A guide to building a video game in Python

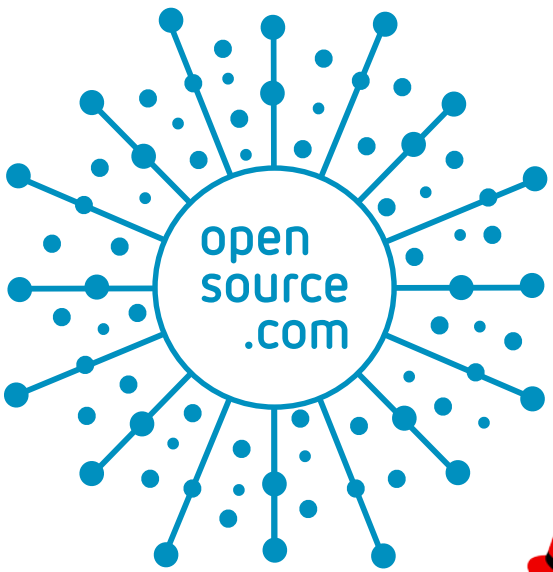
ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: opensource.com/story

Email us: open@opensource.com



Supported by
Red Hat

SETH KENLON

SETH KENLON is an independent multimedia artist, free culture advocate, and UNIX geek. He has worked in the film and computing industry, often at the same time. He is one of the maintainers of the Slackware-based multimedia production project, http://slackermedia.info.

JESS WEICHLER

JESS WEICHLER Jess Weichler is a digital artist using open source software and hardware to create works digitally and in the physical world at CyanideCupcake.com.

She is also an award-winning educator for (and founder of) MakerBox.org.nz an organization that teaches kids of all ages how to use technology, from sewing needles to Arduinos, to make their ideas a reality.

Follow Jess at @jlweich

The code for this booklet can be found here:
https://gitlab.com/makerbox/scratch2python

CHAPTERS

APPENDICES

Learn how to program in Python by building a simple dice game

Python is a good language for young and old, with or without any programming experience.

PYTHON [1] is an all-purpose programming language that can be used to create desktop applications, 3D graphics, video games, and even websites. It's a great first programming language because it can be easy to learn and it's simpler than complex languages like C, C++, or Java. Even so, Python is powerful and robust enough to create advanced applications, and it's used in just about every industry that uses computers. This makes Python a good language for young and old, with or without any programming experience.

Installing Python

Before learning Python, you may need to install it.

Linux: If you use Linux, Python is already included, but make sure that you have Python 3 specifically. To check which version is installed, open a terminal window and type:

```
python --version
```

Should that reveal that you have version 2 installed, or no version at all, try specifying Python 3 instead:

```
python3 --version
```

If that command is not found, then you must install Python 3 from your package manager or software center. Which package manager your Linux distribution uses depends on the distribution. The most common are **dnf** on Fedora and **apt** on Ubuntu. For instance, on Fedora, you type this:

```
sudo dnf install python3
```

MacOS: If you're on a Mac, follow the instructions for Linux to see if you have Python 3 installed. MacOS does not have a built-in package manager, so if Python 3 is not found, install it from python.org/downloads/mac-osx [2]. Although your version of macOS may already have Python 2 installed, you should learn Python 3.

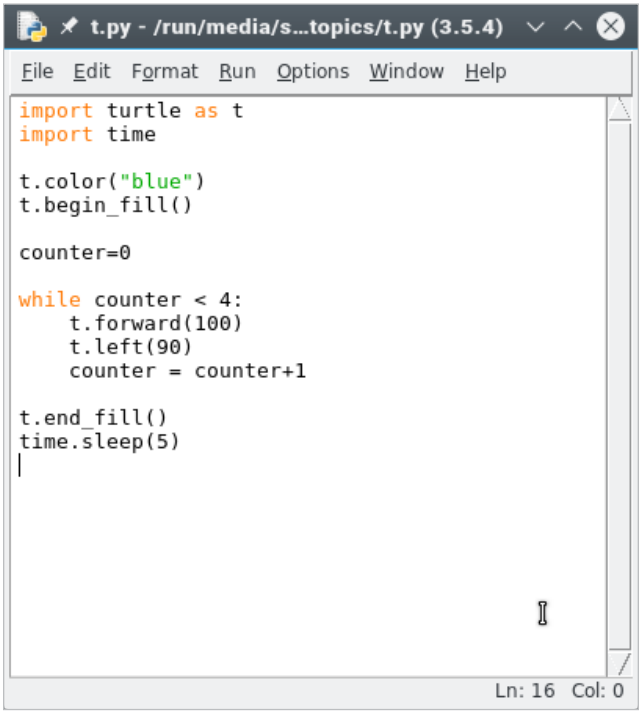
Windows: Microsoft Windows doesn't currently ship with Python. Install it from python.org/downloads/windows [3]. Be sure to select **Add Python to PATH** in the install wizard. Read my article How to Install Python on Windows [4] for instructions specific to Microsoft Windows.

Running an IDE

To write programs in Python, all you really need is a text editor, but it's convenient to have an integrated development environment (IDE). An IDE integrates a text editor with some friendly and helpful Python features. IDLE 3 and PyCharm (Community Edition) are two options among many [5] to consider.

IDLE 3

Python comes with a basic IDE called IDLE.



IDLE

It has keyword highlighting to help detect typing errors, hints for code completion, and a Run button to test code quickly and easily. To use it:

- On Linux or macOS, launch a terminal window and type **idle3**.
- On Windows, launch Python 3 from the Start menu.
  - If you don't see Python in the Start menu, launch the Windows command prompt by typing **cmd** in the 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 [6] for detailed instructions.
  - If that still doesn't work, just use Linux. It's free and, as long as you save your Python files to a USB thumb drive, you don't even have to install it to use it.

PyCharm Community Edition

PyCharm (Community Edition) IDE [7] is an excellent open source Python IDE. 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 test code quickly and easily.

To use it:

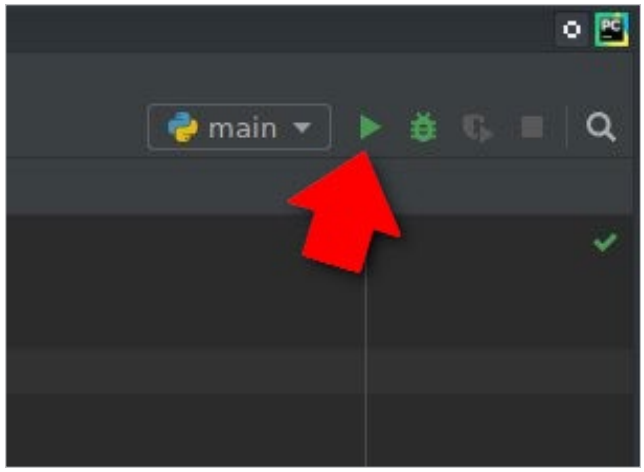
1. Install PyCharm (Community Edition) IDE. On Linux, it's easiest to install it with Flatpak [8]. Alternatively, download [9] the correct installer version from PyCharm's website and install it manually [10]. On MacOS or Windows, download and run the installer from the PyCharm website [11].
2. Launch PyCharm.
3. Create a new project.

Telling Python what to do

Keywords tell Python what you want it to do. In your new project file, type this into your IDE:

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

- If you are using IDLE, go to the Run menu and select Run module option.
- If you are using PyCharm, click the Run File button in the left button bar.



opensource.com

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

That's not very exciting, though. At its core, Python has access to only basic keywords, like **print**, **help**, basic math functions, and so on.

You can use the **import** keyword to load more keywords.

Turtle is a fun module to use. Type this code into your file (replacing the old code), and then run it:

```
import turtle

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?

Advanced turtle

You can try some more complex code for similar results. Instead of hand-coding every line and every turn, you can use a **while loop**, telling Python to do this four times: draw a line and then turn. Python is able to keep track of how many times it's performed these actions with a *variable* called **counter**. You'll learn more about variables soon, but for now see if you can tell how the **counter** and **while loop** interact.

```
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(2)
```

Once you have run your script, it's time to explore an even better module.

Learning Python by making a game

To learn more about how Python works and prepare for more advanced programming with graphics, let's focus on game logic. In this tutorial, we'll also learn a bit about how computer programs are structured by making a text-based game in which the computer and the player roll a virtual die, and the one with the highest roll wins.

Planning your game

Before writing code, it's important to think about what you intend to write. Many programmers write simple documentation [12] *before* they begin writing code, so they have a goal to program toward. Here's how the dice program 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 to do. For example, it tells you that you need the following components to write this game:

- Player: You need a human to play the game.
- AI: The computer must roll a 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 math can compare one number to another to see which is higher.
- A win or lose message.
- A prompt to play again or quit.

Making dice game alpha

Few programs start with all of their features, so the first version will only implement the basics. First a couple of definitions:

A **variable** is a value that is subject to change, and they are used a lot in Python. Whenever you need your program to "remember" something, you use a variable. In fact, almost all the information that code works with is stored in variables. For example, in the math equation **x + 5 = 20**, the variable is *x*, because the letter *x* is a placeholder for a value.

An **integer** is a number; it can be positive or negative. For example, 1 and -1 are both integers. So are 14, 21, and even 10,947.

Variables in Python are easy to create and easy to work with. This initial version of the dice game uses two variables: **player** and **ai**.

Type the following code into a new project called **dice_alpha**:

```
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 much like a game. The player never knows what they rolled or what the computer rolled, and the game ends even if the player would like to play again.

This is common in the first version of software (called an alpha version). Now that you are confident that you can accomplish the main part (rolling a die), it's time to add to the program.

Improving the game

In this second version (called a beta) of your game, a few improvements will make it feel more like a game.

1. Describe the results

Instead of just telling players whether they did or didn't win, it's more interesting 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 will crash because Python thinks you're trying to do math. 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 numbers in the player and ai variables as if they were a word in a sentence (a string) rather than a number in a math equation (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.

2. Slow it down

Computers are fast. Humans sometimes can be fast, but in games, it’s often better to build suspense. You can use Python’s **time** function to slow your game down during the suspenseful parts.

```
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.

3. Detect ties

If you play your game enough, you’ll discover that even though your game appears to be working correctly, it actually has a bug in it: It doesn’t know what to do when the player and the computer roll the same number.

To check whether 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 math.

When you want to have more than just two options (i.e., win or lose), you can using Python’s keyword **elif**, which means *else if*. This allows your code to check to see whether any one of *some* results are true, rather than just checking whether *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 try to tie the computer’s roll.

Programming the final release

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 collection 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 at once. Functions make it possible to 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(ai) )

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

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

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

This version of the game asks the player whether they want to quit the game after they play. If they respond with a **Y** or **y**, Python’s **exit** function is called and the game quits.

More importantly, you've created 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 bottom of your existing code. The first two lines are only for context and to emphasize what gets indented and what does not. Pay 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()
```

The **while True** code block runs first. Because **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 starts. When the dice function is over, your loop either runs again or it exits, depending on how the player answered the prompt.

Using a loop to run a program 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.

Next steps

Now you know the basics of Python programming. The next article in this series describes how to write a video game with PyGame [13], a module that has more features than turtle, but is also a lot more complex.

Links

[1] https://www.python.org/
[2] https://www.python.org/downloads/mac-osx/
[3] https://www.python.org/downloads/windows
[4] https://opensource.com/article/19/8/how-install-python-windows
[5] https://opensource.com/resources/python/ides
[6] https://docs.python.org/3/using/windows.html
[7] https://www.jetbrains.com/pycharm/download
[8] https://flathub.org/apps/details/com.jetbrains.PyCharm-Community
[9] https://www.jetbrains.com/pycharm/download/#section=linux
[10] https://opensource.com/article/18/1/how-install-apps-linux
[11] https://www.jetbrains.com/pycharm/download
[12] https://opensource.com/article/17/8/doc-driven-development
[13] https://www.pygame.org/news

Build a game framework with Python using the Pygame module

The first part of this series explored Python by creating a simple dice game. Now it's time to make your own game from scratch.

IN MY FIRST ARTICLE in this series, I explained how to use Python to create a simple, text-based dice game. You also used the Turtle module to generate some simple graphics. In this article, you start using a module called Pygame to create a game with graphics.

The Turtle module is included with Python by default. Anyone who's got Python installed also has Turtle. The same is not true for an advanced module like Pygame. Because it's a specialized code library, you must install Pygame yourself. Modern Python development uses the concept of virtual *environments*, which provides your Python code its own space to run in, and also helps manage which code libraries your application uses. This ensures that when you pass your Python application to another user to play, you know exactly what they need to install to make it work.

You can manage your Python virtual environment manually, or you can let your IDE help you. For now, you can let PyCharm do all the work. If you're not using PyCharm, read László Kiss Kollár's article about managing Python packages [1].

Getting started with Pygame

Pygame is a library, or *Python module*. It's a collection of common code that prevents you from having to reinvent the wheel with every new game you write. You've already used the Turtle module, and you can imagine how complex that could have been if you'd had to write the code to create a pen before drawing with it. Pygame offers similar advantages, but for video games.

A video game needs a setting, a world in which it takes place. In Pygame, there are two different ways to create your setting:

- Set a background color
- Set a background image

Either way, your background is only an image or a color. Your video game characters can't interact with things in the background, so don't put anything too important back there. It's just set dressing.

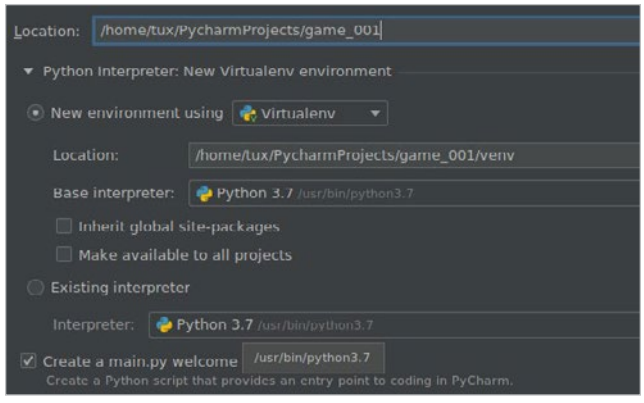
Setting up your first Pygame script

To start a new Python project, you would normally create a new folder on your computer and place all 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.

PyCharm (or whatever IDE you use) can do this for you.

To create a new project in PyCharm, navigate to the **File** menu and select **New Project**. In the window that appears, enter a name for your project (such as **game_001**.) Notice that this project is saved into a special **PycharmProjects** folder in your home directory. This ensures that all the files your game needs stays in one place.

When creating a new project, let PyCharm create a new environment using Virtualenv, and accept all defaults. Enable the option to create a **main.py** file (and to set up some important defaults.)



opensource.com

After you've clicked the **Create** button, a new project appears in your PyCharm window. The project consists of a virtual environment (the **venv** directory listed in the left column) and a demo file called **main.py**.

Delete all the contents of **main.py** so you can enter your own custom code. A Python script starts with the file type, your name, and the license you want to use. Use an open source license so 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 tell Python what modules you want to use. In this code sample, you don't have to type the # character or anything after it on each line. The # character in Python represents a *comment*, notes in code left for yourself and other coders.

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

Notice that PyCharm doesn't understand what Pygame is, and underlines it to mark it as an error. This is because Pygame, unlike sys and os, isn't included with Python by default. You need to include Pygame in your project directory before you can use it successfully in code. To include it, hover your mouse over the word **pygame** until you see a notification popup explaining the error.

Click the **Install package pygame** link in the alert box, and wait for PyCharm to install Pygame into your virtual environment.

Once it's installed, the error disappears.

Without an IDE like PyCharm, you can install Pygame into your virtual environment manually using the **pip** command.

Code sections

Because you'll be working a lot with this script file, it helps to make sections within the file so you know where to put stuff. You do this with block comments, which are comments that are visible only when looking at your source code. Create four blocks in your code. These are all comments that Python ignores, but they're good placeholders for you as you follow along with these lessons. I still use placeholders when I code, because it helps me organize and plan.

```
'''
Variables
'''

# put variables here


'''
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.

There is a way to toggle full-screen mode, the way many modern video games do, but since you're just starting out, keep it simple and just set one size.

```
'''
Variables
'''
worldx = 960
worldy = 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 (using the keyword `init`, for *initialize*) Pygame.

Add these variables:

```
fps   = 40  # frame rate
ani   = 4   # animation cycles
```

Add instructions to start Pygame's internal clock in the Setup section:

```
'''
Setup
'''


clock = pygame.time.Clock()
pygame.init()
```

Now you can set your background.

Setting the background

Before you continue, open a graphics application and create a background for your game world. Save it as `stage.png` inside a folder called `images` in your project directory. If you need a starting point, you can download a set of Creative Commons [2] backgrounds from kenny.nl [3].

There are several free graphic applications you can use to create, resize, or modify graphics for your games.

- Pinta [4] is a basic, easy to learn paint application.
- Krita [5] is a professional-level paint materials emulator that can be used to create beautiful images. If you're very interested in creating art for video games, you can even purchase a series of online game art tutorials [6].
- Inkscape [7] is a vector graphics application. Use it to 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 a background, add this code in the setup section of your file:

```
world = pygame.display.set_mode([worldx,worldy])
backdrop = pygame.image.load(os.path.join('images','stage.png'))
backdropbox = world.get_rect()
```

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

```
world = pygame.display.set_mode([worldx, worldy])
```

You also must define a color to use. In your setup section, create some color definitions using values for red, green, and blue (RGB).

```
'''
Variables
'''


BLUE  = (25, 25, 200)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
```

Look out for errors

PyCharm is strict, and that's pretty typical for programming. Syntax is paramount! As you enter your code into PyCharm, you see warnings and errors. The warnings are yellow and the errors are red.

PyCharm can be over-zealous sometimes, though, so it's usually safe to ignore warnings. You may be violating the "Python style", but these are subtle conventions that you'll learn in time. Your code will still run as expected.

Errors, on the other hand, prevent your code (and sometimes PyCharm) from doing what you expect. For instance, PyCharm is very insistent that there's a newline character at the end of the last line of code, so you **must** press **Enter** or **Return** on your keyboard at the end of your code. If you don't, PyCharm quietly refuses to run your code.

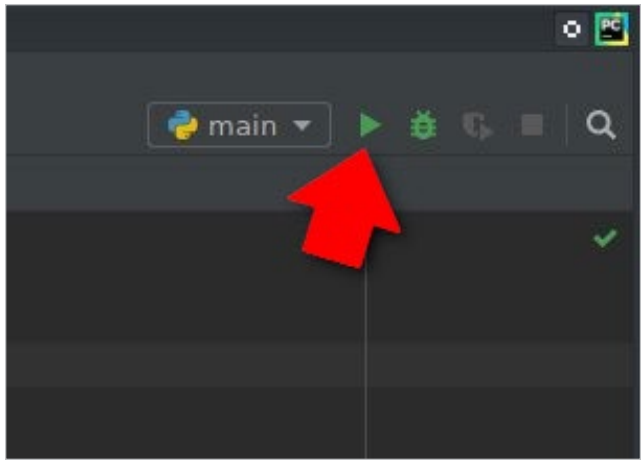
Running the game

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

To prove this, save and then run your game.

If you are using IDLE, run your game by selecting `Run Module` from the Run menu.

If you are using PyCharm, click the `Run file` button in the top right toolbar.



opensource.com

You can also run a Python script straight from a Unix terminal or a Windows command prompt, as long as you're in your virtual environment.

However you launch it, don't expect much, because your game only lasts a few milliseconds right now. You can 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.

To force your game to stay open and active long enough for someone to see it (let alone play it), use a `while` loop.

To make your game remain open, you can set a variable to some value, then tell a `while` loop to keep looping for as long as the variable remains unchanged.

This is often called a "main loop," and you can use the term `main` as your variable. Add this anywhere in your Variables section:

```
main = True
```

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

```
'''
Main loop
'''

while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

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

Be sure to press **Enter** or **Return** after the final line of your code so there's an empty line at the bottom of your file.

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

If you are using an image for the background:

```
world.blit(backdrop, backdropbox)
```

If you are using a color for the background:

```
world.fill(BLUE)
```

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

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

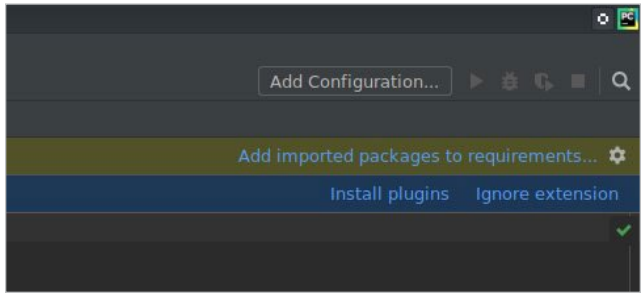
Save your file, and run it again to see the most boring game ever created.

To quit the game, press q on your keyboard.

Freeze your Python environment

PyCharm is managing your code libraries, but your users aren't going to run your game from PyCharm. Just as you save your code file, you also need to preserve your virtual environment.

Go to the **Tools** menu and select **Sync Python Requirements**. This saves your library dependencies to a special file called **requirements.txt**. The first time you sync your requirements, PyCharm prompts you to install a plugin and to add dependencies. Click to accept these offers.



opensource.com

A **requirements.txt** is generated for you, and placed into your project directory.

Code

Here's what your code should look like so far:

```
#!/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/>.

import pygame
import sys
import os

'''
Variables
'''
```

```
worldx = 960
worldy = 720
fps = 40  # frame rate
ani = 4   # animation cycles
main = True

BLUE = (25, 25, 200)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)


'''
Objects
'''

# put Python classes and functions here


'''
Setup
'''

clock = pygame.time.Clock()
pygame.init()
world = pygame.display.set_mode([worldx, worldy])
backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
backdropbox = world.get_rect()


'''
Main Loop
'''
while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

        if event.type == pygame.KEYDOWN:
            if event.key == ord('q'):
                pygame.quit()
            try:
                sys.exit()
            finally:
                main = False
    world.blit(backdrop, backdropbox)
    pygame.display.flip()
    clock.tick(fps)
```

What to do next

In the next article of this series, I'll show you how to add to your currently empty game world, so start creating or finding some graphics to use!

Links

[1] https://opensource.com/article/19/4/managing-python-packages
[2] https://opensource.com/article/20/1/what-creative-commons
[3] https://kenney.nl/assets/background-elements-redux
[4] https://www.pinta-project.com/
[5] http://krita.org/
[6] https://gumroad.com/l/krita-game-art-tutorial-1
[7] http://inkscape.org/

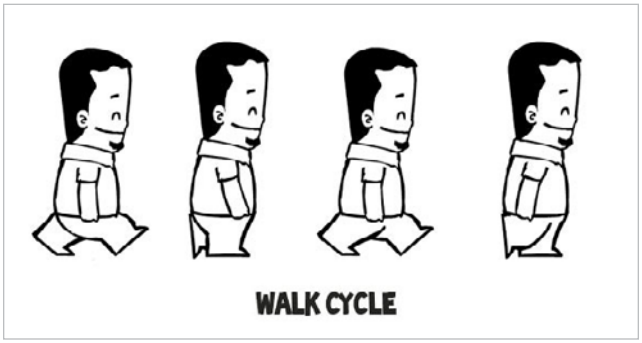
How to add a player to your Python game

Part three of a series on building a game from scratch with Python.

IN THE FIRST ARTICLE of this series, I explained how to use Python to create a simple, text-based dice game. In the second part, I showed you how to build a game from scratch, starting with creating the game's environment [1]. But every game needs a player, and every player needs a playable character, so that's what you'll do next in this third part of the series.

In Pygame, the icon or avatar that a player controls is called a sprite. If you don't have any graphics to use for a player sprite yet, download the walk-0.png, walk-2.png, walk-4.png, and walk-5.png files [2] from the classic open source game Supertux [3] and rename them hero1.png to hero4.png Alternately, you can create something for yourself using Krita [4] or Inkscape [5], or search OpenGameArt [6]. org for other options. Then, if you didn't already do so in the previous article, create a directory called `images` within your Python project directory. Put the images you want to use in your game into the `images` folder.

To make your game truly exciting, you ought to use an animated sprite for your hero. If you're drawing your characters yourself, this means you have to draw more assets, but it makes a big difference. The most common animation is a walk cycle, a series of drawings that make it look like your sprite is walking. The quick and dirty version of a walk cycle requires four drawings.



Ajay Karat, CC BY-SA 3.0

Note: The code samples in this article allow for both a static player sprite and an animated one.

Name your player sprite hero.png. If you're creating an animated sprite for a walk cycle, append a digit after the name, starting with hero1.png. Save you hero image into a directory called images in your Python project directory.

Create a Python class

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, in the Objects section, add the code to create a player. If you're using a static image with no walk cycle, use this code (note that this code goes in the Objects section of your file):

```
'''
Objects
'''


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()
```

This code block creates a virtual "object" for Python to use when referencing your hero sprite. In object-oriented programming, an "object" is referred to as a class. The object template (specifically, `pygame.sprite.Sprite`) is provided by Pygame. That's what makes it possible for you to define an image to represent the player character. If you had to program that from scratch, you'd have to learn a lot more about Python before you could start creating a game, and that's the advantage of using a framework like Pygame.

If you have a walk cycle for your playable character, save each drawing as an individual file called `hero1.png` to `hero4.`

png in your project's images folder. Then use a loop to tell Python to cycle through each file. This is one of the features of object-oriented programming: each class can have tasks assigned exclusively to it, which occurs without affecting the "world" around it. In this case, your player character sprite is programmed to cycle through four different images to create the illusion of walking, and this can happen regardless of what else is happening around it.

