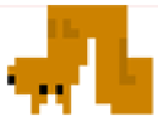


Making Games with Python & Pygame

A guide to programming with graphics, animation, and sound

Al Sweigart

Making Games with Python & Pygame

ROUGH DRAFT v4

Note: This is a rough draft. Most sections are incomplete, and this book is only meant to give a preview of what the final book will look like.

About This Book

Hello! This book will teach you how to make computer games by programming in the Python programming language. This book will also give you an introduction to the Pygame framework (also called the Pygame library), which makes it easy to create 2D games and programs with graphics. Both Python and the Pygame framework can be downloaded for free from http://python.org and http://pygame.org. All you need is a computer and this book to begin making your own games.

This book is an intermediate programming book. If you are completely new to programming, you can still try to follow along with the source code examples and pick some things up. However, it might be easier to learn how to program in Python first. There is a free book called "Invent Your Own Computer Games with Python" that is available completely for free from http://inventwithpython.com. That book covers programming non-graphical, text-based games for complete beginners.

However, if you already know how to program in Python (or even some other language, since Python is so easy to pick up) and want to start making games beyond just text, then this is the book for you. The book starts with a short introduction to how the Pygame library works and the functions it provides. Then it covers actual games, so you can see how actual game programs make use of Pygame.

This book features eleven different games that are clones of popular games that you've probably already played. Each chapter presents the full source code for a game. The games are a lot more fun and interactive than the text-based games in Invent with Python, but are still fairly short. All of the programs are less than 600 lines long. This is pretty small when you consider that professional games you download or buy in a store can be hundreds of thousands of lines long. These games require an entire team of programmers and artists working with each other for months or years to make.

If you've read "Invent with Python", you'll find that the chapters in this book don't go into as much detail about how the programs work as Invent with Python did. This book assumes that you can understand how the individual lines of code work, and so it only explains the broader concepts of how the games work.

The website for this book is http://inventwithpython.com/pygame. All the programs and files mentioned in this book can be downloaded for free from this website, including this book itself. Programming is a great creative activity, so please share this book as widely as possible. The

Creative Commons license that this book is released under gives you the right to copy and duplicate this book as much as you want (as long as you don't charge money for it.)

If you ever have questions about how these programs work, feel free to email me at al@inventwithpython.com.

Chapter 1 – Installing Python and Pygame

What You Should Know Before You Begin

It might help if you know a bit about Python programming (or how to program in another language besides Python) before you read through this book. However, even if you haven't you can still read this book anyway. Programming isn't nearly as hard as people think it is. If you ever run into some trouble, you can read the first book online at http://inventwithpython.com or look up a topic that you find confusing on the Invent with Python wiki at http://inventwithpython.com/wiki.

You don't need to know how to use the Pygame library before reading this book. The next chapter is a brief tutorial on all of Pygame's major features and functions.

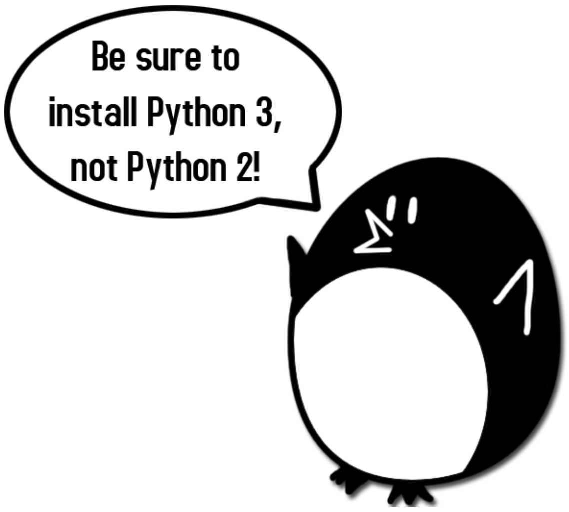
Just in case you haven't read the first book and already installed Python and Pygame, the installation instructions are in this chapter. If you already have installed both of these pieces of software, then you can skip this chapter.

Downloading and Installing Python

Before we can begin programming you'll need to install software called the Python interpreter. (You may need to ask an adult for help here.) The **interpreter** is a program that understands the instructions that you'll write in the Python language. Without the interpreter, your computer won't be able to run your Python programs. We'll just refer to "the Python interpreter" as "Python" from now on.

The Python interpreter software can be downloaded from the official website of the Python programming language, http://www.python.org. You might want the help of someone else to download and install the Python software. The installation is a little different depending on if your computer's operating system is Windows, Mac OS X, or a Linux OS such as Ubuntu. You can also find videos of people installing the Python software online. A list of these videos is at http://invpy.com/ installing

Important Note! Be sure to install Python 3, and not Python 2. The programs in this book use Python 3, and you'll get errors if you try to run them with Python 2. It is so important, I am adding a cartoon penguin telling you to install Python 3 so that you do not miss this message:



"Be sure to install Python 3, not Python 2!", says the penguin. Otherwise the programs in this book won't work.

Windows Instructions

When you get to python.org, you should see a list of links on the left (such as "About", "News", "Documentation", "Download", and so on.) Click on the **Download** link to go to the download page, then look for the file called Python 3.2 Windows Installer (Windows binary -- does not include source) and click on its link to download Python for Windows.

Double-click on the python-3.2.msi file that you've just downloaded to start the Python installer. (If it doesn't start, try right-clicking the file and choosing Install.) Once the installer starts up, click the Next button and just accept the choices in the installer as you go (no need to make any changes). When the install is finished, click Finish.

Important Note! Be sure to install Python 3, and not Python 2. The programs in this book use Python 3, and you'll get errors if you try to run them with Python 2.

Mac OS X Instructions

The installation for Mac OS X is similar to the Windows installation. Instead of downloading the .msi file from the Python website, download the .dmg Mac Installer Disk Image file instead. The link to this file will look something like "Mac Installer disk image (3.2)" on the "Download Python Software" web page.

Ubuntu and Linux Instructions

If your operating system is Ubuntu, you can install Python by opening a terminal window (from the desktop click on **Applications** > **Accessories** > **Terminal**) and entering "sudo apt-get install python3.2" then pressing Enter. You will need to enter the root password to install Python, so ask the person who owns the computer to type in this password if you do not know it.

You also need to install the IDLE software. From the terminal, type in "sudo apt-get idle3". The root password is also needed to install IDLE.

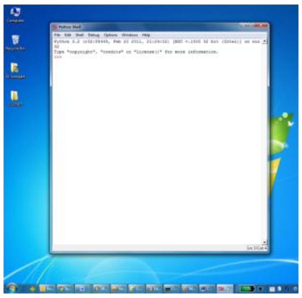
Starting Python

We will be using the IDLE software to type in our programs and run them. IDLE stands for **I**nteractive **D**eve**L**opment **E**nvironment. The development environment is software that makes it easy to write Python programs.

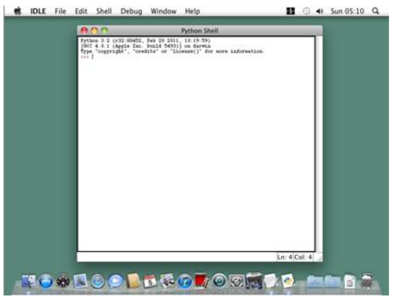
If your operating system is Windows XP, you should be able to run Python by clicking the Start button, then selecting Programs, Python 3.1, IDLE (Python GUI). For Windows Vista or Windows 7, just click the Windows button in the lower left corner, type "IDLE" and select "IDLE (Python GUI)".

If your operating system is Max OS X, start IDLE by opening the Finder window and click on Applications, then click Python 3.2, then click the IDLE icon.

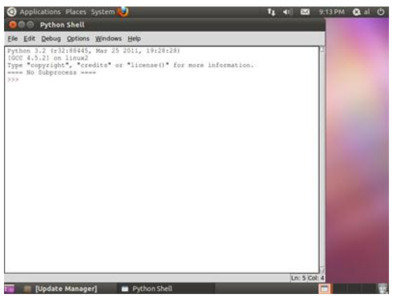
If your operating system is Ubuntu or Linux, start IDLE by opening a terminal window and then type "idle3" and press Enter. You may also be able to click on Applications at the top of the screen, and then select Programming, then IDLE 3.



```
Python 3.2 (r32:88445, Feb 20 2011, 21:29:02) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>>
```



```
Python 3.2 (r32:88452, Feb 20 2011, 10:19:59)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> |
```


Applications Places System
9:13 PM
Python Shell
File Edit Debug Options Windows Help
Ln: 5 Col: 4
[Update Manager]
Python Shell

```
Python 3.2 (r32:88445, Mar 25 2011, 19:28:28)
[GCC 4.5.2] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
```

The window that appears when you first run IDLE is called the **interactive shell**. A **shell** is a program that lets you type instructions into the computer. The Python shell lets you type Python instructions, and the shell sends these instructions to the Python interpreter to perform. We can type Python instructions into the shell and, because the shell is interactive, the computer will read our instructions and respond in some way.

How to Use This Book

“Making Games with Python & Pygame” is different from other programming books because it focuses on the complete source code for different game programs. Instead of teaching you programming concepts and leaving it up to you to figure out how to make programs with those concepts, this book shows you some programs and then explains how they are put together.

In general, you should read these chapters in order. There are many concepts that are used over and over in these games, and they are only explained in detail in the first game they appear in. But if there is a game you think is interesting, go ahead and jump to that chapter. You can always read the previous chapters later if you got ahead of yourself.

The Featured Programs

Each chapter focuses on a single game program and explain how different parts of the code work. It is very helpful to copy these programs by typing in the code line by line from this book.

You can also download the source code file from this book's website. In a web browser, go to the URL http://invpy.com/source and follow the instructions to download the source code file.

Downloading Graphics and Sound Files

While you can just type in the code you read out of this book, you will need to download the graphics and sound files used by the games in this book from http://invpy.com/downloads. Make sure that these image and sound files are located in the same folder as the .py Python file, otherwise your Python program will not be able to find these files.

Line Numbers and Spaces

When entering the source code yourself, do not type the line numbers that appear at the beginning of each line. For example, if you see this in the book:

```
1. number = random.randint(1, 20)
2. spam = 42
3. print('Hello world!')
```

You do not need to type the "1." on the left side, or the space that immediately follows it. Just type it like this:

```
number = random.randint(1, 20)
spam = 42
```

```
print('Hello world!')
```

Those numbers are only used so that this book can refer to specific lines in the code. They are not a part of the actual program.

Aside from the line numbers, be sure to enter the code exactly as it appears. Notice that some of the lines don't begin at the leftmost edge of the page, but are indented by four or eight spaces. Be sure to put in the correct number of spaces at the start of each line. (Since each character in IDLE is the same width, you can count the number of spaces by counting the number of characters above or below the line you're looking at.)

For example, you can see that the second line is indented by four spaces because the four characters ("whil") on the line above are over the indented space. The third line is indented by another four spaces (the four characters, "if n" are above the third line's indented space):

```
while spam < 10:
    if number == 42:
        print('Hello')
```

Text Wrapping in This Book

Some lines of code are too long to fit on one line on the page, and the text of the code will wrap around to the next line. When you type these lines into the file editor, enter the code all on one line without pressing Enter.

You can tell when a new line starts by looking at the line numbers on the left side of the code. For example, the code below has only two lines of code, even though the first line wraps around:

```
1. print('This is the first line! xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxx')
2. print('This is the second line, not the third line.')
```

The Source Code In This Book

In my previous book, "Invent Your Own Computer Games with Python", I listed the entire source code for the program the chapter was explaining at the very beginning of the chapter. Then, as each section of the code was explained, the book had that section of code reprinted for the reader's convenience. Due to space restrictions, this book cannot have the complete source code listed at the start of the chapter; it only has the source code printed section by section throughout the chapter. Because you might want to flip back to earlier or later lines of code while reading this book, it might help to also have the entire source code for the program displayed on your computer. At the start of each source code listing will be the URL for a website where the program is typed out in full. This will keep you from having to flip through several pages just to see other lines in the program.

Checking Your Code Online

Some of the programs in this book are a little long. Although it is very helpful to learn Python by typing out the source code for these programs, you may accidentally make typos that cause your programs to crash. It may not be obvious where the typo is.

You can copy and paste the text of your source code to the online diff tool on the book's website. The diff tool will show any differences between the source code in the book and the source code you've typed. This is an easy way of finding any typos in your program.

Copying and pasting text is a very useful computer skill, especially for computer programming. There is a video tutorial on copying and pasting at this book's website at http://invpy.com/videos.

The online diff tool is at this web page: http://invpy.com/diff. There is also a video tutorial on how to use this tool on the book’s website.

More Info Links on http://invpy.com

There is a lot that you can learn about programming. But you don’t need to learn all of it now. There are several times in this book where you might like to learn these additional details and explanations, but if I included them in this book then it would add many more pages. If this larger, heavier book accidentally fell on you the weight of these many additional pages would crush you, resulting in death. Instead, I have included “more info” links in this book that you can follow on this book’s website. You do not have to read this additional information to understand anything in this book, but it is there if you are curious. These (and other) links have been shortened and begin with http://invpy.com.

Even though this book is not dangerously heavy, please do not let it fall on you anyway.

Chapter 2 – Pygame Basics

Just like how Python comes with several modules like `random`, `math`, or `time` that provide additional functions for your programs, the Pygame framework includes several modules with functions for drawing graphics, playing sounds, and handling mouse input.

This chapter will cover the basic modules and functions that Pygame provides and assumes you already know basic Python programming. If you have trouble with some of the programming concepts, you can read through the "Invent Your Own Computer Games with Python" book online at http://invpy.com/book. This book is aimed at complete beginners to programming.

GUI vs. CLI

The Python programs that you can write with Python's built-in functions only deal with text through the `print()` and `input()` functions. Your program can display text on the screen and let the user type in text from the keyboard. This type of program has a **Command Line Interface**, or **CLI** (which sounds like the first part of "climb" and rhymes with "sky"). These programs are somewhat limited because they can't display pictures, have colors, or use the mouse. These CLI programs only get input from the keyboard with the `input()` function and even then user must press enter before the program can respond to the input. This means **real-time** (that is, continuing to run code without waiting for the user) action games are impossible to make.

Pygame provides functions for creating programs with a **Graphical User Interface**, or **GUI** (pronounced, "gooey"). Instead of a text-based CLI, programs with a graphics-based GUI can show a window with images and colors.

Hello World with Pygame

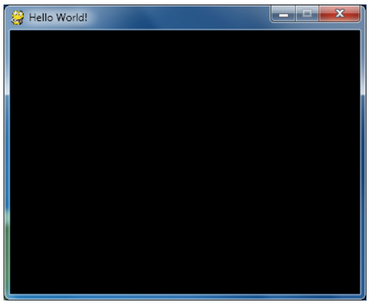
Our first program made with Pygame is a small program that makes a window that says "Hello World!" appear on the screen. Type in the following code into IDLE's file editor and save it as *hellopygameworld.py*. Then run the program by pressing **F5** or selecting **Run > Run Module** from the menu at the top of the file editor.

Remember, do not type the numbers or the periods at the beginning of each line (that's just for reference in this book.)

```
1. import pygame, sys
2. from pygame.locals import *
```

```
 3.
 4. pygame.init()
 5. WINDOWSURF = pygame.display.set_mode((400, 300))
 6. pygame.display.set_caption('Hello World!')
 7. while True: # main game loop
 8.     for event in pygame.event.get():
 9.         if event.type == QUIT:
10.             pygame.quit()
11.             sys.exit()
12.     pygame.display.update()
```

When you run this program, a black window like this will appear:



Yay! You've just made the world's most boring video game! It's just a blank window with "Hello World!" at the top of the window (in what is called the window's **title bar** or **caption**). But creating a window is the first step to making graphical games. When you click on the X in the corner of the window, the program will end and the window will disappear.

Calling the `print()` function to make text appear in the window won't work because `print()` is a function for CLI programs. The same goes for `input()` to get keyboard input from the user. Pygame uses other functions for input and output, which are explained later in this chapter. For now, let's look at each line in our "Hello World" program in more detail.

```
 1. import pygame, sys
```

Line 1 is a simple `import` statement that imports the `pygame` and `sys` modules so that our program can use the functions in them. All of the Pygame functions dealing with graphics, sound, and other features that Pygame provides are in the `pygame` module.

```
 2. from pygame.locals import *
```

Line 2 is also an `import` statement. However, instead of the `import modulename` format, it uses the `from modulename import *` format. Normally if you want to call a function that is in a module, you must use the `modulename.functionname()` format after importing the module. However, with `from modulename import *`, you can skip the `modulename.` portion and simply use `functionname()` (just like Python's built-in functions).

The reason we use this form of `import` statement for `pygame.locals` is because `pygame.locals` contains several constant variables that are easy to identify as being in the `pygame.locals` module without `pygame.locals.` in front of them. But for all other modules, you generally want to use the regular `import modulename` format. (There is more information about why you want to do this at http://invpy.com/namespaces.)

```
4. pygame.init()
```

Line 4 is the `pygame.init()` function, which always needs to be called after importing the `pygame` module and before calling any Pygame function. This function contains a lot of code that gets the Pygame library ready to use. You don't need to know what this function does, you just need to know that it needs to be called first in order for many Pygame functions to work. If you see an error message like, `pygame.error: font not initialized`, check to see if you forgot to call `pygame.init()`.

```
5. WINDOWSURF = pygame.display.set_mode((400, 300))
```

Line 5 is a call to the `pygame.display.set_mode()` function, which returns the `pygame.Surface` object for the window. (Surface objects are described later in this chapter.) Notice that we pass a tuple value of two integers to the function: `(400, 300)`. This tuple tells the `set_mode()` function how wide and how high to make the window in pixels. `(400, 300)` will make a window with a width of 400 pixels and height of 300 pixels. Pass a tuple of two integers to `set_mode()`, not just two integers themselves. The `pygame.Surface` object returned is stored in the `WINDOWSURF` variable. (We will just call them Surface objects for short.)

```
6. pygame.display.set_caption('Hello World!')
```

Line 6 sets the caption text that will appear at the top of the window by calling the `pygame.display.set_caption()` function. The string value `'Hello World!'` is passed in this function call to make that text appear as the caption.

Game Loops and Game States

```
7. while True: # main game loop
```

```
8.     for event in pygame.event.get():
```

Line 7 is a `while` loop that has a condition of simply the value `True`. This means that it is an **infinite loop** that never exits due to its condition being the value `False`. The only way the program execution will ever exit out of an infinite loop is if a `break` statement is executed (which moves execution to the first line after the loop) or `sys.exit()` (which terminates the program). Also, if the infinite loop is inside a function, a `return` statement will also move execution out of the loop (and the function too).

The games in this book all have infinite loops in them along with a comment calling it the "main game loop". A **game loop** or **main loop** is a loop where most of the code that handles events, updates the game state, and draws the screen is done. Main loops are common in **event-driven programming** (also called **event-based programming**), which is where objects called "events" are created when the user does some event like moves the mouse, clicks a mouse button, or pushes down (or lets up on) a key on the keyboard. Inside the main loop is code that detects which events have happened, called **event detection**. (The event detection part of the loop is really just the single call to `pygame.event.get()`.) The main loop also has code that updates the game state based on which events have been detected, called **event handling**.

The **game state** is simply a way of referring to the values in all the variables in a game program. In many games, the game state includes the values in the variables that tracks the player's health and position (and the health and position of any enemies), which marks have been made on a board, or whose turn it is. Whenever something happens like the player taking damage (which lowers their health value), or an enemy moves somewhere, or it becomes someone else's turn we say that the game state has changed.

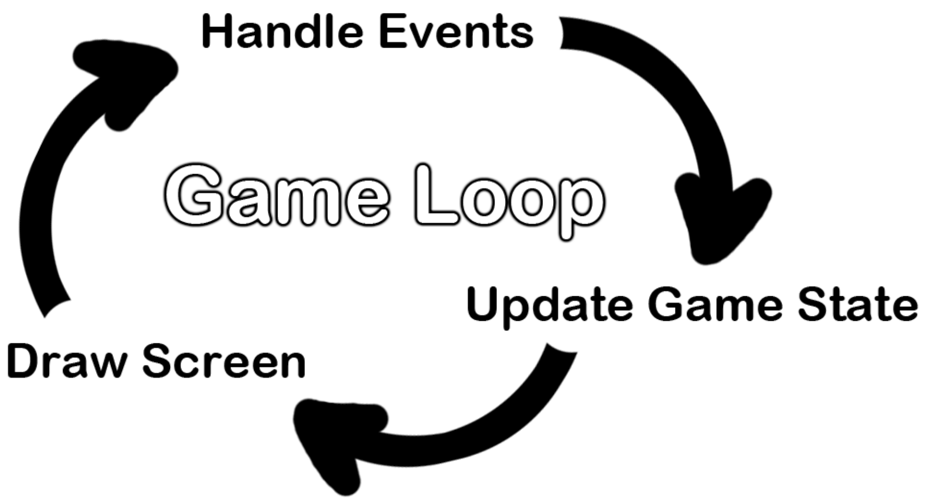
The text-based CLI games in "Invent Your Own Computer Games with Python" did not have these "game loops". That's because these programs would make calculations or display text to the screen, and then program execution would hit an `input()` function call and wait for the player to type something in. Until the player pressed the Enter key, the entire program execution stopped. These programs did not run in real-time, so nothing could really change until the player pressed Enter. The computer would either be quickly carrying out code in the program, or would be doing nothing while it waits for the player.

Pygame.event.Event Objects

GUI programs are not like that. These programs can run in real-time and update the game state instead of stopping to wait for the user. Any time the user pressed a keyboard button or clicks the mouse on the program's window, a `pygame.event.Event` object is created by the Pygame library. (This is a type of object called `Event` that exists in the `event` module, which itself is in the `pygame` module.) We can find out which events have happened by calling the `pygame.event.get()` function, which returns a list of `pygame.event.Event` objects (which we will just call `Event` objects for short.)

`Event` objects are created by the Pygame library when the user presses a keyboard button, moves the mouse, clicks a mouse button, quits the program by clicking the X in the corner of the window, or does a few other actions. The program can see all the `Event` objects that Pygame has created by calling the `pygame.event.get()` function to return a list of `Event` objects for each event that has happened since the last time the `pygame.event.get()` function was called. (Or, if `pygame.event.get()` has never been called, the events that have happened since the start of the program.)

In general, the game loop in a game program does three things over and over again until the program terminates: it handles any events (such as user input), updates the game state (possibly because of something the player did, or just things about the game state that simply change over time), and then draws a representation of game state to the screen for the player to see.



Line 8 is a `for` loop that will iterate over the list of `Event` objects that were returned by `pygame.event.get()`. On each iteration through the `for` loop, a variable named `event` will be assigned the value of one of the event objects in this list. These assignments are done in order, so if the user clicked the mouse and then pressed a keyboard key, the event variable will have a mouse click event object on the first iteration and then a keyboard press event on the second iteration. If no event have been generated, then `pygame.event.get()` will return a blank list.

The QUIT Event and pygame.quit() Function

```
 9.         if event.type == QUIT:
10.             pygame.quit()
11.             sys.exit()
```

Event objects have a **member variable** (also called **attributes**) named type which tells us what kind of event the object represents. Pygame has a constant variable for each of possible types in the pygame.locals modules. Line 9 checks if the Event object's type is equal to the constant QUIT. (Remember that since we used the from pygame.locals import * form of the import statement, we only have to type QUIT instead of pygame.locals.QUIT.)

If the Event object is a quit event, then the block following the if statement executes. This block (which covers lines 10 and 11) has a call to the pygame.quit() and sys.exit() functions. The pygame.quit() function is sort of the opposite of the pygame.init() function: it runs code that deactivates the Pygame library. Your programs should always call pygame.quit() before they call sys.exit() to terminate the program. (Normally it doesn't really matter. But there is a bug in IDLE that causes IDLE to hang if a Pygame program terminates before pygame.quit() is called.)

Since we have no if statements that run code for other types of Event object, there is no event-handling code for when the user clicks the mouse, presses keyboard keys, or causes any other Event objects to be created. The user can do these things and it doesn't change anything in the program. After the for loop on line 8 is done handling all the Event objects that have been returned by pygame.event.get(), the program execution continues to line 12.

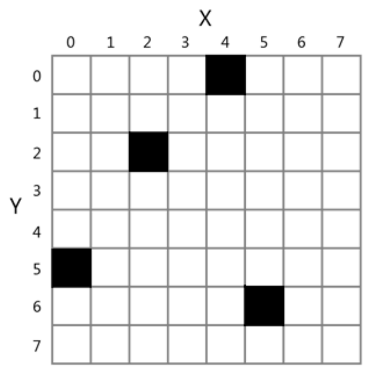
```
12.     pygame.display.update()
```

Line 12 calls the pygame.display.update() function, which draws the Surface object returned by pygame.display.set_mode() to the screen (remember we stored this object in the WINDOWSURF variable). Since the Surface object hasn't changed (by, say, some of the drawing functions that are explained later in this chapter), the same black image is redrawn to the screen each time pygame.display.update() is called.

That is the entire program. After line 12 is done, the infinite while loop on line 7 starts again. This program does nothing besides make a black window appear on the screen, constantly check for a QUIT event, and then redraws the unchanged black window to the screen over and over again. Let's learn how to make interesting things appear on this window instead of just blackness by learning about pixels, Surface objects, and the Pygame drawing functions.

Pixels and Screen Coordinates

The window that the "Hello World" program creates is just composed of little square dots on your screen called pixels. Each pixel starts off as black but can be set to a different color. Imagine that instead of a Surface object that is 400 pixels wide and 300 pixels tall, we just had a Surface object that was 8 pixels by 8 pixels. If that tiny 8x8 Surface was enlarged so that each pixel looks like a square in a grid, and we added numbers for the X and Y axis, then a good representation of it could look something like this:



We can refer to a specific pixel by using a Cartesian Coordinate system. Each column (called the **X-axis**) and each row (called the **Y-axis**) will have an "address" of an integer from 0 to 7 so that we can locate any pixel by specifying the X and Y axis.

For example, in the above 8x8 image, we can see that the pixels at the XY coordinates (4, 0), (2, 2), (0, 5), and (5, 6) have been painted black, while all the other pixels are painted white. XY coordinates are also called **points**. If you've taken a math class and learned about Cartesian Coordinates, you might notice that the Y-axis starts at 0 at the *top* and then increases going *down*, rather increasing as it goes up. This is just how Cartesian Coordinates work in Pygame (and almost every programming language.)

The Pygame framework often represents Cartesian Coordinates as a **tuple** of two integers, such as (4, 0) or (2, 2). The first integer is the X coordinate and the second is the Y coordinate. (Cartesian Coordinates are covered in more detail in chapter 12 of "Invent Your Own Computer Games with Python" at http://invpy.com/chap12)

Surface Objects and The Window

Surface objects are rectangular representations of a 2D image. You can program the computer to draw on them to change the color of their pixels, and then display a Surface object on the screen. The Surface object returned by `pygame.display.set_mode()` is called the **display**

Surface. The window border and the title bar and buttons are not part of the display Surface object.

Anything that is drawn on the display Surface object will be displayed on the window when the `pygame.display.update()` function is called. It is a lot faster to draw on a Surface object (which only exists in the computer's memory) than it is to draw a Surface object to the computer screen. Computer memory is much faster to change than pixels on a monitor. Often, your program will draw several different things to a Surface object. Once you are done drawing this image (called a **frame**, just like a still image on a paused DVD is called) on a Surface object, it can be drawn to the screen. The computer can draw frames very quickly, and our programs will often run around 30 frames per second. (This is called the "framerate" and is explained later in this chapter.)

Drawing on Surface objects will be covered in the "Primitive Drawing Functions" and "Drawing Images" sections later this chapter.

Colors

There are three primary colors of light: red, green and blue. (Red, blue, and yellow are the primary colors for paints and pigments, but the computer monitor uses light, not paint.) By combining different amounts of these three colors you can form any other color. In Python, we represent colors with tuples of three integers. The first value in the tuple is how much red is in the color. An integer value of `0` means there is no red in this color, and a value of `255` means there is a maximum amount of red in the color. The second value is for green and the third value is for blue. These tuples of three integers used to represent a color are often called **RGB values**.

Because you can use any combination of `0` to `255` for each of the three primary colors, this means Pygame can draw 16,581,375 different colors (that is, 255 x 255 x 255 colors). However, if try to use a number larger than `255` or a negative number, you will get an error that looks like "`ValueError: invalid color argument`".

For example, we will create the tuple `(0, 0, 0)` and store it in a variable named `BLACK`. With no amount of red, green, or blue, the resulting color is completely black. The color black is the absence of any color.

The tuple `(255, 255, 255)` for a maximum amount of red, green, and blue to result in white. The color white is the full combination of red, green, and blue. The tuple `(255, 0, 0)` represents the maximum amount of red but no amount of green and blue, so the resulting color is red. Similarly, `(0, 255, 0)` is green and `(0, 0, 255)` is blue.

You can mix the amount of red, green, and blue to form other colors. For example, yellow is `(255, 255, 0)`, a mixture of red and green. Combining a little bit of red and green as `(128, 128, 0)` will give you purple. RGB values are used not only in Pygame, but in many other libraries and pieces of software.

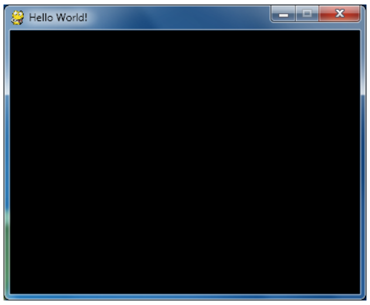
Transparent Colors

When you look through a glass window that has a deep red tint, all of the colors behind it have a red shade added to them. You can mimic this effect by adding a fourth `0` to `255` integer value to your color values.

This value is known as the **alpha value**. It is a measure of how transparent a color is. Normally when you draw a pixel onto a surface object, the new color completely replaces whatever color was already there. But with colors that have an alpha value, you can instead just add a colored tint to the color that is already there.

For example, this tuple of three integers is for the color green: `(0, 255, 0)`. But if we add a fourth integer for the alpha value, we can make this a half transparent green color: `(0, 255, 0, 128)`. An alpha value of `255` is completely opaque, that is, no transparency at all (the colors `(0, 255, 0)` and `(0, 255, 0, 255)` look exactly the same. An alpha value of `0` means the color is completely transparent. If you draw any color that has an alpha value of `0` to a surface object, it will have no effect, because this color is completely transparent and invisible.

If we were to create a color tuple to draw the legendary Invisible Pink Unicorn, we would use `(255, 192, 192, 0)`, which ends up looking completely invisible just like any other color that has a 0 for its alpha value. (How people can know what color an invisible unicorn is, I have no idea.)



(Above is a screenshot of a Pygame program that has a drawing of the Invisible Pink Unicorn.)

pygame.Color Objects

You need to know how to represent a color because Pygame's drawing functions need a way to know what color you want to draw with. A tuple of three or four integers is one way. Another way is as a `pygame.Color` object. You can create `Color` objects by calling the `pygame.Color()` constructor function and passing either three or four integers. You can store this `Color` object in variables just like you can store tuples in variables. Try typing the following into the interactive shell:

```
>>> import pygame
>>> pygame.Color(255, 0, 0)
(255, 0, 0, 255)
>>> myColor = pygame.Color(255, 0, 0, 128)
>>> myColor == (255, 0, 0, 128)
True
>>>
```

Any drawing function in Pygame (which we will learn about in a bit) that has a color parameter can have either the tuple form or `Color` object form of a color passed for it.

Now that you know how to represent colors (as a `pygame.Color` object or a tuple of three or four integers for red, green, blue, and optionally alpha) and coordinates (as a tuple of two integers for X and Y), let's learn about `pygame.Rect` objects so we can start using Pygame's drawing functions.

