self.rect.left = x
        self.rect.top  = y


class Platform(pygame.sprite.Sprite):
    def createBlock(self,x,y,width,height
        self.image = pygame.Surface([widt
        self.rect = self.image.get_rect(
        self.rect.x = x
        self.rect.y = y
        self.image = pygame.image.load(
    def __init__(self,x,y,width,height)
        pygame.sprite.Sprite.__init__(s
```

Installing Python

Before learning Python, you may need to install it.

Linux

If you use Linux [https://software.opensuse.org/distributions/leap], Python is already included, but make sure that you have `Python 3`, specifically. To check what version you have installed, open a terminal window and type:

```
python3 -V
```

If that command is not found, then install Python 3 [https://software.opensuse.org/package/python3-base] from your package manager.

Mac OS

Follow the instructions for Linux, above. Macintosh does not have an inbuilt package manager, so if Python 3 is not found, install it from python.org/downloads/mac-osx [https://www.python.org/downloads/mac-osx/].

Warning

Mac OS does have Python 2 installed, but you should learn Python 3.

Windows

At the time of this writing, Microsoft Windows doesn't yet ship with Python. Install it from python.org/downloads/windows [https://www.python.org/downloads/windows]. Be sure to select Add Python to PATH in the install wizard.

Running an IDE

To write programmes in the Python language, all you really need is a text editor, but it's convenient to have what's called an IDE, or an *Integrated Development Environment*. An IDE integrates a text editor with some friendly and helpful Python features.

IDLE 3

Python comes with a basic IDE called IDLE. It has keyword highlighting to help detect typos and a Run button to quickly and easily test code.

Launching IDLE

- On Linux or Mac OS, launch a terminal window and type **idle3**
- On Windows, launch Python 3 from the Start menu.

  If there is no Python in the Start menu, launch the Windows command prompt, called cmd in your start menu, and type **C:\Windows\py.exe**

  If that doesn't work, try reinstalling Python. Be sure to select Add Python to PATH in the install wizard. Refer to docs.python.org/3/using/windows.html [https://docs.python.org/3/using/windows.html] for detailed instructions.

  If that doesn't work, make sure you have Python installed, or just use Linux [https://software.opensuse.org/distributions/leap]. It's free and, as long as you save your Python files to a USB thumbdrive, doesn't even require you to install it to use it.

Ninja-IDE

An excellent Python IDE is Ninja-IDE [http://ninja-ide.org/]. It has keyword highlighting to help detect typos, quotation and parenthesis completion to avoid syntax errors, line numbers (helpful when debugging), indentation markers, and a Run button to quickly and easily test code.

Using Ninja-IDE

Click the Save button to commit your changes.

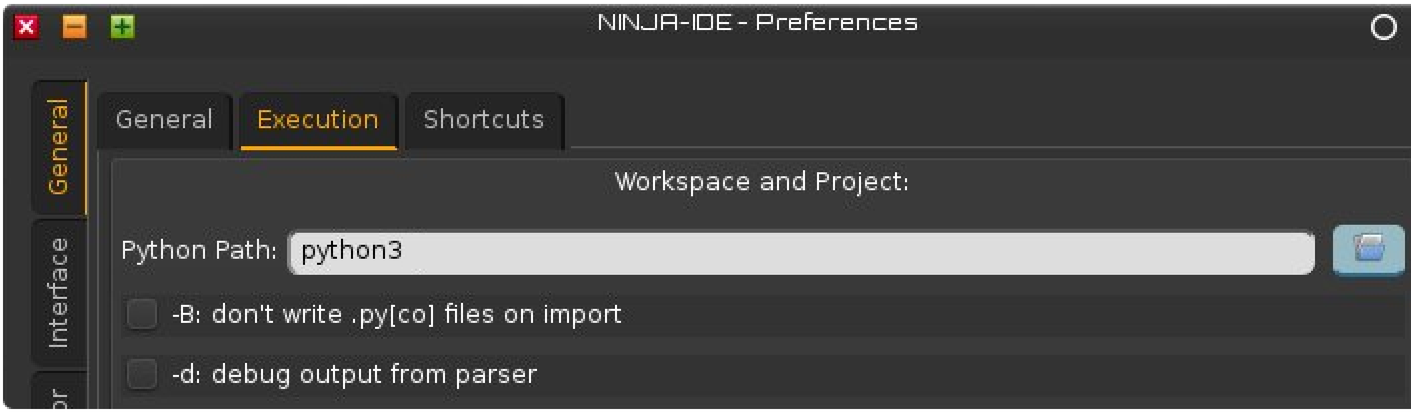
1. Install Ninja-IDE [http://ninja-ide.org/] by downloading its installer from the web site. If you're using Linux, it's easier to just use your package manager.

2. Launch Ninja-IDE.

3. Go to the Edit menu and select Preferences.

   In the Preferences window, click the Execution tab.

   In the Execution tab, change `python` to `python3`.



Workspace and Project:

Python Path: python3

□ -B: don't write .py[co] files on import

□ -d: debug output from parser

Telling Python what to do

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

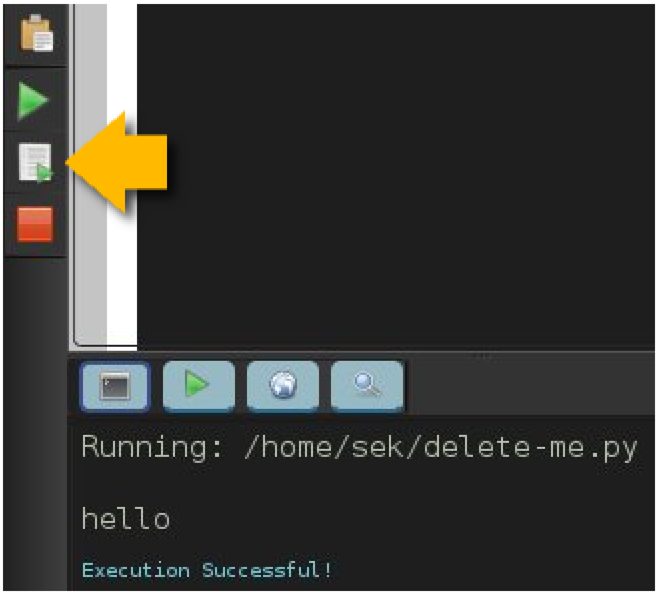
In either IDLE or Ninja, go to the File menu and create a new file. Ninja users: do not create a new project, just a new file.

In your new, empty file, type this into IDLE or Ninja:

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

- If you are using IDLE, go to the Run menu and select Run module option.

- If you are using Ninja, click the Run file button in the left button bar.



```
Running: /home/sek/delete-me.py

hello

Execution Successful!
```

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 at least a hundred 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. Start a new file in IDLE or Ninja and name it `pen.py`.

> ## Warning
>
> Do not call your file `turtle.py`, because `turtle.py` is the name of the file that contains the turtle programme that you are controlling. Naming your file `turtle.py` will confuse Python, because it will think that you want to `import` your own file.

Type this code into your file, and then run it:

```
import turtle
```

Turtle is a fun module to use. It's a lot like the Pen code blocks in Scratch. Try this:

```
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()
```

See what shapes you can draw with the turtle module.

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

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

Try some more complex code:

```

import turtle as t
import time

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

counter=0

while counter < 4:
    t.forward(100)
    t.left(90)
    counter = counter+1

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

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.

Installing Pygame

Before continuing into the exciting world of Pygame, you must install it. Python 3, like Linux, has a *package manager*, called pip3, that lets you install extra modules from the Internet.

Linux and BSD

Linux is a great operating system that powers everything from super computers to space stations to movie studios. It has two important advantages over others: it's open source, so you have access to all the code you run, and it accepts commands directly from the user to the operating system itself. While there are ways to control Linux through desktop icons and menus, it's often faster to just tell Linux exactly what you want it to do. And that's why many programmers prefer Linux: they can tell the computer exactly what to do with just a few quickly-typed commands.