```
'''
Objects
'''


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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.images = []
        for i in range(1, 5):
            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()
```

Bring the player into the game world

Now that a Player class exists, you must use it to spawn a player sprite in your game world. If you never call on the Player class, it never runs, and there will be no player. You can test this out by running your game now. The game runs just as well as it did at the end of the previous article, with the exact same results: an empty game world.

To bring a player sprite into your world, you must "call" the Player class to generate a sprite and then add it to a Pygame sprite group. Add these lines to your Setup section:

```
player = Player()   # spawn player
player.rect.x = 0   # go to x
player.rect.y = 0   # go to y
player_list = pygame.sprite.Group()
player_list.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. How do you fix something that only happens for a millisecond? You might recall from the previous article that you need to add something to the *main loop*. To make the player spawn for longer than a millisecond, tell Python to draw it once per loop.

Change the drawing clause of your main loop to look like this:

```
    world.blit(backdrop, backdropbox)
    player_list.draw(world) # draw player
    pygame.display.flip()
    clock.tick(fps)
```

Launch your game now. Your player spawns!

Setting the alpha channel

Depending on how you created your player sprite, it may have a colored block around it. What you are seeing is the space that ought to be occupied by an *alpha channel*. It's meant to be the "color" of invisibility, but Python doesn't know to make it invisible yet. What you are seeing, then, is the space within the bounding box (or "hit box," in modern gaming terms) around the sprite.

You can tell Python what color to make invisible by setting an alpha channel and using RGB values. If you don't know the RGB values your drawing uses as alpha, open your drawing in Pinta or Inkscape and fill the empty space around your drawing with a unique color, like #00ff00 (more or less a "greenscreen green"). Take note of the color's hex value (#00ff00, for greenscreen green) and use that in your Python script as the alpha channel.

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

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

Python doesn't know what to use as alpha unless you tell it.

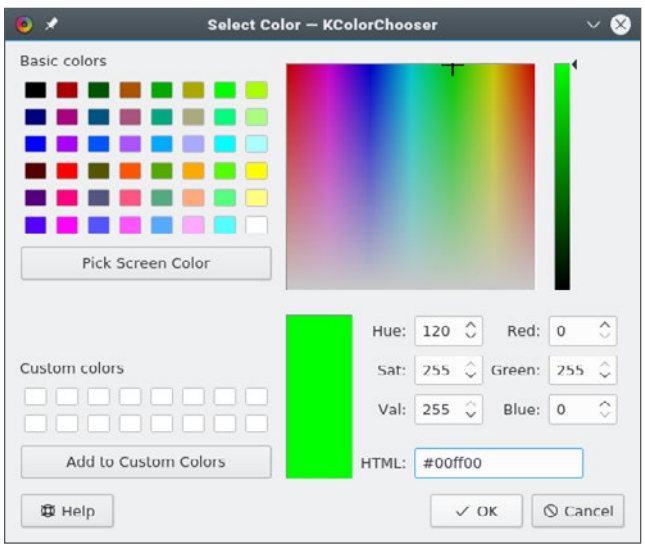
If you believe your image already has an alpha channel, you can try setting a variable ALPHA to 0 or 255, both of which are common places for alpha to be stored. One of those may work, but maybe due to my background in film production, I prefer to explicitly create and set my own alpha channel.

Setting your own alpha

In the Variable section of your code, add this variable definition:

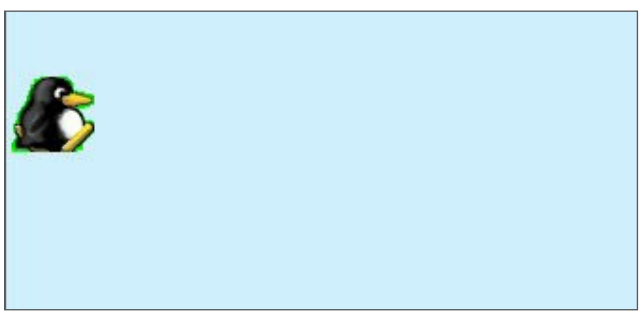
```
ALPHA = (0, 255, 0)
```

In this example code, **0,255,0** is used, which is the same value in RGB as #00ff00 is in hex. You can get all of these color values from a good graphics application like GIMP [7], Krita, or Inkscape. Alternately, you can also detect color values with a good system-wide color chooser, like KColorChooser [8] or ColourPicker [9].



If your graphics application is rendering your sprite's background as some other value, adjust the values of your alpha variable as needed. No matter what you set your alpha value, it will be made "invisible." RGB values are very strict, so if you need to use 000 for alpha, but you need 000 for the black lines of your drawing, just change the lines of your drawing to 111, which is close enough to black that nobody but a computer can tell the difference.

Launch your game to see the results.



If you're having trouble setting an alpha channel for your character, refer to Appendix 3 for detailed instructions on getting it right.

Here's the code in its entirety so far:

```
#!/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/>.
from typing import Tuple

import pygame
import sys
import os

'''
Variables
'''

worldx = 960
worldy = 720
fps = 40  # frame rate
ani = 4  # animation cycles
world = pygame.display.set_mode([worldx, worldy])

BLUE = (25, 25, 200)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
ALPHA = (0, 255, 0)

'''
Objects
'''


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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.images = []
        for i in range(1, 5):
```

```
            img = pygame.image.load(os.path.join('images',
                  'hero' + str(i) + '.png')).convert()
            img.convert_alpha()  # optimise alpha
            img.set_colorkey(ALPHA)  # set alpha
            self.images.append(img)
            self.image = self.images[0]
            self.rect = self.image.get_rect()


'''
Setup
'''

backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
clock = pygame.time.Clock()
pygame.init()
backdropbox = world.get_rect()
main = True

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


'''
Main Loop
'''

while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

        if event.type == pygame.KEYDOWN:
            if event.key == ord('q'):
                pygame.quit()
            try:
                sys.exit()
            finally:
                main = False
    world.blit(backdrop, backdropbox)
    player_list.draw(world)
    pygame.display.flip()
    clock.tick(fps)
```

In the fourth part of this series, I'll show you how to make your sprite move. How exciting!

Links

[1] https://opensource.com/article/17/12/program-game-python-part-2-creating-game-world
[2] https://github.com/SuperTux/supertux/tree/master/data/images/creatures/tux/small
[3] https://www.supertux.org/
[4] http://krita.org/
[5] http://inkscape.org/
[6] https://opengameart.org/
[7] http://gimp.org/
[8] https://github.com/KDE/kcolorchooser
[9] https://github.com/stuartlangridge/ColourPicker

Using Pygame to move your game character around

In the fourth part of this series, learn how to code the controls needed to move a game character.

IN THE FIRST ARTICLE in this series, I explained how to use Python to create a simple, text-based dice game [1]. In the second part, you began building a game from scratch, starting with creating the game's environment [2]. And in the third installment, you created a player sprite [3] and made it spawn in your (rather empty) game world. As you've probably noticed, a game isn't much fun when you can't move your character around. In this article, you'll use Pygame to add keyboard controls so you can direct your character's movement.

There are functions in Pygame to add other kinds of controls (such as a mouse or game controller), but since you certainly have a keyboard if you're typing out Python code, that's what this article covers. Once you understand keyboard controls, you can explore other options on your own.

You created a key to quit your game in the second article in this series, and the principle is the same for movement. However, getting your character to move is a little more complex.

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

Setting up keys for controlling your player sprite

Open your Python game script in IDLE, PyCharm, or a text editor.

Because the game must constantly "listen" for keyboard events, you'll be writing code that needs to run continuously. Can you figure out where to put code that needs to run constantly for the duration of the game?

If you answered "in the main loop," you're correct! Remember that unless code is in a loop, it runs (at most) only once—and it may not run at all if it's hidden away in a class or function that never gets used.

To make Python monitor 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:
    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
```

Some people prefer to control player characters with the keyboard characters W, A, S, and D, and others prefer to use arrow keys. Be sure to include *both* options.

Note: It's vital that you consider all of your users when programming. If you write code that works only for you, it's very likely that you'll be the only one who uses your application. More importantly, if you seek out a job writing code for money, you are expected to write code that works for everyone. Giving your users choices, such as the option to use either arrow keys or WASD (it's called *accessibility*), is a sign of a good programmer.

Launch your game using Python, and watch the console window for output when you press the right, left, and up arrows, or the A, D, and W keys.

```
$ python ./your-name_game.py
  left
  left stop
```

```
right
right stop
jump
```

This confirms that Pygame detects your key presses correctly. Now it's time to do the hard work of making the sprite 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 should you decide to animate it in the future, you also must track frames so the walk cycle stays on track.

Create these variables in the Player class. The first two lines are for context (you already have them in your code, if you've been following along), so add only the last three:

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

With those variables set, it's time to code the sprite's movement.

The player sprite doesn't need to respond to control all the time because sometimes it isn't being told to move. The code that controls the sprite, therefore, is only one small part of all the things the player sprite can do. 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`, which stands for *define*.

Make a function in your Player class to add *some number* of pixels to your sprite's position on screen. Don't worry about how many pixels you add yet; that will be decided in later code.

```
    def control(self,x,y):
        """
        control player movement
        """
        self.movex += x
        self.movey += y
```

To move a sprite in Pygame, you must tell Python to redraw the sprite in its new location—and where that new location is.

Since the Player sprite isn't always moving, make these updates a dedicated function within the Player class. Add this function after the control function you created earlier.

To make it appear that the sprite is walking (or flying, or whatever it is your sprite is supposed to do), you need to change its position on screen when the appropriate key is pressed. To get it to move across the screen, you redefine its position, designated by the `self.rect.x` and `self.rect.y` properties, to its current position plus whatever amount of `movex` or `movey` is applied. (The number of pixels the move requires is set later.)

```
    def update(self):
        """
        Update sprite position
        """
        self.rect.x = self.rect.x + self.movex
```

Do the same thing for the Y position:

```
        self.rect.y = self.rect.y + self.movey
```

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

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

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

Tell the code how many pixels to add to your sprite's position by setting a variable, then use that variable when triggering the functions of your Player sprite.

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

```
player_list = pygame.sprite.Group()
player_list.add(player)
steps = 10  # how many pixels to move
```

Now that you have the appropriate function and 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 player sprite to move with each loop.

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

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

Remember, `steps` is a variable representing how many pixels your sprite moves when a key is pressed. If you add 10 pixels to the location of your player sprite when you press D or the right arrow, then when you stop pressing that key you must subtract 10 (`-steps`) to return your sprite's momentum back to 0.

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

Updating the sprite graphic

Why doesn't your sprite move yet? Because the main loop doesn't call the update function.

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

```
player.update()  # update player position
player_list.draw(world)
pygame.display.flip()
clock.tick(fps)
```

Launch your game again to witness your player sprite move across the screen at your bidding. There's no vertical movement yet because those functions will be controlled by gravity, but that's another lesson for another article.

Movement works, but there's still one small problem: your hero graphic doesn't turn to face the direction it's walking. In other words, if you designed your hero facing right, then it looks like it's walking backwards when you press the left arrow key. Normally, you'd expect your hero to turn left when walking left, and turn right again to walk to the right.

Flipping your sprite

You can flip a graphic with Pygame's `transform` function. This, like all the other functions you've been using for this game, is a lot of complex code and maths distilled into a single, easy to use, Python keyword. This is a great example of why a framework helps you code. Instead of having to learn basic principles of drawing pixels on screen, you can let Pygame do all the work and just make a call to a funciton that already exists.

You only need the transform on the instance when your graphic is walking the opposite way it's facing by default. My graphic faces right, so I apply the transform to the left code block. The `pygame.transform.flip` function takes three arguments, according to Pygame documentation [4]: what to flip, whether to flip horizontally, and whether to flip vertically. In this case, those are the graphic (which you've already defined in the existing code), True for horizontal, and False for a vertical flip.

Update your code:

```
        # moving left
        if self.movex < 0:
            self.frame += 1
            if self.frame > 3*ani:
                self.frame = 0
            self.image = pygame.transform.flip
                        (self.images[self.frame // ani],
                        True, False)
```

Notice that the transform function is inserted into your existing code. The variable `self.image` is still getting defined as an image from your list of hero images, but it's getting "wrapped" in the transform function.

Try your code now, and watch as your hero does an about-face each time you point it in a different direction.

That's enough of a lesson for now. Until the next article, you might try exploring other ways to control your hero. For intance, should you have access to a joystick, try reading Pygame's documentation for its joystick [5] module and see if you can make your sprite move that way. Alternately, see if you can get the mouse [6] to interact with your sprite.

Most importantly, have fun!

All the code used in this tutorial

For your reference, here is all the code used in this series of articles so far.

```
#!/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/>.
from typing import Tuple

import pygame
import sys
import os

'''
Variables
'''

worldx = 960
worldy = 720
fps = 40  # frame rate
ani = 4  # animation cycles
world = pygame.display.set_mode([worldx, worldy])

BLUE = (25, 25, 200)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
ALPHA = (0, 255, 0)

'''
Objects
'''


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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.movex = 0
        self.movey = 0
        self.frame = 0
        self.images = []
        for i in range(1, 5):
            img = pygame.image.load(os.path.join('images',
                  'hero' + str(i) + '.png')).convert()
            img.convert_alpha()  # optimise alpha
            img.set_colorkey(ALPHA)  # set alpha
            self.images.append(img)
            self.image = self.images[0]
            self.rect = self.image.get_rect()

    def control(self, x, y):
        """
        control player movement
        """
        self.movex += x
        self.movey += y

    def update(self):
        """
        Update sprite position
        """

        self.rect.x = self.rect.x + self.movex
        self.rect.y = self.rect.y + self.movey

        # moving left
        if self.movex < 0:
            self.frame += 1
            if self.frame > 3*ani:
                self.frame = 0
            self.image = pygame.transform.flip(self.images[self.
                         frame // ani], True, False)

        # moving right
        if self.movex > 0:
            self.frame += 1
            if self.frame > 3*ani:
                self.frame = 0
            self.image = self.images[self.frame//ani]


'''
Setup
'''

backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
clock = pygame.time.Clock()
pygame.init()
backdropbox = world.get_rect()
main = True

player = Player()  # spawn player
player.rect.x = 0  # go to x
player.rect.y = 0  # go to y
player_list = pygame.sprite.Group()
player_list.add(player)
steps = 10

'''
Main Loop
'''

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

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

world.blit(backdrop, backdropbox)
player.update()
player_list.draw(world)
pygame.display.flip()
clock.tick(fps)
```

You've come far and learned much, but there's a lot more to do. In the next few articles, you'll add enemy sprites [7], emulated gravity, and lots more. In the mean time, practice with Python!

Links

[1] https://opensource.com/article/17/10/python-101
[2] https://opensource.com/article/17/12/program-game-python-part-2-creating-game-world
[3] https://opensource.com/article/17/12/program-game-python-part-3-spawning-player
[4] https://www.pygame.org/docs/ref/transform.html#pygame.transform.flip
[5] http://pygame.org/docs/ref/joystick.html
[6] http://pygame.org/docs/ref/mouse.html#module-pygame.mouse
[7] https://opensource.com/article/18/5/pygame-enemy

What's a hero without a villain?
How to add one to your Python game

In part five of this series on building a Python game from scratch, add a bad guy for your good guy to battle.

IN THE PREVIOUS ARTICLES in this series (see part 1, part 2, part 3, and part 4), you learned how to use Pygame and Python to spawn a playable hero character in an as-yet empty video game world. But what's a hero without a villain?

It would make for a pretty boring game if you had no enemies, so in this article, you'll add an enemy to your game and construct a framework for building levels.

It might seem strange to jump ahead to enemies when there's still more to be done to make the player sprite fully functional, but you've learned a lot already, and creating villains is very similar to creating a player sprite. So relax, use the knowledge you already have, and see what it takes to stir up some trouble.

For this exercise, you need an enemy sprite. If you haven't downloaded one already, you can find Creative Commons [1] assets on OpenGameArt.org [2].

Creating the enemy sprite

Whether you realize it or not, you already know how to implement enemies. The process is similar to creating a player sprite:

1. Make a class so enemies can spawn.
2. Create an `update` function for the enemy, and update the enemy in your main loop.
3. Create a move 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, and you set the sprite's starting position.

Before continuing, make sure you have placed your enemy graphic in your game project's `images` directory (the same directory where you placed your player image). In this article's example code, the enemy graphic is named `enemy.png`.

A game looks a lot better when everything *alive* is animated. Animating an enemy sprite is done the same way as animating a player sprite. For now, though, keep it simple, and use a non-animated sprite.

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
```

If you want to animate your enemy, do it the same way you animated your player [3].

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 you want the sprite to appear. This means 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, and tell it which image to use, along with the X and Y coordinates of your desired spawn point.

Ao you did when spawning a player sprite, add code to designate a spawn point in the setup section of your script:

```
enemy = Enemy(300,0,'enemy.png')      # spawn enemy
enemy_list = pygame.sprite.Group()    # create enemy group
enemy_list.add(enemy)                 # add enemy to group
```

In that sample code, you spawn an enemy by creating a new object (called `enemy`), at 300 pixels on the X axis and 0 on the Y axis. Spawning the enemy at 0 on the Y axis means that its top left corner is located at 0, with the graphic itself descending down from that point. You might need to adjust these numbers, or the numbers for your hero sprite, depending on how big your sprites are, but try to get it to spawn in a place you can reach with your player sprite (accounting for your game's current lack of vertical movement). In the end, I placed my enemy at 0 pixels on the Y axis and my hero at 30 pixels to get them boh to appear on the same plane. Experiment with the spawn points for yourself, keeping in mind that greater Y axis numbers are lower on the screen.

Your hero graphic had an image "hard coded" into its class because there's only one hero, but you may want to use different graphics for each enemy, so the image file is some-

thing you can define at sprite creation. The image used for this enemy sprite is enemy.png.

Drawing a sprite on screen

If you were to launch your game now, it would run but you wouldn't see an enemy. You might recall the same problem when you created your player sprite. Do you remember how to fix it?

To get a sprite to appear on screen, you must add them to your main loop. If something is not in your main loop, then it only happens once, and only for a millisecond. If you want something to persist in your game, it must happen in the main loop.

You must add code to draw all enemies in the enemy group (called `enemy_list`), which you established in your setup section, on the screen. The middle line in this example code is the new line you need to add:

```
player_list.draw(world)
enemy_list.draw(world)  # refresh enemies
pygame.display.flip()
```

Right now, you have only one enemy, but you can add more later if you want. As long as you add an enemy to the enemies group, it will be drawn to the screen during the main loop.

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

Level one

Your game is in its infancy, but you will probably want to add a series of levels, eventually. It's important to plan ahead when you program so your game can grow as you learn more about programming. Even though you don't even have one complete level yet, you should code as if you plan on having many levels.

Think about what a "level" is. How do you know you are at a certain level in a game?

You can think of a level as a collection of items. In a platformer, such as the one you are building here, a level consists of a specific arrangement of platforms, placement of enemies and loot, and so on. You can build a class that builds a level around your player. Eventually, when you create more than one level, you can use this class to generate the next level when your player reaches a specific goal.

Move the code you wrote to create an enemy and its group into a new function that gets called along with each new level. It requires some modification so that each time you create a new level, you can create and place several enemies:

```
class Level():
    def bad(lvl,eloc):
        if lvl == 1:
            enemy = Enemy(eloc[0],eloc[1],'enemy.png') # spawn enemy
            enemy_list = pygame.sprite.Group() # create enemy group
            enemy_list.add(enemy)              # add enemy to group
        if lvl == 2:
            print("Level " + str(lvl) )

        return enemy_list
```

The return statement ensures that when you use the `Level.bad` function, you're left with an `enemy_list` containing each enemy you defined.

Since you are creating enemies as part of each level now, your setup section needs to change, too. Instead of creating an enemy, you must define where the enemy will spawn and what level it belongs to.

```
eloc = []
eloc = [300,0]
enemy_list = Level.bad( 1, eloc )
```

Run the game again to confirm your level is generating correctly. You should see your player, as usual, and the enemy you added in this chapter.

Hitting the enemy

An enemy isn't much of an enemy if it has no effect on the player. It's common for enemies to cause damage when a player collides with them.

Since you probably want to track the player's health, the collision check 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 are 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. The first line in this code sample is for context, so add the second line to your Player class:

```
self.frame  = 0
self.health = 10
```

In the `update` function of your Player class, add this code block:

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

This code establishes a collision detector using the Pygame function `sprite.spritecollide`, called `enemy_hit`. This collision detector sends out a signal any time the hitbox of its parent sprite (the player sprite, where this detector has been created) touches the hitbox of any sprite in `enemy_list`. The

for loop is triggered when such a signal is received and deducts a point from the player's health.