Rect Objects

Remember that in Pygame we can represent points as a tuple of two integers for the X and Y coordinate. Pygame also has two ways to represent rectangular areas. The first is a tuple of four integers for the X coordinate of the top left corner, the Y coordinate of the top left corner, and the width and then height of the rectangle. The second way is as a `pygame.Rect` object, which we will call Rect objects for short. (This is just like how colors can be represented as a tuple or a `Color` object.)

For example, this creates a Rect object with a top left corner at (10, 20) that is 200 pixels wide and 300 pixels tall:

```
>>> import pygame
>>> spamRect = pygame.Rect(10, 20, 200, 300)
```

The handy thing about this is that the Rect object automatically calculates the coordinates for other features of the rectangle. For example, if you need to know the x coordinate of the right edge of the `pygame.Rect` object we stored in the `spamRect` variable, you can just access the Rect object's `right` attribute:

```
>>> spamRect.right
210
```

The Pygame code for the Rect object automatically calculated that if the left edge is at the X coordinate 10 and the rectangle is 200 pixels wide, then the right edge must be at the X coordinate 210. If you reassign the right attribute, all the other attributes are automatically recalculated:

```
>>> spam.right = 350
>>> spam.left
150
```

Here's a list of all the attributes that `pygame.Rect` objects provide (in our example, the variable where the Rect object is stored is named `myRect`):

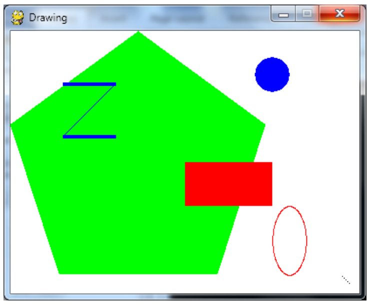
Attribute Name	**Description**
`myRect.left`	The int value of the X-coordinate of the left side of the rectangle.
`myRect.right`	The int value of the X-coordinate of the right side of the rectangle.
`myRect.top`	The int value of the Y-coordinate of the top side of the rectangle.
`myRect.bottom`	The int value of the Y-coordinate of the bottom side.
`myRect.centerx`	The int value of the X-coordinate of the center of the rectangle.
`myRect.centery`	The int value of the Y-coordinate of the center of the rectangle.
`myRect.width`	The int value of the width of the rectangle.
`myRect.height`	The int value of the height of the rectangle.
`myRect.size`	A tuple of two ints: (width, height)
`myRect.topleft`	A tuple of two ints: (left, top)
`myRect.topright`	A tuple of two ints: (right, top)
`myRect.bottomleft`	A tuple of two ints: (left, bottom)
`myRect.bottomright`	A tuple of two ints: (right, bottom)
`myRect.midleft`	A tuple of two ints: (left, centery)
`myRect.midright`	A tuple of two ints: (right, centery)
`myRect.midtop`	A tuple of two ints: (centerx, top)
`myRect.midbottom`	A tuple of two ints: (centerx, bottom)

Primitive Drawing Functions

Pygame provides several different functions for drawing different shapes onto a surface object. These shapes such as rectangles, circles, ellipses, lines, or individual pixels are often called **drawing primitives**. Open IDLE's file editor and type in the following program, and save it as *drawing.py*.

```
import pygame, sys
from pygame.locals import *

pygame.init()

# set up the window
WINDOWSURF = pygame.display.set_mode((500, 400), 0, 32)
pygame.display.set_caption('Drawing')

# set up the colors
BLACK = (  0,   0,   0)
WHITE = (255, 255, 255)
RED   = (255,   0,   0)
GREEN = (  0, 255,   0)
BLUE  = (  0,   0, 255)

# draw on the surface object
WINDOWSURF.fill(WHITE)
pygame.draw.polygon(WINDOWSURF, GREEN, ((146, 0), (291, 106), (236, 277),
(56, 277), (0, 106)))
pygame.draw.line(WINDOWSURF, BLUE, (60, 60), (120, 60), 4)
pygame.draw.line(WINDOWSURF, BLUE, (120, 60), (60, 120))
pygame.draw.line(WINDOWSURF, BLUE, (60, 120), (120, 120), 4)
pygame.draw.circle(WINDOWSURF, BLUE, (300, 50), 20, 0)
pygame.draw.ellipse(WINDOWSURF, RED, (300, 250, 40, 80), 1)
pygame.draw.rect(WINDOWSURF, RED, (200, 150, 100, 50))

pixObj = pygame.PixelArray(WINDOWSURF)
pixObj[480][380] = BLACK
pixObj[482][382] = BLACK
pixObj[484][384] = BLACK
pixObj[486][386] = BLACK
pixObj[488][388] = BLACK
del pixObj

# run the game loop
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    pygame.display.update()
```

When this program is run, the following window is displayed until the user closes the window:

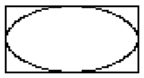


Notice how we make constant variables for each of the colors. Doing this makes our code more readable, because seeing `GREEN` in the source code is much easier to understand as representing the color green than `(0, 255, 0)` is.

The drawing functions are named after the shapes they draw. The parameters you pass these functions tell them which surface to draw on, where to draw the shape (and what size), in what color, and how wide to make the lines. You can see how these functions are called in the *drawing.py* program, but here is a short description of each function:

- **fill(color)** – The `fill()` method is not a function but a method of `pygame.Surface` objects. It will completely fill in the entire Surface object with whatever color value you pass as for the `color` parameter. It is often called on the display Surface to start drawing each frame from scratch.
- **pygame.draw.polygon(surface, color, pointlist, width)** – A polygon is shape made up of only flat sides. Triangles, squares, and pentagons are all examples of polygons. Circles are not polygons because they do not have flat sides. The `surface` and `color` parameters tell the function on what surface to draw the polygon, and what color to make it.
  The third parameter is a tuple or list of points (that is, tuple or list of two-integer tuples for XY coordinates.) The polygon is drawn by drawing lines between each point and the point that comes after it in the tuple. Then a line is drawn from the last point to the first point. You can also pass a list of points instead of a tuple of points.
  The `width` parameter is optional. If you leave it out, the polygon that is drawn will be filled in, just like our green polygon on the screen is filled in with the green color. If you do pass an integer value for the `width` parameter, only the outline of the polygon will be drawn. The integer represents how many pixels width the polygon's outline will be. Passing `1` for the `width` parameter will make a skinny polygon, while passing `4` or `10` or `20` will make

thicker polygons. If you pass the integer `0` for the `width` parameter, the polygon will be filled in (just like if you left the `width` parameter out entirely.)
All of the `pygame.draw` drawing functions have optional `width` parameters at the end, and they work the same way as `pygame.draw.polygon()`'s `width` parameter. Probably a better name for the `width` parameter would have been "thickness", since that parameter controls how thick the shapes you draw are.

- **pygame.draw.line(surface, color, start_point, end_point, width)** – This function draws a line between the `start_point` and `end_point` parameters. You can also set what color and the width of the line. (Pygame draws really, really thick lines kind of funny. Most of the time this doesn't matter, but you can read http://invpy.com/thicklines for more information.)
- **pygame.draw.lines(surface, color, closed, pointlist, width)** – This function draws a series of lines from one point to the next, much like `pygame.draw.polygon()`. The only difference is that if you pass `False` for the `closed` parameter, there will not be a line from the last point in the `pointlist` parameter to the first point. (If you pass `True`, then it will draw a line from the last point to the first.)
- **pygame.draw.circle(surface, color, center_point, radius, width)** – This function draws a circle. The center of the circle is at the `center_point` parameter. The size of the circle is determined by the integer passed for the `radius` parameter.
  The radius of a circle is the distance from the center to the edge. (The radius of a circle is always half of the diameter.) Passing `20` for the `radius` parameter will draw a circle that has a radius of 20 pixels.
- **pygame.draw.ellipse(surface, color, bounding_rectangle, width)** – This function draws an ellipse, which is like a squashed or stretched circle. This function has all the usual parameters, but in order to tell the function how large and where to draw the ellipse, you must specify the bounding rectangle of the ellipse. A **bounding rectangle** is the smallest rectangle that can be drawn around a shape. Here's an example of an ellipse and its bounding rectangle:
  
  Remember that instead of passing a tuple of four integers for the `bounding_rectangle` parameter, you can pass a `pygame.Rect` object. Note that you do not specify the center point for the ellipse like you do for the `pygame.draw.circle()` function, just the bounding rectangle.
- **pygame.draw.rect(surface, color, rectangle_tuple, width)** – This function draws a rectangle. The `rectangle_tuple` is either a tuple of four integers (for the XY coordinates of the top left corner, and the width and height) or a `pygame.Rect` object can be passed instead. If the `rectangle_tuple` has the same size for the width and height, a square will be drawn.

pygame.PixelArray Objects

Unfortunately, there isn't a single function you can call that will set a single pixel to a color (unless you call `pygame.draw.rect()` with a 1x1 rectangle or call `pygame.draw.line()` with the same start and end point). The Pygame framework needs to run some code behind the scenes before and after drawing to a Surface object. If it had to do this for every single pixel you wanted to set, your program would run much slower. (By my quick testing, drawing pixels becomes two or three times slower.)

Instead, you should tell Pygame to do the set up just once. This is what line 27 does when it creates a `pygame.PixelArray` object (we'll call them PixelArray objects for short). The Surface object that we want to draw pixels on is passed as an argument. Creating a PixelArray object of a Surface object will "lock" the Surface object. While a Surface object is locked, the drawing functions can still be called on it, but it cannot have images like PNG or JPG images drawn on it with the `blit()` method. (This method is explained later in this chapter.)

The PixelArray object that is returned can have individual pixels set by accessing them with two indexes. For example, line 28's `pixObj[480][380] = BLACK` will set the pixel at X coordinate 480 and Y coordinate 380 to be black (remember that the `BLACK` variable stores the color tuple `(0, 0, 0)`).

To tell Pygame that you are finished drawing individual pixels, delete the PixelArray object with a `del` statement. This is what line 33 does. Deleting the PixelArray object will "unlock" the Surface object so that you can once again draw images on it. If you forget to delete the PixelArray object, the next time who try to blit (that is, draw) an image to the Surface, the program will raise an error that says, "`pygame.error: Surfaces must not be locked during blit`".

The `pygame.display.update()` Function

After you are done calling the drawing functions to make the display Surface object look the way you want, you must call `pygame.display.update()` to make the display Surface actually appear on the user's monitor.

The one thing that you must remember is that `pygame.display.update()` will only make the display Surface (that is, the `pygame.Surface` object that was returned from the call to `pygame.display.set_mode()`) appear on the screen. If you want the images on other Surface objects to appear on the screen, you must "blit" them to the display Surface object first with the `blit()` method (which is explained next in the "Drawing Images" section.)

Drawing Images

Making good graphics by just using the drawing functions can be hard. It's usually much easier to use graphics software to draw images and then save them to an image file. Or you can just go on the Internet and download image files that you want use in your games. Pygame has functions for loading and displaying images stored in these files on the screen. The differences between these image file formats is explained at http://invpy.com/formats, but in general I recommend always using PNG images.

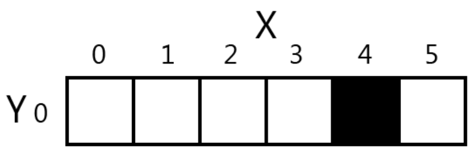
You can load an image file with the pygame.image.load() function. It has one parameter, which is a string of the filename of the image file. The function then returns a Surface object that has the image drawn on it. This function stores a Surface object of some cat picture in the `catImg` variable:

```
catImg = pygame.image.load('cat.png')
```

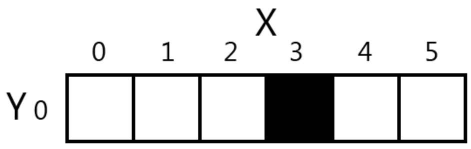
Of course, only the display Surface object can be drawn to the screen when `pygame.display.update()` is called. So we have to copy the `catImg` Surface object to the display Surface object by "blitting" it to the display Surface. Blitting is done with the `blit()` method, which is a method of Surface objects. Here is a `blit()` method call that will copy

Animation

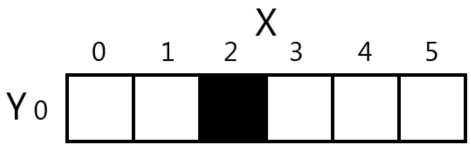
Now that we know how to get the Pygame framework to draw to the screen, let's learn how to make animated pictures. A game with only still, unmoving images would be fairly dull. (Sales of my games "Look At This Rock" and "Look At This Rock 2: A Different Rock" have been disappointing. http://invpy.com/rock) Animated images are the result of drawing an image on the screen, then a split second later drawing a slightly different image on the screen. Imagine the program's window was 6 pixels wide and 1 pixel tall, with all the pixels white except for a black pixel at 4, 0. It would look like this:



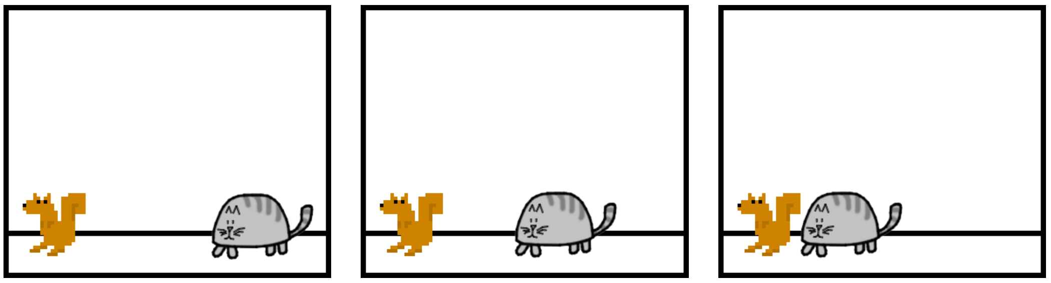
If you changed the window so that 3, 0 was black and 4,0 was white, it would look like this:



To the user, it looks like the black pixel has “moved” over to the left. If you redrew the window to have the black pixel at 2, 0, it would continue to look like the black pixel is moving left:



It may look like the black pixel is moving, but this is just an illusion. To the computer, it is just showing three different images that each have just one black pixel. Consider if the three following images were rapidly shown on the screen:



To the user, it would look like the cat is moving towards the squirrel. But to the computer, they’re just a bunch of pixels. The trick to making believable looking animation is to have your program draw a picture to the window, wait a fraction of a second, and then draw a slightly different picture.

Here is an example program demonstrating a simple animation. Type this code into IDLE’s file editor and save it as *catanimation.py*. It will also require the image file cat.png to be in the same folder as the *catanimation.py* file. You can download this image from http://invpy.com/cat.png. This code is available at http://invpy.com/catanimation.py

```
import pygame, sys
from pygame.locals import *

pygame.init()

FPS = 30 # frames per second setting
fpsClock = pygame.time.Clock()
```

```

# set up the window
WINDOWSURF = pygame.display.set_mode((400, 300), 0, 32)
pygame.display.set_caption('Animation')

WHITE = (255, 255, 255)
catImg = pygame.image.load('cat.png')
catx = 10
caty = 10
direction = 'right'

while True: # the main game loop
    WINDOWSURF.fill(WHITE)

    if direction == 'right':
        catx += 5
        if catx == 280:
            direction = 'down'
    elif direction == 'down':
        caty += 5
        if caty == 220:
            direction = 'left'
    elif direction == 'left':
        catx -= 5
        if catx == 10:
            direction = 'up'
    elif direction == 'up':
        caty -= 5
        if caty == 10:
            direction = 'right'

    WINDOWSURF.blit(catImg, (catx, caty))

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

    pygame.display.update()
    fpsClock.tick(FPS)
```

Frames Per Second and pygame.time.Clock Objects

The number of pictures that the program draws per second is commonly called the **framerate**, and is measured in **FPS** or **frames** per second. (On computer monitors, the common name for

FPS is hertz. Many monitors have a framerate of 60 hertz, or 60 frames per second.) A low frame rate in video games can make the game look choppy or jumpy. This is usually caused when the program has to compute a large amount of code in between drawing frames to the screen. If the program cannot run fast enough to draw to the screen frequently, then the FPS goes down. (But the games in this book are simple enough that this won't be issue even on old computers.)

A `pygame.time.Clock` object can help us make sure our program runs at a certain FPS. This `Clock` object will set the general speed that our programs run at by putting in small pauses on each iteration of the game loop. If we didn't have these pauses, our game program would run as fast as the computer could run it. This is often too fast for the player, and as computers get faster they would run the game faster too. A `Clock` object makes sure the game runs at the same speed no matter how fast of a computer it runs on. The `Clock` object is created on line 7 of the *animation.py* program.

```
7. fpsClock = pygame.time.Clock()
```

The `Clock` object's `tick()` method is called to put a pause in the program. This method should be called at the very end of the game loop and after the call to `pygame.display.update()`. The length of the pause is calculated based on how long it took to execute the code in the game loop. In the animation program, is it run on line 47 right before the end of the game loop.

```
47.     fpsClock.tick(FPS)
```

Try modifying the `FPS` constant variable to run the same program at different frame rates. Setting it to a lower value would make the program run slower. Setting it to a higher value would make the program run faster.

Drawing Images

The drawing functions are fine if you want to draw simple shapes on the screen, but many games have images (also called **sprites**) as the main part of their graphics. Pygame is able to draw images to Surface objects from PNG, JPG, GIF, and BMP files. (The differences between these image file formats is described at http://invpy.com/formats.)

The string with the image file's filename is passed to the `pygame.image.load()` function, which then returns a Surface object that has the image drawn on it. (If you get an error message like "`pygame.error: Couldn't open cat.png`", then make sure the *cat.png* file is in the same folder as the *animation.py* file before you run the program.) This Surface object will be a separate Surface from the display Surface object, so we must blit (that is, copy) the image's

Surface object to the display Surface object. **Blitting** is a name for drawing the contents of one Surface onto another and in Pygame it is done with the `blit()` method of Surface objects.

```
39.     WINDOWSURF.blit(catImg, (catx, caty))
```

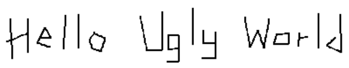
Line 39 of the animation program calls the `blit()` method on the Surface object stored in `WINDOWSURF`. There are two parameters for `blit()`. The first is the source Surface object, which is what will be copied onto the `WINDOWSURF` Surface object. The second parameter is a two-integer tuple for the x and y values of the topleft corner where the image should be blitted to.

If `catx` and `caty` were set to `100` and `200` and the width of `catImg` was `25` and the height was `30`, this blit() call would copy this image onto `WINDOWSURF` so that the topleft corner of the catImg was at the XY coordinate (100, 200) and the bottomright XY coordinate was (125, 230).

The rest of the game loop is just changing the `catx`, `caty`, and `direction` variables so that the cat moves around the window. There is also a call to `pygame.event.get()` to handle the `QUIT` event.

Fonts

If you want to draw text to the screen, you *could* write several calls to `pygame.draw.line()` to draw out lines of each letter. This would be a headache to type out all those `pygame.draw.line()` calls and figure out all the XY coordinates, and probably wouldn't look very good.



The above message would take forty one calls to the `pygame.draw.line()` function to make. Instead, Pygame provides some much simpler functions for fonts and creating text. Here is a small Hello World program using Pygame's font functions. Type it into IDLE's file editor and save it as *fonttext.py*:

```
import pygame, sys
from pygame.locals import *

pygame.init()
WINDOWSURF = pygame.display.set_mode((400, 300))
pygame.display.set_caption('Hello World!')

WHITE = (255, 255, 255)
GREEN = (0, 255, 0)
```

```
10. BLUE = (0, 0, 128)
11.
12. fontObj = pygame.font.Font('freesansbold.ttf', 32)
13. textSurfaceObj = fontObj.render('Hello world!', True, GREEN, BLUE)
14. textRectObj = textSurfaceObj.get_rect()
15. textRectObj.center = (200, 150)
16.
17. while True: # main game loop
18.     WINDOWSURF.fill(WHITE)
19.     WINDOWSURF.blit(textSurfaceObj, textRectObj)
20.     for event in pygame.event.get():
21.         if event.type == QUIT:
22.             pygame.quit()
23.             sys.exit()
24.     pygame.display.update()
```

There are six steps to making text appear on the screen:

1. Create a `pygame.font.Font object`. (Like on line 12)
2. Create a Surface object with the text drawn on it by calling the Font object's `render()` method. (Line 13)
3. Create a Rect object from the Surface object by calling the Surface object's `get_rect()` method. (Line 14)
4. Set the position of the Rect object by changing one of its attributes. On line 15, we set the center of the Rect object to be at 200, 150.
5. Blit the Surface object with the text onto the Surface object returned by `pygame.display.set_mode()`. (Line 19)
6. Call `pygame.display.update()` to make the display Surface appear on the screen. (Line 24)

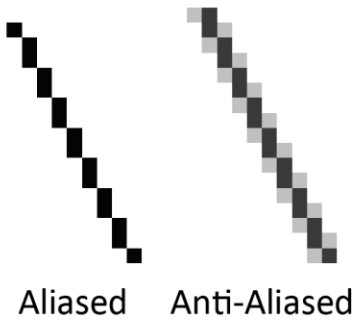
The parameters to the `pygame.font.Font()` constructor function is a string of the font file to use, and an integer of the size of the font (in points, like how word processors measure font size). On line 12, we pass `'freesansbold.ttf'` (this is a font that comes with Pygame) and the integer 32 (for a 32-point sized font.)

The parameters to the `render()` method call are a string of the text to render, a Boolean value to specify if we want anti-aliasing (explained later in this chapter), the color of the text, and the color of the background. If you want a transparent background, then simply leave off the background color parameter in the method call.

Anti-Aliasing

Anti-aliasing is a graphics technique for making text and shapes look less blocky by adding a little bit of blur to their edges. It takes a little more computation time to do anti-aliasing, so although the graphics may look better, your program may run slower (but only just a little).

If you zoom in on an aliased line and an anti-aliased line, they look like this:



To make Pygame's text use anti-aliasing, just pass `True` for the second parameter of the `render()` method. There are also two other primitive drawing functions. The `pygame.draw.aaline()` and `pygame.draw.aalines()` functions have the same parameters as `pygame.draw.line()` and `pygame.draw.lines()`, except they will draw anti-aliased (smooth) lines instead of aliased (blocky) lines.

Playing Sounds

Playing sounds that are stored in sound files is even simpler than displaying images from image files. First, you must create a `pygame.mixer.Sound` object (which we will call Sound objects for short) by calling the `pygame.mixer.Sound()` constructor function. It takes one string parameter, which is the filename of the sound file. Pygame can load WAV, MP3, or OGG files. (The difference between these audio file formats is explained at http://invpy.com/formats.) To play this sound, call the Sound object's `play()` method. If you want to stop the Sound object from playing immediately, just call the `stop()` method. The `stop()` method has no arguments. Here is some sample code:

```
soundObj = pygame.mixer.Sound('beeps.wav')
soundObj.play()
import time
time.sleep(1) # wait a second and let the sound play
soundObj.stop()
```

The play() method call will immediately return and keep playing the sound while the program execution continues on.

The Sound objects are good for sound effects to play when the player takes damage, slashes a sword, or collects a coin. But your games might also be better if they had background music playing. To load a background music file, call the pygame.mixer.music.load() function and pass it a string argument of the sound file to load. This file can be WAV, MP3, or MID format. To begin playing the sound file as the background music, call the pygame.mixer.music.play(-1, 0.0) function. The -1 argument makes the background music loop when it reaches the end of the sound file. (If you set it to an integer 0 or larger, then the music will only loop that number of times instead of looping forever.) The 0.0 means to start playing the sound file from the beginning. (If you set it to a larger integer, the music will begin playing that many seconds into the sound file. For example, if you pass 13.5 for the second parameter, the sound file with begin playing at the point 13.5 seconds from the beginning.)

To stop playing the background music immediately, call the pygame.mixer.music.stop() function. This function has no arguments.

Here is some example code of the sound methods and functions:

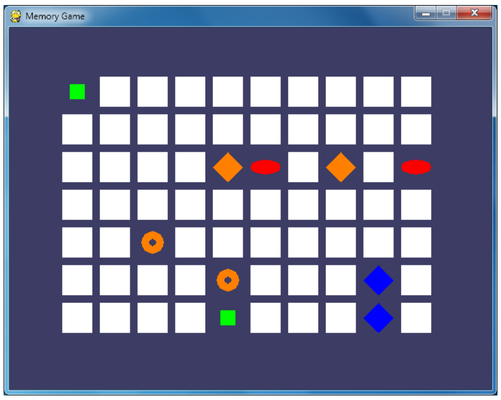
```
# Loading and playing a sound effect:
soundObj = pygame.mixer.Sound('beepingsound.wav')
soundObj.play()

# Loading and playing background music:
pygame.mixer.music.load(backgroundmusic.mp3')
pygame.mixer.music.play(-1, 0.0)
...some more code...
pygame.mixer.music.stop()
```

Summary

This covers the basics of making graphical games with the Pygame framework. Of course, just reading about these functions probably isn't enough to help you learn how to make games using these functions. The rest of the chapters in this book each focus on the source code for a small, complete game. This will give you an idea of what complete game programs "look like", so you can then apply those same ideas to your own game programs.

Unlike the "Invent Your Own Computer Games with Python" book, this book assumes that you know the basics of Python programming. If you have trouble remembering how variables, functions, loops, if-else statements, and conditions work, you can probably figure it out just by seeing what’s in the code and how the program behaves. But if you are still stuck, you can read the “Invent with Python” book (it’s for people who are completely new to programming) for free online at http://inventwithpython.com.

Chapter 3 – Memory Puzzle



How to Play Memory Puzzle

In the Memory Puzzle game, several shapes covered up by white boxes. There are two of each possible shape and color combination (in this game, they're called icons). The player can click on two boxes to see what shape is behind them. If the shapes match, then those boxes remain uncovered. The player wins when all the boxes are uncovered. To give the player a hint, the boxes are quickly uncovered once at the beginning of the game.

Nested `for` Loops

One concept that you will see in Memory Puzzle (and most of the games in this book) is the use of a `for` loop inside of another `for` loop. These are called nested `for` loops. Nested `for` loops are handy for going through every possible combination of two lists. Type the following into the interactive shell:

```
>>> for x in [0, 1, 2, 3, 4]:
...     for y in ['a', 'b', 'c']:
...         print(x, y)
...
0 a
0 b
0 c
1 a
1 b
1 c
2 a
2 b
```

```
2 c
3 a
3 b
3 c
4 a
4 b
4 c
>>>
```

There are several times in the Memory Puzzle code that we need to iterate through every possible X and Y coordinate on the board. We'll use nested `for` loops to make sure that we get every combination. Note that the inner `for` loop (the `for` loop inside the other `for` loop) will go through all of its iterations before going to the next iteration of the outer `for` loop. If we reverse the order of the `for` loops, the same values will be printed but they will be printed in a different order. Type the following code into the interactive shell, and compare the order it prints values to the order in the previous nested `for` loop example:

```
>>> for y in ['a', 'b', 'c']:
...     for x in [0, 1, 2, 3, 4]:
...         print(x, y)
...
0 a
1 a
2 a
3 a
4 a
0 b
1 b
2 b
3 b
4 b
0 c
1 c
2 c
3 c
4 c
>>>
```

Source Code of Memory Puzzle

This source code can be downloaded from http://invpy.com/memorypuzzle.py.

Go ahead and first type in the entire program into IDLE's file editor, save it as *memorypuzzle.py*, and run it. If you get any error messages, look at the line number that is mentioned in the error message and check your code for any typos. You can also copy and paste your code into the web form at http://invpy.com/diff to see if the differences between your code and the code in the book.

You'll probably pick up a few ideas about how the program works just by typing it in once. And when you're done typing it in, you can then play the game for yourself.

Program Credits and Imports

```
1. # Memory Puzzle
2. # By Al Sweigart al@inventwithpython.com
3. # http://inventwithpython.com/pygame
4. # Creative Commons BY-NC-SA 3.0 US
5.
6. import random, pygame, sys
7. from pygame.locals import *
8.
```

At the top of the program are comments about what the game is, who made it, and where the user could find more information. There's also a note that the source code is freely copyable under a Creative Commons license (just like this book!) The program makes use of many functions in other modules, so it imports those modules on line 6. Line 7 is also an `import` statement in the `from (module name) import *` format, which means you do not have to type the module name in front of it. There are no functions in the `pygame.locals` module, but there are several constant variables in it that we want to use such as `MOUSEMOTION`, `KEYUP`, or `QUIT`. Using this style of `import` statement, we only have to type `MOUSEMOTION` rather than `pygame.locals.MOUSEMOTION`.

Magic Numbers are Bad

```
FPS = 30 # frames per second, the general speed of the program
WINDOWWIDTH = 640 # size of window's width in pixels
WINDOWHEIGHT = 480 # size of windows' height in pixels
REVEALSPEED = 8 # speed boxes' sliding reveals and covers
BOXSIZE = 40 # size of box height & width in pixels
GAPSIZE = 10 # size of gap between boxes in pixels
```

The game programs in this book use a lot of constant variables. You might not realize why they're so handy. Instead of using, say the `BOXSIZE` variable in our code we just type the integer `40` directly in the code when we need it? There are two reasons.

First, if we ever wanted to change the size of each box later, we would have to go through the entire program and find and replace each time we typed `40`. By just using the `BOXSIZE` constant, we only have to change line 13 and the rest of the program is already up to date. This is much better, especially since we might use the integer value `40` for something else besides the size of the white boxes, and changing that `40` accidentally would cause bugs in our program.

Second, it makes the code more readable. Go down to the next section and look at line 18. This sets up a calculation for the `XMARGIN` constant, which is how many pixels are on the side of the entire board. It is a complicated looking expression, but you can carefully piece out what it means. Line 18 looks like this:

```
XMARGIN = int((WINDOWWIDTH - (BOARDWIDTH * (BOXSIZE + GAPSIZE))) / 2)
```

But if line 18 didn't use constant variables, it would look like this:

```
XMARGIN = int((640 – (10 * (40 + 10))) / 2)
```

Now it becomes impossible to remember what exactly the programmer intended to mean. These unexplained numbers in the source code are often called **magic numbers**. Whenever you find yourself entering magic numbers, you should consider replacing them with a constant variable instead. To the Python interpreter, both of the previous lines are the exact same. But to a human programmer who is reading the source code and trying to understand how it works, the second version of line 18 doesn't make much sense at all! Constants really help the readability of source code.

Of course, you can go too far replacing numbers with constant variables. Look at the following code:

```
ZERO = 0
ONE = 1
TWO = 99999999
TWOANDTHREEQUARTERS = 2.75
```

Don't write code like that. That's just silly.

Sanity Checks with `assert` Statements

```
 15. BOARDWIDTH = 10 # number of columns of icons
 16. BOARDHEIGHT = 7 # number of rows of icons
 17. assert (BOARDWIDTH * BOARDHEIGHT) % 2 == 0, 'Board needs to have an even
number of boxes for pairs of matches.'
 18. XMARGIN = int((WINDOWWIDTH - (BOARDWIDTH * (BOXSIZE + GAPSIZE))) / 2)
```

```
19. YMARGIN = int((WINDOWHEIGHT - (BOARDHEIGHT * (BOXSIZE + GAPSIZE))) / 2)
20.
```

The `assert` statement on line 15 ensures that the board width and height we've selected will result in an even number of boxes (since we will have pairs of icons in this game.) There are three parts to an `assert` statement: the `assert` keyword, an expression which, if `False`, results in crashing the program. The third part (after the comma after the expression) is a string that appears if the program crashes because of the assertion.

The assert statement with an expression basically says, "The programmer asserts that this expression must be `True`, otherwise crash the program." This is a good way of adding a sanity check to your program to make sure that if the execution ever passes an assertion, then we can at least know that that code is working as expected.