To install Pygame with Python 3 on Linux or BSD, open a terminal window.

```
$ sudo pip3 install pygame
```

Enter your administrative password when prompted and wait for the install to complete.

```
Password: [type your password]
Collecting pygame...

Installing collected packages: pygame
Successfully installed pygame-1.9.3
```

Confirm that it has been installed with this command:

```
$ python3 -c 'import pygame' || echo 'failed'
```

If you get the message `failed` back, then PyGame has not installed correctly. Try running the install command again. Make sure you enter the correct administrative password for your computer.

If you do not have the admin password, install PyGame to just your own home directory:

```
$ pip3 install pygame --user
```

Confirm it again:

```
$ python3 -c 'import pygame' || echo 'failed'
```

As long as everything went well, move on to the next chapter.

Mac

Beneath the non-open source interface of the Mac operating system is UNIX, the same technology that powers Linux and BSD. Broadly, terminal usage on a Mac is limited compared to what you can do in a Linux terminal, but for Python it's mostly the same.

To install Pygame with Python 3 on an Apple Mac computer, open a terminal window. Use the same command as you would on Linux:

```
$ sudo pip3 install pygame
```

Enter your administrative password when prompted and wait for the install to complete.

```
Password: [type your password]
Collecting pygame...

Installing collected packages: pygame
Successfully installed pygame-1.9.3
```

Confirm that it has been installed with this command:

```
$ python3 -c 'import pygame' || echo 'failed'
```

If you get the message `failed` back, then PyGame has not installed correctly. Try running the install command again. Make sure you enter the adminstrative password for your computer.

If you do not have the admin password, install PyGame to just your own home directory:

```
$ pip3 install pygame --user
```

Confirm it again:

```
$ python3 -c 'import pygame' || echo 'failed'
```

As long as everything went well, move on to the next chapter.

Windows

On Microsoft Windows, go to the Start menu and launch run cmd:

Initially, try the same command as Linux uses:

```
pip3 install pygame
```

If that fails, then you have installed Python without adding it to your `PATH`. Use this command instead:

```
C:\Windows\py.exe -m 'pip' install pygame --user
```

As long as everything went well, move on to the next chapter.

Dice game

Pygame is a collection of special Python functions and keywords that make it easy to create games. Before using it, you need to understand the common parts of code.

In this chapter, you're making a text-based game in which the computer rolls a virtual die, and the player rolls a virtual die, and the one with the highest roll wins.

Game prep

Before writing code, it's important to think about what you intend to write, first. The best programmers write simple documentation for their code *before* writing the code itself so that they have a goal to programme toward. Here's how the dice programme might look, if you shipped documentation along with the game:

1. Start the dice game and press **Return** or **Enter** to roll.

2. The results are printed out to your screen.

3. You are prompted to roll again or to quit.

It's a simple game, but the documentation tells you a lot about what you need. Can you think of the components you will need to write this game? If you need to, think about it in terms of Scratch.

Ultimately, you need these components:

- Player: you need a human to play the game.

- AI: the computer must roll die, too, or else the player has no one to win or lose to.

- Random number: a common six-sided die renders a random number between 1 and 6.

- Operator: simple maths can compare one number to another to see which is higher.

- A win or lose message.

- A prompt to play again or quit.

Few programmes start with all of its features implemented, so your first version only implements the basics.

Dice game alpha

A common use of variables in Scratch is for keeping score or tracking health, so you may not have used variables much in Scratch. In Python, however, variables are used a lot. Any time you need your programme to "remember" something, you use a variable. In fact, almost all the information that code works with is stored in variables.

Important terms

Variable

A value subject to change.

For example, in the maths equation `x + 5 = 20`, the variable is *x*, because the letter *x* is a placeholder for a value.

Integer

A number. It can be positive or negative.

For example, 1 and -1 are both integers. So are 14, 21, and even 10947.

Variables in Python are easy to create, and easy to work with. In this initial version of the dice game, two variables are used: `player` and `ai`.

Type the following code into a new text file called `dice_alpha.py`:

```
import random

player = random.randint(1,6)
ai = random.randint(1,6)

if player > ai :
    print("You win")  # notice indentation
else:
    print("You lose")
```

Launch your game to make sure it works.

This basic version of your dice game works pretty well. It accomplishes the basic goals of the game, but it doesn't feel that much like a game. The player never gets to know what they rolled, or what the computer rolled, and the game ends even if the player would rather play again.

This is common of the first version of software (called an *alpha* version). Now that you are confident that you can accomplish the main part of the game, it's time to add to the programme to improve it.

Dice game beta

In this section, you make a series of improvements to your game so that it "feels" more like a game.

Descriptive results

Instead of just telling the player that they did or didn't win, it's more interesting for the player if they know what they rolled. Try making these changes to your code:

```
player = random.randint(1,6)
print("You rolled " + player)

ai = random.randint(1,6)
print("The computer rolled " + ai)
```

If you run the game now, it crashes because Python thinks you're trying to do maths. It thinks you're trying to add the letters `You rolled` and whatever number is currently stored in the `player` variable.

You must tell Python to treat the number in the `player` and `ai` variables as if it were a word in a sentence (called a *string* by programmers) rather than a number in a maths equation (called an *integer*).

Make these changes to your code:

```
player = random.randint(1,6)
print("You rolled " + str(player) )

ai = random.randint(1,6)
print("The computer rolled " + str(ai) )
```

Run your game now to see the result.

Taking time

Computers are very fast. Humans sometimes can be fast, but in games sometimes it's better to build suspense. You can use the `time` function of Python to slow your game down at the suspenseful parts of your game.

```
import random
import time

player = random.randint(1,6)
print("You rolled " + str(player) )

ai = random.randint(1,6)
print("The computer rolls...." )
time.sleep(2)
print("The computer has rolled a " + str(player) )

if player > ai :
    print("You win")  # notice indentation
else:
    print("You lose")
```

Launch your game to test your changes.

Tie detection

If you play your game often enough, you discover that even though your game appears to be working correctly, it actually has a bug in it. See if you can find it, if you haven't already.

Your game doesn't know what to do when the player and the computer roll the same number.

To check if a value is equal to another value, Python uses `==`. That's *two* equal signs, not just one. If you use only one, Python thinks you're trying to create a new variable, but you're actually trying to do maths.

You have used `if` statements in Scratch, and now you have also used an `if-else` statement in Python, but what do you use when you want to have more than just two options? Python has a keyword called `elif`, meaning *else if*. This makes it possible for your code to check to see *if* any one of some results are true, rather than just checking if *one* thing is true.

Modify your code like this:

```
if player > ai :
    print("You win")  # notice indentation
elif player == ai:
    print("Tie game.")
else:
    print("You lose")
```

Launch your game a few times to see if you can tie the computer's roll.

Dice game final

The beta release of your dice game is functional and feels more like a game than the alpha. For the final release, create your first Python `function`.

A function is a group of code that you can call upon as a distinct unit. Functions are important because most applications have a lot of code in them, but not all of that code has to run all at once. Functions make it so you can start an application and control what happens, and when.

Change your code to this:

```
import random
import time

def dice():
    player = random.randint(1,6)
    print("You rolled " + str(player) )
    ai = random.randint(1,6)
    print("The computer rolls...." )
    time.sleep(2)
    print("The computer has rolled a " + str(player) )

    if player > ai :
        print("You win")  # notice indentation
    else:
        print("You lose")

    print("Quit? Y/N")
    cont = input()

    if cont == "Y" or cont == "y":
        exit()
    elif cont == "N" or cont == "n":
        pass
    else:
        print("I did not understand that. Playing again.")
```

In this version of your game, you ask the player if they want to quit the game. If they respond with a `Y` or `y`, then Python's `exit` function is called and the game quits.

More importantly, you create your own function called `dice`. The dice function doesn't run right away. In fact, if you try your game at this stage, it won't crash but it doesn't exactly run, either. To make the `dice` function actually do something, you have to *call it* in your code.