Since this code appears in the `update` function of your player class and `update` is called in your main loop, Pygame checks for this collision once every clock tick.

Moving the enemy

An enemy that stands still is useful if you want, for instance, spikes or traps that can harm your player, but the game is 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.

Eventually, your game world will scroll, so how do you get an enemy to move back and forth within the game world when the game world itself is moving?

You tell your enemy sprite to take, for example, 10 paces to the right, 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 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 variable
```

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 on any number from 0 to 100.
- Move left if the counter is on any number from 100 to 200.
- Reset the counter back to 0 if the counter is greater than 200.

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

The actual numbers you use for how far the enemy will move in either direction depending on your screen size, and possibly, eventually, the size of the platform your enemy is walking on. Start small and work your way up as you get used to the results. Try this first:

```
    def move(self):
        '''
        enemy movement
        '''
        distance = 80
        speed = 8

        if self.counter >= 0 and self.counter <= distance:
            self.rect.x += speed
        elif self.counter >= distance and self.counter
            <= distance*2:
            self.rect.x -= speed
        else:
            self.counter = 0

        self.counter += 1
```

After you enter this code, PyCharm will offer to simplify the "chained comparison". You can accept its suggestion to optimize your code, and to learn some advanced Python syntax. You can also safely ignore PyCharm. The code works, either way.

You can adjust the distance and speed as needed.

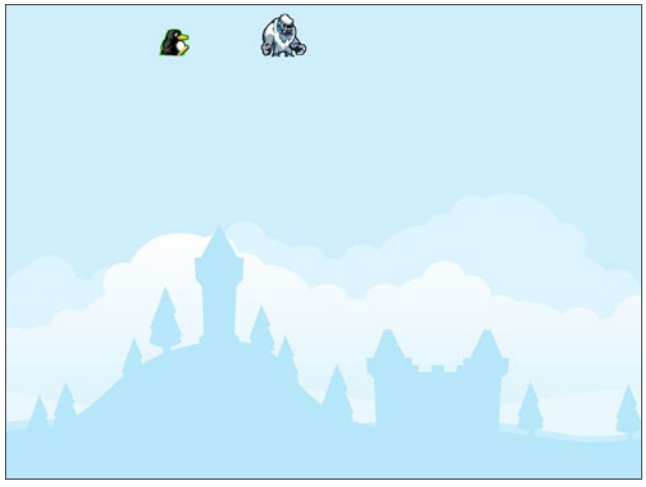
The question is: does this code work if you launch your game now?

Of course not! And you know why: you must call the move function in your main loop.

The first line in this sample code is for context, so add the last two lines:

```
    enemy_list.draw(world) #refresh enemy
    for e in enemy_list:
        e.move()
```

Launch your game and see what happens when you hit your enemy. You might have to adjust where the sprites spawn so that your player and your enemy sprite can collide. When they do collide, look in the console of IDLE or PyCharm to see the health points being deducted.



You may notice that health is deducted for every moment your player and enemy are touching. That's a problem, but it's a problem you'll solve later, after you've had more practice with Python.

For now, try adding some more enemies. Remember to add each enemy to the `enemy_list`. As an exercise, see if you can think of how you can change how far different enemy sprites move.

Code so far

For you reference, here's the code so far:

```
#!/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/>.

from typing import Tuple

import pygame
import sys
import os

'''
Variables
'''

worldx = 960
worldy = 720
fps = 40
ani = 4
world = pygame.display.set_mode([worldx, worldy])

BLUE = (25, 25, 200)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
ALPHA = (0, 255, 0)

'''
Objects
'''


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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.movex = 0
        self.movey = 0
        self.frame = 0
        self.health = 10
        self.images = []
        for i in range(1, 5):
            img = pygame.image.load(os.path.join('images',
                  'hero' + str(i) + '.png')).convert()
            img.convert_alpha()
            img.set_colorkey(ALPHA)
            self.images.append(img)
            self.image = self.images[0]
            self.rect = self.image.get_rect()

    def control(self, x, y):
        """
        control player movement
        """
        self.movex += x
        self.movey += y

    def update(self):
        """
        Update sprite position
        """

        self.rect.x = self.rect.x + self.movex
        self.rect.y = self.rect.y + self.movey

        # moving left
        if self.movex < 0:
            self.frame += 1
            if self.frame > 3*ani:
                self.frame = 0
            self.image = pygame.transform.flip(self.images[self.
                         frame // ani], True, False)

        # moving right
        if self.movex > 0:
            self.frame += 1
            if self.frame > 3*ani:
                self.frame = 0
            self.image = self.images[self.frame//ani]

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


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
        self.counter = 0

    def move(self):
        '''
        enemy movement
        '''
        distance = 80
        speed = 8

        if self.counter >= 0 and self.counter <= distance:
            self.rect.x += speed
        elif self.counter >= distance and self.counter
            <= distance*2:
            self.rect.x -= speed
        else:
            self.counter = 0

        self.counter += 1


class Level():
    def bad(lvl, eloc):
        if lvl == 1:
            enemy = Enemy(eloc[0],eloc[1],'enemy.png')
            enemy_list = pygame.sprite.Group()
            enemy_list.add(enemy)
        if lvl == 2:
            print("Level " + str(lvl) )

        return enemy_list


'''
Setup
'''

backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
clock = pygame.time.Clock()
pygame.init()
backdropbox = world.get_rect()
main = True

player = Player()  # spawn player
player.rect.x = 0  # go to x
player.rect.y = 30  # go to y
player_list = pygame.sprite.Group()
player_list.add(player)
steps = 10

eloc = []
eloc = [300, 0]
enemy_list = Level.bad(1, eloc )

'''
Main Loop
'''

while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

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

    world.blit(backdrop, backdropbox)
    player.update()
    player_list.draw(world)
    enemy_list.draw(world)
    for e in enemy_list:
        e.move()
    pygame.display.flip()
    clock.tick(fps)
```

Links

[1] https://opensource.com/article/20/1/what-creative-commons

[2] https://opengameart.org/content/opp2017-sprites-characters-objects-effects

[3] https://opensource.com/article/17/12/game-python-moving-player

Put platforms in a Python game with Pygame

In part six of this series on building a Python game from scratch, create some platforms for your characters to travel.

A PLATFORMER GAME needs platforms. In Pygame [1], the platforms themselves are sprites, just like your playable sprite. That's important because having platforms that are objects makes it a lot easier for your player sprite to interact with them.

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

Coding platform objects

To build a platform object, you create a class called `Platform`. It's a `sprite` [2], just like your Player sprite, with many of the same properties.

Your `Platform` class needs to know a lot of information about what kind of platform you want, where it should appear 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 article [3], you don't have to tell `Platform` everything upfront.

In the objects section of your script, create a new class:

```
# x location, y location, img width, img height, img file
class Platform(pygame.sprite.Sprite):
    def __init__(self, xloc, yloc, imgw, imgh, img):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(os.path.join
                    ('images', img)).convert()
        self.image.convert_alpha()
        self.image.set_colorkey(ALPHA)
        self.rect = self.image.get_rect()
        self.rect.y = yloc
        self.rect.x = xloc
```

When called, this class creates an object onscreen in *some* X and Y location, with *some* width and height, using *some* image file for texture. It's very similar to how players or enemies are drawn onscreen. You probably recognize this same code structure from the Player and Enemy classes.

Types of platforms

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

The tile method

There are a few different ways to implement a platform game world. In the original side-scroller games, such as Mario Super Bros. and Sonic the Hedgehog, the technique was to use "tiles," meaning that there were a few blocks to represent the ground and various platforms, and these blocks were used and reused to make a level. You have only eight or 12 different kinds of blocks, and you line them up onscreen to create the ground, floating platforms, and whatever else your game needs. Some people find this the easier way to make a game since you just have to make (or download) a small set of level assets to create many different levels. The code, however, requires a little more math.

SuperTux, a tile-based video game.

The hand-painted method

Another method is to make each and every asset as one whole image. If you enjoy creating assets for your game

world, this is a great excuse to spend time in a graphics application, building each and every part of your game world. This method requires less math, because all the platforms are whole, complete objects, and you tell Python where to place them onscreen.

Each method has advantages and disadvantages, and the code you must use is slightly different depending on the method you choose. I'll cover both so you can use one or the other, or even a mix of both, in your project.

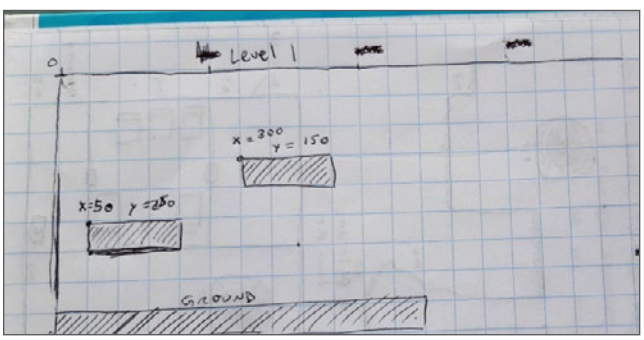
Level mapping

Mapping out your game world is a vital part of level design and game programming in general. It does involve math, but nothing too difficult, and Python is good at math so it can help some.

You might find it helpful to design on paper first. Get a sheet of paper and draw a box to represent your game window. Draw platforms in the box, labeling each with its X and Y coordinates, as well as its intended width and height. The actual positions in the box don't have to be exact, as long as you keep the numbers realistic. For instance, if your screen is 720 pixels wide, then you can't fit eight platforms at 100 pixels each all on one screen.

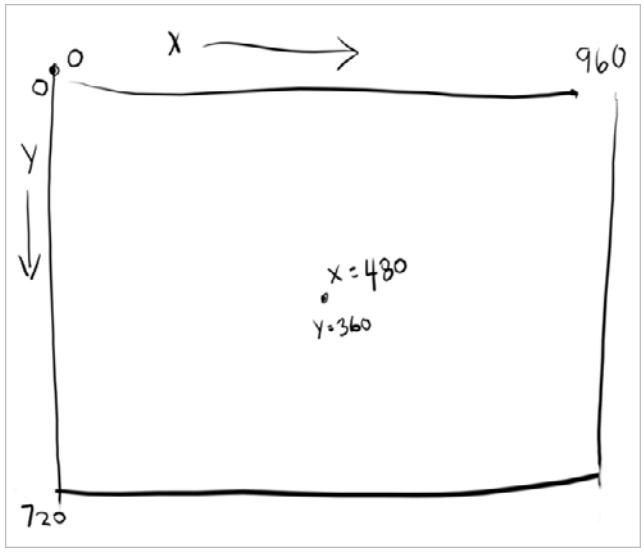
Of course, not all platforms in your game have to fit in one screen-sized box, because your game will scroll as your player walks through it. So keep drawing your game world to the right of the first screen until the end of the level.

If you prefer a little more precision, you can use graph paper. This is especially helpful when designing a game with tiles because each grid square can represent one tile.



Coordinates

You may have learned in school about the Cartesian coordinate system [4]. What you learned applies to Pygame, except that in Pygame, your game world's coordinates place 0,0 in the top-left corner of your screen instead of in the middle, which is probably what you're used to from Geometry class.



Example of coordinates in Pygame.

The X axis starts at 0 on the far left and increases infinitely to the right. The Y axis starts at 0 at the top of the screen and extends down.

Image sizes

Mapping out a game world is meaningless if you don't know how big your players, enemies, and platforms are. You can find the dimensions of your platforms or tiles in a graphics program. In Krita [5], for example, click on the **Image** menu and select **Properties**. You can find the dimensions at the very top of the **Properties** window.

Alternately, you can create a simple Python script to tell you the dimensions of an image. To do that, you must install a Python module called Pillow, which provides the Python Image Library (PIL). Add Pillow to your project's `requirements.txt` file:

```
pygame~=1.9.6
Pillow
```

Create a new Python file in PyCharm and name it `identify`. Type this code into it:

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

# GNU All-Permissive License
# Copying and distribution of this file, with or without
# modification, are permitted in any medium without royalty
# provided the copyright notice and this notice are preserved.
# This file is offered as-is, without any warranty.

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]
img = Image.open(pic)
X   = img.size[0]
Y   = img.size[1]

print(X, Y)
```

Click on the **Terminal** tab at the bottom of the PyCharm window to open a terminal within your virtual environment. Now you can install the Pillow module into your environment:

```
(venv) pip install -r requirements.txt
Requirement already satisfied: pygame~=1.9.6 [...]
Installed Pillow [...]
```

Once that is installed, run your script from within your game project directory:

```
(venv) python ./identify.py images/ground.png
(1080, 97)
```

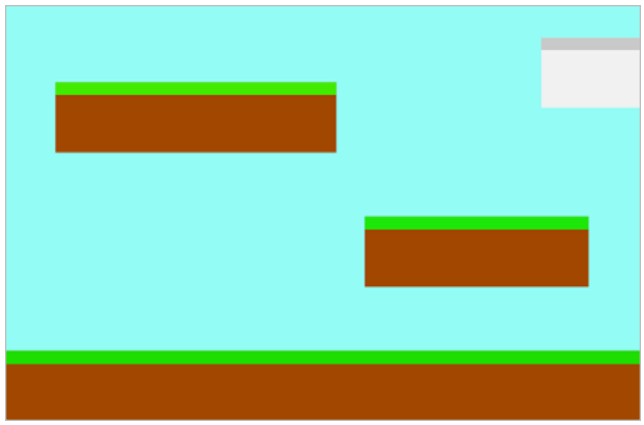
The image size of the ground platform in this example is 1080 pixels wide and 97 high.

Platform blocks

If you choose to draw each asset individually, you must create several platforms and any other elements you want to insert into your game world, each within its own file. In other words, you should have one file per asset, like this:

One image file per object.

You can reuse each platform as many times as you want, just make sure that each file only contains one platform. You cannot use a file that contains everything, like this:



Your level cannot be one image file.

You might want your game to look like that when you've finished, but if you create your level in one big file, there is no way to distinguish a platform from the background, so either paint your objects in their own file or crop them from a large file and save individual copies.

Note: As with your other assets, you can use GIMP [6], Krita [5], MyPaint [7], or Inkscape [8] to create your game assets.

Platforms appear on the screen at the start of each level, so you must add a `platform` function in your `Level` class. The special case here is the ground platform, which is important enough to be treated as its own platform group. By treating the ground as its own special kind of platform, you can choose whether it scrolls or whether it stands still while other platforms float over the top of it. It's up to you.

Add these two functions to your Level class:

```
def ground(lvl,x,y,w,h):
    ground_list = pygame.sprite.Group()
    if lvl == 1:
        ground = Platform(x,y,w,h,'block-ground.png')
        ground_list.add(ground)

    if lvl == 2:
        print("Level " + str(lvl) )

    return ground_list

def platform( lvl ):
    plat_list = pygame.sprite.Group()
    if lvl == 1:
        plat = Platform(200, worldy-97-128, 285,67,'block-big.png')
        plat_list.add(plat)
        plat = Platform(500, worldy-97-320, 197,54,'block-small.png')
        plat_list.add(plat)
    if lvl == 2:
        print("Level " + str(lvl) )

    return plat_list
```

The `ground` function requires an X and Y location so Pygame knows where to place the ground platform. It also requires the width and height of the platform so Pygame knows how far the ground extends in each direction. The function uses your `Platform` class to generate an object onscreen, and then adds that object to the `ground_list` group.

The `platform` function is essentially the same, except that there are more platforms to list. In this example, there are only two, but you can have as many as you like. After entering one platform, you must add it to the `plat_list` before listing another. If you don't add a platform to the group, then it won't appear in your game.

> **Tip:** It can be difficult to think of your game world with 0 at the top, since the opposite is what happens in the real world; when figuring out how tall you are, you don't measure yourself from the sky down, you measure yourself from your feet to the top of your head.
>
> If it's easier for you to build your game world from the "ground" up, it might help to express Y-axis values as negatives. For instance, you know that the bottom of your game world is the value of `worldy`. So `worldy` minus the height of the ground (97, in this example) is where your player is normally standing. If your character is 64 pixels tall, then the ground minus 128 is exactly twice as tall as your player. Effectively, a platform placed at 128 pixels is about two stories tall, relative to your player. A platform at -320 is three more stories. And so on.

As you probably know by now, none of your classes and functions are worth much if you don't *use* them. Add this code to your setup section:

```
ground_list = Level.ground(1, 0, worldy-97, 1080, 97)
plat_list = Level.platform(1)
```

And add these lines to your main loop (again, the first line is just for context):

```
enemy_list.draw(world)   # refresh enemies
ground_list.draw(world)  # refresh ground
plat_list.draw(world)    # refresh platforms
```

Tiled platforms

Tiled game worlds are considered easier to make because you just have to draw a few blocks upfront and can use them over and over to create every platform in the game. There are sets of tiles with a Creative Commons license [9] for you to use on sites like kenney.nl [10] and OpenGameArt.org [11].

The simplified-platformer-pack from kenney.nl are 64 pixels square, so that's the dimension for tiles this article uses. Should you download or create tiles with a different size, adjust the code as needed.

The `Platform` class is the same as the one provided in the previous sections.

The `ground` and `platform` in the `Level` class, however, must use loops to calculate how many blocks to use to create each platform.

If you intend to have one solid ground in your game world, the ground is simple. You just "clone" your ground tile across the whole window. For instance, you could create a list of X and Y values to dictate where each tile should be placed, and then use a loop to take each value and draw one tile. This is just an example, so don't add this to your code:

```
# Do not add this to your code
gloc = [0,656,64,656,128,656,192,656,256,656,320,656,384,656]
```

If you look carefully, though, you can see all the Y values are always the same (656, to be specific), and the X values increase steadily in increments of 64, which is the size of the tile. That kind of repetition is exactly what computers are good at, so you can use a little bit of math logic to have the computer do all the calculations for you:

Add this to the setup part of your script:

```
gloc = []
tx   = 64
ty   = 64

i=0
while i <= (worldx/tx)+tx:
    gloc.append(i*tx)
    i=i+1

ground_list = Level.ground( 1,gloc,tx,ty )
```

With this code, regardless of the size of your window, Python divides the width of the game world by the width of the tile and creates an array listing each X value. This doesn't calculate the Y value, but that never changes on flat ground anyway.

To use the array in a function, use a `while` loop that looks at each entry and adds a ground tile at the appropriate location. Add this function to your `Level` class:

```
def ground(lvl,gloc,tx,ty):
    ground_list = pygame.sprite.Group()
    i=0
    if lvl == 1:
        while i < len(gloc):
            ground = Platform(gloc[i],worldy-ty,tx,ty,
```

```
                'tile-ground.png')
            ground_list.add(ground)
            i=i+1

    if lvl == 2:
        print("Level " + str(lvl) )

    return ground_list
```

This is nearly the same code as the `ground` function for the block-style platformer, provided in the previous section, aside from the `while` loop.

For moving platforms, the principle is similar, but there are some tricks you can use to make your life easier.

Rather than mapping every platform by pixels, you can define a platform by its starting pixel (its X value), the height from the ground (its Y value), and how many tiles to draw. That way, you don't have to worry about the width and height of every platform.

The logic for this trick is a little more complex, so copy this code carefully. There is a `while` loop inside of another `while` loop because this function must look at all three values within each array entry to successfully construct a full platform. In this example, there are only three platforms defined as `ploc.append` statements, but your game probably needs more, so define as many as you need. Of course, some won't appear yet because they're far offscreen, but they'll come into view once you implement scrolling.

```
def platform(lvl,tx,ty):
    plat_list = pygame.sprite.Group()
    ploc = []
    i=0
    if lvl == 1:
        ploc.append((200,worldy-ty-128,3))
        ploc.append((300,worldy-ty-256,3))
        ploc.append((500,worldy-ty-128,4))
        while i < len(ploc):
            j=0
            while j <= ploc[i][2]:
                plat = Platform((ploc[i][0]+(j*tx)),ploc[i]
                       [1],tx,ty,'tile.png')
                plat_list.add(plat)
                j=j+1
            print('run' + str(i) + str(ploc[i]))
            i=i+1

    if lvl == 2:
        print("Level " + str(lvl) )

    return plat_list
```

Of course, this has only created a function to calculate platforms for each level. You code doesn't invoke the function yet.

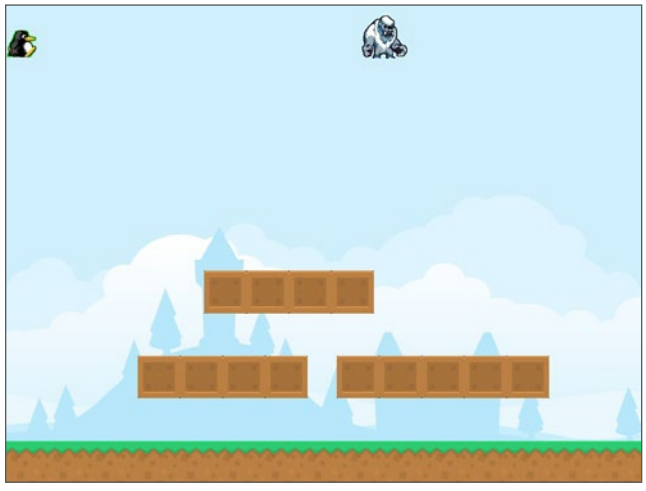
In the setup section of your program, add this line:

```
plat_list = Level.platform(1, tx, ty)
```

To get the platforms to appear in your game world, they must be in your main loop. If you haven't already done so, add these lines to your main loop (again, the first line is just for context):

```
    enemy_list.draw(world)  # refresh enemies
    ground_list.draw(world) # refresh ground
    plat_list.draw(world)   # refresh platforms
```

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 offscreen; you'll fix that soon.



Applying what you know

I haven't demonstrated how to place your enemy in your game world, but apply what you've learnt so far to position the enemy sprite either on a platform or down on the ground.

Don't position your hero sprite yet. That must be managed by the forces of gravity (or at least an emulation of it), which you'll learn in the next two articles.