Telling If a Number is Even or Odd

If the product of the board width and height is divided by two and has a remainder of 0 (the `%` **modulous operator** evaluates what the remainder is) then the number is even. Even numbers divided by two will always have a remainder of zero. Odd numbers divided by two will always have a remainder of one. This is a good trick to remember if you need your code to tell if a number is even or odd.

Crash Early and Crash Often!

Having your program crash is a bad thing. It happens when your program has some mistake in the code. So why would anyone want their program to crash?

If the values we chose for `BOARDWIDTH` and `BOARDHEIGHT` that we chose on line 15 and 16 result in a board with an odd number of boxes (such as if the width were 3 and the height were 5), then there would always be one left over icon that would not have a pair to be matched with. This would cause a bug later on in the program, and it could take a lot of debugging work to figure out that the real source of the bug is at the very beginning of the program. In fact, just for fun, try commenting out the assertion so it doesn't run, and then setting the `BOARDWIDTH` and `BOARDHEIGHT` constants both to odd numbers. When you run the program, it will immediately show an error happening on a line 149 in random.py, which is in `getRandomizedBoard()` function!

```
Traceback (most recent call last):
  File "C:\book2svn\src\memorypuzzle.py", line 292, in <module>
    main()
  File "C:\book2svn\src\memorypuzzle.py", line 58, in main
    mainBoard = getRandomizedBoard()
  File "C:\book2svn\src\memorypuzzle.py", line 149, in getRandomizedBoard
```

```
    columns.append(icons[0])
IndexError: list index out of range
```

We could be spending a lot of time looking at `getRandomizedBoard()` trying to figure out what's wrong with it, before realizing that `getRandomizedBoard()` is perfectly fine: the real source of the bug was on line 15 and 16 where we set the `BOARDWIDTH` and `BOARDHEIGHT` constants.

The assertion makes sure that this never happens. If our code is going to crash, we want it to crash as soon as it detects something is terribly wrong, because otherwise the bug may not become apparent until much later in the program. Crash early!

You want to add `assert` statements whenever there is some condition in your program that must always, always, always be `True` (or always, always, always `False`, and just put the `not` operator in front of the `assert` statement's expression.) You don't have to go overboard and put `assert` statements everywhere, but crashing often with asserts goes a long way in detecting the true source of a bug. Crash early and crash often! (In your code, that is. Not, say, when riding a pony.)

Making the Source Code Look Pretty

```
21. #            R    G    B
22. GRAY     = (100, 100, 100)
23. NAVYBLUE = ( 60,  60, 100)
24. WHITE    = (255, 255, 255)
25. RED      = (255,   0,   0)
26. GREEN    = (  0, 255,   0)
27. BLUE     = (  0,   0, 255)
28. YELLOW   = (255, 255,   0)
29. ORANGE   = (255, 128,   0)
30. PURPLE   = (255,   0, 255)
31. CYAN     = (  0, 255, 255)
32.
33. BGCOLOR = NAVYBLUE
34. LIGHTBGCOLOR = GRAY
35. BOXCOLOR = WHITE
36. HIGHLIGHTCOLOR = BLUE
37.
```

Remember that colors in Pygames are represented by a tuple of three integers from `0` to `255`. These three integers represent the amount of red, green, and blue in the color which is why the tuples are called RGB values. Notice the spacing of the tuples on lines 22 to 31 are such that the R, G, and B integers line up. In Python the indentation (the space at the beginning of the line) is significant, but the spacing in the rest of the line is not so strict. By spacing the integers in the tuple out, we can clearly see how the RGB values compare to each other.

This is a nice thing to make your code more readable, but don't bother spending too much time doing it. Code doesn't have to be pretty to work. At a certain point, you'll just be spending more time typing spaces than you would have saved by having readable tuple values.

Using Constant Variables Instead of Strings

```
DONUT = 'donut'
SQUARE = 'square'
DIAMOND = 'diamond'
LINES = 'lines'
OVAL = 'oval'

```

The program also sets up constant variables for some strings. These constants will be used in the data structure for the board, tracking which spaces on the board have which icons. These strings aren't magic numbers, so you might wonder why we use constants for them. Look at the following code, which comes from line 187:

```
    if shape == DONUT:
```

The shape variable will be set to one of the strings `'donut'`, `'square'`, `'diamond'`, `'lines'`, or `'oval'` and then compared to the `DONUT` constant. If we made a typo when writing line 187, say, something like this:

```
    if shape == DUNOT:
```

Then Python would crash, giving an error message saying that there is no variable named `DUNOT`. This is good. Since the program has crashed on line 187, when we check that line it will be easy to see that the bug was caused by a typo. However, if we were using strings instead of constant variables and made the same typo, line 187 would look like this:

```
    if shape == 'dunot':
```

This is perfectly acceptable Python code, so it won't crash. However, this will lead to weird bugs later on in our program. Because the code does not immediately crash where the problem is caused, it can be much harder to find it.

Making Sure We Have Enough Icons

```
 44. ALLCOLORS = (RED, GREEN, BLUE, YELLOW, ORANGE, PURPLE, CYAN)
 45. ALLSHAPES = (DONUT, SQUARE, DIAMOND, LINES, OVAL)
 46. assert len(ALLCOLORS) * len(ALLSHAPES) * 2 >= BOARDWIDTH * BOARDHEIGHT,
"Board is too big for the number of shapes/colors defined."
 47.
```

In order for our game program to be able to create icons of every possible color and shape combination, we need to make a tuple that holds all of these values. (It just happens that these tuples never change, so they're also a constant.) There is also another assertion on line 46 to make sure that there are enough color/shape combinations for the size of the board we have. If there isn't, then the program will crash on line 46 and we will know that we either have to add more colors and shapes, or make the board width and height smaller. With 7 colors and 5 shapes, we can make 35 (that is, 7 x 5) different icons. And because we'll have a pair of each icon, that means we can have a board with up to 70 (that is, 35 x 2, or 7 x 5 x 2) spaces.

Tuples vs. Lists, Immutable vs. Mutable

You might have noticed that the `ALLCOLORS` and `ALLSHAPES` variables are tuples instead of lists. When do we want to use tuples and when do we want to use lists? And what's the difference between them anyway?

Tuples and lists are the exact same in everyway except two: tuples use parentheses instead of square brackets, and tuples cannot be modified while lists can. We often call lists **mutable** (meaning they can be changed) and tuples **immutable** (meaning they cannot be changed).

For an example of trying to change values in lists and tuples, look at the following code:

```
>>> listVal = [1, 1, 2, 3, 5, 8]
>>> tupleVal = (1, 1, 2, 3, 5, 8)
>>> listVal[4] = 'hello!'
>>> listVal
[1, 1, 2, 3, 'hello!', 8]
>>> tupleVal[4] = 'hello!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tupleVal
(1, 1, 2, 3, 5, 8)
>>> tupleVal[4]
5
```

Notice that when we try to change the item at index `2` in the tuple, Python gives us an error message saying that tuple objects do not support “item assignment”.

There is a silly benefit and an important benefit to tuple’s immutability. The silly benefit is that code that uses tuples is slightly faster than code that uses lists. (Python is able to make some optimizations knowing that the values in a tuple will never change.) But having your code run a few nanoseconds faster is not important.

The important benefit to using tuples is similar to the benefit of using constant variables: it’s a sign that the value in the tuple will never change, so anyone reading the code later will be able to say, “I can expect that this tuple will always be the same. Otherwise they would have used a list.” This also lets the future programmer reading your code say, “If I see a list value, I know that it could be modified at some point in this program. Otherwise, the programmer who wrote this code would have used a tuple.”

You can still assign a new tuple value to a variable:

```
>>> tupleVal = (1, 2, 3)
>>> tupleVal = (1, 2, 3, 4)
```

The reason this code works is because the code isn’t changing the `(1, 2, 3)` tuple on the second line. It is assigning an entirely new tuple `(1, 2, 3, 4)` to the tupleVal, and overwriting the old tuple value. You cannot however, use the square brackets to modify an item in the tuple.

Strings are also an immutable data type. You can use the square brackets to read a single letter in a string, but you cannot change a single letter in a string:

```
>>> strVal = 'Hello'
>>> strVal[1]
'e'
>>> strVal[1] = 'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

One Item Tuples Need a Trailing Comma

Also, one minor details about tuples: if you ever need to write code about a tuple that has one value in it, then it needs to have a trailing comma in it, such as this:

```
oneValueTuple = (42, )
```

If you forget this comma (and it is very easy to forget), then Python won't be able to tell the difference between this and a set of parentheses that just change the order of operations. For example, look at the following two lines of code:

```
variableA = (5 * 6)
variableB = (5 * 6, )
```

The value that is stored in variableA is just the integer `30`. However, the expression for variableB's assignment statement is the single-item tuple value `(30, )`. Blank tuple values do not need a comma in them, they can just be a set of parenetheses by themselves: ().

The global statement, and Why Global Variables are Evil

```
48. def main():
49.     global FPSCLOCK, WINDOWSURF
50.     pygame.init()
51.     FPSCLOCK = pygame.time.Clock()
52.     WINDOWSURF = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
53.
54.     mousex = 0 # used to store x coordinate of mouse event
55.     mousey = 0 # used to store y coordinate of mouse event
56.     pygame.display.set_caption('Memory Game')
57.
```

This is the start of the `main()` function, which is where (oddly enough) the main part of the game code is. The main function makes calls to the other functions in our program as it needs them. Think of the `main()` function as the conductor of the orchestra, and the other functions as the muscians.

Note that many of the functions called in the `main()` function will be explained later in this chapter.

Line 49 is a `global` statement. The `global` statement is the `global` keyword followed by a comma-delimited list of variable names. These variable names are then marked as global variables.. Inside the `main()` function, those names are not for local variables that might just happen to have the same name as global variables. They *are* global variables. Any values assigned to them in the `main()` function will persist outside the `main()` function. We are marking the `FPSCLOCK` and `WINDOWSURF` variables as globals because they are used in several other functions in the program.

Data Structures and 2D Lists

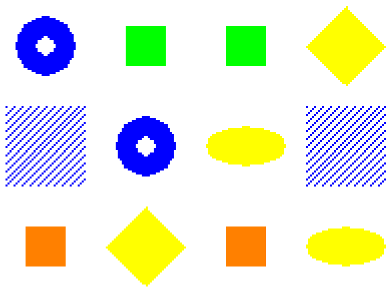
```
58.     mainBoard = getRandomizedBoard()
```

```
59.     revealedBoxes = generateRevealedBoxesData(False)
60.
```

The `getRandomizedBoard()` function returns a data structure that represents the state of the board. The `generateRevealedBoxesData()` function returns a data structure that represents which boxes are covered, respectively. The return values of these functions are two dimensional (2D) lists, or lists of lists. (A list of lists of lists of values would be a 3D list.) Another word for two or more dimensionals lists is a **multidimensional** list.

If we have a list value stored in a variable named `spam`, we could access a value in that list with the square brackets, such as `spam[2]` to retrieve the third value in the list. If the value at `spam[2]` is itself a list, then we could use another set of square brackets to retrieve a value *in that list*. This would look like, say, `spam[2][4]`, which would retrieve the fifth value in the list that is the third value in `spam`. Using the this notation of lists of lists makes it easy to map a 2D board to a 2D list value. Since the `mainBoard` variable will store icons in it, if we wanted to get the icon on the board at the position (4, 5) then we could just use the expression `mainBoard[4][5]`. Since the icons themselves are stored as two-item tuples with the shape and color, the complete data structure is a list of list of two-item tuples. Whew!

Here's an small example. Say the board looked like this:



The corresponding data structure would be:

```
[[(DONUT, BLUE), (LINES, BLUE), (SQUARE, ORANGE)], [(SQUARE, GREEN), (DONUT,
BLUE), (DIAMOND, YELLOW)], [(SQUARE, GREEN), (OVAL, YELLOW), (SQUARE, ORANGE)],
[(DIAMOND, YELLOW), (LINES, BLUE), (OVAL, YELLOW)]]
```

(If your book is in black and white, you can see a color version of the above picture at http://invpy.com/memoryboard.) You'll notice that `mainBoard[x][y]` will correspond to the icon at the (x, y) coordinate on the board.

Meanwhile, the "revealed boxes" data structure is a similar 2D list, except instead of a two-item tuple like the board data structure, it has a Boolean value: `True` if the box at that x, y coordinate is revealed, and `False` if it is covered up. Passing `False` to the `generateRevealedBoxesData()` function sets all of the Boolean values to `False`. (This function is explained in detail later.)

These two data structures are used to keep track of the state of the board.

The "Start Game" Animation

```
61.     firstSelection = None # stores the (x, y) of the first box clicked.
62.
63.     WINDOWSURF.fill(BGCOLOR)
64.     startGameAnimation(mainBoard)
65.
```

Line 61 sets up a variable called `firstSelection` with the value `None`. When the player clicks on an icon on the board, the program needs to track if this was the first icon of the pair that was clicked on or the second icon. If `firstSelection` is `None`, the click was on the first icon and we store the XYcoordinates in the `firstSelection` variable as a tuple of two integers (one for the X value, the other for Y). Line 63 fills the entire surface with the background color. This will also paint over anything that used to be on the surface, which gives us a clean slate to start drawing graphics on.

If you've played the Memory Puzzle game, you'll notice that at the beginning of the game, all of the boxes are quickly covered and uncovered randomly to give the player a sneak peek at which icons are under which boxes. This all happens in the `startGameAnimation()` function, which is explained later in this chapter.

It's important to give the player this sneak peek (but not long enough of a peek to let the player easily memorize the icon locations), because otherwise they would have no clue where any icons are. Blindly clicking on the icons isn't as much fun as having a little hint.

The Game Loop

```
66.     while True: # main game loop
67.         mouseClicked = False
68.
69.         WINDOWSURF.fill(BGCOLOR) # drawing the window
70.         drawBoard(mainBoard, revealedBoxes)
71.
```

The game loop is an infinite loop that starts on line 66 that keeps iterating for as long as the game is in progress. Remember that the game loop handles events, updates the game state, and draws the game state to the screen.

The game state for the Memory Puzzle program is stored in the following variables:

- `mainBoard`
- `revealedBoxes`
- `firstSelection`
- `mouseClicked`
- `mousex`
- `mousey`

On each iteration of the game loop in the Memory Puzzle program, the `mouseClicked` variable stores a Boolean value that is `True` if the player has clicked the mouse during this iteration through the game loop. (This is part of keeping track of the game state.)

On line 69, the surface is painted over with the background color to erase anything that was previously drawn on it. The program then calls `drawBoard()` to draw the current state of the board based on the board and "revealed boxes" data structures that we pass it. (These lines of code are part of drawing and updating the screen.)

Remember that our drawing functions only draw on the in-memory display Surface object. This surface will not actually appear on the screen until we call `pygame.display.update()`, which is done at the end of the game loop.

The Event Handling Loop

```
 72.         for event in pygame.event.get(): # event handling loop
 73.             if event.type == QUIT or (event.type == KEYUP and event.key == 
K_ESCAPE):
 74.                 pygame.quit()
 75.                 sys.exit()
 76.             elif event.type == MOUSEMOTION:
 77.                 mousex, mousey = event.pos
 78.             elif event.type == MOUSEBUTTONUP:
 79.                 mousex, mousey = event.pos
 80.                 mouseClicked = True
 81.
```

The for loop on line 72 executes code for every event that has happened since the last iteration of the game loop. This loop is called the event handling loop (which is different from the game loop, although the event handling loop is inside of the game loop). The list of `pygame.Event` objects returned by the `pygame.event.get()` call are stored one at a time in the `event` variable on each iteration through the event handling loop.

If the event object was a either a `QUIT` event or a `KEYUP` event of the Esc key, then the program should quit. Otherwise, in the event of a `MOUSEMOTION` (that is, the mouse cursor has moved) or `MOUSEBUTTONUP` (that is, a mouse button was pressed earlier and now the button was let up) event, the position of the mouse cursor should be stored in the `mousex` and `mousey` variables. If this was a `MOUSEBUTTONUP` event, `mouseClicked` should also be set to `True`.

Once we have handled all of the events, the values stored in `mousex`, `mousey`, and `mouseClicked` will tell us any input that player has given us. Now we should update the game state and draw the results to the screen.

Checking Which Box The Mouse Cursor is Over

```
82.         boxx, boxy = getBoxAtPixel(mousex, mousey)
83.         if boxx != None and boxy != None:
84.             # The mouse is currently over a box.
85.             if not revealedBoxes[boxx][boxy]:
86.                drawHighlightBox(boxx, boxy)
```

The `getBoxAtPixel()` function will return the board coordinates of the box that the mouse coordinates are over. How `getBoxAtPixel()` does this is explained later. All we have to know for now is that if the `mousex` and `mousey` coordinates were over a box, a tuple of the XY board coordinates are returned by the function and stored in `boxx` and `boxy`. If the mouse cursor was not over any box (for example, if it was off to the side of the board or in a gap in between boxes) then the tuple `(None, None)` is returned by the function and `boxx` and `boxy` both have `None` stored in them.

We are only interested in the case where `boxx` and `boxy` do not have `None` in them, so the next several lines of code are in the block following the `if` statement on line 83 that checks for this case. If execution has come inside this block, we know the user has the mouse cursor over a box (and maybe has also clicked the mouse, depending on the value stored in `mouseClicked`).

The `if` statement on line 85 checks if the box is covered up or not by reading the value stored in `revealedBoxes[boxx][boxy]`. If it is `False`, then we know the box is covered. Whenever the mouse is over a covered up box, we want to draw a blue highlight around the box to inform the player that they can click on it. (This highlighting is not done for boxes that are already uncovered.) The highlight drawing is handled by our `drawHighlightBox()` function, which is explained later.

```
87.             if not revealedBoxes[boxx][boxy] and mouseClicked:
 88.                 revealBoxesAnimation(mainBoard, [(boxx, boxy)])
 89.                 revealedBoxes[boxx][boxy] = True # set the box as
"revealed"
```

On line 87, we check if the mouse cursor is not only over a covered up box but if the mouse has also been clicked. In that case, we want to play the "reveal" animation for that box by calling our `revealBoxesAnimation()` function (which is, as with all the other functions `main()` calls, explained later in this chapter). You should note that calling this function only draws the animation of the box being uncovered. It isn't until line 89 when we set `revealedBoxes[boxx][boxy] = True` that the data structures that track the game state is updated.

If you comment out line 89 and then run the program, you'll notice that after clicking on a box the reveal animation is played, but then the box immediately appears covered up again. This is because `revealedBoxes[boxx][boxy]` is still set to `False`, so on the next iteration of the game loop, the board is drawn with this box covered up. Not having line 89 would cause quite an odd bug in our program.

Handle the First Clicked Box

```
 90.                 if firstSelection == None: # the current box was the first
box clicked
 91.                     firstSelection = (boxx, boxy)
 92.                 else: # the current box was the second box clicked
 93.                     # Check if there is a match between the two icons.
 94.                     icon1shape, icon1color = getShapeAndColor(mainBoard,
firstSelection[0], firstSelection[1])
 95.                     icon2shape, icon2color = getShapeAndColor(mainBoard,
boxx, boxy)
 96.
```

Before we entered the game loop, the `firstSelection` variable was set to `None`. Our program will interepret this to mean that no boxes have been clicked, so if line 90's condition is `True`, that means this is the first of the two possibly matching boxes that was clicked. We want to play the reveal animation for the box and then keep that box uncovered. We also set the `firstSelection` variable to a tuple of the box coordinates for the box that was clicked.

If this is the second box the player has clicked on, we want to play the reveal animation for that box but then check if the two icons under the boxes are matching. The `getShapeAndColor()` function (explained later) will retrieve the shape and color values of the icons. These values will be one of the values in the `ALLCOLORS` and `ALLSHAPES` tuples.

Handling a Mismatched Pair of Icons

```
 97.                 if icon1shape != icon2shape or icon1color !=
icon2color:
 98.                     # Icons don't match. Re-cover up both selections.
 99.                     pygame.time.wait(1000) # 1000 milliseconds = 1 sec
100.                     coverBoxesAnimation(mainBoard,
[(firstSelection[0], firstSelection[1]), (boxx, boxy)])
101.                     revealedBoxes[firstSelection[0]][firstSelection
[1]] = False
102.                     revealedBoxes[boxx][boxy] = False
```

The `if` statement on line 97 checks if either the shapes or colors of the two icons don't match. If this is the case, then we want to pause the game for 1000 milliseconds (which is the same as 1 second) by calling `pygame.time.wait(1000)` (so that the player has a chance to see that the two icons don't match). Then play the "cover up" animations for both boxes. We also want to update the game state to mark these boxes as not revealed (i.e. covered up).

Handling If the Player Won

```
103.                 elif hasWon(revealedBoxes): # check if all pairs found
104.                     gameWonAnimation(mainBoard)
105.                     pygame.time.wait(2000)
106.
107.                     # Reset the board
108.                     mainBoard = getRandomizedBoard()
109.                     revealedBoxes = generateRevealedBoxesData(False)
110.
111.                     # Show the fully unrevealed board for a second.
112.                     drawBoard(mainBoard, revealedBoxes)
113.                     pygame.display.update()
114.                     pygame.time.wait(1000)
115.
116.                     # Replay the start game animation.
117.                     startGameAnimation(mainBoard)
118.                 firstSelection = None # reset firstSelection variable
119.
```

Otherwise, if line 97's condition was `False`, then the two icons are a match. The program doesn't really have to do anything else at that point: it can just leave both boxes in the revealed state. However, the program should check if this was the last pair of icons on the board to be matched. This is done inside our `hasWon()` function, which returns `True` if the board is in a winning state (that is, all of the boxes are revealed). If this is the case, we want to play the "game won" animation by calling `gameWonAnimation()`, then pause slightly to let the player revel in their victory, and then reset the data structures in `mainBoard` and `revealedBoxes` to start a new game.

Line 117 plays the “start game” animation again. After that, the program execution will just loop through the game loop as usual, and the player can continue playing until they quit the program.

No matter if the two boxes were matching or not, after the second box was clicked line 118 will set the `firstSelection` variable back to `None` so that the next box the player clicks on will be interpreted as the first clicked box of a pair of possibly matching icons.

Draw the Game State to the Screen

```
        # Redraw the screen and wait a clock tick.
        pygame.display.update()
        FPSCLOCK.tick(FPS)


```

At this point, the game state has been updated depending on the player’s input, and the latest game state has been drawn to the `WINDOWSURF` display Surface object. We’ve reached the end of the game loop, so we call `pygame.display.update()` to draw the `WINDOWSURF` Surface object to the screen.

Line 9 set the FPS constant to the integer value `30`, meaning we want the game to run (at most) 30 frames per second. If we want the program to run faster, we can increase this number. If we want the program to run slower, we can decrease this number. (It can even be set to a float value like `0.5`, which will run the program at half a frame per second, that is, one frame per two seconds.)

In order to run at 30 frames per second, each frame must be drawn in 1/30th of a second. This means that `pygame.display.update()` and all the code in the game loop must execute in under 33.3 milliseconds. Any modern computer can do this easily with plenty of time left over. To prevent the program from running too fast, we call the `tick()` method of the `pygame.Clock` object in `FPSCLOCK` to have to it pause the program for the rest of the 33.3 milliseconds.

Since this is done at the very end of the game loop, it ensures that each iteration of the game loop takes (at least) 33.3 milliseconds. If for some reason the `pygame.display.update()` call and the code in the game loop has taken longer than 33.3 milliseconds, then the `tick()` method will not wait at all and immediately return.

I've kept saying that the other functions would be explained later in the chapter. Now that we've gone over the `main()` function and you have an idea for how the general program works, let's go into the details of all the other functions that are called from `main()`.

Creating the "Revealed Boxes" Data Structure

```
def generateRevealedBoxesData(val):
    revealedBoxes = []
    for i in range(BOARDWIDTH):
        revealedBoxes.append([val] * BOARDHEIGHT)
    return revealedBoxes


```

The `generateRevealedBoxesData()` function needs to create a list of lists of Boolean values. The Boolean value will just be the one that is passed to the function as the `val` parameter. We start the data structure as an empty list in the `revealedBoxes` variable.

In order to make the data structure have the `revealedBoxes[x][y]` structure, we need to make sure that the inner lists represent the vertical columns of the board and not the horizontal rows. Otherwise, the data structure will have a `revealedBoxes[y][x]` structure.

The `for` loop will create the columns and then append them to `revealedBoxes`. The columns are created using list replication, so that the column list has as many `val` values as the `BOARDHEIGHT` dictates.

Creating the Board Data Structure – Step 1 – Get All Possible Icons

```
def getRandomizedBoard():
    # Get a list of every possible shape in every possible color.
    icons = []
    for color in ALLCOLORS:
        for shape in ALLSHAPES:
            icons.append( (shape, color) )

```

The board data structure is just a list of lists of tuples, where each tuple has a two values: one for the icon's shape and one for the icon's color. But creating this data structure is a little complicated. We need to be sure to have exactly as many icons for the number of boxes on the board and also be sure there are two and only two icons of each type.

The first step to do this is to create a list with every possible combination of shape and color. Recall that we have a list of each color and shape in `ALLCOLORS` and `ALLSHAPES`, so nested for loops will go through every possible shape for every possible color. These are all added to the list in the icons variable.

Step 2 – Shuffling and Truncating the List of All Icons

```
139.     random.shuffle(icons) # randomize the order of the icons list
140.     numIconsUsed = int(BOARDWIDTH * BOARDHEIGHT / 2) # calculate how many
icons are needed
141.     icons = icons[:numIconsUsed] * 2 # make two of each
142.     random.shuffle(icons)
143.
```

But remember, there may be more possible combinations than spaces on the board. We need to calculate the number of spaces on the board by multiplying BOARDWIDTH by BOARDHEIGHT. Then we divide that number by 2 because we will have pairs of icons. On a board with 70 spaces, we'd only need 35 different icons, since there will be two of each icon. This number will be stored in numIconsUsed.

Line 141 uses list slicing to grab the first numIconsUsed number of icons in the list. (If you've forgotten how list slicing works, check out http://invpy.com/slicing.) This list has been shuffled on line 139, so it won't always be the same icons each game. Then this list is replicated by 2 using the * operator so that there are two of each of the icons. This new doubled up list will overwrite the old list in the icons variable. Since the first half of this new list is identical to the last half, we call the shuffle() method again to randomly mix up the order of the icons.

Step 3 – Placing the Icons on the Board

```
144.     # Create the board data structure, with randomly placed icons.
145.     board = []
146.     for x in range(BOARDWIDTH):
147.         columns = []
148.         for y in range(BOARDHEIGHT):
149.             columns.append(icons[0])
150.             del icons[0] # remove the icons as we assign them
151.         board.append(columns)
152.     return board
153.
154.
```

Now we need to create a list of lists data structure for the board. We can do this with nested for loops just like the generateRevealedBoxesData() function did. For each column on the board, we will create a list of randomly selected icons. As we add icons to the column, we will then delete them. This way, as the `icons` list gets shorter and shorter, there will always be a new icon stored in `icons[0]`.

To picture this better, type the following code into the interactive shell. Notice how the `del` statement changes the `myList` list.

```
>>> myList = ['cat', 'dog', 'mouse', 'lizard']
>>> del myList[0]
>>> myList
['dog', 'mouse', 'lizard']
>>> del myList[0]
>>> myList
['mouse', 'lizard']
>>> del myList[0]
>>> myList
['lizard']
>>> del myList[0]
>>> myList
[]
>>>
```

Splitting a List into a List of Lists

```
155. def splitIntoGroupsOf(groupSize, theList):
156.     # splits a list into a list of lists, where the inner lists have at
157.     # most groupSize number of items.
158.     result = []
159.     for i in range(0, len(theList), groupSize):
160.         result.append(theList[i:i + groupSize])
161.     return result
162.
163.
```

The `splitIntoGroupsOf()` function (which will be called by the `startGameAnimation()` function) splits a list into a list of lists, where the inner lists have groupSize number of items in them. (The last list could have less if there are only leftovers.)

The call to `range()` on line 159 uses the three-parameter form of `range()`. (If you are unfamiliar with this form, take a look at http://invpy.com/range.) Let's use an example. If the length of the list is `20` and the `groupSize` parameter is `8`, then `range(0, len(theList), groupSize)` evaluates to `range(0, 20, 8)`. This will give the `i` variable the values `0`, `8`, and `16`.

The list slicing on line 160 with `theList[i:i + groupSize]` creates the lists that are added to the `result` list. On each iteration where `i` is `0`, `8`, and `16` (and `groupSize` is `8`), this list slicing expression would be `theList[0:8]`, then `theList[8:16]`, and then `theList[16:24]`.

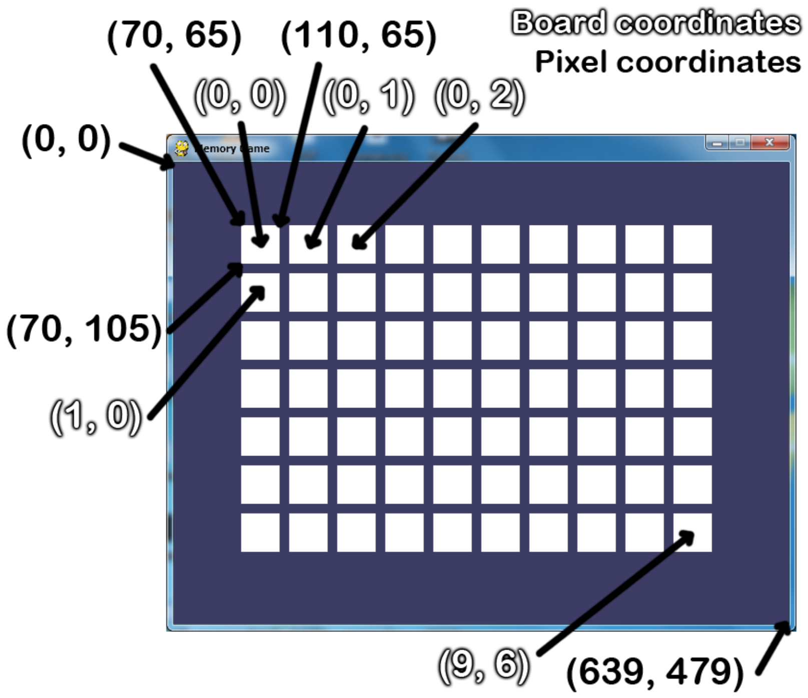
Note that even though the largest index of `theList` would be `19` in our example, `theList[16:24]` won't create an `IndexError` error. It will just create a list slice with the remaining items in the list. Remember that list slicing doesn't destroy the original list stored in `theList`. It just copies a section of it to make a new list value. This is new list value is the list that is appended to the list in result. So when we return result at the end of this function, we are returning a list of lists.

Different Coordinate Systems

```
def leftTopCoordsOfBox(boxx, boxy):
    # Convert board coordinates to pixel coordinates
    left = boxx * (BOXSIZE + GAPSIZE) + XMARGIN
    top = boxy * (BOXSIZE + GAPSIZE) + YMARGIN
    return (left, top)


```

By now you are familiar with Cartesian Coordinate systems. (If you'd like a refresher on this topic, read http://invpy.com/coordinates.) In most of our games we will be using multiple Cartesian Coordinate systems. One system that is used in the Memory Puzzle game is for pixel coordinates. But we will also be using another coordinate system for the boxes. This is because it will be easier to use (3, 2) to refer to the 4th box from the left and 3rd from the top (remember that the numbers start with 0, not 1) instead of using (220, 165). However, the box at box coordinates (3, 2) would have it's top left corner located at pixel (220, 165). We need a way to translate between these two coordinate systems.

Here's a picture of the game and the two different coordinate systems. Remember that the window is 640 pixels wide and 480 pixels tall, so (639, 479) is the bottom right corner (because the top left corner is (0, 0), and not (1, 1)).



The leftTopCoordsOfBox() function will take box coordinates and return pixel coordinates. Because a box takes up multiple pixels on the screen, we will always return the single pixel at the top left corner of the box. This value will be returned as a two-integer tuple. The leftTopCoordsOfBox() function will often be used when we need pixel coordinates for drawing these boxes.

Converting from Pixel Coordinates to Box Coordinates

```
def getBoxAtPixel(x, y):
    for boxx in range(BOARDWIDTH):
        for boxy in range(BOARDHEIGHT):
            left, top = leftTopCoordsOfBox(boxx, boxy)
            boxRect = pygame.Rect(left, top, BOXSIZE, BOXSIZE)
            if boxRect.collidepoint(x, y):
                return (boxx, boxy)
    return (None, None)


```

We will also need a function to convert from pixel coordinates (which the mouse clicks and mouse movement events use) to box coordinates (so we can find out over which box the mouse event happened.) Rect objects have a collidepoint() method that you can pass X and Y coordinates too and it will return True if the coordinates are inside the Rect object’s area.

In order to find which box the mouse coordinates are over, we will go through each box's coordinates and call the `collidepoint()` method on a Rect object with those coordinates. When `collidepoint()` returns `True`, we know we have found the box that was clicked on or moved over and will return the box coordinates. If none of them return `True`, then the `getBoxAtPixel()` function will return the value `(None, None)`. This tuple is returned instead of simply `None` because the caller of `getBoxAtPixel()` is expecting a tuple of two values to be returned.

Drawing the Icon, and Syntactic Sugar

```
def drawIcon(shape, color, boxx, boxy):
    quarter = int(BOXSIZE * 0.25) # syntactic sugar
    half =    int(BOXSIZE * 0.5)  # syntactic sugar

    left, top = leftTopCoordsOfBox(boxx, boxy) # get pixel coords from board coords
```

The `drawIcon()` function will draw an icon (with the specified `shape` and `color`) at the space whose coordinates are given in the `boxx` and `boxy` parameters. Each possible shape has a different set of Pygame drawing function calls for it, so we must have a large set of `if` and `elif` statements to differentiate between them.

The X and Y coordinates of the left and top edge of the box can be obtained by calling the `leftTopCoordsOfBox()` function. The width and height of the box are both set in the `BOXSIZE` constant. However, many of the shape drawing function calls use the midpoint and quarter-point of the box as well. We can calculate this and store it in the variables quarter and half. We could just as easily have the code `int(BOXSIZE * 0.25)` instead of the variable `quarter`, but this way the code becomes easier to read since it is more obvious what `quarter` means rather than `int(BOXSIZE * 0.25)`.

Such variables are an example of syntactic sugar. **Syntactic sugar** is when we add code that could have been written in another way (probably with less actual code and variables), but does make the source code easier to read. Constant variables are one form of syntactic sugar. Pre-calculating a value and storing it in a variable is another type of syntactic sugar. (For example, in the `getRandomizedBoard()` function, we could have easily made the code on lines 140 and line 141 into a single line of code. But it's easier to read as two separate lines.) We don't need to have the extra `quarter` and `half` variables, but having them makes the code easier to read. Code that is easy to read is easy to debug and upgrade in the future.