Add this loop to the very bottom of your existing code. The first two lines are only for context, and to emphasize what gets indented and what does not. Pay very close attention to indentation.

```

    else:
        print("I did not understand that. Playing again.")

# main loop
while True:
    print("Press return to roll your die.")
    roll = input()
    dice()
```

What runs first is the `while True` code block. Since `True` is always true by definition, this code block always runs until Python tells it to quit.

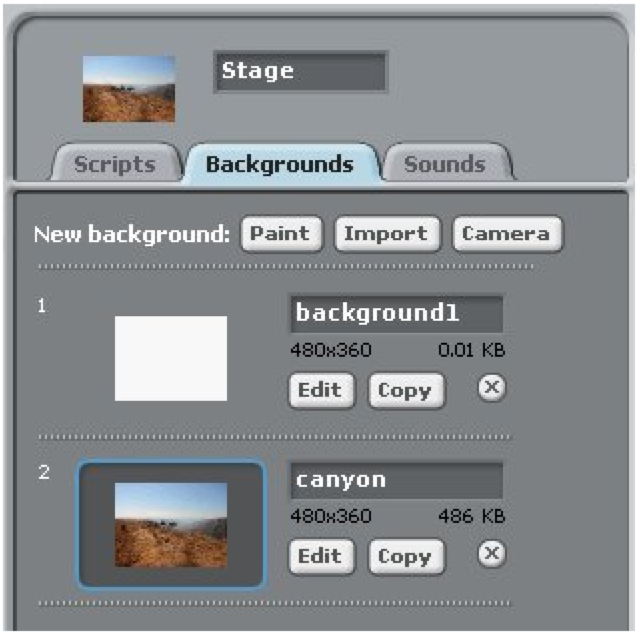
The `while True` code block is a loop. It first prompts the user to start the game, then it calls your `dice` function. That's how the game is started. When the dice function is over, your loop either runs again or it exits, depending on how the player answered the prompt to quit.

Using a loop to run a programme is the most common way to code an application. The loop ensures that the application stays open long enough for the computer user to use functions within the application. Pygame uses a loop, so you'll get to practise with loops as you continue through this book.

Now you know most of the basic parts of a computer programme. It's time to write a game with Pygame.

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.



1 background1 480x360 0.01 KB

2 canyon 480x360 486 KB

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

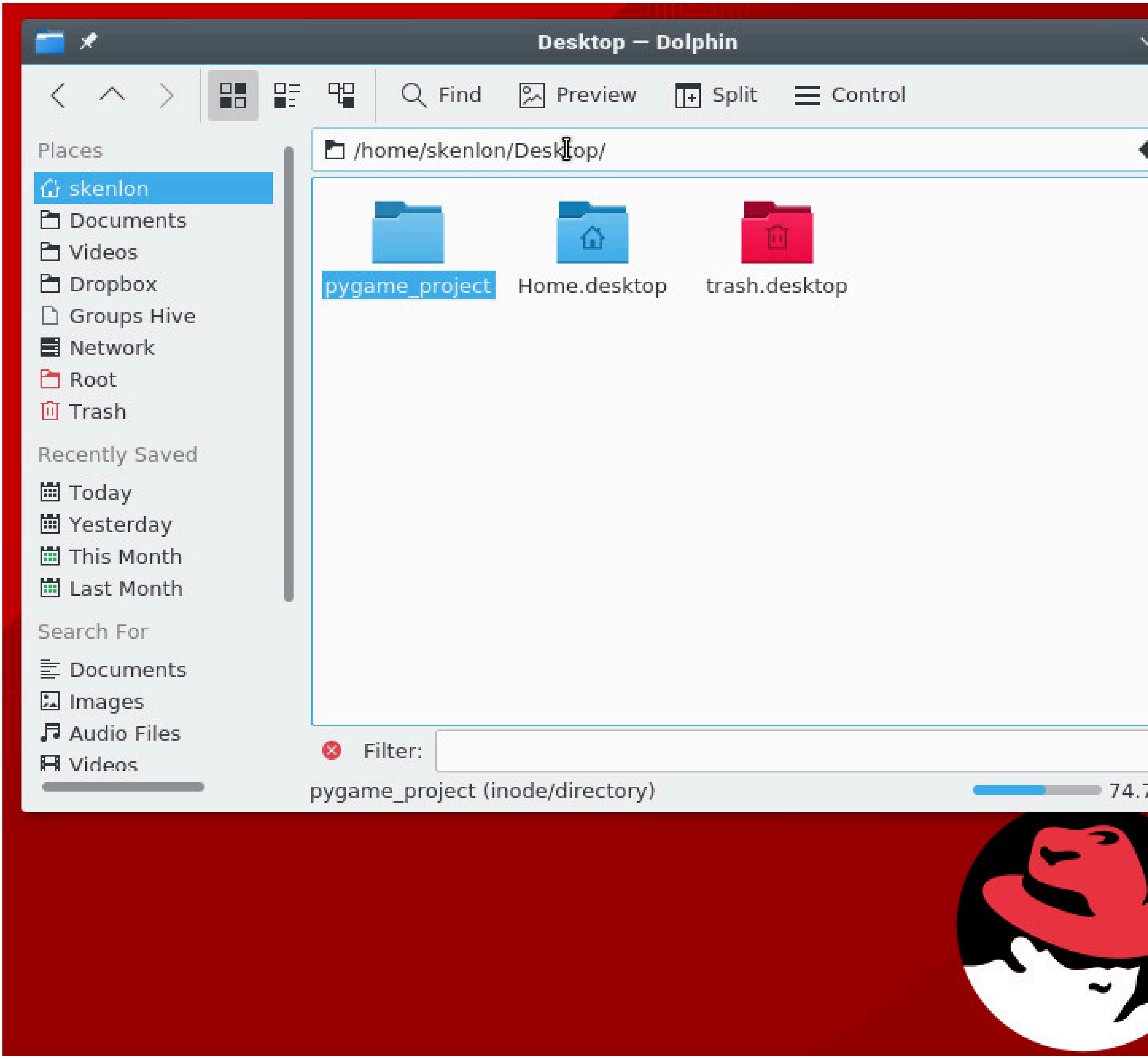
1. Set a background colour.

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

To start a new Pygame project, create a folder on your computer. All of your game files go into this directory. It's vitally important that you keep all the files needed to run your game inside of your project folder.



A Python script starts with the file type, 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 fits on most people's computers.

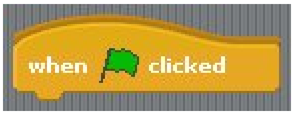
```
'''
Setup
```

```
'''
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.

Add the last line shown in this code sample to yours:

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

Before you continue, open a graphics application and create a background for your game world. Save it as `stage.png` inside of a folder called `images` in your project directory.

There are several free graphics applications you can use.

Open source graphics applications

Krita A professional-level paint materials emulator. Beautiful images can be made with Krita [http://krita.org].

Pinta Pinta [https://pinta-project.com/pintaproject/pinta/releases] is a basic paint application. Easy to learn.

Inkscape Inkscape [http://inkscape.org] is a vector graphics application. Draw with shapes, lines, splines, and Bézier curves.

Your graphic doesn't have to be complex, and you can always go back and change it later. Once you have it, add this code in the setup section of your file:

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

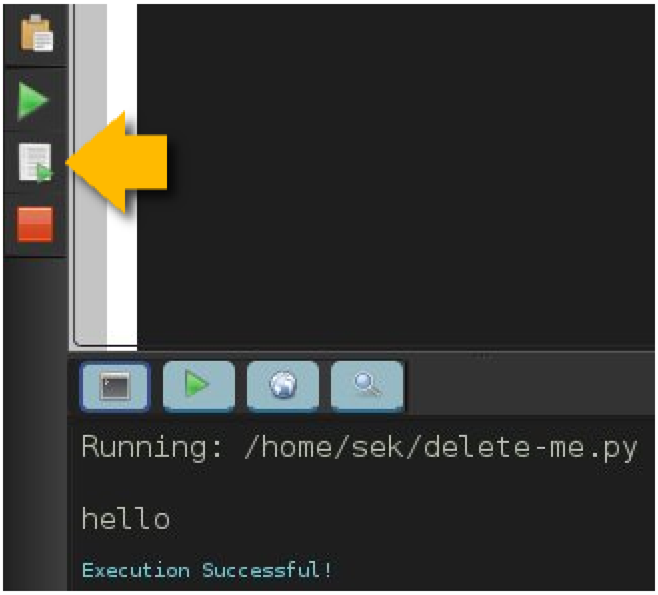
If you're just going to fill the background of your game screen with a colour, all you need is:

```
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.