For now, here's the code so far:

```
#!/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/>.

import pygame
import sys
import os

'''
Variables
'''



worldx = 960
worldy = 720
fps = 40
ani = 4
world = pygame.display.set_mode([worldx, worldy])

BLUE = (25, 25, 200)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
ALPHA = (0, 255, 0)

'''
Objects
'''

# x location, y location, img width, img height, img file
class Platform(pygame.sprite.Sprite):
    def __init__(self, xloc, yloc, imgw, imgh, img):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(os.path.join
                    ('images', img)).convert()
        self.image.convert_alpha()
        self.image.set_colorkey(ALPHA)
        self.rect = self.image.get_rect()
        self.rect.y = yloc
        self.rect.x = xloc

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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.movex = 0
        self.movey = 0
        self.frame = 0
        self.health = 10
        self.images = []
        for i in range(1, 5):
            img = pygame.image.load(os.path.join('images',
                  'hero' + str(i) + '.png')).convert()
            img.convert_alpha()
            img.set_colorkey(ALPHA)
            self.images.append(img)
            self.image = self.images[0]
            self.rect = self.image.get_rect()

    def control(self, x, y):
        """
        control player movement
        """
        self.movex += x
        self.movey += y

    def update(self):
        """
        Update sprite position
        """

        self.rect.x = self.rect.x + self.movex
        self.rect.y = self.rect.y + self.movey

        # moving left
        if self.movex < 0:
            self.frame += 1
            if self.frame > 3*ani:
                self.frame = 0
            self.image = pygame.transform.flip(self.images[self.
                         frame // ani], True, False)

        # moving right
        if self.movex > 0:
            self.frame += 1
            if self.frame > 3*ani:
                self.frame = 0
            self.image = self.images[self.frame//ani]

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

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
        self.counter = 0

    def move(self):
        '''
        enemy movement
        '''
        distance = 80
        speed = 8

        if self.counter >= 0 and self.counter <= distance:
            self.rect.x += speed
        elif self.counter >= distance and self.counter
            <= distance*2:
            self.rect.x -= speed
        else:
            self.counter = 0

        self.counter += 1


class Level():
    def ground(lvl, gloc, tx, ty):
        ground_list = pygame.sprite.Group()
        i = 0
        if lvl == 1:
            while i < len(gloc):
                ground = Platform(gloc[i], worldy - ty, tx, ty,
                         'tile-ground.png')
                ground_list.add(ground)
                i = i + 1

        if lvl == 2:
            print("Level " + str(lvl) )

        return ground_list

    def bad(lvl, eloc):
        if lvl == 1:
            enemy = Enemy(eloc[0],eloc[1],'enemy.png')
            enemy_list = pygame.sprite.Group()
            enemy_list.add(enemy)
        if lvl == 2:
            print("Level " + str(lvl) )

        return enemy_list

    # x location, y location, img width, img height, img file
    def platform(lvl,tx,ty):
    plat_list = pygame.sprite.Group()
    ploc = []
    i=0
        if lvl == 1:
            ploc.append((200,worldy-ty-128,3))
            ploc.append((300,worldy-ty-256,3))
            ploc.append((500,worldy-ty-128,4))
            while i < len(ploc):
                j=0
                while j <= ploc[i][2]:
                    plat = Platform((ploc[i][0]+(j*tx)),ploc[i]
                           [1],tx,ty,'tile.png')
                    plat_list.add(plat)
                    j=j+1
                print('run' + str(i) + str(ploc[i]))
                i=i+1

        if lvl == 2:
            print("Level " + str(lvl) )

        return plat_list


'''
Setup
'''

backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
clock = pygame.time.Clock()
pygame.init()
backdropbox = world.get_rect()
main = True

player = Player()  # spawn player
player.rect.x = 0  # go to x
player.rect.y = 30  # go to y
player_list = pygame.sprite.Group()
player_list.add(player)
steps = 10

eloc = []
eloc = [300, 0]
enemy_list = Level.bad(1, eloc )

gloc = []
tx   = 64
ty   = 64

i = 0
while i <= (worldx / tx) + tx:
    gloc.append(i * tx)
    i = i + 1

ground_list = Level.ground(1, gloc, tx, ty)
plat_list = Level.platform(1, tx, ty)
```

```
'''
Main Loop
'''

while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

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

    world.blit(backdrop, backdropbox)
    player.update()
    player_list.draw(world)
    enemy_list.draw(world)
    ground_list.draw(world)
    plat_list.draw(world)
    for e in enemy_list:
        e.move()
    pygame.display.flip()
    clock.tick(fps)
```

Links

[1] https://www.pygame.org/news
[2] https://opensource.com/article/17/12/game-python-add-a-player
[3] https://opensource.com/article/17/12/game-python-moving-player
[4] https://en.wikipedia.org/wiki/Cartesian_coordinate_system
[5] https://krita.org/en/
[6] https://www.gimp.org/
[7] http://mypaint.org/about/
[8] https://inkscape.org/en/
[9] https://opensource.com/article/20/1/what-creative-commons
[10] https://kenney.nl/assets/simplified-platformer-pack
[11] https://opengameart.org/content/simplified-platformer-pack

Simulate gravity in your Python game

Learn how to program video games with Python's Pygame module and start manipulating 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 most video games, there are basically only two aspects of physics that are important: gravity and collision.

You implemented some collision detection when you added an enemy [1] to your game, but this article adds more because gravity requires collision detection. Think about why gravity might involve collisions. If you can't think of any reasons, don't worry—it'll become apparent as you work through the sample code.

Gravity in the real world is the tendency for objects with mass to be drawn toward one another. The larger the object, the more gravitational influence it exerts. In video game physics, you don't have to create objects with mass great enough to justify a gravitational pull; you can just program a tendency for objects to fall toward the presumed largest object in the video game world: the world itself.

Adding a gravity function

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

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—if you add gravity to your entire game world, all of your platforms would fall to the ground. Instead, you add gravity just to your player and enemy sprites.

First, add a **gravity** function in your **Player** class:

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

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 basically gravity.

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

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

```
player.gravity() # check gravity
player.update()
```

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

Your gravity simulation is working, but maybe too well.

As an experiment, try changing the rate at which your player falls.

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 happen when a player falls off the world, you have to be able to detect when the player disappears offscreen.

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 falls so far that it reaches the bottom of the screen, then you can do *something*. To keep things simple, 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.movey += 3.2 # how fast player falls

    if self.rect.y > worldy and self.movey >= 0:
        self.movey = 0
        self.rect.y = worldy-ty
```

Then launch your game. Your sprite still falls, but it stops at the bottom of the screen. You may not be able to see your sprite behind the ground layer, though. An easy fix is to make your player sprite bounce higher by adding another **-ty** to its new Y position after it hits the bottom of the game world:

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

        if self.rect.y > worldy and self.movey >= 0:
            self.movey = 0
            self.rect.y = worldy-ty-ty
```

Now your player bounces at the bottom of the screen, just behind your ground sprites.

What your player really needs is a way to fight gravity. The problem with gravity is, you can't fight it unless you have something to push off of. So, in the next article, you'll add ground and platform collision and the ability to jump. In the meantime, try applying gravity to the enemy sprite.

Here's all the code so far:

```
#!/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/>.

import pygame
import sys
import os

'''
Variables
'''

worldx = 960
worldy = 720
fps = 40
ani = 4
world = pygame.display.set_mode([worldx, worldy])

BLUE = (25, 25, 200)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
ALPHA = (0, 255, 0)

'''
Objects
'''

# x location, y location, img width, img height, img file
class Platform(pygame.sprite.Sprite):
    def __init__(self, xloc, yloc, imgw, imgh, img):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(os.path.join
                    ('images', img)).convert()
        self.image.convert_alpha()
        self.image.set_colorkey(ALPHA)
        self.rect = self.image.get_rect()
        self.rect.y = yloc
        self.rect.x = xloc

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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.movex = 0
        self.movey = 0
        self.frame = 0
        self.health = 10
        self.images = []
        for i in range(1, 5):
            img = pygame.image.load(os.path.join('images',
                  'hero' + str(i) + '.png')).convert()
            img.convert_alpha()
            img.set_colorkey(ALPHA)
            self.images.append(img)
            self.image = self.images[0]
            self.rect = self.image.get_rect()

    def gravity(self):
        self.movey += 3.2
        if self.rect.y > worldy and self.movey >= 0:
            self.movey = 0
            self.rect.y = worldy-ty-ty

    def control(self, x, y):
        """
        control player movement
        """
        self.movex += x
        self.movey += y
```

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

        self.rect.x = self.rect.x + self.movex
        self.rect.y = self.rect.y + self.movey

        # moving left
        if self.movex < 0:
            self.frame += 1
            if self.frame > 3*ani:
                self.frame = 0
            self.image = pygame.transform.flip(self.images[self.
                         frame // ani], True, False)

        # moving right
        if self.movex > 0:
            self.frame += 1
            if self.frame > 3*ani:
                self.frame = 0
            self.image = self.images[self.frame//ani]

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


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
        self.counter = 0

    def move(self):
        '''
        enemy movement
        '''
        distance = 80
        speed = 8

        if self.counter >= 0 and self.counter <= distance:
            self.rect.x += speed
        elif self.counter >= distance and self.counter
            <= distance*2:
            self.rect.x -= speed
        else:
            self.counter = 0

        self.counter += 1


class Level():
    def ground(lvl, gloc, tx, ty):
        ground_list = pygame.sprite.Group()
        i = 0
        if lvl == 1:
            while i < len(gloc):
                ground = Platform(gloc[i], worldy - ty, tx, ty,
                         'tile-ground.png')
                ground_list.add(ground)
                i = i + 1

        if lvl == 2:
            print("Level " + str(lvl) )

        return ground_list

    def bad(lvl, eloc):
        if lvl == 1:
            enemy = Enemy(eloc[0],eloc[1],'enemy.png')
            enemy_list = pygame.sprite.Group()
            enemy_list.add(enemy)
        if lvl == 2:
            print("Level " + str(lvl) )

        return enemy_list

    # x location, y location, img width, img height, img file
    def platform(lvl,tx,ty):
    plat_list = pygame.sprite.Group()
    ploc = []
    i=0
    if lvl == 1:
        ploc.append((200,worldy-ty-128,3))
        ploc.append((300,worldy-ty-256,3))
        ploc.append((500,worldy-ty-128,4))
        while i < len(ploc):
            j=0
            while j <= ploc[i][2]:
                plat = Platform((ploc[i][0]+(j*tx)),ploc[i]
                       [1],tx,ty,'tile.png')
                plat_list.add(plat)
                j=j+1
            print('run' + str(i) + str(ploc[i]))
            i=i+1
```

```
    if lvl == 2:
        print("Level " + str(lvl) )

    return plat_list


'''
Setup
'''

backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
clock = pygame.time.Clock()
pygame.init()
backdropbox = world.get_rect()
main = True

player = Player()  # spawn player
player.rect.x = 0  # go to x
player.rect.y = 30  # go to y
player_list = pygame.sprite.Group()
player_list.add(player)
steps = 10

eloc = []
eloc = [300, 0]
enemy_list = Level.bad(1, eloc )

gloc = []
tx   = 64
ty   = 64

i = 0
while i <= (worldx / tx) + tx:
    gloc.append(i * tx)
    i = i + 1

ground_list = Level.ground(1, gloc, tx, ty)
plat_list = Level.platform(1, tx, ty)

'''
Main Loop
'''

while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

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

    world.blit(backdrop, backdropbox)
    player.update()
    player_list.draw(world)
    enemy_list.draw(world)
    ground_list.draw(world)
    plat_list.draw(world)
    for e in enemy_list:
        e.move()
    pygame.display.flip()
    clock.tick(fps)
```

Links

[1] https://opensource.com/article/18/5/pygame-enemy

Add jumping to your Python platformer game

Learn how to fight gravity with jumping in this installment on programming video games with Python's Pygame module.

IN THE PREVIOUS ARTICLE in this series, you simulated gravity, but now 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 falling 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. Once the player sprite is jumping, then gravity is 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 lines above the comment are for context, so just add the final two lines:

```
self.frame = 0
self.health = 10
# jump code below
self.is_jumping = True
self.is_falling = False
```

These new values are called Boolean values, which is a term (named after mathematician George Boole) meaning *either true* or *false*. In programming, this is a special data type indicating that a variable is either "on" or "off". In this case, the hero sprite can either be falling or not falling, and it can be jumping or not jumping.

The first variable (**is_jumping**) is set to **True** because I'm spawing the hero in the sky and need it to fall immediately to the ground, as if it were in mid-jump. This is a little counter-intuitive, because the hero isn't actually jumping. The hero has only just spawned. This is theoretically an abuse of this Boolean value, and it is admittedly "cleaner" code to have True and False statements that actually reflect reality. However, I find it easier to let gravity help the hero find the ground rather than having to hard code a spawn position every level. It also evokes classic platformers, and gives the player the sense of "jumping into" the game world. In other words, this is a small initial lie that serves the program, so set it to **True**.

The other variable (**is_falling**) is also set to **True** because the hero does indeed need to descend to the ground.

Conditional gravity

In the real world, jumping is an act of moving against gravity. In your game, though, gravity only needs to be "on" when the hero sprite isn't standing on solid ground. When you have gravity on all the time (in Pygame), you risk getting a bounce-effect on your hero sprite as gravity constantly tries to force the hero down while the collision with the ground resists. Not all game engines require this much interaction with gravity, but Pygame isn't designed exclusively for platformers (you could write a top-down game instead, for example) so gravity isn't managed by the engine.

Your code is only *emulating* gravity in your game world. The hero sprite isn't actually falling when it appears to fall, it's being moved by your **gravity** function. To permit your hero sprite to fight gravity and jump, or to collide with solid objects (like the ground and floating platforms), you must modify your **gravity** function to activate only when the hero is jumping. This code replaces the entire **gravity** function you wrote for the previous article:

```
def gravity(self):
    if self.is_jumping:
        self.movey += 3.2
```

This causes your hero sprite to fall right through the bottom of the screen, but you can fix that with some collision detection on the ground.

Programming solid ground

In the previous article, a quick hack was implemented to keep the hero sprite from falling through the bottom of the screen. It kept the hero on screen, but only by creating an invisible wall across the bottom of the screen. It's cleaner to use objects as objects, and besides it's pretty common in platformers to allow players to fall off the world as a penalty for a poorly timed jump.

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

```
ground_hit_list = pygame.sprite.spritecollide(self,
                  ground_list, False)
for g in ground_hit_list:
    self.movey = 0
    self.rect.bottom = g.rect.top
    self.is_jumping = False  # stop jumping

# fall off the world
if self.rect.y > worldy:
    self.health -=1
    print(self.health)
    self.rect.x = tx
    self.rect.y = ty
```

This code block checks for collisions happening between ground sprites and the hero sprite. This is the same principle you used when detecting a hit against your hero by an enemy.

In the event of a collision, it uses builtin information provided by Pygame to find the bottom of the hero sprite (**self.rect.bottom**), and set its position to the top of the ground sprite (**p.rect.top**). This provides the illusion that the hero sprite is "standing" on the ground, and prevents it from falling through the ground.

It also sets **self.is_falling** to 0 so that the program is aware that the hero is not in mid-jump. Additionally, it sets **self.movey** to 0 so the hero is not pulled by gravity (it's a quirk of game physics that you don't need to continue to pull a sprite toward Earth once the sprite has been grounded).

The **if** statement at the end detects whether the player has descended *below* the level of the ground; if so, it deducts health points as a penalty, and then respawns the hero sprite back at the top left of the screen (using the values of **tx** and **ty**, the size of tiles. as quick and easy starting values.) This assumes that you want your player to lose health points and respawn for falling off the world. That's not strictly necessary; it's just a common convention in platformers.

Jumping in Pygame

The code to **jump** happens in several places. First, create a jump function to "flip" the **is_jumping** and **is_falling** values:

```
def jump(self):
    if self.is_jumping is False:
        self.is_falling = False
        self.is_jumping = True
```

The actual lift-off from the jump action happens in the **update** function of your **Player** class:

```
    if self.is_jumping and self.is_falling is False:
        self.is_falling = True
        self.movey -= 33  # how high to jump
```

This code executes only when the **is_jumping** variable is True while the **is_falling** variable is False. When these conditions are satisfied, the hero sprite's Y position is adjusted to 33 pixels in the "air". It's *negative* 33 because a lower number on the Y axis in Pygame means it's closer to the top of the screen. That's effectively a jump. You can adjust the number of pixels for a lower or higher jump. This clause also sets **is_falling** to True, which prevents another jump from being registered. If you set it to False, a jump action would compound on itself, shooting your hero into space, which is fun to witness but not ideal for gameplay.

Calling the jump function

The problem is that nothing in your main loop is calling the **jump** function yet. You made a placeholder keypress for it early on, but right now, all the **jump** key does is print jump to the terminal.

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

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

If you would rather use the Spacebar for jumping, set the key to **pygame.K_SPACE** instead of **pygame.K_UP**. Alternately, you can use both (as separate if statements) so that the player has a choice.

Landing on a platform

So far, you've defined an anti-gravity condition for when the player sprite hits the ground, but the game code keeps platforms and the ground in separate lists. (As with so many choices made in this article, that's not strictly necessary,

and you can experiment with treating the ground as just another platform.) To enable a player sprite to stand on top of a platform, you must detect a collision between the player sprite and a platform sprite, and stop gravity from "pulling" it downward.

Place this code into your **update** function:

```
plat_hit_list = pygame.sprite.spritecollide(self,
                plat_list, False)
for p in plat_hit_list:
    self.is_jumping = False  # stop jumping
    self.movey = 0

    # approach from below
    if self.rect.bottom <= p.rect.bottom:
       self.rect.bottom = p.rect.top
    else:
       self.movey += 3.2
```

This code scans through the list of platforms for any collisions with your hero sprite. If one is detected, then **is_jumping** is set to False and any movement in the sprite's Y position is cancelled.

Platforms hang in the air, meaning the player can interact with them by approaching them from either above or below. It's up to you how you want the platforms to react to your hero sprite, but it's not uncommon to block a sprite from accessing a platform from below. The code in the second code block treats platforms as a sort of ceiling or pergola, such that the hero can jump onto a platform as long as it jumps higher than the platform's topside, but obstructs the sprite when it tries to jump from beneath:

The first clause of the **if** statement detects whether the bottom of the hero sprite is less than (higher on the screen) than the platform. If it is, then the hero "lands" on the platform, because the value of the bottom of the hero sprite is made equal to the top of the platform sprite. Otherwise, the hero sprite's Y position is increased, causing it to "fall" away from the platform.

Falling

If you try your game now, you find that jumping works mostly as expected, but falling isn't consistent. For instance, after your hero jumps onto a platform, it can't walk off of a platform to fall to the ground. It just stays in the air, as if there was still a platform beneath it. However, you are able to cause the hero to *jump* off of a platform.

The reason for this is the way gravity has been implemented. Colliding with a platform turns gravity "off" so the hero sprite doesn't fall through the platform. The problem is, nothing turns gravity back on when the hero walks off the edge of a platform.

You can force gravity to reactivate by activating gravity during the hero sprite's movement. Edit the movement code in the **update** function of your **Player** class, adding a statement to activate gravity during movement. The two lines you need to add are commented:

```
        if self.movex < 0:
            self.is_jumping = True  # turn gravity on
            self.frame += 1
            if self.frame > 3 * ani:
                self.frame = 0
            self.image = pygame.transform.flip(self.images[self.
                         frame // ani], True, False)

        if self.movex > 0:
            self.is_jumping = True  # turn gravity on
            self.frame += 1
            if self.frame > 3 * ani:
                self.frame = 0
            self.image = self.images[self.frame // ani]
```

This activates gravity long enough to cause the hero sprite to fall to the ground upon a failed platform collision check.

Try your game now. Everything works as expected, but try changing some variables to see what's possible.

In the next article, you'll make your world scroll.

Here's all the code so far:

```
#!/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/>.

import pygame
import sys
import os

'''
Variables
'''

worldx = 960
worldy = 720
```

```
fps = 40
ani = 4
world = pygame.display.set_mode([worldx, worldy])

BLUE = (25, 25, 200)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
ALPHA = (0, 255, 0)

'''
Objects
'''

# x location, y location, img width, img height, img file
class Platform(pygame.sprite.Sprite):
    def __init__(self, xloc, yloc, imgw, imgh, img):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(os.path.join
                    ('images', img)).convert()
        self.image.convert_alpha()
        self.image.set_colorkey(ALPHA)
        self.rect = self.image.get_rect()
        self.rect.y = yloc
        self.rect.x = xloc


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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.movex = 0
        self.movey = 0
        self.frame = 0
        self.health = 10
        self.is_jumping = True
        self.is_falling = True
        self.images = []
        for i in range(1, 5):
            img = pygame.image.load(os.path.join('images',
                  'hero' + str(i) + '.png')).convert()
            img.convert_alpha()
            img.set_colorkey(ALPHA)
            self.images.append(img)
            self.image = self.images[0]
            self.rect = self.image.get_rect()

    def gravity(self):
        if self.is_jumping:
            self.movey += 3.2

    def control(self, x, y):
        """
        control player movement
        """
        self.movex += x

    def jump(self):
        if self.is_jumping is False:
            self.is_falling = False
            self.is_jumping = True

    def update(self):
        """
        Update sprite position
        """

        # moving left
        if self.movex < 0:
            self.is_jumping = True
            self.frame += 1
            if self.frame > 3 * ani:
                self.frame = 0
            self.image = pygame.transform.flip(self.images[self.
                         frame // ani], True, False)

        # moving right
        if self.movex > 0:
            self.is_jumping = True
            self.frame += 1
            if self.frame > 3 * ani:
                self.frame = 0
            self.image = self.images[self.frame // ani]

        # collisions
        enemy_hit_list = pygame.sprite.spritecollide(self,
                         enemy_list, False)
        for enemy in enemy_hit_list:
            self.health -= 1
            # print(self.health)

        ground_hit_list = pygame.sprite.spritecollide(self,
                          ground_list, False)
        for g in ground_hit_list:
            self.movey = 0
            self.rect.bottom = g.rect.top
            self.is_jumping = False  # stop jumping

        # fall off the world
        if self.rect.y > worldy:
            self.health -=1
            print(self.health)
            self.rect.x = tx
            self.rect.y = ty

        plat_hit_list = pygame.sprite.spritecollide(self,
```

```
                    plat_list, False)
        for p in plat_hit_list:
            self.is_jumping = False  # stop jumping
            self.movey = 0
            if self.rect.bottom <= p.rect.bottom:
                self.rect.bottom = p.rect.top
            else:
                self.movey += 3.2

        if self.is_jumping and self.is_falling is False:
            self.is_falling = True
            self.movey -= 33  # how high to jump

        self.rect.x += self.movex
        self.rect.y += self.movey

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
        self.counter = 0

    def move(self):
        '''
        enemy movement
        '''
        distance = 80
        speed = 8

        if self.counter >= 0 and self.counter <= distance:
            self.rect.x += speed
        elif self.counter >= distance and self.counter
            <= distance*2:
            self.rect.x -= speed
        else:
            self.counter = 0

        self.counter += 1


class Level():
    def ground(lvl, gloc, tx, ty):
        ground_list = pygame.sprite.Group()
        i = 0
        if lvl == 1:
            while i < len(gloc):
                ground = Platform(gloc[i], worldy - ty, tx, ty,
                        'tile-ground.png')
                ground_list.add(ground)
                i = i + 1

        if lvl == 2:
            print("Level " + str(lvl) )

        return ground_list

    def bad(lvl, eloc):
        if lvl == 1:
            enemy = Enemy(eloc[0],eloc[1],'enemy.png')
            enemy_list = pygame.sprite.Group()
            enemy_list.add(enemy)
        if lvl == 2:
            print("Level " + str(lvl) )

        return enemy_list

    # x location, y location, img width, img height, img file
    def platform(lvl,tx,ty):
        plat_list = pygame.sprite.Group()
        ploc = []
        i=0
        if lvl == 1:
        ploc.append((200, worldy - ty - 128, 3))
        ploc.append((300, worldy - ty - 256, 3))
        ploc.append((500, worldy - ty - 128 , 4))
        while i < len(ploc):
            j=0
            while j <= ploc[i][2]:
                plat = Platform((ploc[i][0] + (j*tx)),ploc[i]
                       [1], tx, ty, 'tile.png')
                plat_list.add(plat)
                j = j + 1
            print('run' + str(i) + str(ploc[i]))
            i = i + 1

    if lvl == 2:
        print("Level " + str(lvl) )

    return plat_list


'''
Setup
'''

backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
clock = pygame.time.Clock()
pygame.init()
backdropbox = world.get_rect()
```