```
    # Draw the shapes
    if shape == DONUT:
        pygame.draw.circle(WINDOWSURF, color, (left + half, top + half), half - 5)
        pygame.draw.circle(WINDOWSURF, BGCOLOR, (left + half, top + half), quarter - 5)
    elif shape == SQUARE:
        pygame.draw.rect(WINDOWSURF, color, (left + quarter, top + quarter, BOXSIZE - half, BOXSIZE - half))
    elif shape == DIAMOND:
        pygame.draw.polygon(WINDOWSURF, color, ((left + half, top), (left + BOXSIZE - 1, top + half), (left + half, top + BOXSIZE - 1), (left, top + half)))
    elif shape == LINES:
        for i in range(0, BOXSIZE, 4):
            pygame.draw.line(WINDOWSURF, color, (left, top + i), (left + i, top))
            pygame.draw.line(WINDOWSURF, color, (left + i, top + BOXSIZE - 1), (left + BOXSIZE - 1, top + i))
    elif shape == OVAL:
        pygame.draw.ellipse(WINDOWSURF, color, (left, top + quarter, BOXSIZE, half))


```

Each of the donut, square, diamond, lines, and oval functions require different drawing primitive function calls to make.

Syntactic Sugar with Getting a Board Space's Icon's Shape and Color

```
def getShapeAndColor(board, boxx, boxy):
    # shape value for x, y spot is stored in board[x][y][0]
    # color value for x, y spot is stored in board[x][y][1]
    return board[boxx][boxy][0], board[boxx][boxy][1]


```

The `getShapeAndColor()` function only has one line. You might wonder why we would want a function instead of just typing in that one line of code whenever we need it. This is done for the same reason we use constant variables: it improves the readability of the code.

It's easy to figure out what a code like `shape, color = getShapeAndColor()` does. But if you looked a code like `shape, color = board[boxx][boxy][0], board[boxx][boxy][1]`, it would be a bit more difficult to figure out what that code exactly does.

Drawing the Box Cover

```
208. def drawBoxCovers(board, boxes, coverage):
209.     # Draws boxes being covered/revealed. "boxes" is a list
210.     # of two-item lists, which have the x & y spot of the box.
211.     for box in boxes:
212.         left, top = leftTopCoordsOfBox(box[0], box[1])
213.         pygame.draw.rect(WINDOWSURF, BGCOLOR, (left, top, BOXSIZE,
BOXSIZE))

214.         shape, color = getShapeAndColor(board, box[0], box[1])
215.         drawIcon(shape, color, box[0], box[1])
216.         if coverage > 0: # only draw the cover if there is an coverage
217.             pygame.draw.rect(WINDOWSURF, BOXCOLOR, (left, top, coverage,
BOXSIZE))
218.     pygame.display.update()
219.     FPSCLOCK.tick(FPS)
220.
221.
```

The `drawBoxCovers()` function has three parameters: the board data structure, a list of (X, Y) tuples for each box that should have the cover drawn, and then the amount of coverage to draw for the boxes.

Since we want to use the same drawing code for each box in the boxes parameter, we will use a `for` loop on line 211 so we execute the same code on each box in the list. Inside this `for` loop, the code should do three things: draw the background color (to paint over anything that was there before), draw the icon, then draw however much of the white box over the icon that is needed. The `leftTopCoordsOfBox()` function will return the pixel coordinates of the top left corner of the box. The `if` statement on line 216 makes sure that if the number in coverage happens to be less than `0`, we won't call the `pygame.draw.rect()` function.

When the `coverage` parameter is `0`, there is no coverage at all. When the `coverage` is set to `20`, there is a 20 pixel wide white box covering the icon. The largest size we'll want the `coverage` set to is the number in `BOXSIZE`.

`drawBoxCovers()` is going to be called from a separate loop than the game loop. Because of this, it needs to have its own calls to `pygame.display.update()` and `FPSCLOCK.tick(FPS)` to display the animation. (This does mean that while inside this loop, there is no code being run to handle any events being generated. That’s fine, since the cover and reveal animations only take a second or so to play.)

Handling the Revealing and Covering Animation

```python
def revealBoxesAnimation(board, boxesToReveal):
    # Do the "box reveal" animation.
    for coverage in range(BOXSIZE, (-REVEALSPEED) - 1, - REVEALSPEED):
        drawBoxCovers(board, boxesToReveal, coverage)


def coverBoxesAnimation(board, boxesToCover):
    # Do the "box cover" animation.
    for coverage in range(0, BOXSIZE + REVEALSPEED, REVEALSPEED):
        drawBoxCovers(board, boxesToCover, coverage)


```

Remember that an animation is simply just displaying different images for brief moments of time, and together they make it seem like things are moving on the screen. The `revealBoxesAnimation()` and `coverBoxesAnimation()` only need to draw an icon with a varying amount of coverage by the white box. We can write a single function called `drawBoxCovers()` which can do this, and then have our animation functions just make several calls to `drawBoxCovers()`. As we saw in the last section, `drawBoxCovers()` makes calls to `pygame.display.update()` and `FPSCLOCK.tick(FPS)` itself.

To do this, we’ll set up a for loop to make decreasing (in the case of `revealBoxesAnimation()`) or increasing (in the case of `coverBoxesAnimation()`) numbers for the third parameter. The amount that the coverage variable will decrease/increase by is the number in the `REVEALSPEED` constant. On line 12 we set this constant to `8`, meaning that on each call to `drawBoxCovers()`, the white box will decrease/increase by 8 pixels. If we increase this number, then more pixels will be drawn on each call, meaning that the white box will decrease/increase in size faster. If we set it to `1`, then the white box will only appear to decrease or increase by 1 pixel at a time, making the entire reveal or cover animation take longer.

Think of it like climbing stairs. If on each step you take, you climbed one stair, then it would take a normal amount of time to climb the entire staircase. But if you climbed two stairs at a time on each step (and the steps took just as long as before), you could climb the entire staircase twice as fast. If you could climb the staircase 8 stairs at a time, then you would climb the entire staircase 8 times as fast.

This code handles drawing to the display Surface, calling pygame.display.update() and then adding a pause with a call to FPSCLOCK's tick() method. But this code doesn't have any event handling, just code for doing animations. I informally call code like this an **animation loop**.

Drawing the Entire Board

```
234. def drawBoard(board, revealed):
235.     # Draws all of the boxes in their covered or revealed state.
236.     for boxx in range(BOARDWIDTH):
237.         for boxy in range(BOARDHEIGHT):
238.             left, top = leftTopCoordsOfBox(boxx, boxy)
239.             if not revealed[boxx][boxy]:
240.                 # Draw a covered box.
241.                 pygame.draw.rect(WINDOWSURF, BOXCOLOR, (left, top, 
BOXSIZE, BOXSIZE))

242.             else:
243.                 # Draw the (revealed) icon.
244.                 shape, color = getShapeAndColor(board, boxx, boxy)
245.                 drawIcon(shape, color, boxx, boxy)
246.
247.
```

The `drawBoard()` function makes a call to `drawIcon()` for each of the boxes on the board. The nested `for` loops on lines 236 and 237 will loop through every possible X and Y coordinate for the boxes, and will either draw the icon at that location or draw a white square instead (to represent a covered up box.)

Drawing the Highlight

```
248. def drawHighlightBox(boxx, boxy):
249.     left, top = leftTopCoordsOfBox(boxx, boxy)
250.     pygame.draw.rect(WINDOWSURF, HIGHLIGHTCOLOR, (left - 5, top - 5, 
BOXSIZE + 10, BOXSIZE + 10), 4)

251.
252.
```

To help the player recognize that they can click on a covered box to reveal it, we will make a blue outline appear around a box to highlight it. This outline is drawn with a call to `pygame.draw.rect()` to make a rectangle with a width of 4 pixels.

The "Start Game" Animation

```
253. def startGameAnimation(board):
```

```
254.     # Randomly reveal the boxes 8 at a time.
255.     coveredBoxes = generateRevealedBoxesData(False)
256.     boxes = []
257.     for x in range(BOARDWIDTH):
258.         for y in range(BOARDHEIGHT):
259.             boxes.append( (x, y) )
260.     random.shuffle(boxes)
261.     boxGroups = splitIntoGroupsOf(8, boxes)
262.
```

The animation that plays at the beginning of the game gives the player a quick hint as to where all the icons are located. In order to make this animation, we have to reveal and cover up groups of boxes one after another. To do this, first we'll create a list of every possible space on the board. This we have nested `for` loops that add (X, Y) tuples to a list in the `boxes` variable.

We will reveal and cover up the first 8 boxes in this list, then the next 8, then the next 8 after that, and so on. However, since the order of the (X, Y) tuples in boxes would be the same each time, then the same order of boxes would be displayed. (Try commenting out line 260 and then running to program to see this effect.)

To change up the boxes each time a game starts, we will call the `random.shuffle()` function to randomly shuffle the order of the tuples in the boxes list. Then when we reveal and cover up the first 8 boxes in this list (and each group of 8 boxes afterwards), it will be random group of 8 boxes.

To get the lists of 8 boxes, we call our `splitIntoGroupsOf()` function, passing `8` and the list in `boxes`. The list of lists that the function returns will be stored in a variable named `boxGroups`.

Revealing and Covering the Groups of Boxes

```
263.     drawBoard(board, coveredBoxes)
264.     for boxGroup in boxGroups:
265.         revealBoxesAnimation(board, boxGroup)
266.         coverBoxesAnimation(board, boxGroup)
267.
268.
```

First, we draw the board. Since every value in `coveredBoxes` is set to `False`, this call to `drawBoard()` will end up drawing only covered up white boxes. The `revealBoxesAnimation()` and `coverBoxesAnimation()` functions will draw over these boxes.

The `for` loop will go through each of the inner lists in the `boxGroups` lists. We pass these to `revealBoxesAnimation()`, which will perform the animation of the white boxes being pulled away to reveal the icon underneath. Then the call to `coverBoxesAnimation()` will animate the white boxes expanding to cover up the icons. Then the for loop goes to the next iteration to animate the next set of 8 boxes.

The "Game Won" Animation

```
269. def gameWonAnimation(board):
270.     # flash the background color when the player has won
271.     coveredBoxes = generateRevealedBoxesData(True)
272.     color1 = LIGHTBGCOLOR
273.     color2 = BGCOLOR
274.
275.     for i in range(13):
276.         color1, color2 = color2, color1 # swap colors
277.         WINDOWSURF.fill(color1)
278.         drawBoard(board, coveredBoxes)
279.         pygame.display.update()
280.         pygame.time.wait(300)
281.
282.
```

When the player has uncovered all of the boxes by matching every pair on the board, we want to congratulate them by flashing the background color. The `for` loop will draw the color in the `color1` variable for the background color and then draw the board over it. However, on each iteration of the `for` loop, the values in `color1` and `color2` will swap with each other on line 276. This way the program will alternate between drawing two different background colors.

Remember that this function needs to call `pygame.display.update()` to actually make the `WINDOWSURF` surface appear on the screen.

Telling if the Player Has Won

```
283. def hasWon(revealedBoxes):
284.     # Returns True if all the boxes have been revealed, otherwise False
285.     for i in revealedBoxes:
286.         if False in i:
287.             return False # return False if any boxes are covered.
288.     return True
289.
290.
```

The player has won the game when all of the icon pairs have been matched. Since the “revealed” data structure gets values in it set to `True` as icons have been matched, we can simply loop through every space in `revealedBoxes` looking for a `False` value. If even one `False` value is in `revealedBoxes`, then we know there are still unmatched icons on the board.

Note that because `revealedBoxes` is a list of lists, the `for` loop on line 285 will set the inner list as the values of `i`. But we can use the in operator to search for a `False` value in the entire inner list. This way we don’t need to write an additional line of code and have two nested `for` loops like this:

```
for x in revealedBoxes:
    for y in revealedBoxes[x]:
        if False == revealedBoxes[x][y]:
            return False
```

Why Bother Having a `main()` Function?

```
if __name__ == '__main__':
    main()
```

It may seem pointless to have a `main()` function, since you could just put that code in the global scope at the bottom of the program instead, and the code would run the exact same. However, there are two good reasons to put them inside of a `main()` function.

First, this lets you have local variables whereas otherwise the local variables in the `main()` function would have to become global variables. Limiting the number of global variables is a good way to keep the code simple and easier to debug. (See the “Why Global Variables are Evil” section in this chapter.)

Second, this also lets you import the program so that you can call and test individual functions. If the *memorypuzzle.py* file is in the C:\Python32 folder, then you can import it from the interactive shell. Type the following to test out the `splitIntoGroupsOf()` and `getBoxAtPixel()` functions to make sure they return the correct return values:

```
>>> import memorypuzzle
>>> memorypuzzle.splitIntoGroupsOf(3, [0,1,2,3,4,5,6,7,8,9])
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
>>> memorypuzzle.getBoxAtPixel(0, 0)
(None, None)
>>> memorypuzzle.getBoxAtPixel(150, 150)
(1, 1)
```

When a module is imported, all of the code in it is run. If we didn't have the `main()` function, and had it's code in the global scope, then the game would have automatically started as soon as we imported it, which really wouldn't let us call individual functions in it.

That's why the code is in a separate function that we have named `main()`. Then we check the built-in Python variable `__name__` to see if we should call the `main()` function or not. This variable is automatically set bt the Python interpreter to the string `'__main__'` if the program itself is being run and `'memorypuzzle'` if it is being imported. This is why the `main()` function is not run when we executed the `import memorypuzzle` statement in the interactive shell.

This is a handy technique for being able to import the program you are working on from the interactive shell and make sure individual functions are returning the correct values by testing them one call at a time.

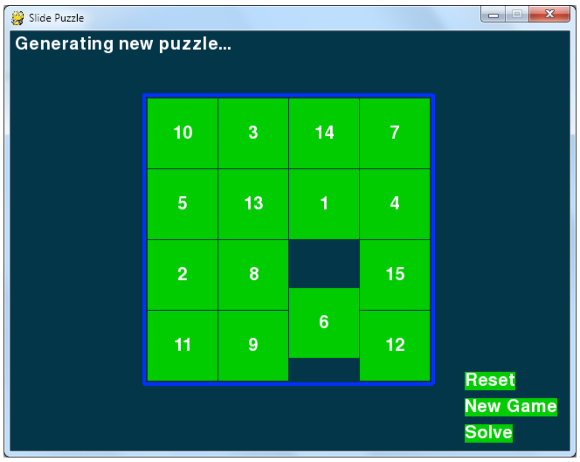
Summary, and a Hacking Suggestion

That is the entire explanation of how the Memory Puzzle program works. Read over the chapter and the source code again to understand it better. Many of the other game programs in this book make use of the same programming concepts (like nested `for` loops, syntactic sugar, and different coordinate systems in the same program) so they won't be explained again to keep this book short.

One idea to try out to understand how the code works is to intentionally break it by commenting out random lines. Doing this to many of the lines will probably cause a syntactic error that will prevent the script from running. But commenting out other lines will result in weird bugs and other cool effects. Try doing this and then figure out why a program has the bugs it does.

This is also the first step in being able to add your own secret cheats or hacks to the program. By breaking the program from what it normally does, you can learn how to change it to do something neat effect (like secretly giving you hints on how to solve the puzzle.) Feel free to experiment. You can always save a copy of the unchanged source code in a different file if you want to play the regular game again.

Chapter 4 – Slide Puzzle



How to Play Slide Puzzle

The board is a 4x4 grid with fifteen tiles (numbered 1 through 15 going left to right) and one blank space. The tiles start out in random positions, and the player must slide tiles around until the tiles are back in their original order.

Source Code to Slide Puzzle

This source code can be downloaded from http://invpy.com/slidepuzzle.py.

Second Verse, Same as the First

```
# Slide Puzzle
# By Al Sweigart al@inventwithpython.com
# http://inventwithpython.com/pygame
# Creative Commons BY-NC-SA 3.0 US

import pygame, sys, random
from pygame.locals import *

# Create the constants (go ahead and experiment with different values)
BOARDWIDTH = 4  # number of columns in the board
BOARDHEIGHT = 4 # number of rows in the board
TILESIZE = 80
WINDOWWIDTH = 640
WINDOWHEIGHT = 480
```

```
FPS = 30
BLANK = None

#                 R    G    B
BLACK =         (  0,   0,   0)
WHITE =         (255, 255, 255)
BRIGHTBLUE =    (  0,  50, 255)
DARKTURQUOISE = (  3,  54,  73)
GREEN =         (  0, 204,   0)

BGCOLOR = DARKTURQUOISE
TILECOLOR = GREEN
TEXTCOLOR = WHITE
BORDERCOLOR = BRIGHTBLUE
BASICFONTSIZE = 20

BUTTONCOLOR = WHITE
BUTTONTEXTCOLOR = BLACK
MESSAGECOLOR = WHITE

XMARGIN = int((WINDOWWIDTH - (TILESIZE * BOARDWIDTH + (BOARDWIDTH - 1))) / 2)
YMARGIN = int((WINDOWHEIGHT - (TILESIZE * BOARDHEIGHT + (BOARDHEIGHT - 1))) / 2)

UP = 'up'
DOWN = 'down'
LEFT = 'left'
RIGHT = 'right'

```

This code at the top of the program just handles all the basic importing of modules and creating constants. This is just like the beginning of the Memory Puzzle game from the last chapter.

Setting Up the Buttons

```
def main():
    global FPSCLOCK, WINDOWSURF, BASICFONT, RESET_SURF, RESET_RECT, NEW_SURF, NEW_RECT, SOLVE_SURF, SOLVE_RECT

    pygame.init()
    FPSCLOCK = pygame.time.Clock()
    WINDOWSURF = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
    pygame.display.set_caption('Slide Puzzle')
    BASICFONT = pygame.font.Font('freesansbold.ttf', BASICFONTSIZE)

    # Store the option buttons and their rectangles in OPTIONS.
    RESET_SURF, RESET_RECT = makeText('Reset',    TEXTCOLOR, TILECOLOR, WINDOWWIDTH - 120, WINDOWHEIGHT - 90)
    NEW_SURF,   NEW_RECT   = makeText('New Game', TEXTCOLOR, TILECOLOR, WINDOWWIDTH - 120, WINDOWHEIGHT - 60)
    SOLVE_SURF, SOLVE_RECT = makeText('Solve',    TEXTCOLOR, TILECOLOR, WINDOWWIDTH - 120, WINDOWHEIGHT - 30)

    mainBoard, solutionSeq = generateNewPuzzle(80)
    SOLVEDBOARD = getStartingBoard() # a solved board is the same as the board in a start state.
```

Just like in the last chapter, the functions that the `main()` function calls will be explained later in the chapter. For now, you just need to know what values they return rather than how they work exactly.

The first part of the `main()` function will handle creating the window, Clock object, and Font object. The `makeText()` function is defined later in the program, but for now you just need to know that it returns a `pygame.Surface` object and `pygame.Rect` object which can be used to make clickable buttons. The Slide Puzzle game will have three buttons: a "Reset" button that will undo any moves the player has made, a "New" button that will create a new slide puzzle, and a "Solve" button that will solve the puzzle for the player.

We will need to have two board data structures for this program. One board will represent the current game state. The other board will have its tiles in the "solved" state, meaning that all the tiles are lined up in order. When the current game state's board is exactly the same as the solved board, then we know the player has won. (We won't ever change this second one. It'll just be there to compare the current game state board to.)

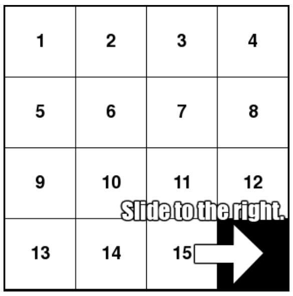
The `generateNewPuzzle()` will create a board data structure that started off in the ordered, solved state and then had 80 random slide moves performed on it (because we passed the integer `80` to it. If we want the board to be even more jumbled, then we can pass a larger integer to it). This will make the board into a randomly jumbled state that the player will have to solve (which will be stored in a variable named `mainBoard`). The `generateNewBoard()` also returns a list of all the random moves that were performed on it (which will be stored in a variable named `solutionSeq`).

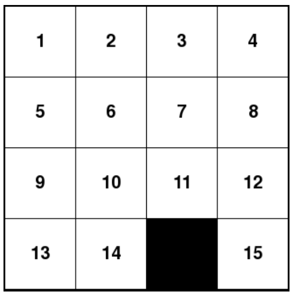
Being Smart By Using Stupid Code

```
    allMoves = [] # list of moves made from the solved configuration

```

Solving a slide puzzle can be really tricky. We could program the computer to do it, but that would require us to figure out an algorithm that can solve the slide puzzle. That would be very difficult and involve a lot of cleverness and effort to put into this program. Fortunately, there's an easier way. We could just have the computer memorize all the random slides it made when it created the board data structure, and then the board can be solved just by performing the opposite slide. Since the board originally started in the solved state, undoing all the slides would return it to the solved state.

For example, below we perform a "right" slide on the board on the left side of the page, which leaves the board in the state that is on the right side of the page:





After the"right" slide, if we do the opposite slide (a "left" slide) then the board will be back in the original state. So to get back to the original state after several slides, we just have to do the opposite slides in reverse order. If we did a right slide, then another right slide, then a down slide, we would have to do an up slide, left slide, and left slide to undo those first three slides. This is much easier than writing a function that can solve these puzzles simply by looking at the current state of them.

The Main Game Loop

```
    while True: # main game loop
        slideTo = None # the direction, if any, a tile should slide
        msg = '' # contains the message to show in the upper left corner.
        if mainBoard == SOLVEDBOARD:
            msg = 'Solved!'

        drawBoard(mainBoard, msg)

```

In the main game loop, the `slideTo` variable will track which direction the player wants to slide a tile (it starts off at the beginning of the game loop as None and is set later) and the `msg` variable tracks what string to display at the top of the window. The program does a quick check on line 64 to see if the board data structure has the same value as the solved board data structure stored in `SOLVEDBOARD`. If so, then the `msg` variable is changed to ontain `'Solved!'`. This won't appear on the screen until `drawBoard()` has been called to draw it to the `WINDOWSURF` Surface object (which is done on line 67) and `pygame.display.update()` is called to draw the display Surface object on the actual computer screen (which is done on line 291 at the end of the game loop).

Clicking on the Buttons

```
 69.         checkForQuit()
 70.         for event in pygame.event.get(): # event handling loop
 71.             if event.type == MOUSEBUTTONUP:
 72.                 spotx, spoty = getSpotClicked(mainBoard, event.pos[0],
event.pos[1])

 73.
 74.                 if (spotx, spoty) == (None, None):
 75.                     # check if the user clicked on an option button
 76.                     if RESET_RECT.collidepoint(event.pos):
 77.                         resetAnimation(mainBoard, allMoves) # clicked on
Reset button

 78.                         allMoves = []
 79.                     elif NEW_RECT.collidepoint(event.pos):
 80.                         mainBoard, solutionSeq = generateNewPuzzle(80) #
clicked on New Game button

 81.                         allMoves = []
 82.                     elif SOLVE_RECT.collidepoint(event.pos):
 83.                         resetAnimation(mainBoard, solutionSeq + allMoves)
# clicked on Solve button

 84.                         allMoves = []
```

Before going into the event loop, the program calls `checkForQuit()` to see if any `QUIT` events have been created (and terminates the program if there have). Why we have a separate function for handling the `QUIT` events will be explained later. The `for` loop on line 70 executes the event handling code for any other event created since the last time `pygame.event.get()` was called (or since the program started, if `pygame.event.get()` has never been called before).

If the type of event was a MOUSEBUTTONUP event (that is, the player had released a mouse button somewhere over the window), then we pass the mouse coordinates to our getSpotClicked() function which will return the board coordinates of the spot on the board the mouse release happened. The event.pos[0] is the X coordinate and event.pos[1] is the Y coordinate.

If the mouse button release did not happen over one of the spaces on the board (but obviously still happened somewhere on the window, since a MOUSEBUTTONUP event was created), then getSpotClicked() will return None. If this is the case, we want to do an additional check to see if the player might have clicked on the Reset, New, or Solve buttons (which are not located on the board). The coordinates of where these buttons are on the window are stored in the pygame.Rect objects that are stored in the RESET_RECT, NEW_RECT, and SOLVE_RECT variables. We can pass the mouse coordinates to the collidepoint() method which will return True if the mouse coordinates are within the Rect object's area and False otherwise.

Sliding Tiles with the Mouse

```
85.             else:
86.                 # check if the clicked tile was next to the blank spot
87.
88.                 blankx, blanky = getBlankPosition(mainBoard)
89.                 if spotx == blankx + 1 and spoty == blanky:
90.                     slideTo = LEFT
91.                 elif spotx == blankx - 1 and spoty == blanky:
92.                     slideTo = RIGHT
93.                 elif spotx == blankx and spoty == blanky + 1:
94.                     slideTo = UP
95.                 elif spotx == blankx and spoty == blanky - 1:
96.                     slideTo = DOWN
97.
```

If getSpotClicked() did not return None, then it will have returned a tuple of two integer values that represent the X and Y coordinate of the spot on the board that was clicked. Then the if and elif statements on lines 89 to 96 check if the spot that was clicked is a tile that is next to the blank spot (otherwise the tile will have no place to slide.)

Our getBlankPosition() function will take the board data structure and return the X and Y board coordinates of the blank spot, which we store in the variables blankx and blanky. Unlike getSpotClicked(), we know getBlankPosition() will always return XY coordinates because there will always be a blank spot on the board. If the spot the user clicked on was next to the blank space, we set the slideTo variable with the value that the tile should slide.

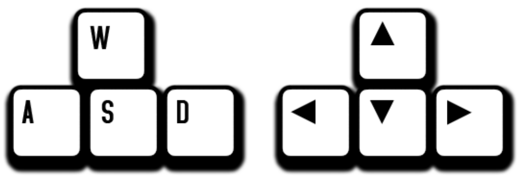
Sliding Tiles with the Keyboard

```
 98.                elif event.type == KEYUP:
 99.                    # check if the user pressed a key to slide a tile
100.                    if (event.key == K_LEFT or event.key == ord('a')) and
isValidMove(mainBoard, LEFT):
101.                        slideTo = LEFT
102.                    elif (event.key == K_RIGHT or event.key == ord('d')) and
isValidMove(mainBoard, RIGHT):
103.                        slideTo = RIGHT
104.                    elif (event.key == K_UP or event.key == ord('w')) and
isValidMove(mainBoard, UP):
105.                        slideTo = UP
106.                    elif (event.key == K_DOWN or event.key == ord('s')) and
isValidMove(mainBoard, DOWN):
107.                        slideTo = DOWN
108.
```

We can also let the user slide tiles by pressing keyboard keys. The if and elif statements on lines 100 to 107 let the user set the slideTo variable by either pressing the arrow keys or the WASD keys (if they prefer using their left hand.) Each if and elif statement also has a call to isValidMove() to make sure that the tile can slide in that direction. (We didn't have to make this call with the mouse clicks because the checks for the neighboring blank space did the same thing.)

WASD and Arrow Keys

The W, A, S, and D keys (together called the WASD keys) are commonly used in computer games to do the same thing as the arrow keys, except the player can use his left hand instead (since the WASD keys are on the left side of the keyboard.) W is for up, A is for left, S is for down, and D is for right. You can easily remember this because the WASD keys have the same layout as the arrow keys:



Actually Performing the Tile Slide

```
        if slideTo:
            slideAnimation(mainBoard, slideTo, 'Click tile or press arrow keys to slide.', 8) # show slide on screen
            makeMove(mainBoard, slideTo)
            allMoves.append(slideTo) # record the slide
        pygame.display.update()
        FPSCLOCK.tick(FPS)


```

Now that the events have all been handled, we should update the game state and display the new state on the screen. If slideTo has been set (either by the mouse event or keyboard event handling code) then we can call slideAnimation() to perform the sliding animation. (The parameters are the board data structure, the direction of the slide, a message to display while sliding the tile, and the speed of the sliding. This is explained in detail later in this chapter.) The slideAnimation() function has its own animation loop. After it returns, we need to update the actual board data structure (which is done in the makeMove() function) and then add the slide to the allMoves list of all the slides. This is done so that if the player clicks on the "Reset" button, we know how to undo all the player's slides.

IDLE and Terminating Pygame Programs

```
def terminate():
    pygame.quit()
    sys.exit()


```

This is a function that we can call that calls both the pygame.quit() and sys.exit() functions. This is a bit of syntactic sugar, so that instead of remembering to make both of these calls, there is just a single function we can call instead.

Checking for a Specific Event, and an Explanation of Pygame's Event Queue

```
def checkForQuit():
    for event in pygame.event.get(QUIT): # get all the QUIT events
        terminate() # terminate if any QUIT events are present
    for event in pygame.event.get(KEYUP): # get all the KEYUP events
        if event.key == K_ESCAPE:
            terminate() # terminate if the KEYUP event was for the Esc key
        pygame.event.post(event) # put the other KEYUP event objects back
```