If you are using IDLE, run your game by selecting Run Module from the Run menu.
<pare> If you are using Ninja, click the Run file button in the left button bar. </pare>



```
Running: /home/sek/delete-me.py

hello

Execution Successful!
```

You can also run a Python script straight from a UNIX terminal or a Windows command prompt.

```
$ python3 ./your-name_game.py
```

If you're using Windows, use this command:

```
py.exe your-name_game.py
```

However you launch it, 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.KEYDOWN:
            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.

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.

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
    '''
    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')).convert()
            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')).convert()
            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.

Creative Commons BY-SA

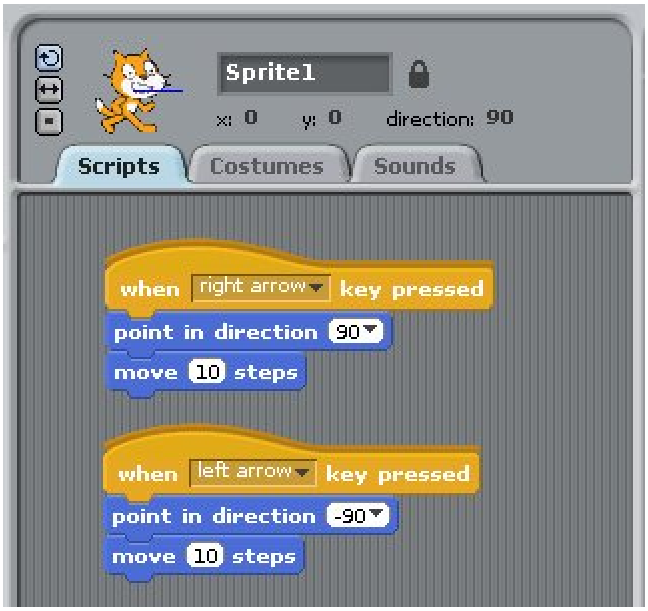
Moving your sprite

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

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.



```
when [right arrow] key pressed
point in direction (90)
move (10) steps

when [left arrow] key pressed
point in direction (-90)
move (10) steps
```

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)
```

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.

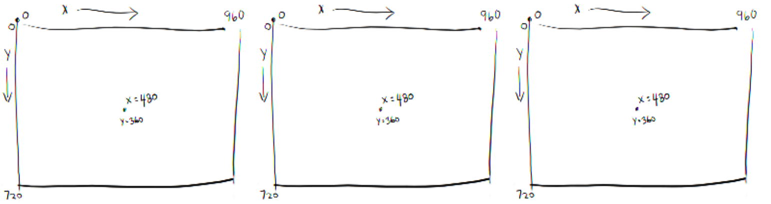
Do maths

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>python3 ./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.

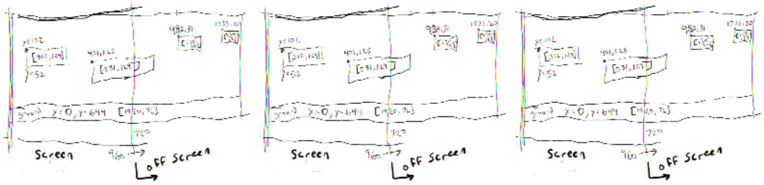
Along the X axis, you want your ground to start at 0. That is, you want the ground to be anchored all the way to the left, and extend from there.

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.

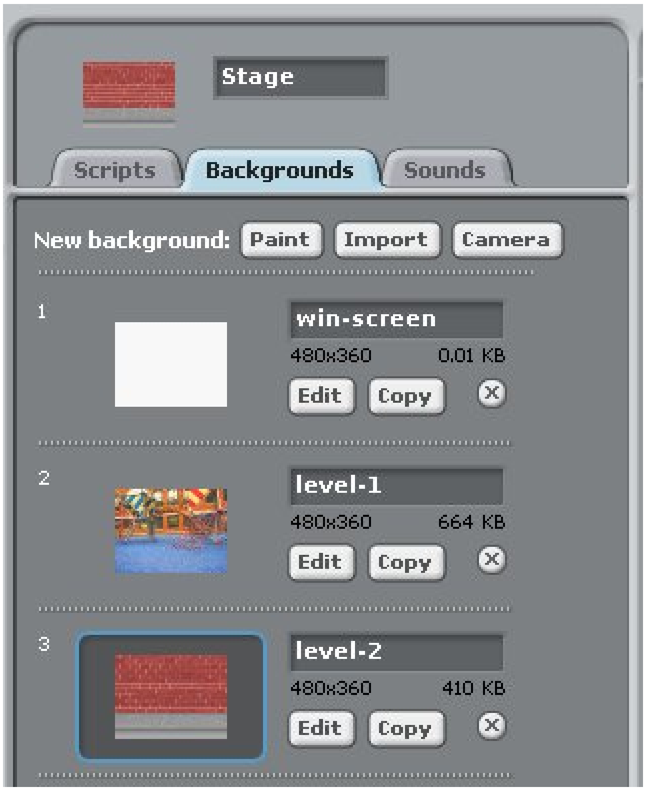


Spawning platforms

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.



1 win-screen 480x360 0.01 KB

2 level-1 480x360 664 KB

3 level-2 480x360 410 KB

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.

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.

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.

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.

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.

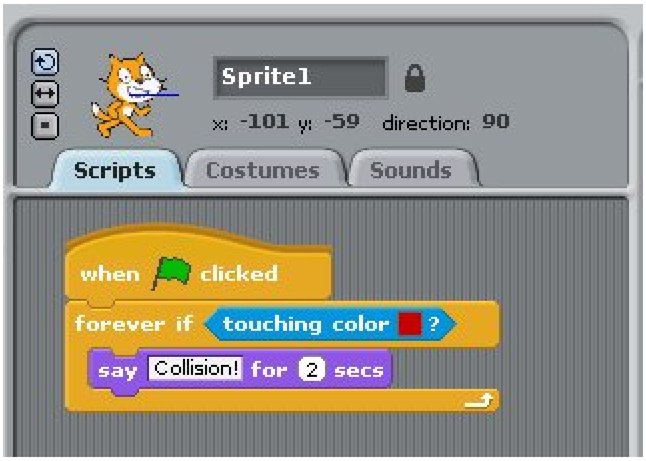
For your side-scroller to work, you simulated gravity so that your player falls toward your game world's Earth just as objects do in real life, but now you must simulate collisions so that your player doesn't fall *through* solid objects.

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



Sprite1

x: -101 y: -59 direction: 90