```
main = True

player = Player()  # spawn player
player.rect.x = 0  # go to x
player.rect.y = 30  # go to y
player_list = pygame.sprite.Group()
player_list.add(player)
steps = 10

eloc = []
eloc = [300, 0]
enemy_list = Level.bad(1, eloc )

gloc = []
tx   = 64
ty   = 64

i = 0
while i <= (worldx / tx) + tx:
    gloc.append(i * tx)
    i = i + 1

ground_list = Level.ground(1, gloc, tx, ty)
plat_list = Level.platform(1, tx, ty)

'''
Main Loop
'''

while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

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

    world.blit(backdrop, backdropbox)
    player.update()
    player_list.draw(world)
    enemy_list.draw(world)
    ground_list.draw(world)
    plat_list.draw(world)
    for e in enemy_list:
        e.move()
    pygame.display.flip()
    clock.tick(fps)
```

Enable your Python game player to run forward and backward

Let your player run free by enabling the side-scroller effect in your Python platformer using the Pygame module.

IN PREVIOUS ENTRIES of this series about creating video games in Python 3 [1] using the Pygame module [2], you designed your level-design layout, but some portion of your level probably extended past your viewable screen. The ubiquitous solution to that problem in platformer games 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 dead zones at either edge of the screen, at which point your avatar stands still while the world scrolls by.

Putting the scroll in side-scroller

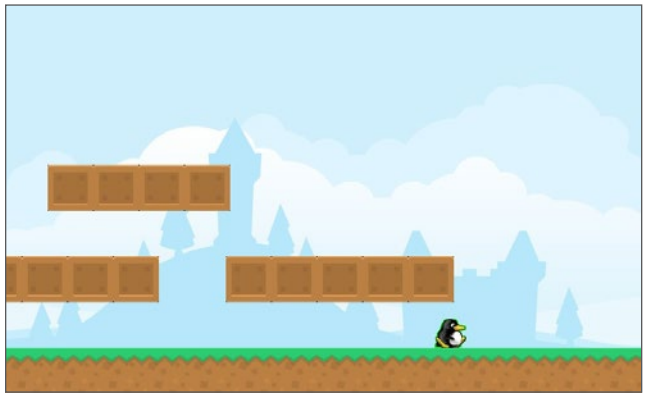
You need one trigger point to go forward and another if you want your player to be able to go backward. These two points are simply two variables. Set them each about 100 or 200 pixels from each screen edge. Create the variables in your variables section:

```
forwardx  = 600
backwardx = 230
```

In the main loop, check to see whether your hero 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 backward. In the following code, the final three lines of code are only for your reference (be careful not to place this code in the for loop checking for keyboard events):

```
        # scroll the world forward
        if player.rect.x >= forwardx:
                scroll = player.rect.x - forwardx
                player.rect.x = forwardx
                for p in plat_list:
                        p.rect.x -= scroll

        # scroll the world backward
        if player.rect.x <= backwardx:
                scroll = backwardx - player.rect.x
                player.rect.x = backwardx
                for p in plat_list:
                        p.rect.x += scroll

    # scrolling code above
    world.blit(backdrop, backdropbox)
    player.gravity() # check gravity
    player.update()
```

Launch your game and try it out.



Scrolling works as expected, but you may notice a small problem that happens when you scroll the world around your player and non-player sprites: the enemy sprite doesn't scroll along with the world. Unless you want your enemy sprite to pursue your player endlessly, you need to modify the enemy code so that when your player makes an expeditious retreat, the enemy is left behind.

Enemy scroll

In your main loop, you must apply the same rules for scrolling platforms to your enemy's position. Because your game

world will (presumably) have more than one enemy in it, the rules are applied to your enemy list rather than an individual enemy sprite. That's one of the advantages of grouping similar elements into lists.

The first two lines are for context, so just add the final two to your main loop:

```
    # scroll the world forward
    if player.rect.x >= forwardx:
        scroll =   ΩAAAAAAAAAAA player.rect.x - forwardx
        player.rect.x = forwardx
        for p in plat_list:
            p.rect.x -= scroll
        for e in enemy_list:    # enemy scroll
            e.rect.x -= scroll  # enemy scroll
```

To scroll in the other direction (again, only add the final two lines to your existing code):

```
    # scroll the world backward
    if player.rect.x <= backwardx:
        scroll = backwardx - player.rect.x
        player.rect.x = backwardx
        for p in plat_list:
            p.rect.x += scroll
        for e in enemy_list:    # enemy scroll
            e.rect.x += scroll  # enemy scroll
```

Launch the game again and see what happens.

Here's all the code you've written for this Python platformer so far:

```
#!/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/>.

import pygame
import sys
import os

'''
Variables
'''

worldx = 960
worldy = 720
fps = 40
ani = 4
world = pygame.display.set_mode([worldx, worldy])
forwardx  = 600
backwardx = 230

BLUE = (25, 25, 200)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
ALPHA = (0, 255, 0)

'''
Objects
'''

# x location, y location, img width, img height, img file
class Platform(pygame.sprite.Sprite):
    def __init__(self, xloc, yloc, imgw, imgh, img):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(os.path.join
                    ('images', img)).convert()
        self.image.convert_alpha()
        self.image.set_colorkey(ALPHA)
        self.rect = self.image.get_rect()
        self.rect.y = yloc
        self.rect.x = xloc


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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.movex = 0
        self.movey = 0
        self.frame = 0
        self.health = 10
        self.is_jumping = True
        self.is_falling = True
        self.images = []
        for i in range(1, 5):
            img = pygame.image.load(os.path.join('images',
                  'hero' + str(i) + '.png')).convert()
            img.convert_alpha()
            img.set_colorkey(ALPHA)
            self.images.append(img)
```

```
        self.image = self.images[0]
        self.rect = self.image.get_rect()

    def gravity(self):
        if self.is_jumping:
            self.movey += 3.2

    def control(self, x, y):
        """
        control player movement
        """
        self.movex += x

    def jump(self):
        if self.is_jumping is False:
            self.is_falling = False
            self.is_jumping = True

    def update(self):
        """
        Update sprite position
        """

        # moving left
        if self.movex < 0:
            self.is_jumping = True
            self.frame += 1
            if self.frame > 3 * ani:
                self.frame = 0
            self.image = pygame.transform.flip(self.images[self.
                         frame // ani], True, False)

        # moving right
        if self.movex > 0:
            self.is_jumping = True
            self.frame += 1
            if self.frame > 3 * ani:
                self.frame = 0
            self.image = self.images[self.frame // ani]

        # collisions
        enemy_hit_list = pygame.sprite.spritecollide(self,
                         enemy_list, False)
        for enemy in enemy_hit_list:
            self.health -= 1
            # print(self.health)

        ground_hit_list = pygame.sprite.spritecollide(self,
                          ground_list, False)
        for g in ground_hit_list:
            self.movey = 0
            self.rect.bottom = g.rect.top
            self.is_jumping = False  # stop jumping

        # fall off the world
        if self.rect.y > worldy:
            self.health -=1
            print(self.health)
            self.rect.x = tx
            self.rect.y = ty

        plat_hit_list = pygame.sprite.spritecollide(self,
                        plat_list, False)
        for p in plat_hit_list:
            self.is_jumping = False  # stop jumping
            self.movey = 0
            if self.rect.bottom <= p.rect.bottom:
                self.rect.bottom = p.rect.top
            else:
                self.movey += 3.2

        if self.is_jumping and self.is_falling is False:
            self.is_falling = True
            self.movey -= 33  # how high to jump

        self.rect.x += self.movex
        self.rect.y += self.movey

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
        self.counter = 0

    def move(self):
        '''
        enemy movement
        '''
        distance = 80
        speed = 8

        if self.counter >= 0 and self.counter <= distance:
            self.rect.x += speed
        elif self.counter >= distance and self.counter
            <= distance*2:
            self.rect.x -= speed
        else:
            self.counter = 0
```

```
        self.counter += 1


class Level():
    def ground(lvl, gloc, tx, ty):
        ground_list = pygame.sprite.Group()
        i = 0
        if lvl == 1:
            while i < len(gloc):
                ground = Platform(gloc[i], worldy - ty, tx, ty,
                         'tile-ground.png')
                ground_list.add(ground)
                i = i + 1

        if lvl == 2:
            print("Level " + str(lvl) )

        return ground_list

    def bad(lvl, eloc):
        if lvl == 1:
            enemy = Enemy(eloc[0],eloc[1],'enemy.png')
            enemy_list = pygame.sprite.Group()
            enemy_list.add(enemy)
        if lvl == 2:
            print("Level " + str(lvl) )

        return enemy_list

    # x location, y location, img width, img height, img file
    def platform(lvl,tx,ty):
        plat_list = pygame.sprite.Group()
        ploc = []
        i=0
        if lvl == 1:
        ploc.append((200, worldy - ty - 128, 3))
        ploc.append((300, worldy - ty - 256, 3))
        ploc.append((500, worldy - ty - 128 , 4))
        while i < len(ploc):
            j=0
            while j <= ploc[i][2]:
                plat = Platform((ploc[i][0] + (j*tx)),ploc[i]
                       [1], tx, ty, 'tile.png')
                plat_list.add(plat)
                j = j + 1
            print('run' + str(i) + str(ploc[i]))
            i = i + 1

    if lvl == 2:
        print("Level " + str(lvl) )

    return plat_list
```

```
'''
Setup
'''

backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
clock = pygame.time.Clock()
pygame.init()
backdropbox = world.get_rect()
main = True

player = Player()  # spawn player
player.rect.x = 0  # go to x
player.rect.y = 30  # go to y
player_list = pygame.sprite.Group()
player_list.add(player)
steps = 10

eloc = []
eloc = [300, 0]
enemy_list = Level.bad(1, eloc )

gloc = []
tx   = 64
ty   = 64

i = 0
while i <= (worldx / tx) + tx:
    gloc.append(i * tx)
    i = i + 1

ground_list = Level.ground(1, gloc, tx, ty)
plat_list = Level.platform(1, tx, ty)

'''
Main Loop
'''

while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

        if event.type == pygame.KEYDOWN:
            if event.key == ord('q'):
                pygame.quit()
                try:
                    sys.exit()
                finally:
                    main = False
            if event.key == pygame.K_LEFT or event.key == ord('a'):
```

```
            player.control(-steps, 0)
        if event.key == pygame.K_RIGHT or event.key == ord('d'):
            player.control(steps, 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(steps, 0)
        if event.key == pygame.K_RIGHT or event.key == ord('d'):
             player.control(-steps, 0)

# scroll the world forward
if player.rect.x >= forwardx:
        scroll = player.rect.x - forwardx
        player.rect.x = forwardx
        for p in plat_list:
                p.rect.x -= scroll
    for e in enemy_list:    # enemy scroll
        e.rect.x -= scroll  # enemy scroll

# scroll the world backward
if player.rect.x <= backwardx:
            scroll = backwardx - player.rect.x
            player.rect.x = backwardx
            for p in plat_list:
                    p.rect.x += scroll
        for e in enemy_list:    # enemy scroll
            e.rect.x += scroll  # enemy scroll

    world.blit(backdrop, backdropbox)
    player.update()
    player.gravity()
    player_list.draw(world)
    enemy_list.draw(world)
    ground_list.draw(world)
    plat_list.draw(world)
    for e in enemy_list:
        e.move()
    pygame.display.flip()
    clock.tick(fps)
```

Links

[1] https://www.python.org/

[2] https://www.pygame.org/news

Put some loot in your Python platformer game

Give your players some treasures to collect and boost their score in this installment on programming video games with Python's Pygame module.

IF YOU'VE FOLLOWED along with the previous articles in this series, then you know all the basics of programming video game mechanics. You can build upon these basics to create a fully functional video game all your own. Following a "recipe" like the code samples in this series is helpful when you're first learning, but eventually, the recipe becomes a constraint. It's time to use the principles you've learned and apply them in new ways.

If that sounds easier said than done, this article demonstrates an example of how to leverage what you already know for new purposes. Specifically, it covers how to implement a looting system using what you have already learned about platforms from previous lessons.

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 provides information leading to your next quest.

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

Before you begin, you must have a loot graphic, such as a coin or a treasure chest. If you've already downloaded my recommended tile set, the simplified-platformer-pack from Kenney.nl [1], then you can use a diamond or key from that.

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 differ from level to level, create a new function called **loot** in your **Level** class, if you don't already have one. Since loot items are not platforms, you must also create a new **loot_list** group and then add loot objects to it. As with platforms, ground, and enemies, this group is used when checking for collisions:

```
def loot(lvl):
    if lvl == 1:
        loot_list = pygame.sprite.Group()
        loot = Platform(tx*9, ty*5, tx, ty, 'loot_1.png')
        loot_list.add(loot)

    if lvl == 2:
        print(lvl)

    return loot_list
```

In this code, I express the location of the loot as multiples of the tile size: **tx** on the X axis and **ty** for the Y axis. I do this because i mapped my level on graph paper, so it's easy to just count the squares on my map and then multiply it by the tile size, rather than calculating the pixel count. This is especially true for very long levels. You can hard code the pixel count, if you prefer.

You can add as many loot objects as you like; just remember to add each one to your loot list. The arguments for the **Platform** class are the X position, the Y position, the width and height of the loot sprite (it's usually easiest to keep your loot sprite the same size as all other tiles), and the image you want to use as loot. Placement of loot can be just as complex as mapping platforms, so use the level design document you created when creating the level.

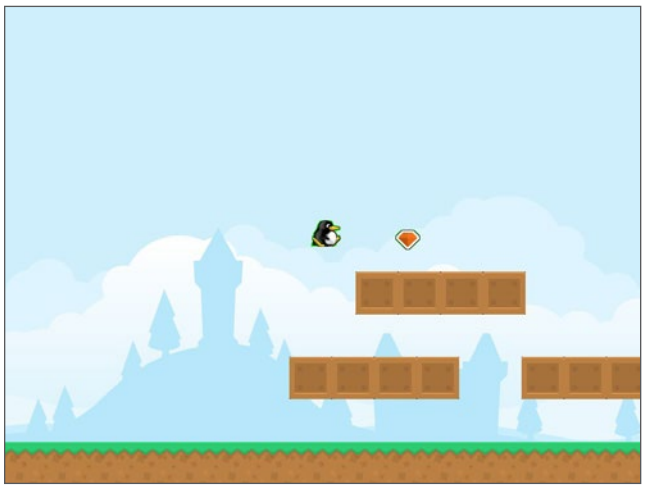
Call your new loot function in the **Setup** section of your script. In the following code, the first three lines are for context, so just add the fourth:

```
loot_list = Level.loot(1)
```

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

```
    loot_list.draw(world)
```

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 loot forward, add the last two lines:

```
for e in enemy_list:
    e.rect.x -= scroll
for l in loot_list:     # loot scroll
    l.rect.x -= scroll  # loot scroll
```

To scroll it backward, add the last two lines:

```
for e in enemy_list:
    e.rect.x += scroll
for l in loot_list:     # loot scroll
    l.rect.x += scroll  # loot 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

As with platforms and enemies, you can check for collisions between loot and your player. The logic is the same as other collisions, except that a hit doesn't (necessarily) affect gravity or health. Instead, a hit causes the loot to disappear and increment the player's score.

When your player touches a loot object, you can remove that object from the **loot_list**. This means that when your main loop redraws all loot items in **loot_list**, it won't redraw that particular object, so it will look like the player has grabbed the loot.

Add the following code above the platform collision detection in the **update** function of your **Player** class (the last line is just 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)

        plat_hit_list = pygame.sprite.spritecollide(self,
                        plat_list, False)
```

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

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

Applying what you know

As you can see, you've got all the basics. All you have to do now is use what you know in new ways. For instance, if you haven't already placed your enemies in a sensible place, take some time to do that now using the same method you've used to place platforms and loot.

There are a few more tips in the next article, but in the meantime, use what you've learned to make a few simple, single-level games. Limiting the scope of what you are trying to create is important so that you don't overwhelm yourself. It also makes it easier to end up with a finished product that looks and feels finished.

Here's all the code you've written for this Python platformer so far:

```
#!/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/>.
```

```
import pygame
import sys
import os

'''
Variables
'''

worldx = 960
worldy = 720
fps = 40
ani = 4
world = pygame.display.set_mode([worldx, worldy])
forwardx  = 600
backwardx = 120

BLUE = (25, 25, 200)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
ALPHA = (0, 255, 0)

'''
Objects
'''

# x location, y location, img width, img height, img file
class Platform(pygame.sprite.Sprite):
    def __init__(self, xloc, yloc, imgw, imgh, img):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(os.path.join
                    ('images', img)).convert()
        self.image.convert_alpha()
        self.image.set_colorkey(ALPHA)
        self.rect = self.image.get_rect()
        self.rect.y = yloc
        self.rect.x = xloc


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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.movex = 0
        self.movey = 0
        self.frame = 0
        self.health = 10
        self.is_jumping = True
        self.is_falling = True
        self.images = []
        for i in range(1, 5):
            img = pygame.image.load(os.path.join('images',
                  'hero' + str(i) + '.png')).convert()
            img.convert_alpha()
            img.set_colorkey(ALPHA)
            self.images.append(img)
            self.image = self.images[0]
            self.rect = self.image.get_rect()

    def gravity(self):
        if self.is_jumping:
            self.movey += 3.2

    def control(self, x, y):
        """
        control player movement
        """
        self.movex += x

    def jump(self):
        if self.is_jumping is False:
            self.is_falling = False
            self.is_jumping = True

    def update(self):
        """
        Update sprite position
        """

        # moving left
        if self.movex < 0:
            self.is_jumping = True
            self.frame += 1
            if self.frame > 3 * ani:
                self.frame = 0
            self.image = pygame.transform.flip(self.images[self.
                         frame // ani], True, False)

        # moving right
        if self.movex > 0:
            self.is_jumping = True
            self.frame += 1
            if self.frame > 3 * ani:
                self.frame = 0
            self.image = self.images[self.frame // ani]

        # collisions
        enemy_hit_list = pygame.sprite.spritecollide(self,
                         enemy_list, False)
        for enemy in enemy_hit_list:
            self.health -= 1
            # print(self.health)

        ground_hit_list = pygame.sprite.spritecollide(self,
                          ground_list, False)
        for g in ground_hit_list:
            self.movey = 0
            self.rect.bottom = g.rect.top
            self.is_jumping = False  # stop jumping

        # fall off the world
        if self.rect.y > worldy:
            self.health -=1
            print(self.health)
```

```
            self.rect.x = tx
            self.rect.y = ty

        plat_hit_list = pygame.sprite.spritecollide(self,
                        plat_list, False)
        for p in plat_hit_list:
            self.is_jumping = False  # stop jumping
            self.movey = 0
            if self.rect.bottom <= p.rect.bottom:
               self.rect.bottom = p.rect.top
            else:
               self.movey += 3.2

        if self.is_jumping and self.is_falling is False:
            self.is_falling = True
            self.movey -= 33  # how high to jump

        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)

        plat_hit_list = pygame.sprite.spritecollide(self,
                        plat_list, False)

        self.rect.x += self.movex
        self.rect.y += self.movey

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
        self.counter = 0

    def move(self):
        '''
        enemy movement
        '''
        distance = 80
        speed = 8

        if self.counter >= 0 and self.counter <= distance:
            self.rect.x += speed
        elif self.counter >= distance and self.counter
            <= distance*2:
            self.rect.x -= speed
        else:
            self.counter = 0

        self.counter += 1


class Level():
    def ground(lvl, gloc, tx, ty):
        ground_list = pygame.sprite.Group()
        i = 0
        if lvl == 1:
            while i < len(gloc):
                ground = Platform(gloc[i], worldy - ty, tx, ty,
                         'tile-ground.png')
                ground_list.add(ground)
                i = i + 1

        if lvl == 2:
            print("Level " + str(lvl) )

        return ground_list

    def bad(lvl, eloc):
        if lvl == 1:
            enemy = Enemy(eloc[0],eloc[1],'enemy.png')
            enemy_list = pygame.sprite.Group()
            enemy_list.add(enemy)
        if lvl == 2:
            print("Level " + str(lvl) )

        return enemy_list

    # x location, y location, img width, img height, img file
    def platform(lvl,tx,ty):
        plat_list = pygame.sprite.Group()
        ploc = []
        i=0
        if lvl == 1:
            ploc.append((200, worldy - ty - 128, 3))
            ploc.append((300, worldy - ty - 256, 3))
            ploc.append((500, worldy - ty - 128 , 4))
            while i < len(ploc):
                j=0
                while j <= ploc[i][2]:
                    plat = Platform((ploc[i][0] + (j*tx)),
                           ploc[i][1], tx, ty, 'tile.png')
                    plat_list.add(plat)
                    j = j + 1
                print('run' + str(i) + str(ploc[i]))
                i = i + 1

        if lvl == 2:
            print("Level " + str(lvl) )

        return plat_list

    def loot(lvl):
        if lvl == 1:
            loot_list = pygame.sprite.Group()
```

```
            loot = Platform(tx*5, ty*5, tx, ty, 'loot_1.png')
            loot_list.add(loot)

        if lvl == 2:
            print(lvl)

        return loot_list


'''
Setup
'''

backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
clock = pygame.time.Clock()
pygame.init()
backdropbox = world.get_rect()
main = True

player = Player()  # spawn player
player.rect.x = 0  # go to x
player.rect.y = 30  # go to y
player_list = pygame.sprite.Group()
player_list.add(player)
steps = 10

eloc = []
eloc = [300, 0]
enemy_list = Level.bad(1, eloc )

gloc = []
tx   = 64
ty   = 64

i = 0
while i <= (worldx / tx) + tx:
    gloc.append(i * tx)
    i = i + 1

ground_list = Level.ground(1, gloc, tx, ty)
plat_list = Level.platform(1, tx, ty)
enemy_list = Level.bad( 1, eloc )
loot_list = Level.loot(1)


'''
Main Loop
'''

while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

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

    # scroll the world forward
    if player.rect.x >= forwardx:
            scroll = player.rect.x - forwardx
            player.rect.x = forwardx
            for p in plat_list:
                    p.rect.x -= scroll
        for e in enemy_list:
            e.rect.x -= scroll