```


```

The checkForQuit() function will check for QUIT events (or if the user has pressed the Esc key) and then call the terminate() function. But this is a bit tricky, and requires some explanation.

Pygame internally has it's own list data structure that it creates and appends Event objects to as they are made. This data structure is called the **event queue**. When the pygame.event.get() function is called with no parameters, the entire list is returned. However, you can pass a constant like QUIT to pygame.event.get() so that it will only return the QUIT events (if any) that are in the internal event queue. The rest of the events will stay in the event queue for the next time pygame.event.get() is called.

You should note that Pygame's event queue only stores up to 127 Event objects. If your program does not call pygame.event.get() frequently enough and the queue fills up, then any new events that happen won't be added to the event queue.

Line 123 essentially pulls out a list of QUIT events from Pygame's event queue and returns them. If there are any QUIT events in the event queue, the program terminates.

Line 125 pulls out all the KEYUP events from the event queue and checks if any of them are for the Esc key. If one of the events is, then the program terminates. However, there could be KEYUP events for keys other than the Esc key. In this case, we need to put the KEYUP event back into Pygame's event queue. We can do this with the pygame.event.post() function. This way, when line 70 calls pygame.event.get() the non-Esc key KEYUP events will still be there. Otherwise calls to checkForQuit() would "consume" all of the KEYUP events and those events would never be handled.

Creating the Board Data Structure

```
def getStartingBoard():
    # Return a board data structure with tiles in the solved state.
    # For example, if BOARDWIDTH and BOARDHEIGHT are both 3, this function
    # returns [[1, 4, 7], [2, 5, 8], [3, 6, None]]
    counter = 1
    board = []
    for x in range(BOARDWIDTH):
        column = []
        for y in range(BOARDHEIGHT):
            column.append(counter)
            counter += BOARDWIDTH
        board.append(column)
        counter -= BOARDWIDTH * (BOARDHEIGHT - 1) + BOARDWIDTH - 1

    board[BOARDWIDTH-1][BOARDHEIGHT-1] = None
    return board


```

The getStartingBoard() data structure will create and return a data structure that represents a “solved” board, where all the numbered tiles are in order and the blank tile is in the lower right corner. This is done with nested for loops, just like the board data structure in the Memory Puzzle game was made.

However, notice that the first column isn’t going to be [1, 2, 3] but instead [1, 4, 7]. This is because the numbers on the tiles increase by 1 going across the row, not down the column. Going down the column, the numbers increase by the size of the board’s width (which is stored in the BOARDWIDTH constant). We will use the counter variable to keep track of the number that should go on the next tile. When the numbering of the tiles in the column is finished, then we need to subtract the counter to the number at the start of the next column.

Not Tracking the Blank Position

```
def getBlankPosition(board):
    # Return the x and y of board coordinates of the blank space.
    for x in range(BOARDWIDTH)):
        for y in range(BOARDHEIGHT):
            if board[x][y] == None:
                return (x, y)


```

Whenever our code needs to find the XY coordinates of the blank space, instead of keeping track of where the blank space is after each slide, we can just create a function that goes through the entire board and finds the blank space coordinates. The None value is used in the board data structure to represent the blank space. The code in getBlankPosition() simply uses nested for loops to find which space on the board is the blank space.

Making a Move by Updating the Board Data Structure

```
def makeMove(board, move):
    # This function does not check if the move is valid.
    blankx, blanky = getBlankPosition(board)

    if move == UP:
        board[blankx][blanky], board[blankx][blanky + 1] =
board[blankx][blanky + 1], board[blankx][blanky]
```

```
    elif move == DOWN:
        board[blankx][blanky], board[blankx][blanky - 1] = board[blankx][blanky - 1], board[blankx][blanky]

    elif move == LEFT:
        board[blankx][blanky], board[blankx + 1][blanky] = board[blankx + 1][blanky], board[blankx][blanky]

    elif move == RIGHT:
        board[blankx][blanky], board[blankx - 1][blanky] = board[blankx - 1][blanky], board[blankx][blanky]


```

The data structure in the board parameter is a 2D list that represents where all the tiles are. Whenever the player makes a move, the program needs to update this data structure. What happens is that the value for the tile is swapped with the value for the blank space. In other programming languages, this swapping of values would usually require a third “temporary” variable. Take a look at this code which swaps the values in the a and b variables:

```
>>> a = 42
>>> b = 5
>>> print(a, b)
42 5
>>> temp = a
>>> a = b
>>> b = temp
>>> print(a, b)
5 42
>>>
```

But Python has a nicer way of doing this:

```
>>> a = 42
>>> b = 5
>>> print(a, b)
42 5
>>> a, b = b, a
>>> print(a, b)
5 42
>>>
```

We will use this swap technique to swap the values for the sliding tile and the blank space. Depending on the directional value that is in the move parameter, we will swap the blank value with a value of a tile next to it. (We get the index of the neighboring tile by adding or subtracting 1 to the blankx or blanky coordinate.)

The makeMove() function doesn't have to return any values, because the board parameter has a list reference passed for its argument. This means that any changes we make to board in this function will be made to the list value that was passed to makeMove(). (You can review the concept of references at http://invpy.com/references.)

When NOT to Use an Assertion

```
171. def isValidMove(board, move):
172.     blankx, blanky = getBlankPosition(board)
173.     return (move == UP and blanky != len(board[0]) - 1) or \
174.            (move == DOWN and blanky != 0) or \
175.            (move == LEFT and blankx != len(board) - 1) or \
176.            (move == RIGHT and blankx != 0)
177.
178.
```

The isValidMove() function is passed a board data structure and a move the player would want to make. The return value is True if this move is possible and False if it is not. For example, you cannot slide a tile to the left one hundred times in a row, because eventually the blank space will be at the edge and there are no more tiles to slide to the left.

Whether a move is valid or not depends on where the blank space is. This function makes a call to getBlankPosition() to find the X and Y coordinates of the blank spot. Lines 173 to 176 are a return statement with a single expression. The \ slashes at the end of the first three lines tells the Python interpreter that that is not the end of the line of code (even though it is at the end of the line.)

Because the parts of this expression in parentheses are joined by or operators, only one of them needs to be True for the entire expression to be True. Each of these parts checks what the intended move is and then sees if the coordinate of the blank space allows that move.

Getting a Not-So-Random Move

```
179. def getRandomMove(board, lastMove=None):
180.     # start with a full list of all four moves
181.     validMoves = [UP, DOWN, LEFT, RIGHT]
182.
183.     # remove moves from the list as they are disqualified
184.     if lastMove == UP or not isValidMove(board, DOWN):
```

```
185.         validMoves.remove(DOWN)
186.     if lastMove == DOWN or not isValidMove(board, UP):
187.         validMoves.remove(UP)
188.     if lastMove == LEFT or not isValidMove(board, RIGHT):
189.         validMoves.remove(RIGHT)
190.     if lastMove == RIGHT or not isValidMove(board, LEFT):
191.         validMoves.remove(LEFT)
192.
193.     # return a random move from the list of remaining moves
194.     return random.choice(validMoves)
195.
196.
```

At the beginning of the game, we start with the board data structure in the solved, ordered state and create the puzzle by randomly sliding around tiles. To decide which of the four directions we should slide, we'll call our getRandomMove() function. Normally we could just use the random.choice() function and pass it a tuple (UP, DOWN, LEFT, RIGHT) to have Python simply randomly choose a direction value for us. But the Sliding Puzzle game has a small requirement that means we can't choose a purely random number.

If you had a slide puzzle and slide a tile to left, and then slide a tile to the right, you would end up with the exact same board you had at the start. It's pointless to make a slide and then make the opposite slide. Also, if the blank space is in the lower right corner than it is impossible to slide a tile up or to the left.

The code in getRandomMove() will take these factors into account. To prevent the function from selecting the last move that was made, the caller of the function can pass a directional value for the lastMove parameter. Line 181 starts with a list of all four directional values stored in the validMoves variable. The lastMove value (if not set to None) is removed from validMoves. Depending on if the blank space is at the edge of the board, lines 184 to 191 will remove other directional values from the lastMove list.

Of the values that are left in lastMove, one of them is randomly selected with a call to random.choice() and returned.

Converting Tile Coordinates to Pixel Coordinates

```
197. def getLeftTopOfTile(tileX, tileY):
198.     left = XMARGIN + (tileX * TILESIZE) + (tileX - 1)
199.     top = YMARGIN + (tileY * TILESIZE) + (tileY - 1)
200.     return (left, top)
201.
202.
```

The getLeftTopOfTile() function converts board coordinates to pixel coordinates. For the XY coordinates of the board space that are passed in, the function calculates and returns the XY coordinates of the pixel at the top left of that board space.

Converting from Pixel Coordinates to Board Coordinates

```
def getSpotClicked(board, x, y):
    # from the x & y pixel coordinates, get the x & y board coordinates
    for tileX in range(len(board[0])):
        for tileY in range(len(board)):
            left, top = getLeftTopOfTile(tileX, tileY)
            tileRect = pygame.Rect(left, top, TILESIZE, TILESIZE)
            if tileRect.collidepoint(x, y):
                return (tileX, tileY)
    return (None, None)


```

The getSpotClicked() function converts from pixel coordinates to board coordinates. The nested loops go through every possible XY board coordinate, and if the pixel coordinates that were passed in are within that space on the board, it returns those board coordinates. Since all of the tiles have a width and height that is set in the TILESIZE constant, we can create a Rect object that represents the space on the board by getting the pixel coordinates of the top left corner of the board space, and then use the collidepoint() Rect method to see if the pixel cooridinates are inside that Rect object's area.

If the pixel coordinates that were passed in were not over any board space, then the value (None, None) is returned.

Drawing a Tile

```
def drawTile(tilex, tiley, number, adjx=0, adjy=0):
    # draw a tile at board coordinates tilex and tiley, optionally a few
    # pixels over (determined by adjx and adjy)
    left, top = getLeftTopOfTile(tilex, tiley)
    pygame.draw.rect(WINDOWSURF, TILECOLOR, (left + adjx, top + adjy, TILESIZE, TILESIZE))

    textSurf = BASICFONT.render(str(number), True, TEXTCOLOR)
    textRect = textSurf.get_rect()
    textRect.center = left + int(TILESIZE / 2) + adjx, top + int(TILESIZE / 2) + adjy

    WINDOWSURF.blit(textSurf, textRect)

```

```

```

The drawTile() function will draw a single numbered tile on the board. The tilex and tiley parameters are the board coordinates of the tile. The number parameter is a string of the tile's number (like '3' or '12'). The adjx and adjy keyword parameters are for making minor adjustments to the position of the tile. For example, passing 5 for adjx would make the tile appear 5 pixels to the right of the space on the board given in tilex and tiley. Passing -10 for adjx would make the tile appear 10 pixels to the left of the space. If no values are passed for these arguments when drawTile() is called, then by default they are set to 0. (This means they will be exactly on the board space given by tilex and tiley.) These adjustment values will be handy when we need to draw the tile in the middle of sliding.

The Pygame drawing functions only use pixel coordinates, so first line 217 converts the board coordinates in tilex and tiley to pixel coordinates, which we will store in variables left and top (since getLeftTopOfTile() returns the top left corner coordinates.) We draw the background square of the tile with a call to pygame.draw.rect() while adding the adjx and adjy values to left and top in case the code needs to adjust the position of the tile.

Lines 219 to 222 then create the Surface object that has the number text drawn on it, position it, and then blit it to the display Surface. The drawTile() function doesn't call pygame.display.update() function, since the caller of drawTile() probably will want to draw more tiles for the rest of the board before making them appear on the screen.

The Making Text Appear on the Screen

```
def makeText(text, color, bgcolor, top, left):
    # create the Surface and Rect objects for some text.
    textSurf = BASICFONT.render(text, True, color, bgcolor)
    textRect = textSurf.get_rect()
    textRect.topleft = (top, left)
    return (textSurf, textRect)


```

The makeText() function handles creating the Surface and Rect objects for positioning text on the screen. Instead of doing all these calls each time we want to make text on the screen, we can just call makeText() instead. This saves us on the amount of typing we have to do for our program. (Though drawTile() makes the calls to render() and get_rect() itself because it positions the text Surface object by the center point, rather than the topleft point.)

Drawing the Board

```
233. def drawBoard(board, message):
234.     WINDOWSURF.fill(BGCOLOR)
235.     if message:
236.         textSurf, textRect = makeText(message, MESSAGECOLOR, BGCOLOR, 5,
5)

237.         WINDOWSURF.blit(textSurf, textRect)
238.
239.     for tilex in range(len(board[0])):
240.         for tiley in range(len(board)):
241.             if board[tilex][tiley]:
242.                 drawTile(tilex, tiley, board[tilex][tiley])
243.
```

This function handles drawing the entire board and all of its tiles to the WINDOWSURF display Surface object. The fill() method on line 234 completely paints over anything that used to be drawn on this Surface object before so that we start fresh.

Line 235 to 237 handles drawing the message at the top of the window. We use this for the “Generating new puzzle…” and other text we want to display at the top of the screen. Remember that if statement conditions consider the blank string to be a False value, so if message is set to '' then lines 236 and 237 are skipped.

Next, nested for loops are used to draw each tile to the display Surface object by calling the drawTile() function.

Drawing the Border of the Board

```
244.     left, top = getLeftTopOfTile(0, 0)
245.     width = BOARDWIDTH * TILESIZE
246.     height = BOARDHEIGHT * TILESIZE
247.     pygame.draw.rect(WINDOWSURF, BORDERCOLOR, (left - 5, top - 5, width +
11, height + 11), 4)

248.
```

Lines 244 to 246 draw a border around the tiles. The top left corner of the boarder will be 5 pixels to the left and 5 pixels above the top left corner of the tile at board coordinates (0, 0). The width and height of the border are calculated from the number of tiles wide and high the board is (stored in the BOARDWIDTH and BOARDHEIGHT constants) multiplied by the size of the tiles (stored in the TILESIZE constant).

The rectangle we draw on line 247 will have a thickness of 4 pixels, so we will move the boarder 5 pixels to the left and above where the top and left variables point so the thickness of the line won't overlap the tiles. We will also add 11 to the width and length (5 of those 11 pixels are to compensate for moving the rectangle to the left and up).

Drawing the Buttons

```
249.     WINDOWSURF.blit(RESET_SURF, RESET_RECT)
250.     WINDOWSURF.blit(NEW_SURF, NEW_RECT)
251.     WINDOWSURF.blit(SOLVE_SURF, SOLVE_RECT)
252.
253.
```

Finally, we draw the buttons off to the slide of the screen. The text and position of these buttons never changes, which is why they were stored in constant variables at the beginning of the main() function.

Animating the Tile Slides

```
254. def slideAnimation(board, direction, message, animationSpeed):
255.     # Note: This function does not check if the move is valid.
256.
257.     blankx, blanky = getBlankPosition(board)
258.     if direction == UP:
259.         movex = blankx
260.         movey = blanky + 1
261.     elif direction == DOWN:
262.         movex = blankx
263.         movey = blanky - 1
264.     elif direction == LEFT:
265.         movex = blankx + 1
266.         movey = blanky
267.     elif direction == RIGHT:
268.         movex = blankx - 1
269.         movey = blanky
270.
```

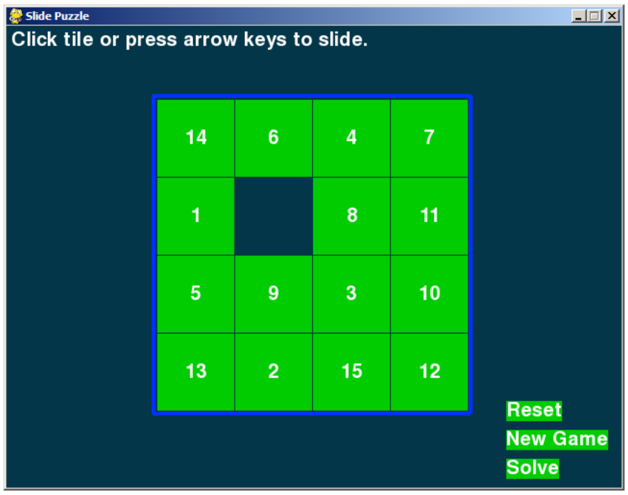
The first thing our code to animate the tile sliding needs to calculate is where the blank space is and where the moving tile is. The comment on line 255 reminds us that the code that calls slideAnimation() should make sure that the slide it passes for the direction parameter is a valid move to make.

The blank space's coordinates can come from a call to getBlankPosition(). From these coordinates and the direction of the slide, we can figure out the XY board coordinates of the tile that will slide. These coordinates will be stored in the movex and movey variables.

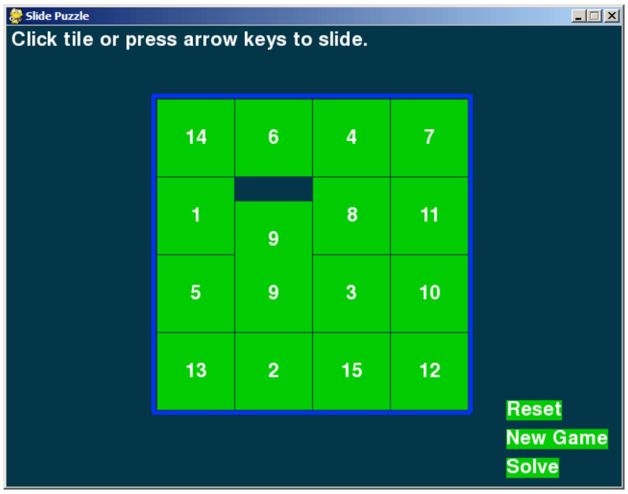
```
    # prepare the base surface
    drawBoard(board, message)
    baseSurf = WINDOWSURF.copy()
    # draw a blank space over the moving tile on the baseSurf Surface.
    moveLeft, moveTop = getLeftTopOfTile(movex, movey)
    pygame.draw.rect(baseSurf, BGCOLOR, (moveLeft, moveTop, TILESIZE,
TILESIZE))

```

The slideAnimation() will contain its own animation loop to handle drawing the tile sliding to the blank space. We will do this by drawing the board to the display Surface object with the drawBoard() function. Then, we will make a copy of this Surface object with the copy() method. The copy() method of Surface objects will return a new Surface object that has the same image drawn to it. But they are two separate Surface objects. After calling the copy() method, if we draw on one Surface object using blit() or the Pygame drawing functions, it will not change the image on the other Surface object. This copy we will store in the baseSurf variable.

Next, we will paint another blank space over the tile that will slide. This is because when we draw each frame of the sliding animation, we will draw the sliding tile over different parts of the baseSurf Surface object. If we didn't blank out the moving tile on the baseSurf Surface, then it would still be there as we draw the sliding tile. In that case, here is what the baseSurf Surface would look like:



And then what it would look like when we draw the sliding tile on top of it:



You can see this for yourself by commenting out line 276 and running the program.

```
278.     for i in range(0, TILESIZE, animationSpeed):
279.         # animate the tile sliding over
280.         checkForQuit()
281.         WINDOWSURF.blit(baseSurf, (0, 0))
282.         if direction == UP:
283.             drawTile(movex, movey, board[movex][movey], 0, -i)
284.         if direction == DOWN:
285.             drawTile(movex, movey, board[movex][movey], 0, i)
286.         if direction == LEFT:
287.             drawTile(movex, movey, board[movex][movey], -i, 0)
288.         if direction == RIGHT:
289.             drawTile(movex, movey, board[movex][movey], i, 0)
290.
291.         pygame.display.update()
292.         FPSCLOCK.tick(FPS)
293.
294.
```

In order to draw the frames of the sliding animation, we must draw the baseSurf surface on the display Surface, then draw the tile closer and closer to its final position where the original blank space was. The space between two adjacent tiles is the same size as a single tile, which we have stored in TILESIZE. The code uses a for loop to go from 0 to TILESIZE. Normally this would mean that we would draw the tile 0 pixels over, then on the next frame draw the tile 1 pixel over, then 2 pixels, then 3, and so on. Each of these frames would take 1/30th of a second. If you have

TILESIZE set to 80 (as the program in this book does on line 12) then sliding a tile would take over two and a half seconds, which is actually kind of slow.

So instead we will have the for loop go from 0 to TILESIZE by several pixels each frame. The number of pixels it jumps over is stored in animationSpeed, which is passed in when slideAnimation() is called. For example, if animationSpeed was set to 8 and the constant TILESIZE was set to 80, then the for loop and range(0, TILESIZE, animationSpeed) would set the i variable to the values 0, 8, 16, 24, 32, 40, 48, 56, 64, 72. (It does not include 80 because the range() function goes up to, but not including, the second argument. This means the entire sliding animation would be done in 10 frames, which would mean it is done in 10/30th of a second (a third of a second.)

Lines 282 to 289 makes sure that we draw the tile sliding in the correct direction (based on what value the direction variable has). After the animation is done, then the function returns. Notice that while the animation is happening, any events being created by the user are not being handled. Those events will be handled the next time execution reaches line 70 in the main() function or the code in the checkForQuit() function.

Creating a New Puzzle

```
295. def generateNewPuzzle(numSlides):
296.     # From a starting configuration, make numSlides number of moves (and
297.     # animate these moves).
298.     sequence = []
299.     board = getStartingBoard()
300.     drawBoard(board, '')
301.     pygame.display.update()
302.     pygame.time.wait(500) # pause 500 milliseconds for effect
```

The generateNewPuzzle() function will be called at the start of each new game. It will create a new board data structure by calling getStartingBoard() and than randomly scramble it. The first few lines of generateNewPuzzle() get the board and then draw it to the screen (freezing for half a second to let the player see the fresh board for a moment.)

```
303.     lastMove = None
304.     for i in range(numSlides):
305.         move = getRandomMove(board, lastMove)
306.         slideAnimation(board, move, 'Generating new puzzle...',
int(TILESIZE / 3))

307.         makeMove(board, move)
308.         sequence.append(move)
309.         lastMove = move
```

```
310.     return (board, sequence)
311.
312.
```

The numSlides parameter will show tell the function how many of these random moves to make. The code for doing a random move is to call getRandomMove() to get the move itself, then call slideAnimation() to perform the animation on the screen. Because doing the slide animation does not actually update the board data structure, we update the board by calling makeMove().

We need to keep track of each of the random moves that was made so that the player can click the "Solve" button later and have the program undo all these random moves. (The "Being Smart By Using Stupid Code" section talks about why and how we do this.)

Then we store the random move in a variable called lastMove which will be passed to getRandomMove() on the next iteration. This prevents the next random move from undoing the random move we just performed.

All of this needs to happen numSlides number of times, so we put lines 305 to 309 inside a for loop. When the board is done being scrambled, then we return the board data structure and also the list of the random moves made on it.

Animating the Board Reset

```
313. def resetAnimation(board, allMoves):
314.     # make all of the moves in allMoves in reverse.
315.     revAllMoves = allMoves[:] # gets a copy of the list
316.     revAllMoves.reverse()
317.
318.     for move in revAllMoves:
319.         if move == UP:
320.             oppositeMove = DOWN
321.         elif move == DOWN:
322.             oppositeMove = UP
323.         elif move == RIGHT:
324.             oppositeMove = LEFT
325.         elif move == LEFT:
326.             oppositeMove = RIGHT
327.         slideAnimation(board, oppositeMove, '', int(TILESIZE / 2))
328.         makeMove(board, oppositeMove)
329.
330.
```

When the player clicks on “Reset” or “Solve”, the Slide Puzzle game program needs to undo all of the moves that were made to the board. The list of directional values for the slides will be passed as the argument for the allMoves parameter.

Line 315 uses list slicing to create a duplicate of the allMoves list. Remember that if you don’t specify a number before the :, then Python assumes the slice should start from the very beginning of the list. And if you don’t specify a number after the :, then Python assumes the slice should keep going to the very end of the list. So allMoves[:] creates a list slice of the entire allMoves list. This makes a copy of the actual list to store in revAllMoves, rather than just a copy of the list reference. (See http://invpy.com/references for details.)

To undo all the moves in allMoves, we need to perform the opposite move of the moves in allMoves, and in reverse order. There is a list method called reverse() which will reverse the order of the items in a list. We call this on line 316.

The for loop on line 318 goes through all these now reversed directional values. Remember, we want the opposite move, so the if and elif statements from line 319 to 326 set the correct directional value in the oppositeMove variable. Then we call slideAnimation() to perform the animation, and makeMove() to update the board data structure.

And Then Actually Start the Program

```
331. if __name__ == '__main__':
332.     main()
```

Just like in the Memory Puzzle game, after all the def statements have been executed to create all the functions, we call the main() function to begin the meat of the program.

That’s all there is to the Slide Puzzle program! But let’s talk about some general programming concepts that came up in this game.

Time vs. Memory Tradeoffs

Of course, there are a few different ways to write the Slide Puzzle game so that it looks and acts the exact same way even though the code is different. There are many different ways the a program that does a task could be written. The most common differences are making tradeoffs between execution time and memory usage.

Usually, the faster a program can run, the better it is. This is especially true with programs that need to do a lot of calculations, whether they are scientific weather simulators or games with a large amount of detailed 3D graphics to draw. It’s also good to use the least amount of memory possible. The more variables and the larger the lists your program uses, the more memory it takes up. (You can find out how to measure your program's memory usage and execution time at http://invpy.com/profiling.)

Right now, the programs in this book aren't big and complicated enough where you have to worry about conserving memory or optimizing the execution time. But it can be something to consider as you become a more skilled programmer.

For example, consider the getBlankPosition() function. This function takes time to run, since it goes through all the possible board coordinates to find where the blank space is. Instead, we could just have a blankspacex and blankspacey variable which would have these XY coordinates so we would not have to look through the entire board each time we want to know where it was. (We would also need code that updates the blankspacex and blankspacey variables whenever a move is done. This code could go in makeMove().) Using these variables would take up more memory, but they would save you on execution time so your program would run faster.

Another example is that we keep a board data structure in the solved state in the SOLVEDBOARD variable, so that we can compare the current board to SOLVEDBOARD to see if the player has solved the puzzle. Each time we wanted to do this check, we could just call the getStartingBoard() function and compare the returned value to the current board. Then we would not need the SOLVEDBOARD variable. This would save us a little bit of memory, but then our program would take longer to run because it is re-creating the solved-state board data structure each time we do this check.

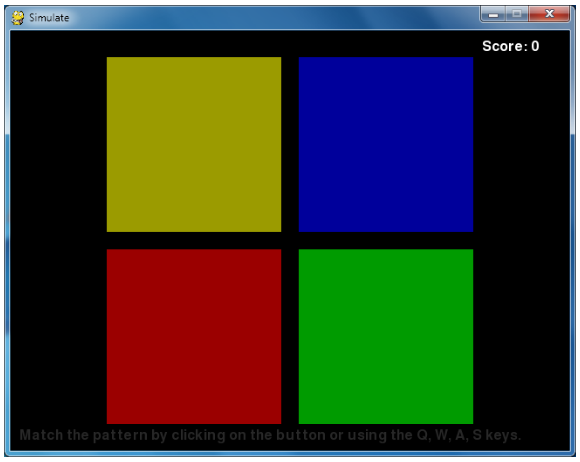
There is one thing you must remember though. Writing code that is readable is a very important skill. Code that is "readable" is code that is easy to understand, especially by programmers who did not write the code. If another programmer can look at your program's source code and figure out what it does without much trouble, then that program is very readable. Readability is important because when you want to fix bugs or add new features to your program (and bugs and ideas for new features **always** come up), then having a readable program makes those tasks much easier.

Nobody Cares About a Few Bytes

Also, there is one thing that might seem kind of silly to say in this book because it seem obvious, but many people wonder about it. You should know that using short variable names like x or num instead of longer, more descriptive variable names like blankx or numSlides does not save you any memory. Using these longer variable names is better because they'll make your program more readable.

Nobody Cares About a Few Million Nanoseconds

Chapter 5 – Simulate


Simulate
Score: 0
Match the pattern by clicking on the button or using the Q, W, A, S keys.

How to Play Simulate

Simulate is a clone of the Simon game. There are four colored buttons on the screen. The buttons light up in a certain random pattern, and then the player must repeat this pattern. Each time the player successfully simulates the pattern, the pattern gets longer. The player continues playing until she eventually loses.

Source Code to Simulate

This source code can be downloaded from http://invpy.com/simulate.py.

The Usual Starting Stuff

```
# Simulate (a Simon clone)
# By Al Sweigart al@inventwithpython.com
# http://inventwithpython.com/pygame
# Creative Commons BY-NC-SA 3.0 US

import random, sys, time, pygame
from pygame.locals import *

FPS = 30
WINDOWWIDTH = 640
WINDOWHEIGHT = 480
FLASHSPEED = 500 # in milliseconds
FLASHDELAY = 200 # in milliseconds
```

```
BUTTONSIZE = 200
BUTTONGAPSIZE = 20
TIMEOUT = 4 # seconds before game over if no button is pushed.