```
when 🏁 clicked
forever if <touching color 🟥 ?>
    say [Collision!] for (2) secs
```

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, platform_list, 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.

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.

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.

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:

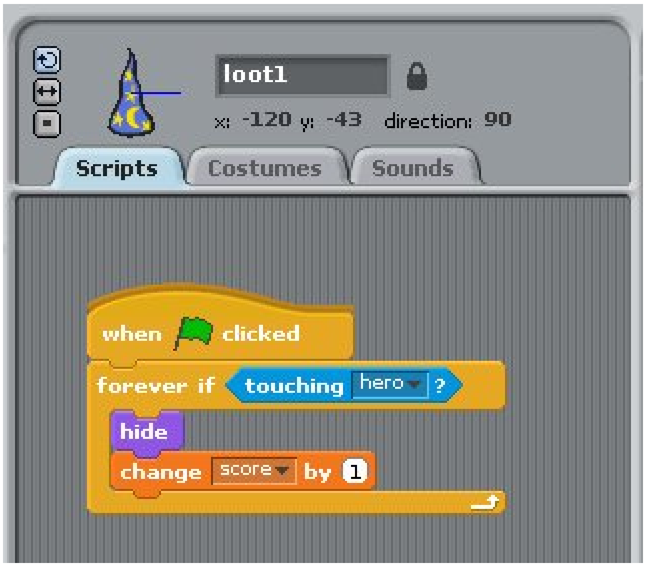
```

        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 clicked
forever if touching hero ?
    hide
    change score by 1
```

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.

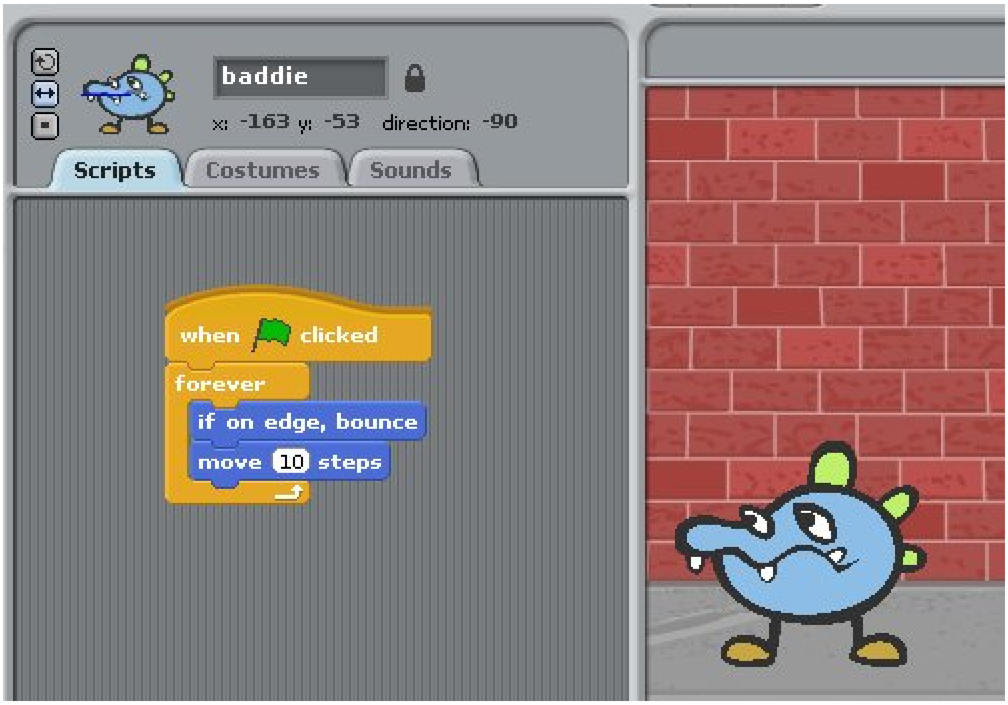
Going behind enemy lines

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.

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.



```
when 🏁 clicked
forever
    if on edge, bounce
    move 10 steps
```

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.

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.

Keeping score

Now that you have loot that your player can collect, there's every reason to keep score so that your player sees just how much loot they've collected.

Conversely, you can also track the player's health so that when they hit one of the enemy's you've just created, it actually means something.

You already have variables tracking both score and health, but it all happens in the background. This chapter teaches you to display these statistics, in a font of your choice, on the game screen, during game play.

Choosing a font

Before displaying text on screen, you must tell PyGame what font to use. Since not everyone in the world has the exact same fonts on their computers, it's important to bundle your font of choice along with your game.

To bundle a font with your game, first create a new directory in your game folder, right along with the directory you created for your images. Call it `fonts`.

Even though several fonts come with your computer, it's not legal to give those fonts away. It seems strange, but that's how the law works. If you want to ship a font with your game, you must find an open source or Creative Commons font that gives you permission to give the font away along with your game.

There are several sites that specialise in free and legal fonts:

- fontlibrary.org [https://fontlibrary.org/]

- https://www.fontsquirrel.com/

- League of Moveable Type [https://www.theleagueofmoveabletype.com/]

When you find a font that you like, download it. Extract the ZIP or tar file, and move the `.ttf` or `.otf` file into the `fonts` folder in your game directory.

Note

You aren't installing the font to your computer. You're just placing it in the `fonts` folder of your game so that Pygame can use it. You can install the font to your computer if you want, but it's not necessary. The imortant thing is to have it in your game directory.

If the font file has a complicated name with spaces or special characters, just rename it. The filename is completely arbitrary, and the simpler it is, the easier it is for you to type into your code.

Rename your font file to `score.ttf` or `score.otf`, depending on what kind of font you downloaded.

Initializing fonts in Pygame

Fonts require a special module within Pygame, so import it along with all the other modules you import at the top of your code:

```
import pygame
import sys
import os
import pygame.freetype # new
```

Just as Pygame itself needs to be initialized in your code, so does its font subsystem. In the Setup section of your game, insert a line to start the font subsystem:

```
pygame.init()
pygame.font.init() # start freetype
main = True
```

And then tell Pygame a little about your font, such as where it is located, and what size it should be. Then create a new variable called `myfont` to serve as your font in the game. All of this also appears in the Setup section:

```
font_path =
 os.path.join(os.path.dirname(os.path.realpath(__file__)),"fonts","amazdoom.ttf")
font_size = 64
myfont    = pygame.font.Font(font_path, font_size)
```

Displaying text in Pygame

Now that you've set the font, you need a function to call to draw the font onto the screen. This is the same principle you use to draw the background and platforms in your game.

First, create a function, and use the `myfont` object to create some text, setting the colour to some RGB value. This must be a global function; it does not exist in a class.

```
def stats(score,health):
    text_score  = myfont.render("Score:"+str(score),  1,(11,11,155))
    text_health = myfont.render("Health:"+str(health), 1,(155,11,11 ))
```

Then blit the text as graphics onto your game screen. The coordinates are the x and y values of where the text should be drawn. Keep in mind the size of your game screen.

```
    screen.blit(text_score, (4, 4))
    screen.blit(text_health, (4, 72))
```

Of course, nothing happens in your game if it's not in the Main loop:

```

    enemies.draw(screen)
    stats(player.score,player.health) # draw text
    pygame.display.flip()
```

Try your game. When the player collects loot, the score goes up. When the player gets hit by an enemy, health goes down. Success.

There is one problem, though. When a player gets hit by an enemy, health goes *way* down, and that's just not fair. Here's how to fix it.

Fixing the health counter

The problem with the current health point system is that health is subtracted for every tick of the Pygame clock that the enemy is touching the player. That means that a slow moving enemy can take a player down to -200 health in just one encounter, and that's not fair. You could, of course, just give your player a starting health score of 10,000 and not worry about it; that would work, and possibly no one would mind. But there is a better way.

Currently, your code detects when a player and an enemy collide. The fix for the health point problem is to detect two things: when the player and enemy collide, and when, once they have collided, they *stop* colliding.

First, in your Player class, create a variable to represent when a Player and enemy have collided:

```

        self.score     = 0
        self.health    = 10
        self.damage    = 0   # player is hit
```

In the update function of your Player class, *remove* this block of code:

```

        for enemy in hit_list:
            self.health -= 1
            print(self.health)
```

And in its place, check for collision as long as the player is not currently being hit:

```

        if self.damage == 0:
            for enemy in hit_list:
                if not self.rect.contains(enemy):
                    self.damage = self.rect.colliderect(enemy)
```

You might see similarity in the block you deleted and the one you just added. They're both doing the same job, but the new code is more complex. Most importantly, the new code only runs if the player is not *currently* being hit already. That means that this code only runs once when a player and enemy collide, instead of 100 times, the way it used to.

The new code uses two new Pygame functions. self.rect.contains checks to see if an enemy is currently within the player's bounding box, and self.rect.colliderect sets your new `self.damage` variable to 1 when it is true, no matter how many times it is true. Now even 3 seconds of getting hit by an enemy still looks like 1 hit to Pygame.

Finally, add another block of code to detect when the player and the enemy are no longer touching. Then and only then, subtract 1 point of health from the player.