    # scroll the world backward
    if player.rect.x <= backwardx:
            scroll = backwardx - player.rect.x
            player.rect.x = backwardx
            for p in plat_list:
                    p.rect.x += scroll
        for e in enemy_list:
            e.rect.x += scroll
        for l in loot_list:
            l.rect.x += scroll

    world.blit(backdrop, backdropbox)
    player.update()
    player.gravity()
    player_list.draw(world)
    enemy_list.draw(world)
    loot_list.draw(world)
    ground_list.draw(world)
    plat_list.draw(world)
    for e in enemy_list:
        e.move()
    pygame.display.flip()
    clock.tick(fps)
```

Links

[1] https://kenney.nl/assets/simplified-platformer-pack

Add scorekeeping to your Python game

In the eleventh article in this series on programming with Python's Pygame module, display your game player's score when they collect loot or take damage.

IF YOU'VE FOLLOWED along with this series, you've learned all the essential syntax and patterns you need to create a video game with Python. However, it still lacks one vital component. This component isn't important just for programming games in Python; it's something you must master no matter what branch of computing you explore: Learning new tricks as a programmer by reading a language's or library's documentation.

Luckily, the fact that you're reading this article is a sign that you're comfortable with documentation. For the practical purpose of making your platform game more polished, in this article, you will add a score and health display to your game screen. But the not-so-secret agenda of this lesson is to teach you how to find out what a library offers and how you can use new features.

Displaying the score in Pygame

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. You can also track the player's health so that when they hit one of the enemies, it has a consequence.

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

Read the docs

Most Python modules have documentation, and even those that do not can be minimally documented by Python's Help function. Pygame's main page [1] links to its documentation. However, Pygame is a big module with a lot of documentation, and its docs aren't exactly written in the same approachable (and friendly and elucidating and helpful) narrative style as articles on Opensource.com. They're technical documents, and they list each class and function available in the module, what kind of inputs each expects, and so on. If you're not comfortable referring to descriptions of code components, this can be overwhelming.

The first thing to do, before bothering with a library's documentation, is to think about what you are trying to achieve. In this case, you want to display the player's score and health on the screen.

Once you've determined your desired outcome, think about what components are required for it. You can think of this in terms of variables and functions or, if that doesn't come naturally to you yet, you can think generically. You probably recognize that displaying a score requires some text, which you want Pygame to draw on the screen. If you think it through, you might realize that it's not very different from rendering a player or loot or a platform on screen.

Technically, you *could* use graphics of numbers and have Pygame display those. It's not the easiest way to achieve your goal, but if it's the only way you know, then it's a valid way. However, if you refer to Pygame's docs, you see that one of the modules listed is **font**, which is Pygame's method for making printing text on the screen as easy as typing.

Deciphering technical documentation

The font documentation page starts with **pygame.font.init()**, which it lists as the function that is used to initialize the font module. It's called automatically by **pygame.init()**, which you already call in your code. Once again, you've reached a point that that's technically *good enough*. While you don't know *how* yet, you know that you can use the **pygame.font** functions to print text on the screen.

If you read further, however, you find that there's yet an even better way to print fonts. The **pygame.freetype** module is described in the docs this way:

> The pygame.freetype module is a replacement for pygame.fontpygame module for loading and rendering fonts. It has all of the functionality of the original, plus many new features.

Further down the **pygame.freetype** documentation page, there's some sample code:

```
import pygame
import pygame.freetype
```

Your code already imports Pygame, but modify your **import** statements to include the Freetype module:

```
import pygame
import sys
import os
import pygame.freetype
```

Using a font in Pygame

From the description of the font modules, it's clear that Pygame uses a font, whether it's one you provide or a default font built into Pygame, to render text on the screen. Scroll through the **pygame.freetype** documentation to find the **pygame.freetype.Font** function:

```
pygame.freetype.Font
Create a new Font instance from a supported font file.

Font(file, size=0, font_index=0, resolution=0, ucs4=False) -> Font

pygame.freetype.Font.name
  Proper font name.

pygame.freetype.Font.path
  Font file path

pygame.freetype.Font.size
  The default point size used in rendering
```

This describes how to construct a font "object" in Pygame. It may not feel natural to you to think of a simple object onscreen as the combination of several code attributes, but it's very similar to how you built your hero and enemy sprites. Instead of an image file, you need a font file. Once you have a font file, you can create a font object in your code with the **pygame.freetype.Font** function and then use that object to render text on the screen.

Asset management

Because not everyone in the world has the exact same fonts on their computers, it's important to bundle your chosen font with your game. To bundle a font, 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 permits you to give the font away along with your game.

Sites that specialize in free and legal fonts include:

- Font Library
- Font Squirrel
- League of Moveable Type

When you find a font that you like, download it. Extract the ZIP or TAR [2] file and move the **.ttf** or **.otf** file into the **fonts** folder in your game project directory.

You aren't installing the font on your computer. You're just placing it in your game's **fonts** folder so that Pygame can use it. You *can* install the font on your computer if you want, but it's not necessary. The important thing is to have it in your game directory, so Pygame can "trace" it onto the screen.

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.

Using a font in Pygame

Now tell Pygame about your font. From the documentation, you know that you'll get a font object in return when you provide at least the path to a font file to **pygame.freetype.Font** (the docs state explicitly that all remaining attributes are optional):

```
Font(file, size=0, font_index=0, resolution=0, ucs4=False) -> Font
```

Create a new variable called **myfont** to serve as your font in the game, and place the results of the **Font** function into that variable. This example uses the **amazdoom.ttf** font, but you can use whatever font you want. Place this code in your Setup section:

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

Displaying text in Pygame

Now that you've created a font object, you need a function to draw the text you want onto the screen. This is the same principle you used to draw the background and platforms in your game.

First, create a function, and use the **myfont** object to create some text, setting the color to some RGB value. This must be a global function; it does not belong to any specific class. Place it in the **objects** section of your code, but keep it as a stand-alone function:

```
def stats(score,health):
    myfont.render_to(world, (4, 4), "Score:"+str(score), BLACK,
                     None, size=64)
```

```
    myfont.render_to(world, (4, 72), "Health:"+str(health),
                     BLACK, None, size=64)
```

Of course, you know by now that nothing happens in your game if it's not in the Main loop, so add a call to your **stats** function near the bottom of the file:

```
    stats(player.score,player.health) # draw text
```

Try your game. If you've been following the sample code in this article exactly, you'll get an error when you try to launch the game now.

Interpreting errors

Errors are important to programmers. When something fails in your code, one of the best ways to understand why is by reading the error output. Unfortunately, Python doesn't communicate the same way a human does. While it does have relatively friendly errors, you still have to interpret what you're seeing.

In this case, launching the game produces this output:

```
Traceback (most recent call last):
  File "/home/tux/PycharmProjects/game_001/main.py", line 41,
    in <module>
    font_size = tx
NameError: name 'tx' is not defined
```

Python is aserting that the variable **tx** is not defined. You know this isn't true, because you've used **tx** in several places by now and it's worked as expected.

But Python also cites a line number. This is the line that caused Python to stop executing the code. It is *not* necessarily the line containing the error.

Armed with this knowledge, you can look at your code in an attempt to understand what has failed.

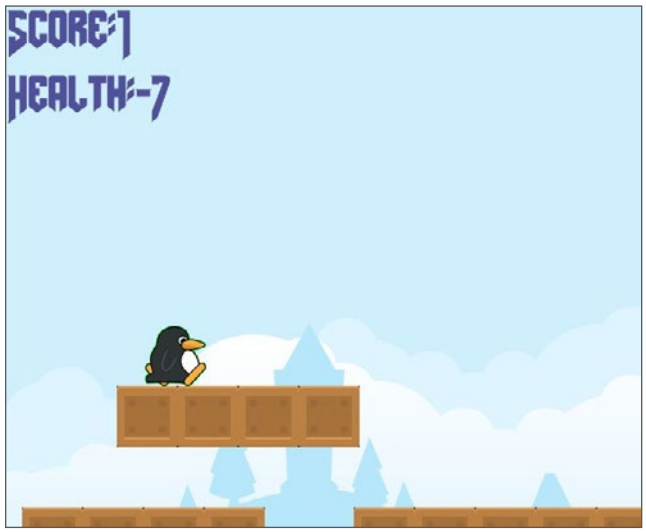
Line 41 attempts to set the font size to the value of **tx**. However, reading through the file in reverse, up from line 41, you might notice that **tx** (and **ty**) are not listed. In fact, **tx** and **ty** were placed haphazardly in your setup section because, at the time, it seemed easy and logical to place them along with other important tile information.

Moving the **tx** and **ty** lines from your setup section to some line above line 41 fixes the error.

When you entcounter errors in Python, take note of the hints it provides, and then read your source code carefully. It can take time to find an error, even for experienced programmers, but the better you understand Python the easier it becomes.

Running the 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 not fair. You have just discovered a non-fatal bug. Non-fatal bugs are those little problems in applications that don't keep the application from starting up or even from working (mostly), but they either don't make sense, or they annoy the user. Here's how to fix this one.

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* separate events: when the player and enemy collide and, once they have collided, when they stop colliding.

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

```
        self.frame = 0
        self.health = 10
        self.damage = 0
```

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

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

You might see similarities between 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 runs only if the player is not *currently* being hit. That means that this code runs once when a player and enemy collide and not constantly for as long as the collision happens, the way it used to.

The new code uses two new Pygame functions. The **self.rect.contains** function 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 one when it is true, no matter how many times it is true.

Now even three seconds of getting hit by an enemy still looks like one hit to Pygame.

I discovered these functions by reading through Pygame's documentation. You don't have to read all the docs at once, and you don't have to read every word of each function. However, it's important to spend time with the documentation of a new library or module that you're using; otherwise, you run a high risk of reinventing the wheel. Don't spend an afternoon trying to hack together a solution to something that's already been solved by the framework you're using. Read the docs, find the functions, and benefit from the work of others!

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

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

Notice that this new code gets triggered *only* when 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 runs, it uses **self.rect.collidelist** to see whether or not the player is *still* touching an enemy in your enemy list (**collidelist** returns negative one when 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 zero and subtract one point of health.

Try your game now.

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 zero 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.

Level up

You already know how to do so much. Now it's time to level up your skills. Go skim the documentation for new tricks and try them out on your own. Programming is a skill you develop, so don't stop with this project. Invent another game, or a useful application, or just use Python to experiment around with crazy ideas. The more you use it, the more comfortable you get with it, and eventually it'll be second nature.

Keep it going, and keep it open!

Here's all the code so far:

```
#!/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/>.
```

```
import pygame
import pygame.freetype
import sys
import os

'''
Variables
'''

worldx = 960
worldy = 720
fps = 40
ani = 4
world = pygame.display.set_mode([worldx, worldy])
forwardx  = 600
backwardx = 120
BLUE = (80, 80, 155)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
ALPHA = (0, 255, 0)

tx = 64
ty = 64

font_path = os.path.join(os.path.dirname(os.path.realpath
 (__file__)),"fonts","amazdoom.ttf")
font_size = tx
pygame.freetype.init()
myfont = pygame.freetype.Font(font_path, font_size)


'''
Objects
'''

def stats(score,health):
    myfont.render_to(world, (4, 4), "Score:"+str(score), BLUE,
                     None, size=64)
    myfont.render_to(world, (4, 72), "Health:"+str(health),
                     BLUE, None, size=64)

# x location, y location, img width, img height, img file
class Platform(pygame.sprite.Sprite):
    def __init__(self, xloc, yloc, imgw, imgh, img):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(os.path.join
                     ('images', img)).convert()
        self.image.convert_alpha()
        self.image.set_colorkey(ALPHA)
        self.rect = self.image.get_rect()
        self.rect.y = yloc
        self.rect.x = xloc
```

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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.movex = 0
        self.movey = 0
        self.frame = 0
        self.health = 10
        self.damage = 0
        self.score = 0
        self.is_jumping = True
        self.is_falling = True
        self.images = []
        for i in range(1, 5):
            img = pygame.image.load(os.path.join('images',
                  'hero' + str(i) + '.png')).convert()
            img.convert_alpha()
            img.set_colorkey(ALPHA)
            self.images.append(img)
            self.image = self.images[0]
            self.rect = self.image.get_rect()

    def gravity(self):
        if self.is_jumping:
            self.movey += 3.2

    def control(self, x, y):
        """
        control player movement
        """
        self.movex += x

    def jump(self):
        if self.is_jumping is False:
            self.is_falling = False
            self.is_jumping = True

    def update(self):
        """
        Update sprite position
        """

        # moving left
        if self.movex < 0:
            self.is_jumping = True
            self.frame += 1
            if self.frame > 3 * ani:
                self.frame = 0
            self.image = pygame.transform.flip(self.images[self.
                         frame // ani], True, False)
```

```
# moving right
if self.movex > 0:
    self.is_jumping = True
    self.frame += 1
    if self.frame > 3 * ani:
        self.frame = 0
    self.image = self.images[self.frame // ani]

# collisions
enemy_hit_list = pygame.sprite.spritecollide(self,
                 enemy_list, False)
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.health -= 1  # subtract 1 hp

ground_hit_list = pygame.sprite.spritecollide(self,
                  ground_list, False)
for g in ground_hit_list:
    self.movey = 0
    self.rect.bottom = g.rect.top
    self.is_jumping = False  # stop jumping

# fall off the world
if self.rect.y > worldy:
    self.health -=1
    print(self.health)
    self.rect.x = tx
    self.rect.y = ty

plat_hit_list = pygame.sprite.spritecollide(self,
                plat_list, False)
for p in plat_hit_list:
    self.is_jumping = False  # stop jumping
    self.movey = 0
    if self.rect.bottom <= p.rect.bottom:
       self.rect.bottom = p.rect.top
    else:
       self.movey += 3.2

if self.is_jumping and self.is_falling is False:
    self.is_falling = True
    self.movey -= 33  # how high to jump

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)

        plat_hit_list = pygame.sprite.spritecollide(self,
                        plat_list, False)

        self.rect.x += self.movex
        self.rect.y += self.movey

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
        self.counter = 0

    def move(self):
        '''
        enemy movement
        '''
        distance = 80
        speed = 8

        if self.counter >= 0 and self.counter <= distance:
            self.rect.x += speed
        elif self.counter >= distance and self.counter
            <= distance*2:
            self.rect.x -= speed
        else:
            self.counter = 0

        self.counter += 1


class Level:
    def ground(lvl, gloc, tx, ty):
        ground_list = pygame.sprite.Group()
        i = 0
        if lvl == 1:
            while i < len(gloc):
                ground = Platform(gloc[i], worldy - ty, tx, ty,
                         'tile-ground.png')
                ground_list.add(ground)
                i = i + 1

        if lvl == 2:
            print("Level " + str(lvl) )
```

```
        return ground_list

    def bad(lvl, eloc):
        if lvl == 1:
            enemy = Enemy(eloc[0],eloc[1],'enemy.png')
            enemy_list = pygame.sprite.Group()
            enemy_list.add(enemy)
        if lvl == 2:
            print("Level " + str(lvl) )

        return enemy_list

    # x location, y location, img width, img height, img file
    def platform(lvl,tx,ty):
        plat_list = pygame.sprite.Group()
        ploc = []
        i=0
        if lvl == 1:
            ploc.append((200, worldy - ty - 128, 3))
            ploc.append((300, worldy - ty - 256, 3))
            ploc.append((500, worldy - ty - 128 , 4))
            while i < len(ploc):
                j=0
                while j <= ploc[i][2]:
                    plat = Platform((ploc[i][0] + (j*tx)),
                           ploc[i][1], tx, ty, 'tile.png')
                    plat_list.add(plat)
                    j = j + 1
                print('run' + str(i) + str(ploc[i]))
                i = i + 1

        if lvl == 2:
            print("Level " + str(lvl))

        return plat_list

    def loot(lvl):
        if lvl == 1:
            loot_list = pygame.sprite.Group()
            loot = Platform(tx*5, ty*5, tx, ty, 'loot_1.png')
            loot_list.add(loot)

        if lvl == 2:
            print(lvl)

        return loot_list


'''
Setup
'''

backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
clock = pygame.time.Clock()
pygame.init()
backdropbox = world.get_rect()
main = True

player = Player()  # spawn player
player.rect.x = 0  # go to x
player.rect.y = 30  # go to y
player_list = pygame.sprite.Group()
player_list.add(player)
steps = 10

eloc = []
eloc = [300, worldy-ty-80]
enemy_list = Level.bad(1, eloc )
gloc = []

i = 0
while i <= (worldx / tx) + tx:
    gloc.append(i * tx)
    i = i + 1

ground_list = Level.ground(1, gloc, tx, ty)
plat_list = Level.platform(1, tx, ty)
enemy_list = Level.bad( 1, eloc )
loot_list = Level.loot(1)


'''
Main Loop
'''

while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

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

# scroll the world forward
if player.rect.x >= forwardx:
        scroll = player.rect.x - forwardx
        player.rect.x = forwardx
        for p in plat_list:
            p.rect.x -= scroll
    for e in enemy_list:
        e.rect.x -= scroll
    for l in loot_list:
        l.rect.x -= scroll

# scroll the world backward
if player.rect.x <= backwardx:
        scroll = backwardx - player.rect.x
        player.rect.x = backwardx
        for p in plat_list:
            p.rect.x += scroll
    for e in enemy_list:
        e.rect.x += scroll
    for l in loot_list:
        l.rect.x += scroll

world.blit(backdrop, backdropbox)
player.update()
player.gravity()
player_list.draw(world)
enemy_list.draw(world)
loot_list.draw(world)
ground_list.draw(world)
plat_list.draw(world)
for e in enemy_list:
    e.move()
stats(player.score, player.health)
pygame.display.flip()
clock.tick(fps)
```

Links

[1] http://pygame.org/news
[2] https://opensource.com/article/17/7/how-unzip-targz-file

Add throwing mechanics to your Python game

Running around avoiding enemies is one thing. Fighting back is another. Learn how in the 12th article in this series on creating a platformer in Pygame.

MY PREVIOUS ARTICLE was meant to be the final article in this series, and it encouraged you to go program your own additions to this game. Many of you did! I got emails asking for help with a common mechanic that I hadn't yet covered: combat. After all, jumping to avoid baddies is one thing, but sometimes it's awfully satisfying to just make them go away. It's common in video games to throw something at your enemies, whether it's a ball of fire, an arrow, a bolt of lightning, or whatever else fits the game.

Unlike anything you have programmed for your platformer game in this series 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 like that, it may disappear when it passes the edge of the screen. If it's a fireball or a bolt of lightning, it might fizzle out after some amount of time.

That means each time a throwable item is spawned, a unique measure of its lifespan must also be spawned. To introduce this concept, this article demonstrates how to throw only one item at a time. (In other words, only one throwable item may exist at a time.) On the one hand, this is a game limitation, but on the other hand, it is a game mechanic in itself. Your player won't be able to throw 50 fireballs at once, since you only allow one at a time, so it becomes a challenge for your player to time when they release a fireball to try to hit an enemy. And behind the scenes, this also keeps your code simple.

If you want to enable more throwable items at once, challenge yourself after you finish this tutorial by building on the knowledge you gain.

Create the throwable class

If you followed along with the other articles in this series, you should be familiar with the basic `__init__` function when spawning a new object on the screen. It's the same function you used for spawning your player [1] and your enemies [2]. Here's an `__init__` function to spawn a throwable object:

```
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 your `Player` class or Enemy class `__init__` function 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 is set to `1`.

Measure 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 lifespan of a throwable object is to detect when it goes off-screen. Which screen edge you need to 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 arrow or a very fast magical force, then you want to monitor the horizontal limit of your game screen. This is defined by `worldx`.
- 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 `worldy`.

This example assumes your throwable object goes a little forward and eventually falls to the ground. 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,worldy):
        '''
        throw physics
        '''
        if self.rect.y < worldy: #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.

Set up your throwable object

Just like 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 there isn't a throwable object when the game starts, the first time a player attempts to throw a weapon, it will fail.

This example assumes your player starts with a fireball as a weapon, so each instance of a throwable object is designated by the `fire` variable. In later levels, as the player acquires new skills, you could introduce a new variable 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 retype them:

```
player_list = pygame.sprite.Group() #context
player_list.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.

Get throwing in the main loop

Code that doesn't appear in the main loop will not be used in the game, so you need to add a few things in your main loop to get your throwable object into your game world.

First, add player controls. Currently, you have no firepower trigger. There are two states for a key on a keyboard: the key can be down, or the key can be up. For movement, you use both: pressing down starts the player moving, and releasing the key (the key is up) stops the player. Firing needs only one signal. It's a matter of taste as to which key event (a key press or a key release) you use to trigger your throwable object.

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

```
            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)
```

Unlike the fireball you created in your setup section, you use a `1` to set `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 `player_list.draw` lines:

```
enemy.move()  # context

if fire.firing:
    fire.update(worldy)
    firepower.draw(world)
player_list.draw(screen)  # context
enemy_list.draw(screen)   # context
```

Notice that these updates are performed only 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.

Detect collisions

If you played your game with the new throwing mechanic, you probably noticed that you can throw objects, but it doesn't have any effect on your foes.