#                R    G    B
WHITE        = (255, 255, 255)
BLACK        = (  0,   0,   0)
BRIGHTRED    = (255,   0,   0)
RED          = (155,   0,   0)
BRIGHTGREEN  = (  0, 255,   0)
GREEN        = (  0, 155,   0)
BRIGHTBLUE   = (  0,   0, 255)
BLUE         = (  0,   0, 155)
BRIGHTYELLOW = (255, 255,   0)
YELLOW       = (155, 155,   0)
DARKGRAY     = ( 40,  40,  40)
BGCOLOR = BLACK

XMARGIN = int((WINDOWWIDTH - (2 * BUTTONSIZE) - BUTTONGAPSIZE) / 2)
YMARGIN = int((WINDOWHEIGHT - (2 * BUTTONSIZE) - BUTTONGAPSIZE) / 2)

```

Here we set up the usual constants for things that we might want to modify later such as the size of the four buttons, the shades of color used for the buttons (the bright colors are used when the buttons light up) and the amount of time the player has to push the next button in the sequence before the game times out.

Setting Up the Buttons

```
# Rect objects for each of the four buttons
YELLOWRECT = pygame.Rect(XMARGIN, YMARGIN, BUTTONSIZE, BUTTONSIZE)
BLUERECT   = pygame.Rect(XMARGIN + BUTTONSIZE + BUTTONGAPSIZE, YMARGIN, BUTTONSIZE, BUTTONSIZE)
REDRECT    = pygame.Rect(XMARGIN, YMARGIN + BUTTONSIZE + BUTTONGAPSIZE, BUTTONSIZE, BUTTONSIZE)
GREENRECT  = pygame.Rect(XMARGIN + BUTTONSIZE + BUTTONGAPSIZE, YMARGIN + BUTTONSIZE + BUTTONGAPSIZE, BUTTONSIZE, BUTTONSIZE)

```

Just like the buttons in the Sliding Puzzle games for “Reset”, “Solve” and “New Game”, the Simulate game has four rectangular areas and code to handle when the player clicks inside of those areas. The program will need Rect objects for the areas of the four buttons so it can call the collidepoint() method. Lines 36 to 39 set up these Rect objects with the appropriate coordinates and sizes. TODO – Add image for the button coordinates.

The main() Function

```
41. def main():
42.     global FPSCLOCK, WINDOWSURF, BASICFONT, BEEP1, BEEP2, BEEP3, BEEP4
43.
44.     pygame.init()
45.     FPSCLOCK = pygame.time.Clock()
46.     WINDOWSURF = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
47.     pygame.display.set_caption('Simulate')
48.
49.     BASICFONT = pygame.font.Font('freesansbold.ttf', 16)
50.
51.     infoSurf = BASICFONT.render('Match the pattern by clicking on the
button or using the Q, W, A, S keys.', 1, DARKGRAY)

52.     infoRect = infoSurf.get_rect()
53.     infoRect.topleft = (10, WINDOWHEIGHT - 25)
54.     # load the sound files
55.     BEEP1 = pygame.mixer.Sound('beep1.ogg')
56.     BEEP2 = pygame.mixer.Sound('beep2.ogg')
57.     BEEP3 = pygame.mixer.Sound('beep3.ogg')
58.     BEEP4 = pygame.mixer.Sound('beep4.ogg')
59.
```

The main() function will call all of the helper functions that are defined elsewhere in the program. The usual Pygame setup functions are called to initialize the library, create a Clock object, create a window, set the caption, and create a Font object that will be used to display the score at the top right and the instructions at the bottom of the window. The objects that are created by these function calls will be stored in global variables so that they can be used in other functions.

Lines 55 to 58 will load sound files so that Simulate can play sound effects as the player clicks on each button. The pygame.mixer.Sound() constructor functions will store a sound object, which we store in the variables BEEP1 to BEEP4 which were made into global variables on line 42.

Some Local Variables Used in This Program

```
60.     # Initialize some variables for a new game
61.     pattern = [] # stores the pattern of colors
62.     currentStep = 0 # the color the player must push next
63.     lastClickTime = 0 # timestamp of the player's last button push
64.     score = 0
```

```
    # when False, the pattern is playing. when True, waiting for the player to click a colored button:
    waitingForInput = False

```

The pattern variable will be a list of color values (either YELLOW, RED, BLUE, or GREEN) to keep track of the pattern that the player must memorize. For example, if the value of pattern was [RED, RED, YELLOW, RED, BLUE, BLUE, RED, GREEN] then the player would have to first click the red button twice, then the yellow button, then the red button, and so on until the final green button. As the player finishes each round, a new random color is added to the end of the list.

The currentStep variable will keep track of which color in the pattern list the player has to click next. If currentStep was 0 and pattern was [GREEN, RED, RED, YELLOW], then the player would have to click the green button. If they clicked on any other button, we will have our code cause a game over.

There is a TIMEOUT constant that makes the player have to click on next button in the pattern within a number of seconds, otherwise the player loses. In order to check if enough time has passed since the last button click, the lastClickTime variable needs to keep track of the last time the player clicked on a button. (Python has a module named time and a time.time() function to return the current time. This will be explained later.)

The score variable keeps track of the score.

There are also two modes that our program will be in. Either the program is playing the pattern of buttons for the player (in which case, waitingForInput is set to False), or the program has finished playing the pattern and is waiting for the user to click the buttons in the correct order (in which case, waitingForInput is set to True).

Drawing the Board and Handling Input

```
    while True: # main game loop
        clickedButton = None # button that was clicked (set to YELLOW, RED, GREEN, or BLUE)
        WINDOWSURF.fill(BGCOLOR)
        drawButtons()

        scoreSurf = BASICFONT.render('Score: ' + str(score), 1, WHITE)
        scoreRect = scoreSurf.get_rect()
        scoreRect.topleft = (WINDOWWIDTH - 100, 10)
```

```
 76.         WINDOWSURF.blit(scoreSurf, scoreRect)
 77.
 78.         WINDOWSURF.blit(infoSurf, infoRect)
 79.
```

Line 68 is the start of the main game loop. The clickedButton will be reset to None at the beginning of each iteration. If a button is clicked during this iteration, then clickedButton will be set to one of the color values to match the button (YELLOW, RED, GREEN, or BLUE).

The fill() method is called on line 70 to repaint the entire display Surface so that we can start drawing from scratch. The four colored buttons are drawn with a call to the drawButtons() (explained later). Then the text for the score is created on lines 73 to 76.

There will also be text that tells the player what their current score is. Unlike the call to the render() method on line 51 for the instruction text, the text for the score changes. It starts off as 'Score: 0' and then becomes 'Score: 1' and then 'Score: 2' and so on. This is why we create new Surface objects by calling the render() method on line 73 inside the game loop.

Checking for Mouse Clicks

```
 80.         checkForQuit()
 81.         for event in pygame.event.get(): # event handling loop
 82.             if event.type == MOUSEBUTTONUP:
 83.                 mousex, mousey = event.pos
 84.                 clickedButton = getButtonClicked(event.pos[0],
event.pos[1])
```

Line 80 does a quick check for any QUIT events, and then line 81 is the start of the event loop. The XY coordinates of any mouse clicks will be stored in the mousex and mousey variables. If the mouse click was over one of the four buttons, then our getButtonClicked() function will return a Color object of the button clicked (otherwise it returns None).

Checking for Keyboard Presses

```
 85.             elif event.type == KEYDOWN:
 86.                 if event.key == K_q:
 87.                     clickedButton = YELLOW
 88.                 elif event.key == K_w:
 89.                     clickedButton = BLUE
 90.                 elif event.key == K_a:
 91.                     clickedButton = RED
 92.                 elif event.key == K_s:
 93.                     clickedButton = GREEN
95.
```

```
 96.
```

Lines 85 to 93 check for any KEYDOWN events (created when the user presses a key on the keyboard). The Q, W, A, and S keys correspond to the buttons because they are arranged in a square shape on the keyboard.

The Q key is in the upper left of the four keyboard keys, just like the yellow button on the screen is in the upper left, so we will make pressing the Q key the same as clicking on the yellow button. We can do this by setting the clickedButton variable to the value in the constant variable YELLOW. We can do the same for the three other keys. This way, the user can play Simulate with either the mouse or keyboard.

The Two States of the Game Loop

```
 97.         if not waitingForInput:
 98.             # play the pattern
 99.             pygame.display.update()
100.             pygame.time.wait(1000)
101.             pattern.append(random.choice((YELLOW, BLUE, RED, GREEN)))
102.             for button in pattern:
103.                 flashButtonAnimation(button)
104.                 pygame.time.wait(FLASHDELAY)
105.             waitingForInput = True
```

There are two different “modes” or “states” that the program can be in. When waitingForInput is False, the program will be displaying the animation for the pattern. When waitingForInput is True, the program will be waiting for the user to use the mouse or keyboard to select buttons.

Lines 97 to 105 will cover the case where the program displays the pattern animation. Since this is done at the start of the game or when the player finishes a pattern, line 101 will add a random color to the pattern list to make the pattern one step longer. Then lines 102 to 104 loops through each of the values in the pattern list and calls flashButtonAnimation() to make that button light up. After it is done lighting up all the buttons in the pattern list, the program sets the waitingForInput variable to True.

Figuring Out if the Player Pressed the Right Buttons

```
106.         else:
107.             # wait for the player to enter buttons
108.             if clickedButton and clickedButton == pattern[currentStep]:
109.                 # pushed the correct button
110.                 flashButtonAnimation(clickedButton)
111.                 currentStep += 1
```

```
112.                 lastClickTime = time.time()
113.
```

If waitingForInput is True, then the code in line 106's else statement will execute. Line 108 checks if the player has clicked on a button during this iteration of the game loop and if that button was the correct one. The currentStep variable keeps track of the index in the pattern list for the button that the player should click on next.

For example, if pattern was set to [YELLOW, RED, RED] and the currentStep variable was set to 0 (like it would be when the player first starts the game), then the correct button for the player to click would be pattern[0] or the yellow button.

If the player has clicked on the correct button, we want to flash the button the player clicked by calling out flashButtonAnimation(), increase the currentStep to the next step, and then update the lastClickTime variable to the current time. (The time.time() function returns an integer of the number of seconds since January 1st, 1970, so we can use it to keep track of time.)

```
114.                 if currentStep == len(pattern):
115.                     # pushed the last button in the pattern
116.                     changeBackgroundAnimation()
117.                     score += 1
118.                     waitingForInput = False
119.                     currentStep = 0 # reset back to first step
120.
```

Lines 114 to 119 are inside the else statement that started on line 106. If the execution is inside that else statement, we know the player clicked on a button and also it was the correct button. Line 114 checks if this was the last correct button in the pattern list by checking if the integer stored in currentStep is equal to the number of values inside the pattern list.

If this is True, then we want to change the background color by calling our changeBackgroundAnimation(). (This is a nice way to tell the player they have entered the entire pattern correctly.) The score is incremented, currentStep is set back to 0, and the waitingForInput variable is set to False so that on the next iteration of the game loop the program will add a new Color value to the pattern list and then flash the buttons.

```
121.             elif (clickedButton and clickedButton != pattern[currentStep])
or (currentStep != 0 and time.time() - TIMEOUT > lastClickTime):
```

If the player did not click on the correct button, the elif statement on line 121 handles the case where either the player clicked on the wrong button or the player has waited too long to click on a button. Either way, we need to show the “game over” animation and start a new game.

The (clickedButton and clickedButton != pattern[currentStep]) part of the elif statement’s condition checks if a button was clicked and it was the wrong button to click. You can compare this to line 108’s if statement’s condition clickedButton and clickedButton == pattern[currentStep] which evaluates to True if the player clicked a button and it was the correct button to click.

The other part of line 121’s elif condition is (currentStep != 0 and time.time() - TIMEOUT > lastClickTime). This handles making sure the player did not “time out”. Notice that this part of the condition has two expressions connected by an and keyword. That means both sides of the and keyword need to evaluate to True.

In order to “time out”, it must not be the player’s first button click. This way, the player can wait as long as they want before starting to enter the pattern. But once they’ve started to click buttons, they must keep clicking the buttons quickly enough until they’ve entered the entire pattern (or have clicked on the wrong pattern and gotten a “game over”.) We can check for this with currentStep != 0.

Epoch Time

Also in order to “time out”, the current time (returned by time.time()) minus four seconds (4 is stored in TIMEOUT) must be greater than the last time clicked a button (stored in lastClickTime). The reason why time.time() - TIMEOUT > lastClickTime works has to do with how epoch time works. Epoch time (also called Unix epoch time) is the number of seconds it has been since January 1st, 1970. (This date is called the Unix epoch, which is considered the birthday of the first Unix computers.)

For example, when I run time.time() from the interactive shell (don’t forget to import the time module first), it looks like this:

```
>>> import time
>>> time.time()
1320460242.118
```

What this number means is that the moment the time.time() function was called was a little over 1,320,460,242 since midnight of January 1st, 1970. (This translates to November 4th, 2011 at 7:30:42pm. You can learn how to convert from Unix epoch time to regular English time at http://invpy.com/epochtime)

If I call time.time() from the interactive shell a few seconds later, it might look like this:

```
>>> time.time()
1320460261.315
```

1320460261.315 seconds after midnight of the Unix epoch is November 4th, 2011 at 7:31:01pm. (Actually, it's 7:31 and 1.315 seconds if you want to be precise.)

Dealing with time would be difficult if we had to deal with strings. It's hard to tell that 19 seconds have passed if we only had the string values '7:30:42 PM' and '7:31:01 PM' to compare. But with epoch time, it's just a matter of subtracting the integers 1320460261.315 - 1320460242.118, which evaluates to 19.197000026702881. This value is the number of seconds between those two times. (The extra 000026702881 comes from very small rounding errors that happen when you do math with floating point numbers. They only happen sometimes and are usually too tiny to matter. You can learn more about floating point rounding errors at http://invpy.com/roundingerrors.)

Going back to line 121, if time.time() - TIMEOUT > lastClickTime evaluates to True, then it has been longer than 4 seconds (the integer 4 is stored in TIMEOUT) since time.time() was called and stored in lastClickTime. If it evaluates to False, then it has been less than 4 seconds.

```
                # pushed the incorrect button, or has timed out
                gameOverAnimation()
                # reset the variables for a new game:
                pattern = []
                currentStep = 0
                waitingForInput = False
                score = 0
                pygame.time.wait(1000)
                changeBackgroundAnimation()

```

If either the player clicked on the wrong button or has timed out, the program should play the "game over" animation and then reset the variables for a new game. This involves setting the pattern list to a blank list, setting currentStep to 0, waitingForInput back to False, and then score to 0. A small pause and a new background color will be set to indicate to the player the start of a new game, which will begin on the next iteration of the game loop.

Drawing the Board to the Screen

```
        pygame.display.update()
        FPSCLOCK.tick(FPS)

```

```

```

Just like the other game programs, the last thing done in the game loop is drawing the display Surface object to the screen and calling the tick() method.

Same as Before

```
def terminate():
    pygame.quit()
    sys.exit()


def checkForQuit():
    for event in pygame.event.get(QUIT): # get all the QUIT events
        terminate() # terminate if any QUIT events are present
    for event in pygame.event.get(KEYUP): # get all the KEYUP events
        if event.key == K_ESCAPE:
            terminate() # terminate if the KEYUP event was for the Esc key
        pygame.event.post(event) # put the other KEYUP event objects back


```

The terminate() and checkForQuit() functions were explained in the Sliding Puzzle chapter, so we will skip describing them again.

Reusing The Constant Variables

```
def flashButtonAnimation(color, animationSpeed=50):
    if color == YELLOW:
        sound = BEEP1
        flashColor = BRIGHTYELLOW
        rectangle = YELLOWRECT
    elif color == BLUE:
        sound = BEEP2
        flashColor = BRIGHTBLUE
        rectangle = BLUERECT
    elif color == RED:
        sound = BEEP3
        flashColor = BRIGHTRED
        rectangle = REDRECT
    elif color == GREEN:
        sound = BEEP4
        flashColor = BRIGHTGREEN
        rectangle = GREENRECT

```

The flashButtonAnimation() has a small animation loop that draws one of the buttons lighting up which playing a sound effect. Depending on which Color value is passed as an argument for the color parameter, the sound, color of the bright flash, and rectangular area of the flash will be different. Line 151 to 166 sets three local variables differently depending on the value in the color parameter: sound, flashColor, and rectangle.

Animating the Button Flash

```
168.     origSurf = WINDOWSURF.copy()
169.     flashSurf = pygame.Surface((BUTTONSIZE, BUTTONSIZE))
170.     flashSurf = flashSurf.convert_alpha()
171.     r, g, b = flashColor
172.     sound.play()
```

The process of animating the button flash is simple: On each frame of the animation, the normal board is drawn and then on top of that, the bright color version of the button that is flashing is drawn over the button. The alpha value of the bright color starts off at 0 for the first frame of animation, but then on each frame after the alpha value is slowly increased until the bright color version completely paints over the normal button color. This will make it look like the button is slowly brightening up.

The brightening up is the first half of the animation. The second half is the button dimming. This is done with the same code, except that instead of the alpha value increasing for each frame, it will be decreasing. As the alpha value gets lower and lower, the bright color painted on top will become more and more invisible, until only the original board with the dull colors is visible.

To do this in code, line 168 creates a copy of the display Surface object and stores it in origSurf. Line 169 creates a new Surface object the size of a single button and stores it in flashSurf. The convert_alpha() method is called on flashSurf so that the Surface object can have transparent colors drawn on it (otherwise, the alpha value in the Color objects we use will be ignored and automatically assumed to be 255).

Line 171 creates individual local variables named r, g, and b to store the RGB values in flashColor. This is just some syntactic sugar that makes the rest of the code in this function easier to read. Before we begin animating the button flash, line 172 will play the sound effect for that button. The program execution keeps going after the sound effect has started to play, so the sound will be playing during the button flash animation.

```
173.     for start, end, step in ((0, 255, 1), (255, 0, -1)):
174.         for alpha in range(start, end, animationSpeed * step):
175.             checkForQuit()
176.             WINDOWSURF.blit(origSurf, (0, 0))
```

```
177.             flashSurf.fill((r, g, b, alpha))
178.             WINDOWSURF.blit(flashSurf, rectangle.topleft)
179.             pygame.display.update()
180.             FPSCLOCK.tick(FPS)
181.     WINDOWSURF.blit(origSurf, (0, 0))
```

Remember that to do the animation, we want to first draw the flashSurf with increasing alpha values from 0 to 255 to do the brightening part of the animation. Then to do the dimming, we want the alpha value to go from 255 to 0. We could do that with code like this:

```
    for alpha in range(0, 255, animationSpeed):
        checkForQuit()
        WINDOWSURF.blit(origSurf, (0, 0))
        flashSurf.fill((r, g, b, alpha))
        WINDOWSURF.blit(flashSurf, rectangle.topleft)
        pygame.display.update()
        FPSCLOCK.tick(FPS)
    for alpha in range(255, 0, -animationSpeed):
        checkForQuit()
        WINDOWSURF.blit(origSurf, (0, 0))
        flashSurf.fill((r, g, b, alpha))
        WINDOWSURF.blit(flashSurf, rectangle.topleft)
        pygame.display.update()
        FPSCLOCK.tick(FPS)
```

The code inside the for loops handles drawing the frame and are identical to each other. If we wrote the code like the above, then the first for loop would handle the brightening part of the animation (where the alpha value goes from 0 to 255) and the second for loop would handle the dimming part of the animation (where the alpha values goes from 255 to 0). Note that for the second for loop, the third argument to the range() call is a negative number.

Whenever we have identical code like this, we can probably shorten our code so we don't have to repeat it. This is what we do with the for loop on line 173, which supplies different values for the range() call on line 174:

```
173.     for start, end, step in ((0, 255, 1), (255, 0, -1)): # animation loop
174.         for alpha in range(start, end, animationSpeed * step):
```

On the first iteration of line 173's for loop, start is set to 0, end is set to 255, and step is set to 1. This way, when the for loop on line 174 is executed, it is calling range(0, 255, animationSpeed). (Note that animationSpeed * 1 is the same as animationSpeed. Multiplying a number by 1 gives us the same number.)

Line 174's for loop then executes and performs the brightening animation.

On the second iteration of line 173's for loop (there are always two and only two iterations of this for loop), start is set to 255, end is set to 0, and step is set to -1. When the line 174's for loop is executed, it is calling range(255, 0, -animationSpeed). (Note that animationSpeed * -1 returns –aniamtionSpeed, since multiplying any number by -1 returns the negative form of that same number.)

This way, we don't have to have two separate for loops and repeat all the code that is inside of them. Here's the code again that is inside line 174's for loop:

```
175.                 checkForQuit()
176.                 WINDOWSURF.blit(origSurf, (0, 0))
177.                 flashSurf.fill((r, g, b, alpha))
178.                 WINDOWSURF.blit(flashSurf, rectangle.topleft)
179.                 pygame.display.update()
180.                 FPSCLOCK.tick(FPS)
```

We check for any QUIT events (in case the user tried to close the program during the animation), then blit the origSurf Surface to the display Surface. Then we paint the flashSurf Surface by calling fill() (supplying the r, g, b values of the color we got on line 171 and the alpha value that the for loop sets in the alpha variable). Then the flashSurf Surface is blitted to the displaySurface.

Then, to make the display Surface appear on the screen, pygame.display.update() is called on line 179. To make sure the animation doesn't play as fast as the computer can draw it, we add short pauses with a call to the tick() method. (If you want to see the flashing animation play very slowly, put a low number like 1 or 2 as the argument to tick() instead of FPS.)

Drawing the Buttons

```
184. def drawButtons():
185.     pygame.draw.rect(WINDOWSURF, YELLOW, YELLOWRECT)
186.     pygame.draw.rect(WINDOWSURF, BLUE,   BLUERECT)
187.     pygame.draw.rect(WINDOWSURF, RED,    REDRECT)
188.     pygame.draw.rect(WINDOWSURF, GREEN,  GREENRECT)
189.
190.
```

Since each of the buttons is just a rectangle of a certain color in a certain place, we just make four calls to pygame.draw.rect() to draw the buttons on the display Surface. The Color object and the Rect object we use to position them are always the same, which is why we stored them in constant variables like YELLOW and YELLOWRECT.

Animating the Background Change

```
191. def changeBackgroundAnimation(animationSpeed=40):
192.     global BGCOLOR
193.     newBgColor = (random.randint(0, 255), random.randint(0, 255),
random.randint(0, 255))
194.
195.     newBgSurf = pygame.Surface((WINDOWWIDTH, WINDOWHEIGHT))
196.     newBgSurf = newBgSurf.convert_alpha()
197.     r, g, b = newBgColor
198.     for alpha in range(0, 255, animationSpeed): # animation loop
199.         checkForQuit()
200.         WINDOWSURF.fill(BGCOLOR)
201.
202.         newBgSurf.fill((r, g, b, alpha))
203.         WINDOWSURF.blit(newBgSurf, (0, 0))
204.
205.         drawButtons() # redraw the buttons on top of the tint
206.
207.         pygame.display.update()
208.         FPSCLOCK.tick(FPS)
209.     BGCOLOR = newBgColor
210.
211.
```

The background color change animation happens whenever the player finishes entering the entire pattern correctly. On each iteration through the animation loop (which is starts on line 198) the entire display Surface has to be redrawn (blended with a less and less transparent new background color, until the background is completely the new color). The steps done on each iteration of the loop are:

- Line 200 fills in the entire display Surface (stored in WINDOWSURF) with the old background color (which is stored in BGCOLOR).
- Line 202 fills in a different Surface object (stored in newBgSurf) with the new background color's RGB values (and the alpha transparency value changes on each iteration since that is what the for loop on line 198 does.)
- Line 203 then draws the newBgSurf Surface to the display Surface in WINDOWSURF. The reason we didn't just paint our semitransparent new background color on WINDOWSURF to begin with is because the fill() method will just replace the color on the Surface, whereas the blit() method will blend the colors.
- Now that we have the background the way we want it, we'll draw the buttons over it with a call to our drawButtons() function. Line 207 and 208 then just draws the display Surface to the screen and adds a pause.

The Game Over Animation

```
212. def gameOverAnimation(color=WHITE, animationSpeed=50):
213.     # play all beeps at once, then flash the background
214.     origSurf = WINDOWSURF.copy()
215.     flashSurf = pygame.Surface(WINDOWSURF.get_size())
216.     flashSurf = flashSurf.convert_alpha()
217.     BEEP1.play() # play all four beeps at the same time, roughly.
218.     BEEP2.play()
219.     BEEP3.play()
220.     BEEP4.play()
221.     r, g, b = color
```

```
222.     for i in range(3): # do the flash 3 times
```

Each of the iterations of the for loop on the next line (line 223 below) will perform a flash. To have three flashes done, we put all of that code in a for loop that has three iterations, which is exactly what line 222 does. If you want more or fewer flashes, then change the integer that is passed to range().

```
223.         for start, end, step in ((0, 255, 1), (255, 0, -1)):
```

The for loop on line 223 is exactly the same as the one line 173. The start, end, and step variables will be used on the next for loop (on line 224) to control how the alpha value changes. Reread the "Animating the Button Flash" section if you need to refresh yourself on how these loops work.

```
224.             # The first iteration in this loop sets the following for loop
225.             # to go from 0 to 255, the second from 255 to 0.
226.             for alpha in range(start, end, animationSpeed * step): #
animation loop
227.                 # alpha means transparency. 255 is opaque, 0 is invisible
228.                 checkForQuit()
229.                 flashSurf.fill((r, g, b, alpha))
230.                 WINDOWSURF.blit(origSurf, (0, 0))
231.                 WINDOWSURF.blit(flashSurf, (0, 0))
232.                 drawButtons()
233.                 pygame.display.update()
234.                 FPSCLOCK.tick(FPS)
235.
236.
237.
```

This animation loop works the same as the previous flashing animation code in the “Animating the Background Change” section. The copy of the original Surface stored in origSurf is drawn on the display Surface, then flashSurf (which has the new flashing color painted on it) is blitted on top of the display Surface. After the background color is set up, the buttons are drawn on top on line 232, and then the display Surface is drawn to the screen.

The for loop on line 226 adjusts the alpha value for each frame of animation (increasing at first, and then decreasing).

Converting from Pixel Coordinates to Buttons

```
238. def getButtonClicked(x, y):
239.     if YELLOWRECT.collidepoint( (x, y) ):
240.         return YELLOW
241.     elif BLUERECT.collidepoint( (x, y) ):
242.         return BLUE
243.     elif REDRECT.collidepoint( (x, y) ):
244.         return RED
245.     elif GREENRECT.collidepoint( (x, y) ):
246.         return GREEN
247.     return None
248.
249.
250. if __name__ == '__main__':
251.     main()
```

The getButtonClicked() function simply takes XY pixel coordinates and returns either the values YELLOW, BLUE, RED, or GREEN if one of the buttons was clicked, or returns None if the XY pixel coordinates are not over any of the four buttons.

Explicit is Better Than Implicit

You may have noticed that the code for getButtonClicked() ends with a return None statement on line 247. This might seem like an odd thing to type out, since all functions return None if they don’t have any return statement at all. We could have left line 47 out entirely and the program would have worked the exact same way. So why bother writing it in?

Normally when a function reaches the end and returns the None value implicitly (that is, there is no return statement outright saying that it is returning None) the code that calls it doesn’t care about the return value. All function calls have to return a value (so that they can evaluate to something and be part of expressions), but we don’t always care about their return value.

For example, think about the print() function. Technically, this function returns the None value, but we never care about it:

```
>>> spam = print('Hello')
Hello
>>> spam == None
True
>>>
```

However, when getButtonClicked() returns None, it means that the coordinates that were passed to it were not over any of the four buttons. To make it clear that in this case the value None is returned from getButtonClicked(), we have the return None line at the end of the function.

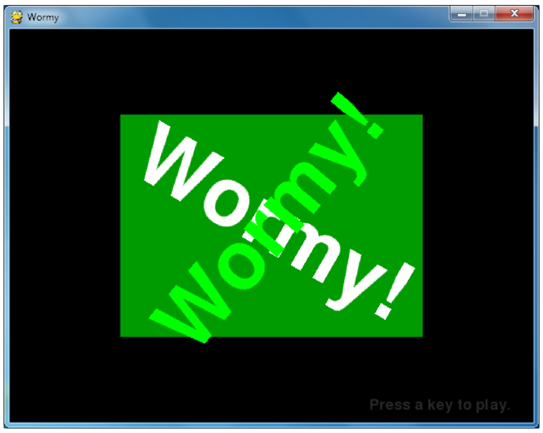
To make your code more readable, it is better to have your code be explicit (that is, clearly state something even if it might be obvious) rather than implicit (that is, leaving it up to the person reading code to know how it works without outright telling them). In fact, “explicit is better than implicit” is one of the Python Koens.

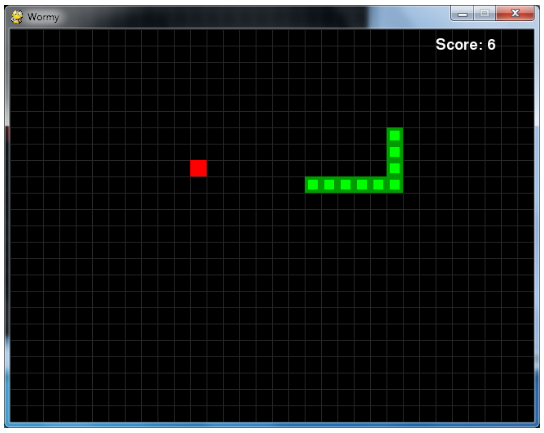
The koens are a group of little sayings about how to write readable code. There’s an easter egg in Python (that is, a little hidden surprise) where if you try to import a feature named this, then it will display “The Zen of Python” koens. Try it out in the interactive shell:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

If you’d like to know more about what these individual koens mean, visit http://invpy.com/zen.

Chapter 6 – Wormy





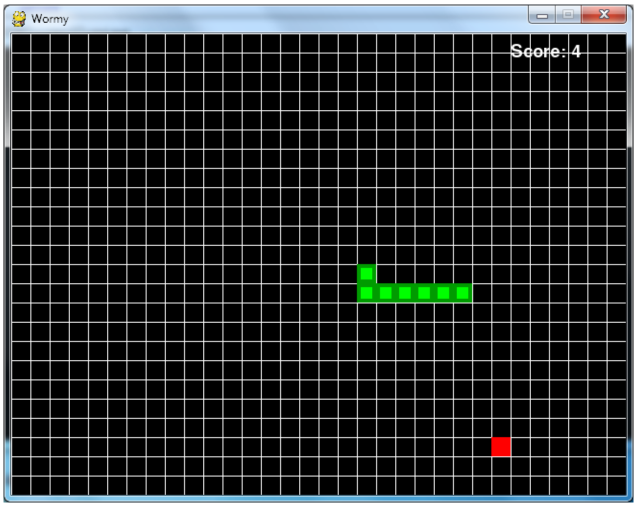
How to Play Wormy

Wormy is a Nibbles clone. The player starts out controlling a short worm that is constantly moving around the screen. The player cannot stop or slow down the worm, but she can control which direction it turns. A red apple appears randomly on the screen, and the player must move the worm so that it eats the apple. Each time the worm eats an apple, the worm grows longer by one segment and a new apply randomly appears on the screen. The game is over if the worm crashes into itself or the edges of the screen.

Source Code to Wormy

This source code can be downloaded from http://invpy.com/wormy.py.

The Grid



If you play the game a little, you'll notice that the apple and the segments of the worm's body always fit along a grid of lines. In the above screenshot, the grid is drawn with white lines (instead of the dark grey lines) so that we can see it better.

We will call each of the squares in this grid a **cell** (it's not always what a space in a grid is called, it's just a name I came up with.) The cells have their own Cartesian coordinate system, with (0, 0) being the top left cell and (31, 23) being the bottom right cell.

The Setup Code

```
# Wormy (a Nibbles clone)
# By Al Sweigart al@inventwithpython.com
# http://inventwithpython.com/pygame
# Creative Commons BY-NC-SA 3.0 US

import random, pygame, sys
from pygame.locals import *

FPS = 15
WINDOWWIDTH = 640
WINDOWHEIGHT = 480
CELLSIZE = 20
assert WINDOWWIDTH % CELLSIZE == 0, "Window width must be a multiple of cell size."
assert WINDOWHEIGHT % CELLSIZE == 0, "Window height must be a multiple of cell size."
```

```
15. CELLWIDTH = int(WINDOWWIDTH / CELLSIZE)
16. CELLHEIGHT = int(WINDOWHEIGHT / CELLSIZE)
17.
```

The code at the start of the program just sets up some constant variables used in the game. The width and height of the cells are stored in CELLSIZE. The assert statements on lines 13 and 14 ensure that the cells fit perfectly in the window. For example, if the CELLSIZE was 10 and the WINDOWWIDTH or WINDOWHEIGHT constants were set to 15, then only 1.5 cells could fit. The assert statements make sure that only an integer number of cells fits in the window.