```
        if self.damage == 1:
            idx = self.rect.collidelist(hit_list)
            if idx == -1:
                self.damage = 0   # set damage back to 0
                self.health -= 1  # subtract 1 hp
```

Notice that this new code only gets triggered if the player has been hit. That means this code doesn't run while your player is running around your game world, exploring or collecting loot. It only runs when the self.damage variable gets activated.

When the code does run, it uses self.rect.collidelist to see whether or not the player is *still* touching an enemy in your enemy list (collidelist returns a -1 if it detects no collision). Once it is not touching an enemy, it's time to pay the self.damage debt: deactivate the self.damage variable by setting it back to 0, and subtract 1 point of health.

Try your game now.

Score reaction

Now that you have a way for your player to know their score and health, you can make certain events occur when your player reaches certain milestones. For instance, maybe there's a special loot item that restores some health points. And maybe a player who reaches 0 health points has to start back at the beginning of a level.

You can check for these events in your code, and manipulate your game world accordingly. You already know how, so this book doesn't cover it in detail, but do try it out on your own.

Throwing mechanics

Running around avoiding enemies is one thing. Fighting back is another.

It's common in video games to be able to throw something at your enemies, whether it's a ball of fire, an arrow, a bolt of lightning, or whatever else might fit the game.

Unlike anything else you have programmed for your game so far, throwable items have a *time to live*. Once you throw an object, it's expected to travel some distance, and then disappear. If it's an arrow or something similar to that, it may disappear when it passes the edge of the screen. If it's a fire ball or bolt of lightning, it might fizzle out after some amount of time.

That means that each time a throwable item is spawned, a unique measure of its life span must also be spawned. You can try to do that yourself, but as an introduction to the concept this chapter only demonstrates how to throw one item at a time. Only one throwable item may exist at one time. On one hand, this is a limitation to your game, but on the other hand it becomes a game mechanic in itself. Your player won't be able to throw 50 fire balls all at once since you only allow one at a time, so it becomes a challenge for your player to budget when they try to hit an enemy. Of course, behind the scenes, it also keeps your code simple.

Note

If you want to try to enable more throwable items at once, you can try it as a challenge to yourself. However, try doing just one first, and then build on the knowledge you gain from this chapter afterwards.

Creating the Throwable class

By now, you're familiar with the basic `__init__` function when spawning a new object on screen. It's the same function you've used for spawning your player and your enemies.

```
class Throwable(pygame.sprite.Sprite):
    '''
    Spawn a throwable object
    '''
    def __init__(self,x,y,img,throw):
        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
        self.firing = throw
```

The primary difference in this function compared to the `__init__` function of your `Player` class or `Enemy` class is that it has a `self.firing` variable. This variable keeps track of whether or not a throwable object is currently alive on screen, so it stands to reason that when a throwable object is created, the variable gets set to `1`.

Time to live

Next, just as with `Player` and `Enemy`, you need an `update` function so that the throwable object moves on its own once it's thrown into the air toward an enemy.

The easiest way to determine the life span of a throwable object is to detect when it has gone off screen. Which screen edge you monitor depends on the physics of your throwable object.

If your player is throwing something that travels quickly along the horizontal axis, like a crossbow bolt or an arrow, or a very fast magical force, then you want to monitor the horizontal limit of your game screen. This is, of course, defined by `screenX`.

If your player is throwing something that travels vertically, or both horizontally and vertically, then you must monitor the vertical limit of your game screen. This is defined by `screenY`.

This example assumes your throwable object goes a little forward, and also falls to the ground eventually. The object does not bounce off the ground, though, and continues to fall off the screen. You can try different settings to see what fits your game best.

```
    def update(self,screenY):
        '''
        throw physics
        '''
        if self.rect.y < screenY: #vertical axis
            self.rect.x  += 15 #how fast it moves forward
            self.rect.y  += 5  #how fast it falls
        else:
            self.kill()     #remove throwable object
            self.firing = 0 #free up firing slot
```

To make your throwable object move faster, increase the momentum of the `self.rect` values.

If the throwable object is off screen, then the object is destroyed, freeing up the RAM that it had occupied. In addition, `self.firing` is set back to `0` to allow your player to take another shot.

Throwing setup

Just as with your player and enemies, you must create a sprite group in your setup section to hold the throwable object.

Additionally, you must create an inactive throwable object to start the game with. If you don't have a throwable object when the game starts, then the first time a player attempts to throw their weapon will fail.

This example assumes your player starts with fireball as a weapon, so each instance of a throwable object is designated by the `fire` variable. In later levels, if the player acquires new skills, a new variable could be introduced, using a different image but leveraging the same `Throwable` class.

In this block of code, the first two lines are already in your code, so don't type them again:

```
movingsprites = pygame.sprite.Group() #context
movingsprites.add(player)             #context
fire = Throwable(player.rect.x,player.rect.y,'fire.png',0)
firepower = pygame.sprite.Group()
```

Notice that a throwable item starts at the same location as the player. That makes it look like the throwable item is coming from the player. The first time the fireball is generated, a `0` is used so that `self.firing` shows as available.

Throwing in the main loop

As usual, code that doesn't appear in the main loop doesn't get used in the game, so there are a few things in your main loop that you need to add to get your throwable object in your game world.

The first thing you need to do is add player controls. Currently, you have no firepower trigger. As you know, there are two states for a key on a keyboard: the key can be down or the key can be up. For movement, you used both; pressing down starts the player moving, and then the key up signal stops the player. Firing only needs one signal. It's a matter of taste as to which you use to actually trigger your throwable object.

In this code block, the first two lines are for context only:

```
            if event.key == pygame.K_UP or event.key == ord('w'):
                player.jump(platform_list)
            if event.key == pygame.K_SPACE:
                if not fire.firing:
                    fire = Throwable(player.rect.x,player.rect.y,'fire.png',1)
                    firepower.add(fire)
```

You can put this code in the `KEYDOWN` or the `KEYUP` section, depending on when you want your throwable item to be released.

Notice that, unlike the fireball created in your setup section, you use a `1` to set the `self.firing` as unavailable.

Finally, you must update and draw your throwable object. The order of this matters, so put this code between your existing `enemy.move` and `movingsprites.draw` lines.

```
    enemy.move()               #context

    if fire.firing:
        fire.update(screenY)
        firepower.draw(screen)
    movingsprites.draw(screen) #context
    enemy_list.draw(screen)       #context
```

Notice that these updates are only performed if the `self.firing` variable is set to 1. If it is set to 0, then `fire.firing` is not true, and the updates are skipped. If you tried to do these updates no matter what, your game would crash because there wouldn't be a `fire` object to update or draw.

Launch your game and try to throw your weapon.

Detecting collisions

If you played your game with the new throwing mechanic, you probably noticed that while throwing objects does work, it doesn't have any affect on your foes.

The reason for this is that your enemies do not check for a collision. An enemy can be hit by your throwable object and never know about it.

You've already done collision detection in your Player class, and this is very similar. In your Enemy class, add a new `update` function:

```
    def update(self,firepower, enemy_list):
        '''
        detect firepower collision
        '''
        fire_hit_list = pygame.sprite.spritecollide(self,firepower,False)
        for fire in fire_hit_list:
            enemy_list.remove(self)
```

The code is simple. Each enemy object checks to see if it has been hit by the `firepower` sprite group. If it has, then the enemy is removed from the enemy group and disappears.

To integrate that function into your game, call the function in your new firing block in the main loop:

```
    if fire.firing:                                # context
        fire.update(screenY)                       # context
        firepower.draw(screen)                     # context
        enemy_list.update(firepower,enemy_list) # update enemy
```

Try your game again, and clear your world of some baddies.

Note

As a bonus challenge to yourself, try incrementing your player's score whenever an enemy is vanquished.

A better hitbox

Currently, all collision detection is based on the image of your sprites. Sometimes that's all you need, but sometimes you want to change what counts as a collision.

Instead of basing collision on the natural bounds of your sprite's image, you can create a *hitbox* variable for sprites.

To add a custom hitbox to your player, create a hitbox variable in the Player `__init__` function:

```
            self.images.append(img)
            self.image = self.images[0]
            self.rect = self.image.get_rect()
            self.hitbox = pygame.Rect(0,0,58, 202)
```

In this example, 58 and 202 are the width and height of your new hitbox. Change them to fit your sprite, and depending on what size hitbox you want.

The hitbox, to be useful, must follow your sprite every where it goes. To force your hitbox to mirror your sprite, set your hitbox to track your sprite's `rect` in your Player's update function. In this code sample, the last two lines are for context, so only add the first line:

```
            self.hitbox.center = self.rect.center
            currentX = self.rect.x
            self.hitbox.center = self.rect.center
```

Your sprite's rect property has several anchor points you can attach your hitbox to. The code sample above uses the center point, but you can also anchor your hitbox to:

- `self.rect.top` (the top border)
- `self.rect.left` (the left border)
- `self.rect.bottom` (the bottom border)
- `self.rect.right` (the right border)
- `self.rect.topleft` (the top left corner)
- `self.rect.bottomleft` (the bottom left corner)
- `self.rect.topright` (the top right corner)
- `self.rect.bottomright` (the bottom right corner)
- `self.rect.center` (the center point)

Sometimes it's hard to get your hitbox just right. You can temporarily see your hitbox as you work by filling it in with a colour.

Add a definition for your colour in your SETUP section:

```
blue  = (0,0,189)
```

And then in your main loop, fill in the hitbox, just before you update gravity:

```
    pygame.Surface.fill(screen,blue,player.hitbox)
    player.gravity() # context
```

Once you're happy with your hitbox's location, you can comment the line out. But before you do, play your game with your hitbox visible and see if anything is different about how your player sprite interacts with the objects in your world.

You should notice that nothing has changed. Do you know why?

Collision detection

Just because you've created a new hitbox doesn't mean your game is using it. All of your collisions are still based on the `self.rect` of your sprite, because that's how you programmed it in the first place.

Go through your game and find the collisions you are detecting against your player sprite and specify `self.hitbox` as the object you want to scan for collisions.

For example, collision detection with enemy sprites is currently set to:

```
            if self.damage == 0:
                 for enemy in enemy_hit_list:
                     if not self.rect.contains(enemy):
                         self.damage = self.rect.colliderect(enemy)

            if self.damage == 1:
                idx = self.rect.collidelist(enemy_hit_list)
                if idx == -1:
                    self.damage = 0 #set damage back to 0
                    self.score -= 1 #subtract 1 hp
```

But you no longer want to detect collisions for `self.rect`, you want to detect collisions for `self.hitbox`:

```
            if self.damage == 0:
                 for enemy in enemy_hit_list:
                     if not self.hitbox.contains(enemy):
                         self.damage = self.hitbox.colliderect(enemy)
                         print(self.score)

            if self.damage == 1:
                idx = self.hitbox.collidelist(enemy_hit_list)
                if idx == -1:
                    self.damage = 0 #set damage back to 0
                    self.score -= 1 #subtract 1 hp
```

Now launch your game. Notice that an enemy hit doesn't register until the enemy touches the *hitbox*, not just the image of your sprite.

You can make this adjustment to any object that collides with another object. Just remember to create a hitbox for that object, and then to change the collision code to *use* the hitbox.

Adding sound

Pygame provides an easy way to integrate sounds into your game. The `mixer` module of Pygame is capable of playing one ore more sounds on command, and mix those sounds together as needed so that you can have, for instance, background music at the same time as the sounds of your hero collecting loot or jumping over enemies.

The mixer module is easy to integrate into an existing game, so rather than giving you code samples showing you exactly where to put code, this chapter is an overview to the four steps required to get sound in your application.

Start the mixer

"

First, in your SETUP section, start the mixer process. Your code already starts Pygame itself and Pygame fonts, so grouping it together with these is a good idea.

```
pygame.init()
pygame.font.init()
pygame.mixer.init() # add this line
main = True
```

Defining the sounds

Next, you must define the sounds you want to use. This, of course, requires you to actually have sounds on your computer, just as using fonts requires you to have fonts, and using graphics requires you to have graphics.

You also must bundle those sounds along with your game so that anyone playing your game also has the sound files.

To bundle a sound with your game, first create a new directory in your game folder, right along with the directory you created for your images and fonts. Call it `sound`.

```
SOUND = os.path.join('sound')
```

Even though there are plenty of sounds on the internet, it's not necessarily *legal* to download them and then give them away with your game. It seems strange, because so many sounds from famous video games are such a part of popular culture, but that's how the law works. If you want to ship a sound with your game, you must find an open source or Creative Commons sound that gives you permission to give the sound away along with your game.

There are several sites that specialise in free and legal sounds:

- freesound.org [https://freesound.org/] hosts sound effects of all sorts.

- Incompetech.com [http://incompetech.com/music/royalty-free/] hosts background music.

- Open Game Art [https://opengameart.org] hosts sound effects and music.

Some sound files are free to use *only* if you give the composer or sound designer credit. Read the conditions of use carefully before bundling with your game! Musicians and sound designers work just as hard on their sounds as you work on your code, so it's nice to give them credit even when they don't require it.

To give your sound sources credit, list the sounds that you use in a text file called `CREDIT`, and place the text file in your game folder.

When you find a sound that you like, download it. If it comes in a ZIP or tar file, extract it and move the sounds into the `sound` folder in your game directory.

If the sound file has a complicated name with spaces or special characters, rename it. The filename is completely arbitrary, and the simpler it is, the easier it is for you to type into your code.

Note

Most video games use `.ogg` (Ogg Vorbis) sound files because it provides high quality in small file sizes. When you download a sound file, it might be an MP3, or WAVE, FLAC, or any one of several audio formats. To keep your compatibility high and your download size low, convert these to Ogg Vorbis with a tool like fre:ac [https://www.freac.org/index.php/en/downloads-mainmenu-33] or Miro [http://getmiro.com/].

For the purpose of this example, assume you have downloaded a sound file called `zap.ogg`.

In your SETUP section, create a variable representing the sound file you want to use:

```
OUCH = pygame.mixer.Sound(SOUND + 'ouch.ogg')
```

Triggering a sound

To use a sound, all you have to do is call the variable when you want to trigger it. For instance, to trigger the `OUCH` sound effect when your player hits an enemy:

```
for enemy in enemy_hit_list:
    OUCH
    score -= 1
```

You can create sounds for all kinds of actions, such as jumping, collecting loot, throwing, colliding, and whatever else you can imagine.

Appendix A. What is open source?

The term "open source" refers to something people can modify and share because its design is publicly accessible.

The term originated in the context of software development to designate a specific approach to creating computer programs. Today, however, "open source" designates a broader set of values—what we call "the open source way [https://opensource.com/open-source-way]." Open source projects, products, or initiatives embrace and celebrate principles of open exchange, collaborative participation, rapid prototyping, transparency, meritocracy, and community-oriented development.

Open source software is software with source code that anyone can inspect, modify, and enhance.

"Source code" is the part of software that most computer users don't ever see; it's the code computer programmers can manipulate to change how a piece of software—a "program" or "application"—works. Programmers who have access to a computer program's source code can improve that program by adding features to it or fixing parts that don't always work correctly.

Some software has source code that only the person, team, or organization who created it—and maintains exclusive control over it—can modify. People call this kind of software "proprietary" or "closed source" software.

Only the original authors of proprietary software can legally copy, inspect, and alter that software. And in order to use proprietary software, computer users must agree (usually by signing a license displayed the first time they run this software) that they will not do anything with the software that the software's authors have not expressly permitted. Microsoft Office and Adobe Photoshop are examples of proprietary software.

Open source software is different. Its authors make its source code available [https://opensource.com/business/13/5/open-source-your-code] to others who would like to view that code, copy it, learn from it, alter it, or share it. LibreOffice [https://www.libreoffice.org/] and the GNU Image Manipulation Program [http://www.gimp.org/] are examples of open source software.

As they do with proprietary software, users must accept the terms of a license [https://opensource.com/law/13/1/which-open-source-software-license-should-i-use] when they use open source software—but the legal terms of open source licenses differ dramatically from those of proprietary licenses.

Open source licenses affect the way people can use, study, modify, and distribute [https://opensource.com/law/10/10/license-compliance-not-problem-open-source-users] software. In general, open source licenses grant computer users permission to use open source software for any purpose they wish [https://opensource.org/docs/osd]. Some open source licenses—what some people call "copyleft" licenses—stipulate that anyone who releases a modified open source program must also release the source code for that program alongside it. Moreover, some open source licenses [https://opensource.com/law/13/5/does-your-code-need-license] stipulate that anyone who alters and shares a program with others must also share that program's source code without charging a licensing fee for it.

By design, open source software licenses promote collaboration and sharing because they permit other people to make modifications to source code and incorporate those changes into their own projects. They encourage computer programmers to access, view, and modify open source software whenever they like, as long as they let others do the same when they share their work.

No. Open source technology and open source thinking both benefit programmers and non-programmers.

Because early inventors built much of the Internet itself on open source technologies—like the Linux operating system [https://opensource.com/resources/what-is-linux] and the Apache Web server application [http://httpd.apache.org/]—anyone using the Internet today benefits from open source software.

Every time computer users view web pages, check email, chat with friends, stream music online, or play multiplayer video games, their computers, mobile phones, or gaming consoles connect to a global network of computers using open source software to route and transmit their data to the "local" devices they have in front of them. The computers that do all this important work are typically located in faraway places that users don't actually see or can't physically access—which is why some people call these computers "remote computers."

More and more, people rely on remote computers when performing tasks they might otherwise perform on their local devices. For example, they may use online word processing, email management, and image editing software that they don't install and run on their personal computers. Instead, they simply access these programs on remote computers by using a Web browser or mobile phone application. When they do this, they're engaged in "remote computing."

Some people call remote computing "cloud computing," because it involves activities (like storing files, sharing photos, or watching videos) that incorporate not only local devices but also a global network of remote computers that form an "atmosphere" around them.

Cloud computing is an increasingly important aspect of everyday life with Internet-connected devices. Some cloud computing applications, like Google Apps, are proprietary. Others, like ownCloud [https://owncloud.org/] and Nextcloud [https://nextcloud.com/], are open source.

Cloud computing applications run "on top" of additional software that helps them operate smoothly and efficiently, so people will often say that software running "underneath" cloud computing applications acts as a "platform [https://opensource.com/life/14/4/why-open-infrastructure-matters]" for those applications. Cloud computing platforms can be open source or closed source. OpenStack [https://opensource.com/resources/what-is-openstack] is an example of an open source cloud computing platform.

People prefer open source software to proprietary software for a number of reasons, including:

Control.
Many people prefer open source software because they have more control [https://opensource.com/life/13/5/tumblr-open-publishing] over that kind of software. They can examine the code to make sure it's not doing anything they don't want it to do, and they can change parts of it they don't like. Users who aren't programmers also benefit from open source software, because they can use this software for any purpose they wish—not merely the way someone else thinks they should.

Training.
Other people like open source software because it helps them become better programmers [https://opensource.com/life/13/6/learning-program-open-source-way]. Because open source code is publicly accessible, students can easily study it as they learn to make better software. Students can also share their work with others, inviting comment and critique, as they develop their skills. When people discover mistakes in programs' source code, they can share those mistakes with others to help them avoid making those same mistakes themselves.

Security.
Some people prefer open source software because they consider it more secure [https://opensource.com/government/10/9/scap-computer-security-rest-us] and stable than proprietary software. Because anyone can view and modify open source software, someone might spot and correct errors or omissions that a program's original authors might have missed. And because so many programmers can work on a piece of open source software without asking for permission from original authors, they can fix, update, and upgrade open source software more quickly [https://opensource.com/government/13/2/bug-fix-day] than they can proprietary software.

Stability.
Many users prefer open source software to proprietary software for important, long-term projects. Because programmers publicly distribute [https://opensource.com/life/12/9/should-we-develop-open-source-openly] the source code for open source

software, users relying on that software for critical tasks can be sure their tools won't disappear or fall into disrepair if their original creators stop working on them. Additionally, open source software tends to both incorporate and operate according to open standards.

No. This is a common misconception [https://opensource.com/education/12/7/clearing-open-source-misconceptions] about what "open source" implies, and the concept's implications are not only economic [https://opensource.com/open-organization/16/5/appreciating-full-power-open].

Open source software programmers can charge money for the open source software they create or to which they contribute. But in some cases, because an open source license might require them to release their source code when they sell software to others, some programmers find that charging users money for *software services and support* (rather than for the software itself) is more lucrative. This way, their software remains free of charge, and they make money helping others [https://opensource.com/business/14/7/making-your-product-free-and-open-source-crazy-talk] install, use, and troubleshoot it.

While some open source software may be free of charge, skill in programming and troubleshooting open source software can be quite valuable [https://opensource.com/business/16/2/add-open-source-to-your-resume]. Many employers specifically seek to hire programmers with experience [https://opensource.com/business/16/5/2016-open-source-jobs-report] working on open source software.

At Opensource.com, we like to say that we're interested in the ways open source values and principles apply to the world *beyond software*. We like to think of open source as not only a way to develop and license computer software, but also an *attitude*.

Approaching all aspects of life "the open source way [https://opensource.com/open-source-way]" means expressing a willingness to share, collaborating with others in ways that are transparent (so that others can watch and join too), embracing failure as a means of improving, and expecting—even encouraging—everyone else to do the same.

It also means committing to playing an active role in improving the world, which is possible only when everyone has access [https://opensource.com/resources/what-open-access] to the way that world is designed.

The world is full of "source code"—blueprints [https://opensource.com/life/11/6/architecture-open-source-applications-learn-those-you], recipes [https://opensource.com/life/12/6/open-source-like-sharing-recipe], rules [https://opensource.com/life/12/4/day-my-mind-became-open-sourced]—that guide and shape the way we think and act in it. We believe this underlying code (whatever its form) should be open, accessible, and shared—so many people can have a hand in altering it for the better.

Here, we tell stories about the impact of open source values on all areas of life—science [https://opensource.com/resources/open-science], education [https://opensource.com/resources/what-open-education], government [https://opensource.com/resources/open-government], manufacturing [https://opensource.com/resources/what-open-hardware], health, law, and organizational dynamics [https://opensource.com/resources/what-open-organization]. We're a community committed to telling others how the open source way is the *best* way, because a love of open source is just like anything else: it's better when it's shared.

This appendix is licensed under the Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) [http://creativecommons.org/licenses/by-sa/4.0/]. It was taken from Opensource.com [http://opensource.com], which hosts a wealth of resources about open source, including a collection of open source FAQs, how-to guides, and tutorials [https://opensource.com/resources].

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 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.

This book is licensed under the Creative Commons Attribution-ShareAlike International 4.0 [https://creativecommons.org/licenses/by-sa/4.0/] license. This means that you are free to share the book with others, or even modify it. If you modify it, you must use the same license to ensure others have the same freedom as you have. See the full license for a complete description.