The reason 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(worldy)                      # context
        firepower.draw(screen)                    # context
        enemy_list.update(firepower,enemy_list) # update enemy
```

You can try your game now, and most everything works as expected. There's still one problem, though, and that's the direction of the throw.

Change the throw mechanic direction

Currently, your hero's fireball moves only to the right. This is because the `update` function of the `Throwable` class adds pixels to the position of the fireball, and in Pygame, a larger number on the X-axis means movement toward the right of the screen. When your hero turns the other way, you probably want it to throw its fireball to the left.

By this point, you know how to implement this, at least technically. However, the easiest solution uses a variable in what may be a new way for you. Generically, you can "set a flag" (sometimes also termed "flip a bit") to indicate the direction your hero is facing. Once you do that, you can check that variable to learn whether the fireball needs to move left or right.

First, create a new variable in your `Player` class to represent which direction your hero is facing. Because my hero faces right naturally, I treat that as the default:

```
        self.score = 0
        self.facing_right = True  # add this
        self.is_jumping = True
```

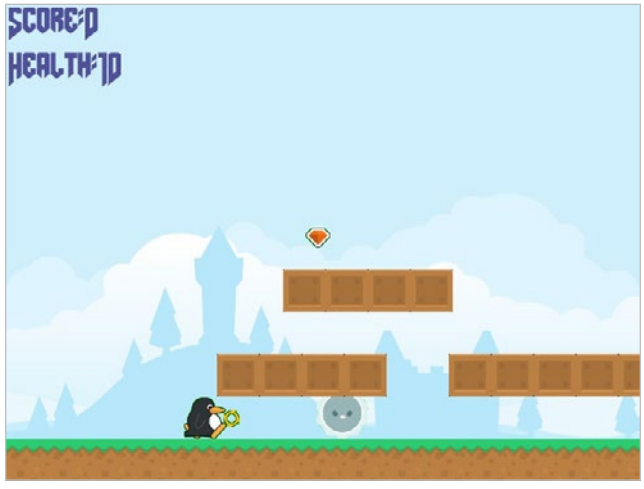
When this variable is `True`, your hero sprite is facing right. It must be set anew every time the player changes the hero's direction, so do that in your main loop on the relevant keyup events:

```
    if event.type == pygame.KEYUP:
        if event.key == pygame.K_LEFT or event.key == ord('a'):
            player.control(steps, 0)
            player.facing_right = False  # add this line
        if event.key == pygame.K_RIGHT or event.key == ord('d'):
            player.control(-steps, 0)
            player.facing_right = True  # add this line
```

Finally, change the `update` function of your `Throwable` class to check whether the hero is facing right or not and to add or subtract pixels from the fireball's position as appropriate:

```
    if self.rect.y < worldy:
        if player.facing_right:
            self.rect.x += 15
        else:
            self.rect.x -= 15
        self.rect.y += 5
```

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



(Seth Kenlon, CC BY-SA 4.0)

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

The complete code:

```
#!/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/>.

import pygame
import pygame.freetype
```

```
import sys
import os

'''
Variables
'''

worldx = 960
worldy = 720
fps = 40
ani = 4
world = pygame.display.set_mode([worldx, worldy])
forwardx  = 600
backwardx = 120

BLUE = (80, 80, 155)
BLACK = (23, 23, 23)
WHITE = (254, 254, 254)
ALPHA = (0, 255, 0)

tx = 64
ty = 64

font_path = os.path.join(os.path.dirname(os.path.realpath
 (__file__)),"fonts","amazdoom.ttf")
font_size = tx
pygame.freetype.init()
myfont = pygame.freetype.Font(font_path, font_size)


'''
Objects
'''

def stats(score,health):
    myfont.render_to(world, (4, 4), "Score:"+str(score), BLUE,
                     None, size=64)
    myfont.render_to(world, (4, 72), "Health:"+str(health),
                     BLUE, None, size=64)


class Throwable(pygame.sprite.Sprite):
    """
    Spawn a player
    """
    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

    def update(self,worldy):
        '''
        throw physics
        '''
        if self.rect.y < worldy:
            if player.facing_right:
            self.rect.x  += 15
            else:
                self.rect.x -= 15
            self.rect.y  += 5
        else:
            self.kill()
            self.firing = 0


# x location, y location, img width, img height, img file
class Platform(pygame.sprite.Sprite):
    def __init__(self, xloc, yloc, imgw, imgh, img):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(os.path.join
                    ('images', img)).convert()
        self.image.convert_alpha()
        self.image.set_colorkey(ALPHA)
        self.rect = self.image.get_rect()
        self.rect.y = yloc
        self.rect.x = xloc


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

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        self.movex = 0
        self.movey = 0
        self.frame = 0
        self.health = 10
        self.damage = 0
        self.score = 0
        self.facing_right = True
        self.is_jumping = True
        self.is_falling = True
        self.images = []
        for i in range(1, 5):
            img = pygame.image.load(os.path.join('images',
                  'walk' + str(i) + '.png')).convert()
            img.convert_alpha()
            img.set_colorkey(ALPHA)
            self.images.append(img)
            self.image = self.images[0]
            self.rect = self.image.get_rect()
```

```
def gravity(self):
    if self.is_jumping:
        self.movey += 3.2

def control(self, x, y):
    """
    control player movement
    """
    self.movex += x

def jump(self):
    if self.is_jumping is False:
        self.is_falling = False
        self.is_jumping = True

def update(self):
    """
    Update sprite position
    """

    # moving left
    if self.movex < 0:
        self.is_jumping = True
        self.frame += 1
        if self.frame > 3 * ani:
            self.frame = 0
        self.image = pygame.transform.flip(self.images[self.
                     frame // ani], True, False)

    # moving right
    if self.movex > 0:
        self.is_jumping = True
        self.frame += 1
        if self.frame > 3 * ani:
            self.frame = 0
        self.image = self.images[self.frame // ani]

    # collisions
    enemy_hit_list = pygame.sprite.spritecollide(self,
                     enemy_list, False)
    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.health -= 1  # subtract 1 hp

    ground_hit_list = pygame.sprite.spritecollide(self,
                      ground_list, False)
    for g in ground_hit_list:
        self.movey = 0
        self.rect.bottom = g.rect.top
        self.is_jumping = False  # stop jumping

    # fall off the world
    if self.rect.y > worldy:
        self.health -=1
        print(self.health)
        self.rect.x = tx
        self.rect.y = ty

    plat_hit_list = pygame.sprite.spritecollide(self,
                    plat_list, False)
    for p in plat_hit_list:
        self.is_jumping = False  # stop jumping
        self.movey = 0
        if self.rect.bottom <= p.rect.bottom:
            self.rect.bottom = p.rect.top
        else:
            self.movey += 3.2

    if self.is_jumping and self.is_falling is False:
        self.is_falling = True
        self.movey -= 33  # how high to jump

    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)

    plat_hit_list = pygame.sprite.spritecollide(self,
                    plat_list, False)

    self.rect.x += self.movex
    self.rect.y += self.movey

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
        self.counter = 0

    def move(self):
```

```
        '''
        enemy movement
        '''
        distance = 80
        speed = 8

        if self.counter >= 0 and self.counter <= distance:
            self.rect.x += speed
        elif self.counter >= distance and self.counter
            <= distance*2:
            self.rect.x -= speed
        else:
            self.counter = 0

        self.counter += 1

    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)


class Level:
    def ground(lvl, gloc, tx, ty):
        ground_list = pygame.sprite.Group()
        i = 0
        if lvl == 1:
            while i < len(gloc):
                ground = Platform(gloc[i], worldy - ty, tx, ty,
                         'tile-ground.png')
                ground_list.add(ground)
                i = i + 1

        if lvl == 2:
            print("Level " + str(lvl) )

        return ground_list

    def bad(lvl, eloc):
        if lvl == 1:
            enemy = Enemy(eloc[0],eloc[1],'enemy.png')
            enemy_list = pygame.sprite.Group()
            enemy_list.add(enemy)
        if lvl == 2:
            print("Level " + str(lvl) )

        return enemy_list

    # x location, y location, img width, img height, img file
    def platform(lvl,tx,ty):
        plat_list = pygame.sprite.Group()
        ploc = []
        i=0
        if lvl == 1:
            ploc.append((200, worldy - ty - 128, 3))
            ploc.append((300, worldy - ty - 256, 3))
            ploc.append((500, worldy - ty - 128 , 4))
            while i < len(ploc):
                j=0
                while j <= ploc[i][2]:
                    plat = Platform((ploc[i][0] + (j*tx)),
                           ploc[i][1], tx, ty, 'tile.png')
                    plat_list.add(plat)
                    j = j + 1
                print('run' + str(i) + str(ploc[i]))
                i = i + 1

        if lvl == 2:
            print("Level " + str(lvl))

        return plat_list

    def loot(lvl):
        if lvl == 1:
            loot_list = pygame.sprite.Group()
            loot = Platform(tx*5, ty*5, tx, ty, 'loot_1.png')
            loot_list.add(loot)

        if lvl == 2:
            print(lvl)

        return loot_list


'''
Setup
'''

backdrop = pygame.image.load(os.path.join('images', 'stage.png'))
clock = pygame.time.Clock()
pygame.init()
backdropbox = world.get_rect()
main = True

player = Player()  # spawn player
player.rect.x = 0  # go to x
player.rect.y = 30  # go to y
player_list = pygame.sprite.Group()
player_list.add(player)
steps = 10
fire = Throwable(player.rect.x, player.rect.y, 'fire.png', 0)
firepower = pygame.sprite.Group()

eloc = []
```

```
eloc = [300, worldy-ty-80]
enemy_list = Level.bad(1, eloc )
gloc = []

i = 0
while i <= (worldx / tx) + tx:
    gloc.append(i * tx)
    i = i + 1

ground_list = Level.ground(1, gloc, tx, ty)
plat_list = Level.platform(1, tx, ty)
enemy_list = Level.bad( 1, eloc )
loot_list = Level.loot(1)


'''
Main Loop
'''

while main:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            try:
                sys.exit()
            finally:
                main = False

        if event.type == pygame.KEYDOWN:
            if event.key == ord('q'):
                pygame.quit()
                try:
                    sys.exit()
                finally:
                    main = False
            if event.key == pygame.K_LEFT or event.key == ord('a'):
                player.control(-steps, 0)
            if event.key == pygame.K_RIGHT or event.key == ord('d'):
                player.control(steps, 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(steps, 0)
                player.facing_right = False
            if event.key == pygame.K_RIGHT or event.key == ord('d'):
                player.control(-steps, 0)
                player.facing_right = True
            if event.key == pygame.K_SPACE:
                if not fire.firing:
                    fire = Throwable(player.rect.x, player.rect.y,
                          'fire.png', 1)
                    firepower.add(fire)

    # scroll the world forward
    if player.rect.x >= forwardx:
            scroll = player.rect.x - forwardx
            player.rect.x = forwardx
            for p in plat_list:
                p.rect.x -= scroll
        for e in enemy_list:
            e.rect.x -= scroll
        for l in loot_list:
            l.rect.x -= scroll

    # scroll the world backward
    if player.rect.x <= backwardx:
            scroll = backwardx - player.rect.x
            player.rect.x = backwardx
            for p in plat_list:
                   p.rect.x += scroll
        for e in enemy_list:
            e.rect.x += scroll
        for l in loot_list:
            l.rect.x += scroll

    world.blit(backdrop, backdropbox)
    player.update()
    player.gravity()
    player_list.draw(world)
    if fire.firing:
        fire.update(worldy)
        firepower.draw(world)
    enemy_list.draw(world)
        enemy_list.update(firepower,enemy_list)
    loot_list.draw(world)
    ground_list.draw(world)
    plat_list.draw(world)
    for e in enemy_list:
        e.move()
    stats(player.score, player.health)
    pygame.display.flip()
    clock.tick(fps)
```

Links

[1] https://opensource.com/article/17/12/game-python-add-a-player

[2] https://opensource.com/article/18/5/pygame-enemy

Add sound to your Python game

Hear what happens when your hero fights, jumps, collects loot, and more by adding sounds to your game. Learn how in the 13th article in this series on creating a platformer in Pygame.

PYGAME PROVIDES an easy way to integrate sounds into your Python video game. Pygame's mixer module [1] can play one or more sounds on command, and by mixing those sounds together, you can have, for instance, background music playing at the same time you hear the sounds of your hero collecting loot or jumping over enemies.

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

Start the mixer

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

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

Define the sounds

Next, you must define the sounds you want to use. This requires that you have the 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 with your game so that anyone playing your game 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:

```
s = 'sound'
```

Even though there are plenty of sounds on the internet, it's not necessarily *legal* to download them and 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 [2] sound that gives you permission to give the sound away with your game.

There are several sites that specialize in free and legal sounds, including:

- Freesound [3] hosts sound effects of all sorts.
- Incompetech [4] hosts background music.
- Open Game Art [5] hosts some 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 any 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.

You might also try making your own music. The excellent LMMS [6] audio workstation is easy to use and ships with lots of interesting sounds. It's available on all major platforms and exports to Ogg Vorbis [7] (OGG) audio format.

Add sound to Pygame

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.

Most video games use OGG sound files because the format provides high quality in small file sizes. When you download a sound file, it might be an MP3, WAVE, FLAC,

or another audio format. To keep your compatibility high and your download size low, convert these to Ogg Vorbis with a tool like fre:ac [8] or Miro [9].

For example, assume you have downloaded a sound file called `ouch.ogg`.

In your code's setup section, create a variable representing the sound file you want to use:

```
ouch = pygame.mixer.Sound(os.path.join(s, 'ouch.ogg'))
```

Trigger 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:
    pygame.mixer.Sound.play(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.

Add background music

If you have music or atmospheric sound effects you want to play in your game's background, you can use the `music` function of Pygame's mixer module. In your setup section, load the music file:

```
music = pygame.mixer.music.load(os.path.join(s, 'music.ogg'))
```

And start the music:

```
pygame.mixer.music.play(-1)
```

The `-1` value tells Pygame to loop the music file infinitely. You can set it to anything from `0` and beyond to define how many times the music should loop before stopping.

Enjoy the soundscapes

Music and sound can add a lot of flavor to your game. Try adding some to your Pygame project!

Links

[1] https://www.pygame.org/docs/ref/mixer.html
[2] https://opensource.com/article/20/1/what-creative-commons
[3] https://freesound.org/
[4] https://incompetech.filmmusic.io/
[5] https://opengameart.org/
[6] https://opensource.com/life/16/2/linux-multimedia-studio
[7] https://en.wikipedia.org/wiki/Vorbis
[8] https://www.freac.org/index.php/en/downloads-mainmenu-330
[9] http://getmiro.com/

How to install Python on Windows

Install Python, run an IDE, and start coding right from your Microsoft Windows desktop.

SO YOU WANT TO LEARN to program? One of the most common languages to start with is Python [1], popular for its unique blend of object-oriented [2] structure and simple syntax. Python is also an *interpreted language*, meaning you don't need to learn how to compile code into machine language: Python does that for you, allowing you to test your programs sometimes instantly and, in a way, while you write your code.

Just because Python is easy to learn doesn't mean you should underestimate its potential power. Python is used by movie studios [3], financial institutions, IT houses, video game studios, makers, hobbyists, artists [4], teachers, and many others.

On the other hand, Python is also a serious programming language, and learning it takes dedication and practice. Then again, you don't have to commit to anything just yet. You can install and try Python on nearly any computing platform, so if you're on Windows, this article is for you.

If you want to try Python on a completely open source operating system, you can install Linux [5] and then try Python [6].

Get Python

Python is available from its website, Python.org [7]. Once there, hover your mouse over the **Downloads** menu, then over the **Windows** option, and then click the button to download the latest release.

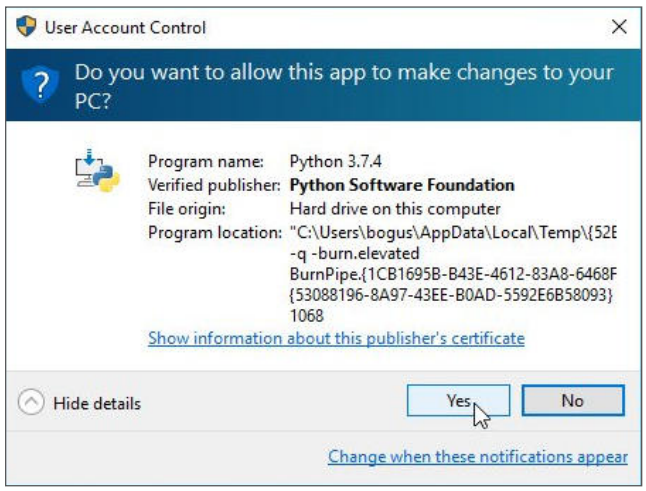
Alternatively, you can click the **Downloads** menu button and select a specific version from the downloads page.

Install Python

Once the package is downloaded, open it to start the installer.

It is safe to accept the default install location, and it's vital to add Python to PATH. If you don't add Python to your PATH, then Python applications won't know where to find Python (which they require in order to run). This is *not* selected by default, so activate it at the bottom of the install window before continuing!

Before Windows allows you to install an application from a publisher other than Microsoft, you must give your approval. Click the **Yes** button when prompted by the **User Account Control** system.



Wait patiently for Windows to distribute the files from the Python package into the appropriate locations, and when it's finished, you're done installing Python.

Time to play.

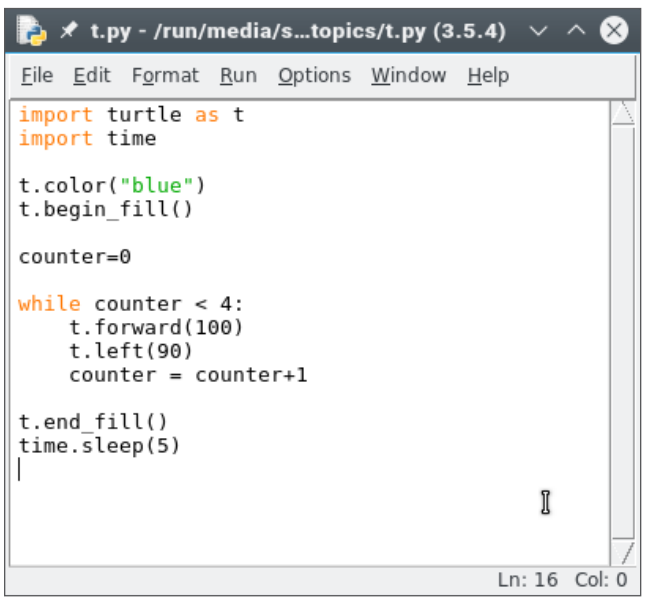
Install an IDE

To write programs in Python, all you really need is a text editor, but it's convenient to have an integrated development environment (IDE). An IDE integrates a text editor with some friendly and helpful Python features. IDLE 3 and Pycharm (Community Edition) are two great open source options to consider.

IDLE 3

Python comes with an IDE called IDLE. You can write code in any text editor, but using an IDE provides you with keyword highlighting to help detect typos, a **Run** button to test code quickly and easily, and other code-specific features that a plain text editor like Notepad++ [8] normally doesn't have.

To start IDLE, click the **Start** (or **Window**) menu and type **python** for matches. You may find a few matches, since Python provides more than one interface, so make sure you launch IDLE.



If you don't see Python in the Start menu, reinstall Python. Be sure to select **Add Python to PATH** in the install wizard. Refer to the Python docs [9] for detailed instructions.

PyCharm IDE

If you already have some coding experience and IDLE seems too simple for you, try PyCharm (Community Edition) [10], an open source IDE for Python. 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 test code quickly and easily.

To install it, visit the PyCharm IDE website, download the installer, and run it. The process is the same as with Python: start the installer, allow Windows to install a non-Microsoft application, and wait for the installer to finish.

Once PyCharm is installed, double-click the PyCharm icon on your desktop or select it from the Start menu.

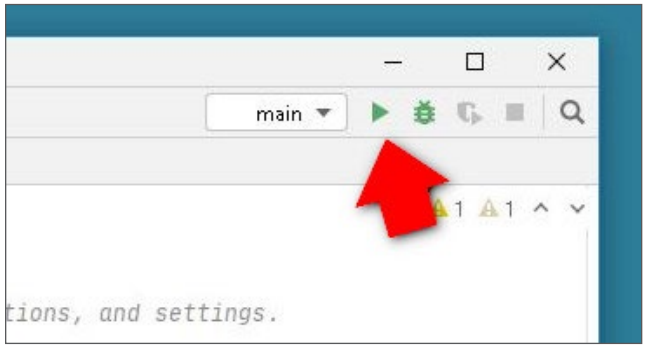
Tell Python what to do

Keywords tell Python what you want it to do. In IDLE, go to the File menu and create a new file. In PyCharm, click the **New Project** button.

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

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

- If you are using IDLE, go to the Run menu and select the Run Module option.
- If you are using PyCharm, click the Run button in the top right corner of the window.



Any time you run code, your IDE prompts you to save the file you're working on. Do that before continuing.

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

That's not very exciting, though. At its core, Python has access to only basic keywords like **print** and **help**, basic math functions, and so on.

Use the **import** keyword to load more keywords. Start a new file 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 program you are controlling. Naming your file **turtle.py** confuses Python because it thinks you want to import your own file.

Turtle [11] is a fun module to use. Add this code to your file:

```
import turtle

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?

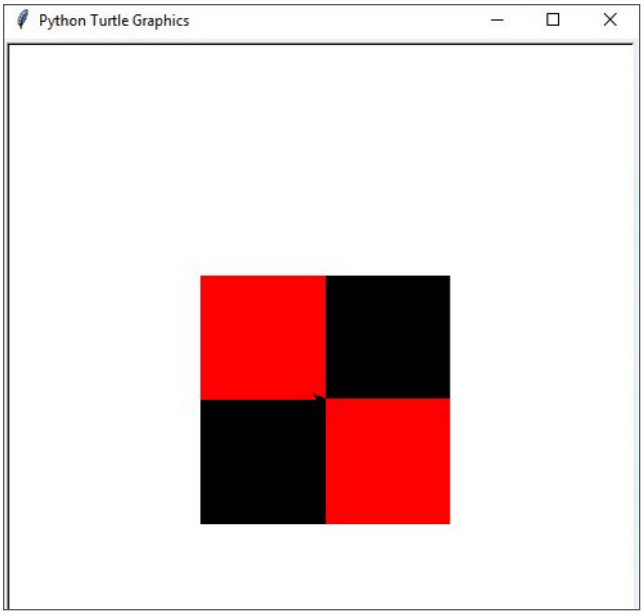
Try more complex code:

```
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(2)
```

Notice that turtle, in this example code, has not only been imported, but it's also been given the shorter nickname **t**, which is quicker and easier to type. This is a convenience function in Python.

Challenge

As a challenge, try changing your script to get this result:



Once you complete that script, you're ready to move on to more exciting modules. A good place to start is this introductory dice game [12].

Stay Pythonic

Python is a fun language with modules for practically anything you can think to do with it. As you can see, it's easy to get started with Python, and as long as you're patient with yourself, you may find yourself understanding and writing Python code with the same fluidity as you write your native language. Work through some Python articles [13] here on Opensource.com, try scripting some small tasks for yourself, and see where Python takes you. To really integrate Python with your daily workflow, you might even try Linux, which is natively scriptable in ways no other operating system is. You might find yourself, given enough time, using the applications you create!

Good luck, and stay Pythonic.

Links

[1] https://www.python.org/
[2] https://opensource.com/article/19/7/get-modular-python-classes
[3] https://github.com/edniemeyer/weta_python_db
[4] https://opensource.com/article/19/7/rgb-cube-python-scribus
[5] https://opensource.com/article/19/7/ways-get-started-linux
[6] https://opensource.com/article/17/10/python-101
[7] https://www.python.org/downloads/
[8] https://notepad-plus-plus.org/
[9] http://docs.python.org/3/using/windows.html
[10] https://www.jetbrains.com/pycharm/download/#section=windows
[11] https://opensource.com/life/15/8/python-turtle-graphics
[12] https://opensource.com/article/17/10/python-101#python-101-dice-game
[13] https://opensource.com/sitewide-search?search_api_views_fulltext=Python

Managing Python packages the right way

By László Kiss Kollár

Don't fall victim to the perils of Python package management.

THE PYTHON PACKAGE INDEX (PYPI) indexes an amazing array of libraries and applications covering every use case imaginable. However, when it comes to installing and using these packages, newcomers often find themselves running into issues with missing permissions, incompatible library dependencies, and installations that break in surprising ways.

The Zen of Python states: "There should be one—and preferably only one—obvious way to do it." This is certainly not always the case when it comes to installing Python packages. However, there are some tools and methods that can be considered best practices. Knowing these can help you pick the right tool for the right situation.

Installing applications system-wide

pip is the de facto package manager in the Python world. It can install packages from many sources, but PyPI [1] is the primary package source where it's used. When installing packages, **pip** will first resolve the dependencies, check if they are already installed on the system, and, if not, install them. Once all dependencies have been satisfied, it proceeds to install the requested package(s). This all happens globally, by default, installing everything onto the machine in a single, operating system-dependent location.