```
18. #             R    G    B
19. WHITE     = (255, 255, 255)
20. BLACK     = (  0,   0,   0)
21. RED       = (255,   0,   0)
22. GREEN     = (  0, 255,   0)
23. DARKGREEN = (  0, 155,   0)
24. DARKGRAY  = ( 40,  40,  40)
25. BGCOLOR = BLACK
26.
27. UP = 'up'
28. DOWN = 'down'
29. LEFT = 'left'
30. RIGHT = 'right'
31.
32. HEAD = 0 # syntactic sugar: index of the worm's head
33.
```

Some more constants are set. The HEAD constant will be explained later in this chapter.

The main() Function

```
34. def main():
35.     global FPSCLOCK, WINDOWSURF, BASICFONT
36.
37.     pygame.init()
38.     FPSCLOCK = pygame.time.Clock()
39.     WINDOWSURF = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
40.     BASICFONT = pygame.font.Font('freesansbold.ttf', 18)
41.     pygame.display.set_caption('Wormy')
42.
43.     showStartScreen()
44.     while True:
45.         runGame()
46.         showGameOverScreen()
47.
```

```
48.
```

In the Wormy game program, we’ve put the main part of the code in a function called runGame(). This is because we only want to show the “start screen” (the animation with the rotating “Wormy” text) once when the program starts (by calling the showStartScreen() function). Then we want to call runGame(), which will start a game of Wormy. This function will return when the player’s worm collides into a wall or into itself.

Then we will show the game over screen by calling showGameOverScreen(). When that function call returns, the loop goes back and calls runGame() again.

A Separate runGame() Function

```
49. def runGame():
50.     # Set a random start point.
51.     startx = random.randint(5, CELLWIDTH - 6)
52.     starty = random.randint(5, CELLHEIGHT - 6)
53.     wormCoords = [{'x': startx,     'y': starty},
54.                   {'x': startx - 1, 'y': starty},
55.                   {'x': startx - 2, 'y': starty}]
56.     direction = RIGHT
57.
58.     # Start the apple in a random place.
59.     apple = getRandomLocation()
60.
```

At the beginning of a game, we want the worm to start in a random position (but not too close to the edges of the board) so we store a random coordinate in startx and starty. (Remember that CELLWIDTH and CELLHEIGHT is the number of cells wide and high the window is, not the number of pixels wide and high).

The body of the worm will be stored in a list of dictionary values. There will be one dictionary value per body segment of the worm. The dictionary will have keys 'x' and 'y' for the XY coordinates of that body segment. Since the worm starts off three segments big and going to the right (which is set on line 56), we want the head of the body to be at startx and starty. The other two body segments will be one and two cells to the left of the head.

The head of the worm will always be the body part at wormCoords[0]. To make this code more readable, we’ve set the HEAD constant to 0 on line 32, so that we can use wormCoords[HEAD] instead.

The Event Handling Loop

```
61.     while True: # main game loop
62.         for event in pygame.event.get(): # event handling loop
63.             if event.type == QUIT:
64.                 terminate()
65.             elif event.type == KEYDOWN:
66.                 if (event.key == K_LEFT or event.key == K_a) and direction
!= RIGHT:

67.                     direction = LEFT
68.                 elif (event.key == K_RIGHT or event.key == K_d) and
direction != LEFT:

69.                     direction = RIGHT
70.                 elif (event.key == K_UP or event.key == K_w) and direction
!= DOWN:

71.                     direction = UP
72.                 elif (event.key == K_DOWN or event.key == K_s) and
direction != UP:

73.                     direction = DOWN
74.                 elif event.key == K_ESCAPE:
75.                     terminate()
76.
```

Line 61 is the start of the main game loop and line 62 is the start of the event handling loop. If the event is a QUIT event, then we call terminate() (which we've defined the same as the terminate() function in the previous game programs.)

Otherwis, if the event is a KEYDOWN event, then we check if the key that was pressed down is an arrow key or WASD key. We want an additional check so that the worm does not turn in on itself. If the worm is moving left, then if the player accidentally presses the right key, the worm will immediate start going right and crash into itself.

That is why we have this check for the current value of the direction variable. That way, if the player accidentally presses an arrow key that would cause them to immediately crash the worm, we just ignore that key press.

Collision Detection

```
77.         # check if the worm has hit itself or the edge
```

```
78.         if wormCoords[HEAD]['x'] == -1 or wormCoords[HEAD]['x'] ==
CELLWIDTH or wormCoords[HEAD]['y'] == -1 or wormCoords[HEAD]['y'] ==
CELLHEIGHT:

79.             return
80.         for wormBody in wormCoords[1:]:
81.             if wormBody['x'] == wormCoords[HEAD]['x'] and wormBody['y'] ==
wormCoords[HEAD]['y']:

82.                 return
83.
```

The worms has crashed when the head has moved off the edge of the grid or if the head moves onto a cell that is already occupied by another body segment.

We can check if the head has moved off the edge of the grid by seeing if either the X coordinate of the head (which is stored in wormCoords[HEAD]['x']) is -1 (which is past the left edge of the grid) or equal to CELLWIDTH (which is past the right edge, since the rightmost Y cell coordinate is one less than CELLWIDTH.)

The head has also moved off the grid if the Y coordinate of the head (which is stored in wormCoords[HEAD]['y']) is either -1 (which is past the top edge) or CELLHEIGHT (which is past the bottom edge.)

All we have to do to end the current game is to return out of runGame(). When runGame() returns to the function call in main(), the next line after the runGame() call (line 46) is the call to showGameOverScreen() which makes the large “Game Over” text appear. This is why we have the return statement on line 79.

Line 80 loops through every body segment in wormCoords after the head (which is at index 0. This is why the for loop iterates over wormCoords[1:] instead of just wormCoords). If both the 'x' and 'y' values of the body segment are the same as the 'x' and 'y' of the head, then we also end the game by returning out of the runGame() function.

Collision Detection with the Apple

```
84.         # check if worm has eaten an apply
85.         if wormCoords[HEAD]['x'] == apple['x'] and wormCoords[HEAD]['y'] ==
apple['y']:

86.             # don't remove worm's tail segment
87.             apple = getRandomLocation() # set a new apple somewhere
88.         else:
89.             del wormCoords[-1] # remove worm's tail segment
```

```
90.
```

We do a similar collision detection check between the head of the worm and the apple's XY coordinates. If they match, we set the coordinates of the apple to a random new location (which we get from the return value of getRandomLocation()).

If the head has not collided with an apple, then we delete the last body segment in the wormCoords list. Remember that negative integers for indexes count from the end of the list. So while 0 is the first item in the list and 1 is the second item, -1 is the last item in the list and -2 is the second to last item.

The code on lines 91 to 100 (described next in the "Moving the Worm" section) will add a new body segment (for the head) in the direction that the worm is going. This will make the worm one segment longer. By not deleting the last body segment when the worm eats an apple, the overall length of the worm increases by one. But when line 89 deletes the last body segment, the size remains the same because a new head segment is added right afterwards.

Moving the Worm

```
 91.         # move the worm by adding a segment in the direction it is moving
 92.         if direction == UP:
 93.             newHead = {'x': wormCoords[HEAD]['x'], 'y':
wormCoords[HEAD]['y'] - 1}

 94.         elif direction == DOWN:
 95.             newHead = {'x': wormCoords[HEAD]['x'], 'y':
wormCoords[HEAD]['y'] + 1}

 96.         elif direction == LEFT:
 97.             newHead = {'x': wormCoords[HEAD]['x'] - 1, 'y':
wormCoords[HEAD]['y']}

 98.         elif direction == RIGHT:
 99.             newHead = {'x': wormCoords[HEAD]['x'] + 1, 'y':
wormCoords[HEAD]['y']}

100.         wormCoords.insert(0, newHead)
```

To move the worm, we add a new body segment to the beginning of the wormCoords list. Because the body segment is being added to the beginning of the list, it will become the new head. The coordinates of the new head will be right next to the old head's coordinates. Whether 1 is added or subtracted from either the X or Y coordinate depends on the direction the worm is going.

This new head segment is added to wormCoords with the insert() list method on line 100.

The insert() List Method

The insert() list method can add items anywhere inside the list (while the append() list method can only add items to the end of the list.) The first parameter for insert() is the index where the item should go (all the items after this index have their indexes increase by one). If the argument passed for the first parameter is larger than the length of the list, the item is simply added to the end of the list (just like what append() does). The second parameter for insert() is the item value itself. Type the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(0, 'frog')
>>> spam
['frog', 'cat', 'dog', 'bat']
>>> spam.insert(10, 'dingo')
>>> spam
['frog', 'cat', 'dog', 'bat', 'dingo']
>>> spam.insert(2, 'horse')
>>> spam
['frog', 'cat', 'horse', 'dog', 'bat', 'dingo']
>>>
```

Drawing the Screen

```
101.         WINDOWSURF.fill(BGCOLOR)
102.         #drawGrid() # DEBUG
103.         drawWorm(wormCoords)
104.         drawApple(apple)
105.         drawScore(len(wormCoords) - 3)
106.         pygame.display.update()
107.         FPSCLOCK.tick(FPS)
108.
```

The code for drawing the screen in the runGame() function is fairly simple. Line 101 fills in the entire display Surface with the background color. Lines 103 to 105 draw the worm, apple, and score to the display Surface. Then the call to pygame.display.update() draws the display Surface to the actual computer screen.

Notice that line 102 is commented out. The drawGrid() function will add white lines to make the grid visible. This may be useful for debugging and understanding the programming, but we don't normally need it in the game, so the call to drawGrid() is commented out and ignored by the Python interpreter.

Drawing “Press a key” Text to the Screen

```
109. def drawPressKeyMsg():
110.     pressKeySurf = BASICFONT.render('Press a key to play.', True,
DARKGRAY)

111.     pressKeyRect = pressKeySurf.get_rect()
112.     pressKeyRect.topleft = (WINDOWWIDTH - 200, WINDOWHEIGHT - 30)
113.     WINDOWSURF.blit(pressKeySurf, pressKeyRect)
114.
115.
```

While the start screen animation is playing or the game over screen is being shown, there will be some small text in the bottom right corner that says “Press a key to play.” Rather than have the code typed out in both the showStartScreen() and the showGameOverScreen(), we put it in a this separate function and simply call the function from showStartScreen() and showGameOverScreen().

The checkForKeyPress() Function

```
116. def checkForKeyPress():
117.     if len(pygame.event.get(QUIT)) > 0:
118.         terminate()
119.
120.     keyUpEvents = pygame.event.get(KEYUP)
121.     if len(keyUpEvents) == 0:
122.         return None
123.     if keyUpEvents[0].key == K_ESCAPE:
124.         terminate()
125.     return keyUpEvents[0].key
126.
127.
```

This function first checks if there are any QUIT events in the event queue. The call to pygame.event.get() on line 117 returns a list of all the QUIT events in the event queue (because we pass QUIT as an argument.) If there are not QUIT events in the event queue, then the list that pygame.event.get() returns will be the empty list: []

The len() call on line 117 will return 0 if pygame.event.get() returned an empty list. If there are more than zero items in list returned by pygame.event.get() (and remember, any items in this list will only be QUIT events because we passed QUIT as the argument to pygame.event.get()), then the terminate() function gets called and the game program terminates.

After that, the call to pygame.event.get() gets a list of any KEYUP events in the event queue. If the key event is for the Esc key, then the program terminates in that case as well. Otherwise, the first key event object in the list that was returned by pygame.event.get() is returned from this checkForKeyPress() function.

Start Screens

```
128. def showStartScreen():
129.     titleFont = pygame.font.Font('freesansbold.ttf', 100)
130.     titleSurf1 = titleFont.render('Wormy!', True, WHITE, DARKGREEN)
131.     titleSurf2 = titleFont.render('Wormy!', True, GREEN)
132.
133.     degrees1 = 0
134.     degrees2 = 0
135.     while True:
136.         WINDOWSURF.fill(BGCOLOR)
```

When the Wormy game program first begins running, the player doesn't automatically begin playing. Instead, a start screen appears which tells the player what program they are running. A start screen also gives the player a chance to prepare for the game to begin (otherwise the player might not be ready and crash on their first game.)

The Wormy start screen requires two Surface objects with the "Wormy!" text drawn on them. These are what the render() method calls create on lines 130 and 131. The text will be large: the Font() constructor function call on line 129 creates a Font object that is 100 points in size. The first "Wormy!" text will have white text with a darkgreen background, and the other will have green text with a transparent background.

Line 135 begins the animation loop for the start screen. During this animation, the two pieces of text will be rotated and drawn to the display Surface object.

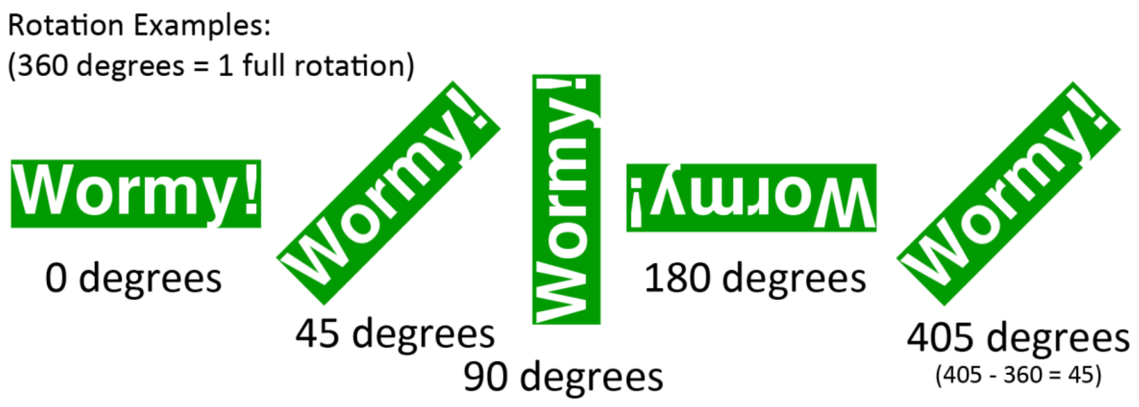
Rotating the Start Screen Text

```
137.         rotatedSurf1 = pygame.transform.rotate(titleSurf1, degrees1)
138.         rotatedRect1 = rotatedSurf1.get_rect()
139.         rotatedRect1.center = (WINDOWWIDTH / 2, WINDOWHEIGHT / 2)
140.         WINDOWSURF.blit(rotatedSurf1, rotatedRect1)
141.
142.         rotatedSurf2 = pygame.transform.rotate(titleSurf2, degrees2)
143.         rotatedRect2 = rotatedSurf2.get_rect()
144.         rotatedRect2.center = (WINDOWWIDTH / 2, WINDOWHEIGHT / 2)
145.         WINDOWSURF.blit(rotatedSurf2, rotatedRect2)
146.
147.         drawPressKeyMsg()
```

```

            if checkForKeyPress():
                pygame.event.get() # clear event queue
                return
            pygame.display.update()
            FPSCLOCK.tick(FPS)
```

To rotate the Surface objects that the “Wormy!” text is written on, we call the pygame.transform.rotate() function. The first parameter is the Surface object to make a rotated copy of. The second argument is the number of degrees to rotate the Surface. The pygame.transform.rotate() function doesn’t change the Surface object you pass it, but rather returns a new Surface object with the rotated image drawn on it.

The amount you rotate it is given in degrees, which is a measure of rotation. There are 360 degrees in a circle. Not being rotated at all is 0 degrees. Rotating to one quarter counter-clockwise is 90 degrees. Rotating 360 degrees is rotating the image all the way around, which means you end up with the same image as if you rotated it 0 degrees. In fact, if the rotation argument you pass to pygame.transform.rotate() is 360 or larger, then Pygame automatically keeps subtracting 360 from it until it gets a number less than 360. This image shows several examples of different rotation amounts:



The drawPressKeyMsg() function call draws the “Press a key to play.” text in the lower corner of the display Surface object. This animation loop will keep looping until checkForKeyPress() returns a value that is not None, which happens if the player presses a key. Before returning, pygame.event.get() is called simply to clear out any other events that have accumulated in the event queue which the start screen was displayed.

Rotations Are Not Perfect

You may wonder why we store the rotated Surface in a separate variable, rather than just overwrite the titleSurf1 and titleSurf2 variables. There are two reasons.

First, rotating a 2D image is never completely perfect. The rotated image is always approximate. If you rotate an image by 10 degrees counterclockwise, and then rotate it back 10 degrees clockwise, the image you have will not be the image you started with. Think of it as making a photocopy, and then a photocopy of the first photocopy, and the another photocopy of that photocopy. If you keep doing this, the image gets worse and worse as the slight distortions add up.

(The only exception to this is if you rotate an image by a multiple of 90 degrees. In that case, the pixels can be rotated without any distortion.)

Second, if you rotate a 2D image then the rotated image will be slightly larger than the original image. If you rotate that rotated image, then the next rotated image will be slightly larger again. If you keep doing this, eventually the image will become too large for Pygame to handle, and your program will crash with the error message, `pygame.error: Width or height is too large`.

These two reasons are why we rotate an original image to get all the different rotated Surface objects. Otherwise, constantly rotating images that have already been rotated will make the image look worse and worse, and increase their size until the program crashes.

```
154.         degrees1 += 3 # rotate by 3 degrees each frame
155.         degrees2 += 7 # rotate by 7 degrees each frame
156.
157.
```

The amount that we rotate the two “Wormy!” text Surface objects is stored in degrees1 and degrees2. On each iteration through the animation loop, we increase the number stored in degrees1 by 3 and degrees2 by 7. This means on the next iteration of the animation loop the white text “Wormy!” Surface object will be rotated by another 3 degrees and the green text “Wormy!” Surface object will be rotated by another 7 degrees. This is why the one of the Surface objects rotates slower than the other.

```
158. def terminate():
159.     pygame.quit()
160.     sys.exit()
161.
162.
```

The terminate() function calls pygame.quit() and sys.exit() so that the game correctly shuts down. This is just like the terminate() functions in the previous game programs.

Deciding Where the Apple Appears

```
def getRandomLocation():
    return {'x': random.randint(0, CELLWIDTH - 1), 'y': random.randint(0, CELLHEIGHT - 1)}


```

The getRandomLocation() function is called whenever the worm eats the apple and it needs to show up in a new cell on the grid. This function simple returns a dictionary with keys 'x' and 'y'. The values of those keys is a random integer between 0 and the number of cells wide or high the board is.

Game Over Screens

```
def showGameOverScreen():
    gameOverFont = pygame.font.Font('freesansbold.ttf', 150)
    gameSurf = gameOverFont.render('Game', True, WHITE)
    overSurf = gameOverFont.render('Over', True, WHITE)
    gameRect = gameSurf.get_rect()
    overRect = overSurf.get_rect()
    gameRect.midtop = (WINDOWWIDTH / 2, 10)
    overRect.midtop = (WINDOWWIDTH / 2, gameRect.height + 10 + 25)

    WINDOWSURF.blit(gameSurf, gameRect)
    WINDOWSURF.blit(overSurf, overRect)
    drawPressKeyMsg()
    pygame.display.update()
```

The Game Over screen is similar to the start screen, except it isn’t animated. The words “Game” and “Over” are drawn to two Surface objects which are then drawn on the screen.

```
    pygame.time.wait(500)
    checkForKeyPress() # clear out any key presses in the event queue

    while True:
        if checkForKeyPress():
            pygame.event.get() # clear event queue
            return

```

The Game Over text will stay on the screen until the player pushes a key. Just to make sure the player doesn’t accidentally press a key too soon, we will put a half second pause with the call to pygame.time.wait() on line 180. (The 500 argument stands for a 500 millisecond pause, which is half of one second.)

Then, checkForKeyPress() is called so that any key events that were made since the showGameOverScreen() function started are ignored. This pause and dropping of the key events is to prevent the following situation: Say the player was trying to turn away from the edge of the screen at the last minute, but pressed the key too late and crashed into the wall. If this happens, then the key press would have happened after the showGameOverScreen() was called, and that key press would cause the Game Over screen to disappear almost instantly. The next game would start immediately after that, and might take the player by surprise. Adding this pause helps the make the game more “user friendly”.

Drawing Functions

The code to draw the score, worm, apple, and grid are all put into separate functions. They will be called from the runGame() function to display the game state on the screen. Note that these functions only draw on the display Surface object. None of them makes a call to pygame.display.update(). (That must be done by the code that calls these drawing functions.)

```
def drawScore(score):
    scoreSurf = BASICFONT.render('Score: %s' % (score), True, WHITE)
    scoreRect = scoreSurf.get_rect()
    scoreRect.topleft = (WINDOWWIDTH - 120, 10)
    WINDOWSURF.blit(scoreSurf, scoreRect)


```

The drawScore() function simply renders and draws the text of the score that was passed in its score parameter on the display Surface object.

```
def drawWorm(wormCoords):
    for coord in wormCoords:
        x = coord['x'] * CELLSIZE
        y = coord['y'] * CELLSIZE
        wormSegmentRect = pygame.Rect(x, y, CELLSIZE, CELLSIZE)
        pygame.draw.rect(WINDOWSURF, DARKGREEN, wormSegmentRect)
        wormInnerSegmentRect = pygame.Rect(x + 4, y + 4, CELLSIZE - 8, CELLSIZE - 8)
        pygame.draw.rect(WINDOWSURF, GREEN, wormInnerSegmentRect)

```

```
204.
```

The drawWorm() function will draw a green box for each of the segments of the worm’s body. The segments are passed in the wormCoords parameter, which is a list of dictionaries with an 'x' key and a 'y' key. The for loop on line 196 loops through each of the dictionary values in wormCoords.

Because the grid coordinates take up the entire window and also begin a 0, 0 pixel, it is fairly easy to convert from grid coordinates to pixel coordinates. Line 197 and 198 simply multiply the coord['x'] and coord['y'] coordinate by the CELLSIZE.

Line 199 creates a Rect object for the worm segment that will be passed to the pygame.draw.rect() function on line 200. Remember that each cell in the grid is CELLSIZE in width and height, so that’s what the size of the segment’s Rect object should be. Line 200 draws a dark green rectangle for the segment. Then on top of this, a smaller bright green rectangle is drawn. (This makes the worm look a little nicer.)

The inner bright green rectangle starts 4 pixels to the right and 4 pixels below the topleft corner of the cell. The width and height of this rectangle are 8 pixels less than the cell size, so there will be a 4 pixel margin on the right and bottom sides as well.

```
205. def drawApple(coord):
206.     x = coord['x'] * CELLSIZE
207.     y = coord['y'] * CELLSIZE
208.     appleRect = pygame.Rect(x, y, CELLSIZE, CELLSIZE)
209.     pygame.draw.rect(WINDOWSURF, RED, appleRect)
210.
211.
```

The drawApple() function is very similar to drawWorm, except since the red apple is just a single rectangle that fills up the cell, all the function needs to do is convert to pixel coordinates (which is what lines 206 and 207 do), create the Rect object with the location and size of the apple (line 208), and then pass this Rect object to the pygame.draw.rect() function.

```
212. def drawGrid(): # A debug function
213.     for x in range(0, WINDOWWIDTH, CELLSIZE): # draw vertical lines
214.         pygame.draw.line(WINDOWSURF, WHITE, (x, 0), (x, WINDOWHEIGHT))
215.     for y in range(0, WINDOWHEIGHT, CELLSIZE): # draw horizontal lines
216.         pygame.draw.line(WINDOWSURF, WHITE, (0, y), (WINDOWWIDTH, y))
217.
218.
```

Normally our code won’t call the drawGrid() function, because it makes the game look ugly to have all these lines running across it. But just to make it easier to visualize the grid, we call pygame.draw.line() for each of the vertical and horizontal lines.

Normally, to draw the 32 vertical lines needed, we would need 32 calls to pygame.draw.line() with the following coordinates:

```
pygame.draw.line(WINDOWSURF, WHITE, (0, 0), (0, WINDOWHEIGHT))
pygame.draw.line(WINDOWSURF, WHITE, (20, 0), (20, WINDOWHEIGHT))
pygame.draw.line(WINDOWSURF, WHITE, (40, 0), (40, WINDOWHEIGHT))
pygame.draw.line(WINDOWSURF, WHITE, (60, 0), (60, WINDOWHEIGHT))
...skipped for brevity...
pygame.draw.line(WINDOWSURF, WHITE, (560, 0), (560, WINDOWHEIGHT))
pygame.draw.line(WINDOWSURF, WHITE, (580, 0), (580, WINDOWHEIGHT))
pygame.draw.line(WINDOWSURF, WHITE, (600, 0), (600, WINDOWHEIGHT))
pygame.draw.line(WINDOWSURF, WHITE, (620, 0), (620, WINDOWHEIGHT))
```

Instead of typing out all these lines of code, we can just have one line of code inside a for loop. Notice that the pattern for the vertical lines is that the X coordinate of the start and end point starts at 0 and goes up to 620, increasing by 20 each time. (The Y coordinate is always 0 for the start point and WINDOWHEIGHT for the end point parameter.) That means the for loop should iterate over range(0, 640, 20). It is no coincidence that the CELLSIZE is 20 and the WINDOWWIDTH is 640. This is why the for loop on line 213 iterates over range(0, WINDOWWIDTH, CELLSIZE).

For the horizontal lines, the coordinates would have to be:

```
pygame.draw.line(WINDOWSURF, WHITE, (0, 0), (WINDOWWIDTH, 0))
pygame.draw.line(WINDOWSURF, WHITE, (0, 20), (WINDOWWIDTH, 20))
pygame.draw.line(WINDOWSURF, WHITE, (0, 40), (WINDOWWIDTH, 40))
pygame.draw.line(WINDOWSURF, WHITE, (0, 60), (WINDOWWIDTH, 60))
...skipped for brevity...
pygame.draw.line(WINDOWSURF, WHITE, (0, 400), (WINDOWWIDTH, 400))
pygame.draw.line(WINDOWSURF, WHITE, (0, 420), (WINDOWWIDTH, 420))
pygame.draw.line(WINDOWSURF, WHITE, (0, 440), (WINDOWWIDTH, 440))
pygame.draw.line(WINDOWSURF, WHITE, (0, 460), (WINDOWWIDTH, 460))
```

The Y coordinate ranges from 0 to 460, increasing by 20 each time. (The X coordinate is always 0 for the start point and WINDOWWIDTH for the end point parameter.) We can also use a for loop here so we don’t have to type out all those pygame.draw.line() calls.

Noticing regular patterns needed by the calls and using loops is a clever programmer trick to save us from a lot of typing. We could have typed out all 56 pygame.draw.line() calls and the program would have worked the exact same. But by being a little bit clever, we can save ourselves a lot of work.

```
219. if __name__ == '__main__':
220.     main()
```

After all the functions and constants and global variables have been defined and created, the main() function is called to start the game.

Don't Reuse Variable Names

Take a look at a few lines of code from the drawWorm() function again:

```
199.         wormSegmentRect = pygame.Rect(x, y, CELLSIZE, CELLSIZE)
200.         pygame.draw.rect(WINDOWSURF, DARKGREEN, wormSegmentRect)
201.         wormInnerSegmentRect = pygame.Rect(x + 4, y + 4, CELLSIZE - 8,
CELLSIZE - 8)

202.         pygame.draw.rect(WINDOWSURF, GREEN, wormInnerSegmentRect)
```

Notice that two different Rect objects are created on lines 199 and 201. The Rect object created on line 199 is stored in the wormSegmentRect local variable and is passed to the pygame.draw.rect() function on line 200. The Rect object created on line 201 is stored in the wormInnerSegmentRect local variable and is passed to the pygame.draw.rect() function on line 202.

Every time you create a variable, it takes up a small amount of the computer's memory. You might think it would be clever to reuse the wormSegmentRect variable for both Rect objects, like this:

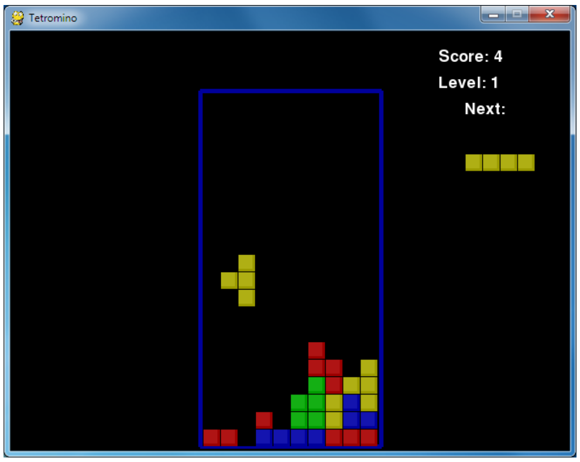
```
199.         wormSegmentRect = pygame.Rect(x, y, CELLSIZE, CELLSIZE)
200.         pygame.draw.rect(WINDOWSURF, DARKGREEN, wormSegmentRect)
201.         wormSegmentRect = pygame.Rect(x + 4, y + 4, CELLSIZE - 8, CELLSIZE
- 8)

202.         pygame.draw.rect(WINDOWSURF, GREEN, wormInnerSegmentRect)
```

Because the Rect object returned by pygame.Rect() on line 199 won't be needed after 200, we can overwrite this value and reuse the variable to store the Rect object returned by pygame.Rect() on line 201. Since we are now using fewer variables we are saving memory, right?

While this is technically true, you really are only saving a few bytes. Modern computers have memory of several billion bytes. So the savings aren't that great. Meanwhile, reusing variables reduces the code readability. If a programmer was reading through this code after it was written, they would see that wormSegmentRect is passed to the pygame.draw.rect() calls on line 200 and 202. If they tried to find the first time the wormSegmentRect variable was assigned a value, they would see the pygame.Rect() call on line 199. They might not realize that the Rect object returned by line 199's pygame.Rect() call isn't the same as the one that is passed to the pygame.draw.rect() call on line 202.

Little things like this make it harder to understand how exactly your program works. It won't just be other programmers looking at your code who will be confused. When you look at your own code a couple weeks after writing it, you may have a hard time remembering how exactly it works. Code readability is much more important than saving a few bytes of memory here and there.

Chapter 7 - Tetromino



How to Play Tetromino

Tetromino is a Tetris clone. Differently shaped blocks fall from the top of the screen, and the player must guide them down to form complete rows that have no gaps in them. When a complete row is formed, the row disappears and each row above it moves down one row. The game keeps going until the screen fills up and a new falling block cannot fit on the screen.

Some Nomenclature

In this chapter, I have come up with a set of terms for the different things in the game program.

- Board – The board is the 10x20 space that the blocks fall and stack up in.
- Box – A box is a single square space on the board.
- Piece – The things that fall from the top of the board that the player can rotate and position. Each piece has a shape and is made up of 4 boxes.
- Shape – The shapes are the different types of pieces in the game. The names of the shapes are T, S, Z, J, I, and O.
- Template – A list of shape data structures that represents all the possible rotations of a shape. These are store in variables with names like S_SHAPE_TEMPLATE or J_SHAPE_TEMPLATE.
- Landed – When a piece has either reached the bottom of the board or is touching a box on the board, we say that the piece has landed. At that point, the next piece should start falling.

Source Code to Tetromino

This source code can be downloaded from http://invpy.com/tetromino.py.

The Usual Setup Code

```
# Tetromino (a Tetris clone)
# By Al Sweigart al@inventwithpython.com
# http://inventwithpython.com/pygame
# Creative Commons BY-NC-SA 3.0 US

import random, time, pygame, sys
from pygame.locals import *

FPS = 25
WINDOWWIDTH = 640
WINDOWHEIGHT = 480
BOXSIZE = 20
BOARDWIDTH = 10
BOARDHEIGHT = 20
BLANK = '.'

```

These are the constants used by our Tetromino game. The board itself is 10 boxes wide and 20 boxes tall. Each box is a square that is 20 pixels wide and high. The BLANK constant will be used as a value to represent blank spaces on the board.

Setting up Timing Constants

```
MOVESIDEWAYSFREQ = 0.15
MOVEDOWNFREQ = 0.15

```

Every time the player pushes the left or right key, the falling piece should move one box over to the left or right. However, the player can also hold down the left or right key. The MOVESIDEWAYSFREQ constant will set it so that every 0.15 seconds that passes with the left or right key held down, the piece will move one box over.

The MOVEDOWNFREQ constant is the same thing except it tells how frequently the piece drops by one box while the player has the down key held down.

More Setup Code

```
XMARGIN = int((WINDOWWIDTH - BOARDWIDTH * BOXSIZE) / 2)
TOPMARGIN = WINDOWHEIGHT - (BOARDHEIGHT * BOXSIZE) - 5
```

The program needs to calculate how many pixels are to the left and right side of the board to the left and right edge of the window. WINDOWWIDTH is the total number of pixels wide the entire window is. If we subtract BOXSIZE pixels from this for each of the boxes wide in the board (which is BOARDWIDTH * BOXSIZE), we’ll have the size of the margin to the left and right of the board. If we divide this by 2, then we will have the size of just one margin. (Since the margins are the same size, we can use XMARGIN for either the left or right margin.)

We can calculate the size of the space between the top of the board and the top of the window in a similar manner. 5 is subtracted in this calculation because we don’t want the bottom of the board to be at the very bottom of the window, but 5 pixels above the bottom.

```
#               R    G    B
WHITE       = (255, 255, 255)
GRAY        = (185, 185, 185)
BLACK       = (  0,   0,   0)
RED         = (155,   0,   0)
LIGHTRED    = (175,  20,  20)
GREEN       = (  0, 155,   0)
LIGHTGREEN  = ( 20, 175,  20)
BLUE        = (  0,   0, 155)
LIGHTBLUE   = ( 20,  20, 175)
YELLOW      = (155, 155,   0)
LIGHTYELLOW = (175, 175,  20)

BORDERCOLOR = BLUE
BGCOLOR = BLACK
TEXTCOLOR = WHITE
TEXTSHADOWCOLOR = GRAY
COLORS      = (     BLUE,      GREEN,      RED,      YELLOW)
LIGHTCOLORS = (LIGHTBLUE, LIGHTGREEN, LIGHTRED, LIGHTYELLOW)
assert len(COLORS) == len(LIGHTCOLORS) # each color must have light color

```

The pieces will come in four colors: blue, green, red, and yellow. When we draw the boxes though, there will be a highlight on the box in a lighter color. So this means we need to create a light blue, light green, light red, and light yellow color as well.