Python 3.7 looks for packages on an Arch Linux system in the following locations:

```
$ python3.7 -c "import sys; print('\n'.join(sys.path))"

/usr/lib/python37.zip
/usr/lib/python3.7
/usr/lib/python3.7/lib-dynload
/usr/lib/python3.7/site-packages
```

One problem with global installations is that only a single version of a package can be installed at one time for a given Python interpreter. This can cause issues when a package is a dependency of multiple libraries or applications, but they require different versions of this dependency. Even if things seem to be working fine, it is possible that upgrading the dependency (even accidentally while installing another package) will break these applications or libraries in the future.

Another potential issue is that most Unix-like distributions manage Python packages with the built-in package manager (**dnf**, **apt**, **pacman**, **brew**, and so on), and some of these tools install into a non-user-writeable location.

```
$ python3.7 -m pip install pytest
Collecting pytest
Downloading...
[...]
Installing collected packages: atomicwrites, pluggy, py,
  more-itertools, pytest
Could not install packages due to an EnvironmentError: [Error 13]
  Permission denied:
'/usr/lib/python3.7/site-packages/site-packages/atomicwrites-
 x.y.z.dist-info'
Consider using '--user' option or check the permissions.
$
```

This fails because we are running **pip install** as a non-root user and we don't have write permission to the **site-packages** directory.

You can technically get around this by running **pip** as a root (using the **sudo** command) or administrative user. However, one problem is that we just installed a bunch of Python packages into a location the Linux distribution's package manager owns, making its internal database and the installation inconsistent. This will likely cause issues anytime we try to install, upgrade, or remove any of these dependencies using the package manager.

As an example, let's try to install **pytest** again, but now using my system's package manager, **pacman**:

```
$ sudo pacman -S community/python-pytest
resolving dependencies...
looking for conflicting packages...
[...]
python-py: /usr/lib/site-packages/py/_pycache_/
  _metainfo.cpython-37.pyc exists in filesystem
python-py: /usr/lib/site-packages/py/_pycache_/
  _builtin.cpython-37.pyc exists in filesystem
python-py: /usr/lib/site-packages/py/_pycache_/
  _error.cpython-37.pyc exists in filesystem
```

Another potential issue is that an operating system can use Python for system tools, and we can easily break these by modifying Python packages outside the system package manager. This can result in an inoperable system, where restoring from a backup or a complete reinstallation is the only way to fix it.

sudo pip install: A bad idea

There is another reason why running **pip install** as root is a

bad idea. To explain this, we first have to look at how Python libraries and applications are packaged.

Most Python libraries and applications today use **setuptools** as their build system. setuptools requires a **setup.py** file in the root of the project, which describes package metadata and can contain arbitrary Python code to customize the build process. When a package is installed from the source distribution, this file is executed to perform the installation and execute tasks like inspecting the system, building the package, etc.

Executing **setup.py** with root permissions means we can effectively open up the system to malicious code or bugs. This is a lot more likely than you might think. For example, in 2017, several packages were uploaded to PyPI [2] with names resembling popular Python libraries. The uploaded code collected system and user information and uploaded it to a remote server. These packages were pulled shortly thereafter. However, these kinds of "typo-squatting" incidents can happen anytime since anyone can upload packages to PyPI and there is no review process to make sure the code doesn't do any harm.

The Python Software Foundation (PSF) recently announced that it will sponsor work to improve the security of PyPI [3]. This should make it more difficult to carry out attacks such as "pytosquatting" [4] and hopefully make this less of an issue in the future.

Security issues aside, **sudo pip install** won't solve all the dependency problems: you can still install only a single version of any given library, which means it's still easy to break applications this way.

Let's look at some better alternatives.

OS package managers

It is very likely that the "native" package manager we use on our OS of choice can also install Python packages. The question is: should we use **pip**, or **apt**, **dnf**, **pacman**, and so on?

The answer is: *it depends*.

pip is generally used to install packages directly from PyPI, and Python package authors usually upload their packages there. However, most package maintainers will not use PyPI, but instead take the source code from the source distribution (sdist) created by the author or a version control system (e.g., GitHub), apply patches if needed, and test and release the package for their respective platforms. Compared to the PyPI distribution model, this has pros and cons:

- Software maintained by native package managers is generally more stable and usually works better on the given platform (although this might not always be the case).
- This also means it takes extra work to package and test upstream Python code:
  1. The package selection is usually much smaller than what PyPI offers.
  2. Updates are slower and package managers will often ship much older versions.

If the package we want to use is available and we don't mind slightly older versions, the package manager offers a convenient and safe way to install Python packages. And, since these packages install system-wide, they are available to all users on the system. This also means that we can use them only if we have the required permissions to install packages on the system.

If we want to use something that is not available in the package manager's selection or is too old, or we simply don't have the necessary permissions to install packages, we can use pip instead.

User scheme installations

pip supports the "user scheme" mode introduced in Python 2.6. This allows for packages to be installed into a user-owned location. On Linux, this is typically **~/.local**. Putting **~/.local/bin/** on our **PATH** will make it possible to have Python tools and scripts available at our fingertips and manage them without root privileges.

```
$ python3.7 -m pip install --user black
Collecting black
 Using cached
[...]
Installing collected packages: click, toml, black
 The scripts black and blackd are installed in
   '/home/tux/.local/bin' which is not on PATH.
 Consider adding this directory to PATH or, if you prefer to
   suppress this warning, use --no-warn-script-location.
Successfully installed black-x.y click-x.y toml-x.y.z
$
```

However, this solution does not solve the issue if and when we need different versions of the same package.

Enter virtual environments

Virtual environments offer isolated Python package installations that can coexist independently on the same system. This offers the same benefits as user scheme installations, but it also allows the creation of self-contained Python installations where an application does not share dependencies with any other application. **Virtualenv** creates a directory that holds a self-contained Python installation, including the Python binary and essential tools for package management: **setuptools**, **pip**, and **wheel**.

Creating virtual environments

virtualenv is a third-party package, but Python 3.3 added the **venv** package to the standard library. As a result, we don't have to install anything to use virtual environments in modern versions of Python. We can simply use **python3.7 -m venv <env_name>** to create a new virtual environment.

After creating a new virtual environment, we must **activate** it by sourcing the activate script in the **bin** directory of the

newly created environment. The activation script creates a new subshell and adds the bin directory to the **PATH** environment variable, enabling us to run binaries and scripts from this location. This means that this subshell will use **python**, **pip**, or any other tool installed in this location instead of the ones installed globally on the system.

```
$ python3.7 -m venv test-env
$ . ./test-env/bin/activate
(test-env) $
```

After this, any command we execute will use the Python installation inside the virtual environment. Let's install some packages.

```
(test-env)$ python3.7 -m pip install --user black
Collecting black
 Using cached
[...]
Installing collected packages: click, toml, black
Successfully installed black-x.y click-x.y toml-x.y.z
(test-env) $
```

We can use **black** inside the virtual environment without any manual changes to the environment variables like **PATH** or **PYTHONPATH**.

```
(test-env) $ black --version
black, version x.y
(test-env) $ which black
/home/tux/test-env/bin/black
(test-env) $
```

When we are done with the virtual environment, we can simply deactivate it with the **deactivate** function.

```
(test-env) $ deactivate
$
```

Virtual environments can also be used without the activation script. Scripts installed in a **venv** will have their *shebang* line rewritten to use the Python interpreter inside the virtual environment. This way, we can execute the script from anywhere on the system using the full path to the script.

```
(test-env) $ head /home/tux/test-env/bin/black
#!/home/tux/test-env/bin/python3.7

# -*- coding: utf-8 -*-
import re
import sys

from black import main

if __name__ == '__main__':
    sys.argv[0] = re.sub(r'(-script\.pyw?|\.exe)?$', '', sys.argv[0])
(test-env) $
```

We can simply run **~/test-env/bin/black** from anywhere on the system and it will work just fine.

It can be useful to add certain commonly used virtual environments to the **PATH** environment variable so we can quickly and easily use the scripts in them without typing out the full path:

```
export PATH=$PATH:~/test-env/bin
```

Now when we execute **black**, it will be picked up from the virtual environment (unless it appears somewhere else earlier on the **PATH**). Add this line to your shell's initialization file (e.g., **~/.bashrc**) to have it automatically set in all new shells.

Virtual environments are very commonly used for Python development because each project gets its own environment where all library dependencies can be installed without interfering with the system installation.

I recommend checking out the virtualenvwrapper [5] project, which can help simplify common **virtualenv**-based workflows.

What about Conda?

Conda [6] is a package management tool that can install packages provided by Anaconda on the repo.continuum.io [7] repository. It has become very popular, especially for data science. It offers an easy way to create and manage environments and install packages in them. One drawback compared to **pip** is that the package selection is much smaller.

A recipe for successful package management

- Never run **sudo pip install**.
- If you want to make a package available to all users of the machine, you have the right permissions, and the package is available, then use your distribution's package manager (**apt**, **yum**, **pacman**, **brew**, etc.).
- If you don't have root permissions or the OS package manager doesn't have the package you need, use **pip install --user** and add the user installation directory to the **PATH** environment variable.
- If you want multiple versions of the same library to coexist, to do Python development, or just to isolate dependencies for any other reason, use virtual environments.

Links

[1] https://pypi.org/
[2] https://github.com/pypa/warehouse/issues/3948
[3] http://pyfound.blogspot.com/2018/12/upcoming-pypi-improvements-for-2019.html
[4] https://pytosquatting.overtag.dk/
[5] https://virtualenvwrapper.readthedocs.io/
[6] https://conda.io/
[7] https://repo.continuum.io/

Easily set image transparency using GIMP

Use chroma key or "green screen" techniques to set transparencies on your video-game graphics.

WHETHER YOU'RE programming a game or an app with Python [1] or Lua [2], you're probably using PNG graphics for your game assets. An advantage of the PNG format, which is not available in a JPEG, is the ability to store an alpha channel. Alpha is, essentially, the "color" of invisibility or transparency. Alpha is the part of an image you don't see. For example, if you were to draw a doughnut, the doughnut hole would be filled with alpha, and you could see whatever was behind it.

A common problem is how to find the alpha part of an image. Sometimes, your programming framework, whether it's Python Arcade [3], Pygame [4], LÖVE, or anything else, detects the alpha channel and treats it (after the appropriate function calls) as transparency. That means it renders no new pixels where there's alpha, leaving that doughnut hole empty. It's 100% transparent or 0% opaque and functionally "invisible."

Other times, your framework and your graphic asset don't agree on where the alpha channel is located (or that an alpha channel exists at all), and you get pixels where you wanted transparency.

This article describes the most sure-fire way I know to work around that.

Chroma key

In computer graphics, there are a few values that contribute to how a pixel is rendered. **Chrominance**, or **chroma**, describes the saturation or intensity of a pixel. The **chroma key** technique (also known as **green screening**) was originally developed as a chemical process, in which a specific color (blue at first and later green) was deliberately obscured by a **matte** during the copying of a film negative, allowing another image to be substituted where there once was a blue or green screen. That's a simplified explanation, but it demonstrates the origins of what is known as the alpha channel in computer graphics.

An alpha channel is information saved in a graphic to identify pixels that are meant to be transparent. RGB graphics, for example, have red, green, and blue channels. RGBA graphics contain red, green, blue, and alpha. The value of alpha can range from 0 to 1, with decimal points being valid entries.

Because an alpha channel can be expressed in several different ways, relying on an embedded alpha channel can be problematic. Instead, you can pick a color and turn it into an alpha value of 0 in your game framework. For that to work, you must know the colors in your graphic.

Prepare your graphic

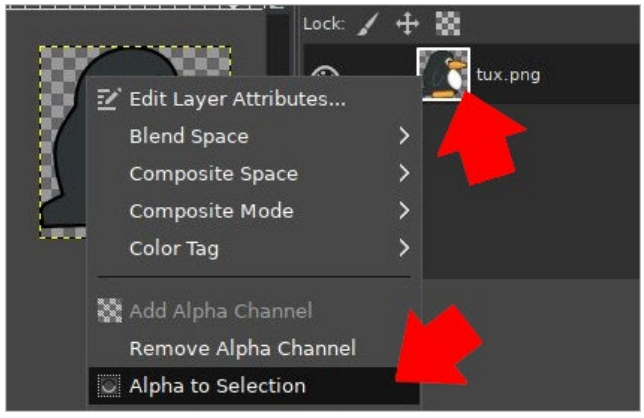
To prepare a graphic with an explicit color reserved exclusively for a chroma key, open the graphic in your favorite graphic editor. I recommend GIMP [5] or Glimpse [6], but mtPaint [7] or Pinta [8] or even Inkscape [9] can work just as well, depending on the nature of your graphics and your ability to translate these instructions to a different tool.

Start by opening this graphic of Tux the penguin:

(Seth Kenlon, CC BY-SA 4.0)

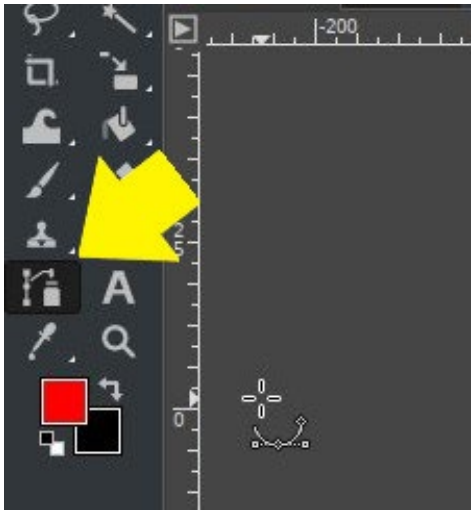
Select the graphic

Once the graphic is open, go to the **Windows** menu and select **Dockable Dialogs** and then **Layers**. Right-click on Tux's layer in the **Layer** palette. From the pop-up menu, select **Alpha to Selection**. If your image does not have a built-in alpha channel, then you must create your own selection manually.



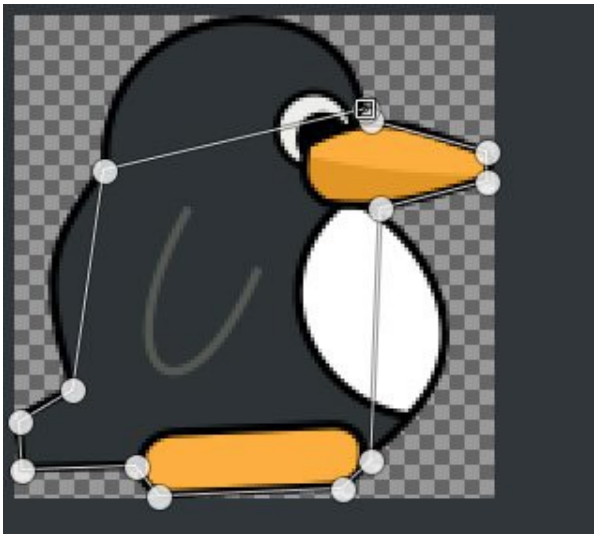
(Seth Kenlon, CC BY-SA 4.0)

To create a selection manually, click the **Paths** tool from the toolbox.



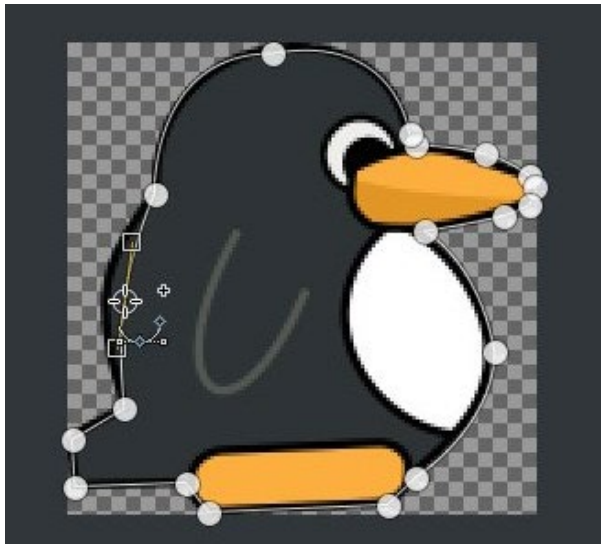
(Seth Kenlon, CC BY-SA 4.0)

Using the **Paths** tool, move your mouse around the graphic, clicking and releasing (do not drag) at each major intersection of its outline. Don't worry about following curves; just find the major intersections and corners. This creates a node at each point, with a straight line drawn between them. You don't need to close your path, so when you reach the final intersection before the one where you started, you're done.



(Seth Kenlon, CC BY-SA 4.0)

Once you've created your outline path, go to the **Windows** menu and select **Dockable Dialogs** and then **Tool Options**. In the **Tool Options** panel, select **Edit (Ctrl)**. With this activated, you can edit the paths you've just drawn by clicking the lines or nodes and adjusting them to fit your graphic better. You can even give the lines curves.



(Seth Kenlon, CC BY-SA 4.0)

Now select the **Paths** panel from the **Windows > Dockable Dialogs** menu. In the Paths panel, click the **Path to Selection** button. Your graphic is now selected.

Grow the selection

If you feel your selection is too tight, you can give yourself a little slack by growing the selection. I sometimes do this when I want to impose or thicken a border around a graphic.

To grow a selection, click the **Select** menu and choose **Grow**. Enter a pixel value and click **OK**.

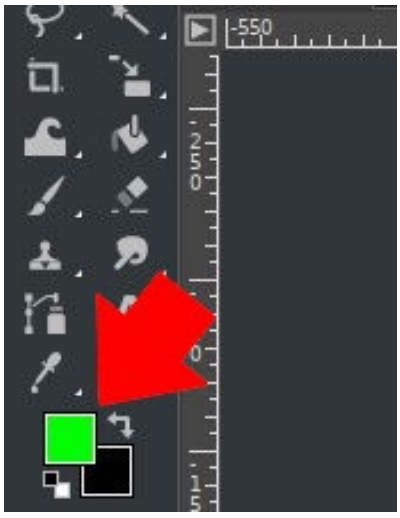
Invert the selection

You have your graphic selected, but what you actually want to select is everything except your graphic. That's because you're creating an alpha matte to define what in your graphic will be replaced by something else. In other words, you need to mark the pixels that will be turned invisible.

To invert the selection, click on the **Select** menu and choose **Invert**. Now everything except your graphic is selected.

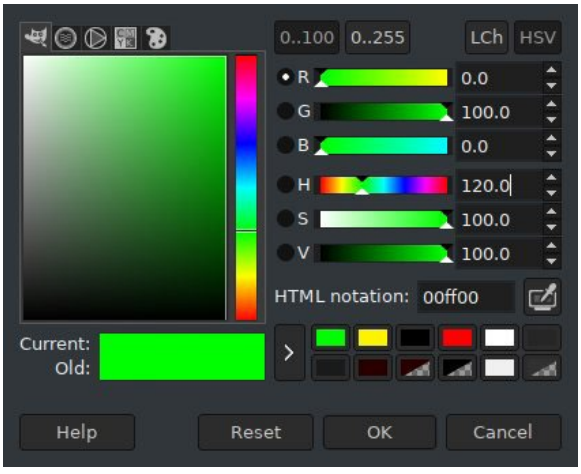
Fill with alpha

With everything except your graphic selected, choose the color you want to use to designate your alpha matte. The most common color is green (as you might guess from the term "green screen"). There's nothing magical about the color green or even a particular shade of green. It's used because humans, frequent subjects of graphic manipulation, contain no green pigment, so it's easy to key out green without accidentally keying out part of the central subject. Of course, if your graphic is a green alien or an emerald or something that does contain green, you should use a different color. You can use any color you want, as long as it's solid and consistent. If you're doing this to many graphics, your choice should be consistent across all graphics.



(Seth Kenlon, CC BY-SA 4.0)

Set your foreground color with the color value you've chosen. To ensure that your choice is exact, use the HTML [10] or HSV [11] representation of the color. For example, if you're using pure green, it can be expressed in GIMP (and most open source graphic applications) as `00ff00` (that's 00 red, FF green, and 00 blue, with F being the maximum amount).



(Seth Kenlon, CC BY-SA 4.0)

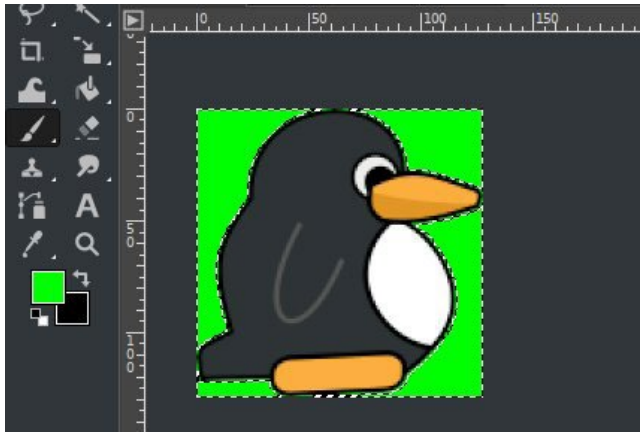
Whatever color you choose, make sure you take note of the HTML or HSV values so you use the exact same color for every graphic.

To fill in your alpha matte, click the **Edit** menu and choose **Fill with FG Color**.

Flatten and export

If you've left a border around your graphic, set your background color to the color you want to use as the border stroke. This is usually either black or white, but it can be anything that suits your game's aesthetic.

Once you have set the background color, click on the **Image** menu and select **Flatten Image**. It's safe to do this whether or not you added a border. This process removes the alpha channel from your image, filling in any "transparent" pixels with the background color.



(Seth Kenlon, CC BY-SA 4.0)

You now have an image ready for your game engine. Export the image to whatever format your game engine prefers, and then import it into your game using whatever function calls it requires. In your code, set the alpha value to `00ff00` (or whatever color you used), and then use the game engine's graphic transforms to treat that color as an alpha channel.

Other methods

This isn't the only way to get transparency in your game graphics. Check your game engine's documentation to find out how it tries to process alpha channels by default, and when you're not certain, try letting your game engine auto-detect transparency in your graphic before you go about editing it. Sometimes, your game engine's expectations and your graphic's preset values happen to match, and you get transparency without having to do any extra work.

When that fails, though, try a little chroma key. It's worked for the film industry for nearly 100 years, and it can work for you too.

Links

[1] https://opensource.com/article/17/10/python-101
[2] https://opensource.com/article/17/4/how-program-games-raspberry-pi
[3] https://opensource.com/article/18/4/easy-2d-game-creation-python-and-arcade
[4] https://opensource.com/article/17/12/game-framework-python
[5] http://gimp.org/
[6] https://glimpse-editor.github.io/
[7] https://opensource.com/article/17/2/mtpaint-pixel-art-animated-gifs
[8] https://www.pinta-project.com/
[9] http://inkscape.org/
[10] https://www.w3schools.com/colors/colors_picker.asp
[11] https://en.wikipedia.org/wiki/HSL_and_HSV