Each of these four colors will be stored in a tuple named COLORS (for the normal colors) and LIGHTCOLORS (for the lighter colors). The white, black, and gray colors are used for other things in the Tetromino game program.

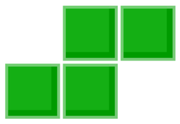
Setting Up the Piece Templates

```
TEMPLATEWIDTH = 5
TEMPLATEHEIGHT = 5

S_SHAPE_TEMPLATE = [['.....',
                     '.....',
                     '..OO.',
                     '.OO..',
                     '.....'],
                    ['.....',
                     '..O..',
                     '..OO.',
                     '...O.',
                     '.....']]

Z_SHAPE_TEMPLATE = [['.....',
                     '.....',
                     '.OO..',
                     '..OO.',
                     '.....'],
                    ['.....',
                     '..O..',
                     '.OO..',
                     '.O...',
                     '.....']]

I_SHAPE_TEMPLATE = [['..O..',
                     '..O..',
                     '..O..',
                     '..O..',
                     '.....'],
                    ['.....',
                     '.....',
                     'OOOO.',
                     '.....',
                     '.....']]

O_SHAPE_TEMPLATE = [['.....',
                     '.....',
                     '.OO..',
                     '.OO..',
                     '.....']]

J_SHAPE_TEMPLATE = [['.....',
                     '.O...',
                     '.OOO.',
                     '.....',
                     '.....'],
                    ['.....',
                     '..OO.',
                     '..O..',
                     '..O..',
                     '.....'],
                    ['.....',
                     '.....',
                     '.OOO.',
                     '...O.',
                     '.....'],
                    ['.....',
                     '..O..',
                     '..O..',
                     '.OO..',
                     '.....']]

L_SHAPE_TEMPLATE = [['.....',
                     '...O.',
                     '.OOO.',
                     '.....',
                     '.....'],
                    ['.....',
                     '..O..',
                     '..O..',
                     '..OO.',
                     '.....'],
                    ['.....',
                     '.....',
                     '.OOO.',
                     '.O...',
                     '.....'],
                    ['.....',
                     '.OO..',
                     '..O..',
                     '..O..',
                     '.....']]

T_SHAPE_TEMPLATE = [['.....',
                     '..O..',
                     '.OOO.',
                     '.....',
                     '.....'],
                    ['.....',
                     '..O..',
                     '..OO.',
                     '..O..',
                     '.....'],
                    ['.....',
                     '.....',
                     '.OOO.',
                     '..O..',
                     '.....'],
                    ['.....',
                     '..O..',
                     '.OO..',
                     '..O..',
                     '.....']]

```

Our game program needs to know how each of the shapes are shaped, including for all of their possible rotations. In order to do this, we will create lists of lists of strings. The inner list of strings will represent a single rotation of a shape, like this:

```
['.....',
 '.....',
 '..OO.',
 '.OO..',
 '.....']
```

We will write the rest of our code so that it interprets a list of strings like the one above to represent a shape where the periods are empty spaces and the Os are boxes, like this:



You can see that this list is spread across multiple lines in the file editor. This is perfectly valid Python, because the Python interpreter realizes that until it sees the] closing square bracket, the list isn't finished. The indentation doesn't matter because Python knows you won't have different indentation in the middle of a list. This is just like how you can write your source code like this and they both work just fine:

```
spam = ['hello', 3.14, 'world', 42, 10, 'fuzz']
eggs = ['hello', 3.14,
   'world'
         , 42,
       10, 'fuzz']
```

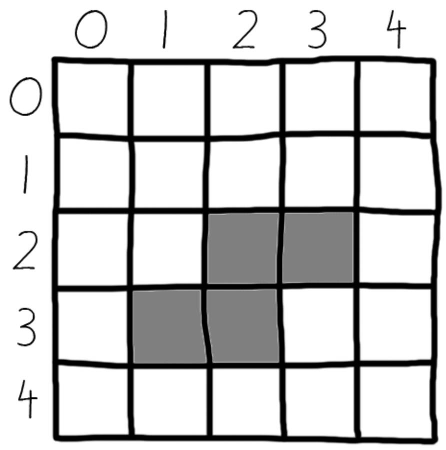
Though, of course, the code for the eggs list would be much more readable if we lined up all the items in the list.

We will make “template” data structures by creating a list of these list of strings, and store them in variables such as S_SHAPE_TEMPLATE. This way, len(S_SHAPE_TEMPLATE) will represent how many possible rotations there are for the S shape, and S_SHAPE_TEMPLATE[0] will represent the S shape’s first possible rotation. Lines 47 to 147 will create “template” data structures for each of the shapes.

Imagine that each possible piece in a tiny 5x5 board of empty space, with some of the spaces on the board filled in with boxes. The following expressions that use S_SHAPE_TEMPLATE[0] are True:

```
S_SHAPE_TEMPLATE[0][2][2] == 'O'
S_SHAPE_TEMPLATE[0][2][3] == 'O'
S_SHAPE_TEMPLATE[0][3][1] == 'O'
S_SHAPE_TEMPLATE[0][3][2] == 'O'
```

If we drew this board on paper, it would look something like this:



This is how we can represent things like Tetromino pieces as Python values such as strings and lists. The TEMPLATEWIDTH and TEMPLATEHEIGHT simply set how large each row and column for each shape’s rotation should be.

```
SHAPES = {'S': S_SHAPE_TEMPLATE,
          'Z': Z_SHAPE_TEMPLATE,
          'J': J_SHAPE_TEMPLATE,
          'L': L_SHAPE_TEMPLATE,
          'I': I_SHAPE_TEMPLATE,
          'O': O_SHAPE_TEMPLATE,
          'T': T_SHAPE_TEMPLATE}

```

The SHAPES variable will be a dictionary that stores all of the different templates. Because each template has all the possible rotations of a single shape, this means that the SHAPES variable contains all possible rotations of every possible shape. This will be the data structure that contains all of the shape data in our game.

The main() Function

```
def main():
    global FPSCLOCK, WINDOWSURF, BASICFONT, BIGFONT
    pygame.init()
    FPSCLOCK = pygame.time.Clock()
    WINDOWSURF = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
    BASICFONT = pygame.font.Font('freesansbold.ttf', 18)
    BIGFONT = pygame.font.Font('freesansbold.ttf', 100)
    pygame.display.set_caption('Tetromino')

    showTextScreen('Tetromino')
```

The main() function handles creating some more global constants and showing the start screen.

```
    while True: # game loop
        if random.randint(0, 1) == 0:
            pygame.mixer.music.load('tetrisb.mid')
        else:
            pygame.mixer.music.load('tetrisc.mid')
        pygame.mixer.music.play(-1, 0.0)
        runGame()
        pygame.mixer.music.stop()
        showTextScreen('Game Over')


```

The code for the actual game is all in runGame(). The main() function here simply randomly decides what background music to start playing (either the tetrisb.mid or tetrisc.mid MIDI music file), then calls runGame() to begin the game. When the player loses, runGame() will return to main(), which then stops the background music and displays the Game Over screen. When the player presses a key, the showTextScreen() call that displays the Game Over screen will return and the game loop will loop back to the beginning at line 169 and start another game.

The Start of a New Game

```
179. def runGame():
180.     # setup variables for the start of the game
181.     board = getBlankBoard()
182.     lastMoveDownTime = time.time()
183.     lastMoveSidewaysTime = time.time()
184.     lastFallTime = time.time()
185.     movingDown = False # note: there is no movingUp variable
186.     movingLeft = False
187.     movingRight = False
188.     score = 0
189.     level, fallFreq = calculateLevelAndFallFreq(score)
190.
191.     fallingPiece = getNewPiece()
192.     nextPiece = getNewPiece()
193.
```

Before the game begins and pieces start falling, we need to initialize some variables to their default values. The fallingPiece variable will be set to the currently falling piece that can be rotated by the player. The nextPiece variable will be set to the piece that shows up in the “Next” part of the screen so that player knows what piece is coming up after setting the falling piece.

The Game Loop

```
194.     while True: # main game loop
195.         if fallingPiece == None:
196.             # No current piece in play, so start a new piece at the top
197.             fallingPiece = nextPiece
198.             nextPiece = getNewPiece()
199.             lastFallTime = time.time() # reset lastFallTime
200.
201.             if not isValidPosition(board, fallingPiece):
202.                 return # can't fit a new piece on the board, so game over
203.
204.         checkForQuit()
```

The main game loop that starts on line 194 handles all of the code for the main part of the game when pieces are falling to the bottom. The fallingPiece variable is set to None after the falling piece has be set on the bottom of the board. This means that the piece in nextPiece should be copied to the fallingPiece variable, and a random new variable should be put into the nextPiece variable (this is returned from our getNewPiece() function). The lastFallTime variable is also reset to the current time.

The pieces that getNewPiece() are positioned a little bit above the board, usually with part of the piece already on the board. But if this is an invalid position because the board is already filled up there (in which case the isValidPosition() call on line 201 will return False), then we know that the board is full and the player should lose the game. Whenever the game ends, the program simply returns out of the runGame() function.

The Event Handling Loop

```
205.         for event in pygame.event.get(): # event handling loop
206.             if event.type == KEYUP:
```

The event handling loop takes care of when the player rotates the falling piece, moves the falling piece, or pauses the game.

Pausing the Game

```
207.                 if (event.key == K_p):
208.                     # Pausing the game
209.                     WINDOWSURF.fill(BGCOLOR)
210.                     pygame.mixer.music.stop()
211.                     showTextScreen('Paused') # pause until a key press
212.                     pygame.mixer.music.play(-1, 0.0)
213.                     lastFallTime = time.time()
214.                     lastMoveDownTime = time.time()
215.                     lastMoveSidewaysTime = time.time()
```

If the player has pressed the P key, then the game should pause. We need to hide the board from the player (otherwise the player could cheat by pausing the game and looking at the board to decide where the next piece should drop).

The code blanks out the window with a call to WINDOWSURF.fill(BGCOLOR) and stops the music. The showTextScreen() function is called to display the “Paused” text and wait for the player to press a key to continue.

Once the player has pressed a key, the execution returns from showTextScreen(). The code should restart the background music. Also, since a large amount of time could have passed since the player paused the game, the lastFallTime, lastMoveDownTime, and lastMoveSidewaysTime variables should all be reset to the current time.

Using Movement Variables to Handle User Input

```
216.                     elif (event.key == K_LEFT or event.key == K_a):
217.                         movingLeft = False
218.                     elif (event.key == K_RIGHT or event.key == K_d):
219.                         movingRight = False
220.                     elif (event.key == K_DOWN or event.key == K_s):
221.                         movingDown = False
222.
```

Letting up on one of the arrow keys (or some of the WASD keys) will set the movingLeft, movingRight, and movingDown variables back to False, indicating that the player no longer wants to move the piece in those directions. The code later will handle what to do based on the Boolean values inside these variables.

Checking if a Slide or Rotation is Valid

```
223.                 elif event.type == KEYDOWN:
224.                     # moving the block sideways
225.                     if (event.key == K_LEFT or event.key == K_a) and
isValidPosition(board, fallingPiece, adjX=-1):

226.                         fallingPiece['x'] -= 1
227.                         movingLeft = True
228.                         movingRight = False
229.                         lastMoveSidewaysTime = time.time()
230.
```

When the left key is pressed down (and moving to the left is a valid move for the falling piece, as determined by the call to isValidPosition), then we should change the position to one space to the left by subtracting the value of fallingPiece['x'] by 1. The movingLeft variable should be set to True (and just to make sure the falling piece won't move both left *and* right, the movingRight variable will be set to False). The lastMoveSidewaysTime variable will be updated to the current time on line 229.

The isValidPosition() function has optional parameters called adjX and adjY. Normally the isValidPosition() function checks the position of the data provided by the piece object that is passed for the second parameter. However, sometimes we don't want to check where the piece is currently located, but rather a few spaces over.

If we pass -1 for the adjX (a short name for "adjusted X"), then it doesn't check if the position of the piece is valid, but rather if the position of where the piece would be if it was one space to the left. Passing 1 for adjX would check one space to the right. There is also an adjY optional parameter. Passing -1 for adjY checks one space above where the piece is currently positioned, and passing a value like 3 for adjY would

These variables are set so that the player can just hold down the arrow key to keep moving the piece over. If the movingLeft variable is set to True, the program can know that the left arrow key (or A key) has been pressed and not yet let go. And if 0.15 seconds (the number stored in MOVESIDEWAYSFREQ) has passed since the time stored in lastMoveSidewaysTime, then it is time for the program to move the falling piece to the left again.

```
231.                 elif (event.key == K_RIGHT or event.key == K_d) and
isValidPosition(board, fallingPiece, adjX=1):

232.                     fallingPiece['x'] += 1
233.                     movingRight = True
234.                     movingLeft = False
235.                     lastMoveSidewaysTime = time.time()
236.
```

This code is almost identical to lines 225 to 229, except that it handles moving the falling piece to the right when the right arrow key (or D key) has been pressed.

```
237.                 # rotating the block (if there is room to rotate)
238.                 elif (event.key == K_UP or event.key == K_w):
239.                     fallingPiece['rotation'] = (fallingPiece['rotation'] +
1) % len(SHAPES[fallingPiece['shape']])
```

The up arrow key (and W key) will rotate the falling piece to its next rotation. All the code has to do is increment the 'rotation' key in the fallingPiece dictionary by 1. However, if incrementing the 'rotation' key makes it larger than the total number of rotations, then “modding” by the total number of possible rotations for that shape (which is what len(SHAPES[fallingPiece['shape']]) is) then it will “roll over” to 0.

Here’s an example of this modding with the J shape, which has 4 possible rotations:

```
>>> 0 % 4
0
>>> 1 % 4
1
>>> 2 % 4
2
>>> 3 % 4
3
>>> 5 % 4
```

```
1
>>> 6 % 4
2
>>> 7 % 4
3
>>> 8 % 4
0
>>>
```

The code on line 239 does the exact same thing as this code would:

```
fallingPiece['rotation'] += 1
while fallingPiece['rotation'] >= len(SHAPES[fallingPiece['shape']]):
    fallingPiece['rotation'] -= len(SHAPES[fallingPiece['shape']])
```

```
240.                     if not isValidPosition(board, fallingPiece):
241.                         fallingPiece['rotation'] =
(fallingPiece['rotation'] - 1) % len(SHAPES[fallingPiece['shape']])
```

If the new rotated position is not valid because it overlaps some boxes already on the board, then we want to switch it back to the original rotation by subtracting 1 from fallingPiece['rotation']. We can also mod it by len(SHAPES[fallingPiece['shape']]) so that if the new value is -1, the modding will change it back to the largest rotation. Here's an example of this modding with the J shape, which has 4 possible rotations (with indexes 0 to 3):

```
>>> -1 % 4
3
```

```
242.                 elif (event.key == K_q): # rotate the other direction
243.                     fallingPiece['rotation'] = (fallingPiece['rotation'] -
1) % len(SHAPES[fallingPiece['shape']])

244.                     if not isValidPosition(board, fallingPiece):
245.                         fallingPiece['rotation'] =
(fallingPiece['rotation'] + 1) % len(SHAPES[fallingPiece['shape']])

246.
```

Lines 242 to 245 do the same thing 238 to 241, except they handle the case where the player has pressed the Q key which rotates the piece in the opposite direction. In this case, we *subtract* 1 from fallingPiece['rotation'] (which is done on line 243) instead of adding 1.

```
247.                     # making the block fall faster with the down key
248.                     elif (event.key == K_DOWN or event.key == K_s):
249.                         movingDown = True
250.                         if isValidPosition(board, fallingPiece, adjY=1):
251.                             fallingPiece['y'] += 1
252.                         lastMoveDownTime = time.time()
253.
```

If the down arrow or S key is pressed down, then the player wants the piece to fall even faster. Line 251 moves the piece down one space on the board (but only if it is a valid space). The movingDown variable is set to True and lastMoveDownTime is reset to the current time. These variables will be checked later so that the piece keeps falling at a faster rate as long as the down arrow or S key is held down.

Finding the Bottom

```
254.                     # move the current block all the way down
255.                     elif event.key == K_SPACE:
256.                         movingDown = False
257.                         movingLeft = False
258.                         movingRight = False
259.                         for i in range(1, BOARDHEIGHT):
260.                             if not isValidPosition(board, fallingPiece,
adjY=i):

261.                                 break
262.                         fallingPiece['y'] += (i-1)
263.
```

While the down arrow key makes the piece fall faster, when the player presses the space key the falling piece will immediately drop down as far as it can go on the board. So we need to find out how many spaces the piece can move until it hits bottom.

First, lines 256 to 258 will set all the moving variables to False (which makes the code in later parts of the programming think that the user has let up on any arrow keys that were held down.) This is done because this code will move the piece to the absolute bottom and begin falling the next piece, and we don't want to surprise the player by having those pieces immediately start moving just because she was holding down an arrow key when she hit the space key.

To find the farthest that the piece can fall, we should call isValidPosition() and pass the integer 1 for the adjY parameter. If isValidPosition() returns False, we know that the piece can fall any further and is already at the bottom. If isValidPosition() returns True, then we know that it can fall 1 space down.

In that case, we should call isValidPosition() with adjY set to 2. If it returns True again, we will call isValidPosition() with adjY set to 3, and so on. This is what the for loop on line 259 handles: calling isValidPosition() with increasing values to pass for adjY until the function call returns False. At that point, we know that the value in i is one space more past the bottom. This is why line 262 increases fallingPiece['y'] by (i – 1).

(Also note that the second parameter to range() on line 259's for statement is set to BOARDHEIGHT, because this is the maximum amount that the piece could fall before it must hit the bottom of the board.)

Moving by Holding Down the Key

```
264.         # handle moving the block because of user input
265.         if (movingLeft or movingRight) and time.time() -
lastMoveSidewaysTime > MOVESIDEWAYSFREQ:

266.             if movingLeft and isValidPosition(board, fallingPiece, adjX=-
1):

267.                 fallingPiece['x'] -= 1
268.             elif movingRight and isValidPosition(board, fallingPiece,
adjX=1):

269.                 fallingPiece['x'] += 1
270.             lastMoveSidewaysTime = time.time()
271.
```

Remember that on line 227 the movingLeft variable was set to True if the player pressed down on the left arrow key? (The same for line 233 where movingRight was set to True if the player pressed down on the right arrow key.) The moving variables were set back to False if the user let up on these keys also (see line 217 and 219).

What also happened when the player pressed down on the left or right arrow key was that the lastMoveSidewaysTime variable was set to the current time (which was the return value of time.time()). If the player continued to hold down the arrow key without letting up on it, then the movingLeft or movingRight variable would still be set to True.

If the user held down on the key for longer than 0.15 seconds (the value stored in MOVESIDEWAYSFREQ is the float 0.15) then the expression time.time() - lastMoveSidewaysTime > MOVESIDEWAYSFREQ would evaluate to True. Line 265's condition is True if the user has both held down the arrow key and 0.15 seconds has passed, and in that case we should move the falling piece to the left or right even though the user hasn't pressed the arrow key again.

This is very useful because it would become tiresome for the player to repeatedly hit the arrow keys to get the falling piece to move over multiple spaces on the board. Instead, he can just hold down an arrow key and the piece will keep moving over until he lets up on the key. When that happens, the code on lines 216 to 221 will set the moving variable to False and the condition on line 265 will be False. That is what stops the falling piece from sliding over more.

To demonstrate why the time.time() - lastMoveSidewaysTime > MOVESIDEWAYSFREQ returns True after the number of seconds in MOVESIDEWAYSFREQ has passed, run this short program:

```
import time

WAITTIME = 4
begin = time.time()

while True:
    now = time.time()
    message = '%s, %s, %s' % (begin, now, (now - begin))
    if now - begin > WAITTIME:
        print(message + ' PASSED WAIT TIME!')
    else:
        print(message + ' Not yet...')
    time.sleep(0.2)
```

This program has an infinite loop, so in order to terminate it, press Ctrl-C. The output of this program will look something like this:

```
1322106392.2, 1322106392.2, 0.0 Not yet...
1322106392.2, 1322106392.42, 0.219000101089 Not yet...
1322106392.2, 1322106392.65, 0.449000120163 Not yet...
1322106392.2, 1322106392.88, 0.680999994278 Not yet...
1322106392.2, 1322106393.11, 0.910000085831 Not yet...
1322106392.2, 1322106393.34, 1.1400001049 Not yet...
1322106392.2, 1322106393.57, 1.3710000515 Not yet...
1322106392.2, 1322106393.83, 1.6360001564 Not yet...
1322106392.2, 1322106394.05, 1.85199999809 Not yet...
1322106392.2, 1322106394.28, 2.08000016212 Not yet...
1322106392.2, 1322106394.51, 2.30900001526 Not yet...
1322106392.2, 1322106394.74, 2.54100012779 Not yet...
1322106392.2, 1322106394.97, 2.76999998093 Not yet...
1322106392.2, 1322106395.2, 2.99800014496 Not yet...
1322106392.2, 1322106395.42, 3.22699999809 Not yet...
1322106392.2, 1322106395.65, 3.45600008965 Not yet...
1322106392.2, 1322106395.89, 3.69200015068 Not yet...
```

```
1322106392.2, 1322106396.12, 3.92100000381 Not yet...
1322106392.2, 1322106396.35, 4.14899992943 PASSED WAIT TIME!
1322106392.2, 1322106396.58, 4.3789999485 PASSED WAIT TIME!
1322106392.2, 1322106396.81, 4.60700011253 PASSED WAIT TIME!
1322106392.2, 1322106397.04, 4.83700013161 PASSED WAIT TIME!
1322106392.2, 1322106397.26, 5.06500005722 PASSED WAIT TIME!
Traceback (most recent call last):
  File "C:\timetest.py", line 13, in <module>
    time.sleep(0.2)
KeyboardInterrupt
```

The first number on each line of output is the return value of time.time() when the program first started (and this value never changes). The second number is the latest return value from time.time() (this value keeps getting updated on each iteration of the loop). And the third number is the current time minus the start time. This third number is the number of seconds that have elapsed since the begin = time.time() line of code was executed.

If this number is greater than 4, the code will start printing “PASSED WAIT TIME!” instead of “Not yet...”. This is how our game program can know if a certain amount of time has passed since a line of code was run.

In our Tetromino program, the time.time() – lastMoveSidewaysTime expression will evaluate to the number of seconds that has elapsed since the last time lastMoveSidewaysTime was set to the current time. If this value is greater than the value in MOVESIDEWAYSFREQ, we know it is time for the code to move the falling piece over one more space.

Don’t forget to update lastMoveSidewaysTime to the current time again! This is what we do on line 270.

```
272.         if movingDown and time.time() - lastMoveDownTime > MOVEDOWNFREQ
and isValidPosition(board, fallingPiece, adjY=1):

273.             fallingPiece['y'] += 1
274.             lastMoveDownTime = time.time()
275.
```

Lines 272 to 274 do almost the same thing as lines 265 to 270 do except for moving the falling piece down. This has a separate move variable (movingDown) and “last time” variable (lastMoveDownTime) as well as a different “move frequency” variable (MOVEDOWNFREQ). Both the MOVEDOWNFREQ and MOVESIDEWAYSFREQ varaibles are set to 0.15, but we use two different variables in case you’d like to change how the game works later on.

Letting the Piece “Naturally” Fall

```
276.         # let the piece fall if it is time to fall
277.         if time.time() - lastFallTime > fallFreq:
278.             # see if the piece has landed
279.             if not isValidPosition(board, fallingPiece, adjY=1):
280.                 # falling piece has landed, set it on the board
281.                 addToBoard(board, fallingPiece)
282.                 score += removeCompleteLines(board)
283.                 level, fallFreq = calculateLevelAndFallFreq(score)
284.                 fallingPiece = None
285.             else:
286.                 # piece did not land, just move the block down
287.                 fallingPiece['y'] += 1
288.                 lastFallTime = time.time()
289.
```

The rate that the piece is naturally moving down (that is, falling) is tracked by the lastFallTime variable. If enough time has elapsed since the falling piece last fell down one space, lines 279 to 288 will handle dropping the piece by one space.

If the condition on line 279 is True, then the piece has landed. The call to addToBoard() will make the piece part of the board data structure (so that future pieces can land on it), and the removeCompleteLines() call will handle erasing any full lines on the board and moving the boxes down. The removeCompleteLines() function also returns an integer value of how many lines were removed, so we add this number to the score.

Because the score may have changed, we call the calculateLevelAndFallFreq() function to update the current level and frequency that the pieces fall. And finally, we set the fallingPiece variable to None to indicate that the next piece should become the new falling piece, and a random new piece should be generated for the new next piece. (That is done on lines 195 to 199 at the beginning of the game loop.)

If the piece has not landed, we simply set it’s Y position down one space and reset lastFallTime to the current time.

Drawing Everything on the Screen

```
290.         # drawing everything on the screen
291.         WINDOWSURF.fill(BGCOLOR)
292.         drawBoard(board)
293.         drawStatus(score, level)
294.         drawNextPiece(nextPiece)
295.         if fallingPiece != None:
```

```
            drawPiece(fallingPiece)

        pygame.display.update()
        FPSCLOCK.tick(FPS)


```

Now that the game loop has handled all events and updated the game state, the game loop just needs to draw the game state to the screen. Most of the drawing is handled by other functions, so the game loop code just needs to call those functions. Then the call to pygame.display.update() makes the display Surface appear on the actual computer screen, and the tick() method call adds a slight pause so the game doesn't run too fast.

A Shortcut Function for Making Text

```
def makeTextObjs(text, font, color):
    surf = font.render(text, True, color)
    return surf, surf.get_rect()


```

This function just provides us with a shortcut. Given the text, Font object, and a Color object, it calls render() for us and returns the Surface and Rect object for this text. This just saves us from typing out the code to create the Surface and Rect object.

The terminate() Function

```
def terminate():
    pygame.quit()
    sys.exit()


```

The terminate() function works the same as in the previous game programs.

The checkForKeyPress() Function

```
def checkForKeyPress():
    # Go through event queue looking for a KEYUP event.
    # Grab KEYDOWN events to remove them from the event queue.
    checkForQuit()

    for event in pygame.event.get([KEYDOWN, KEYUP]):
        if event.type == KEYDOWN:
            continue
        return event.key
    return None


```

The checkForKeyPress() function works almost the same as it did in the Wormy game. First it calls checkForQuit() to handle any QUIT events (or KEYUP events specifically for the Esc key) and terminates the program if there are any. Then it pulls out all the KEYUP and KEYDOWN events from the event queue. It ignores any KEYDOWN events (it only specified KEYDOWN to the pygame.event.get() call to clear them out of the event queue.)

If there were no KEYUP events in the event queue, then the function returns None.

A Generic Text Screen Function

```
def showTextScreen(text):
    # This function displays large text in the
    # center of the screen until a key is pressed.
    # Draw the text drop shadow
    titleSurf, titleRect = makeTextObjs(text, BIGFONT, TEXTSHADOWCOLOR)
    titleRect.center = (int(WINDOWWIDTH / 2), int(WINDOWHEIGHT / 2))
    WINDOWSURF.blit(titleSurf, titleRect)

    # Draw the text
    titleSurf, titleRect = makeTextObjs(text, BIGFONT, TEXTCOLOR)
    titleRect.center = (int(WINDOWWIDTH / 2) - 3, int(WINDOWHEIGHT / 2) - 3)
    WINDOWSURF.blit(titleSurf, titleRect)

    # Draw the additional "Press a key to play." text.
    pressKeySurf, pressKeyRect = makeTextObjs('Press a key to play.', BASICFONT, TEXTCOLOR)
    pressKeyRect.center = (int(WINDOWWIDTH / 2), int(WINDOWHEIGHT / 2) + 100)
    WINDOWSURF.blit(pressKeySurf, pressKeyRect)

```

Instead of separate functions for the start screen and game over screens, we will create one generic function named showTextScreen(). The showTextScreen() function will draw whatever text we pass for the text parameter. Also, the text “Press a key to play.” will be displayed in addition.

Notice that lines 328 to 330 draw the text in a darker shadow color first, and then lines 333 to 335 draw the same text again, except offset by 3 pixels to the left and 3 pixels upward. This creates a “drop shadow” effect that makes the text look a bit prettier. (You can compare the difference by commenting out lines 328 to 330 to see the text without a drop shadow.)

The showTextScreen() will be used for the start screen, the game over screen, and also for a pause screen. (The pause screen is explained later in this chapter.)

```
342.     while checkForKeyPress() == None:
343.         pygame.display.update()
344.         FPSCLOCK.tick()
345.
346.
```

We want the text to stay on the screen until the user presses a key. This small loop will constantly call pygame.display.update() and FPSCLOCK.tick() until checkForKeyPress() returns a value other than None. (This happens when the user presses a key.)

The checkForQuit() Function

```
347. def checkForQuit():
348.     for event in pygame.event.get(QUIT): # get all the QUIT events
349.         terminate() # terminate if any QUIT events are present
350.     for event in pygame.event.get(KEYUP): # get all the KEYUP events
351.         if event.key == K_ESCAPE:
352.             terminate() # terminate if the KEYUP event was for the Esc key
353.         pygame.event.post(event) # put the other KEYUP event objects back
354.
355.
```

The checkForQuit() function can be called to handle any events that will cause the program to terminate. This happens if there are any QUIT events in the event queue (this is handle by lines 348 and 349), or if there is a KEYUP event of the Esc key. (The player should be able to press the Esc key at any time to quit the program.)

Because the pygame.event.get() call on line 350 pulls out all of the KEYUP events (including events for keys other than the Esc key), if the event is not for the Esc key, we want to put it back into the event queue by calling the pygame.event.post() function.

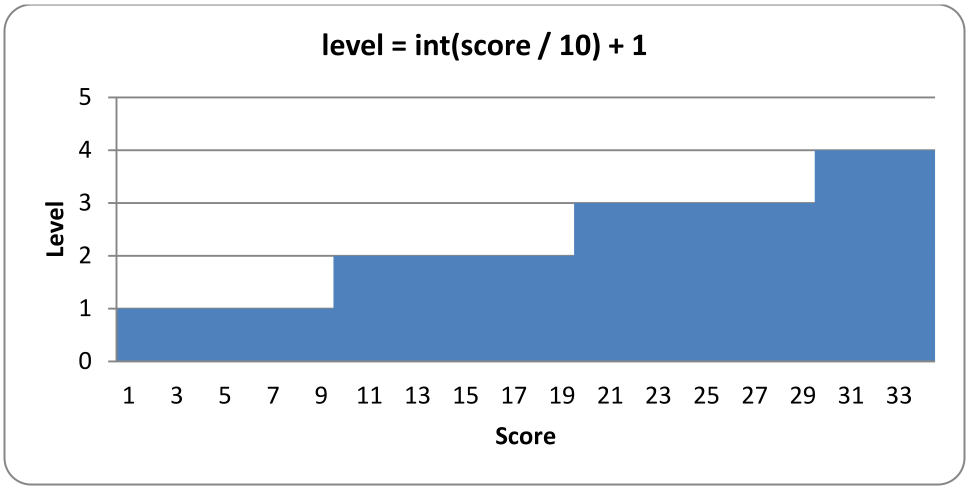
The calculateLevelAndFallFreq() Function

```
356. def calculateLevelAndFallFreq(score):
357.     # Based on the score, return the level the player is on and
358.     # how many seconds pass until a falling piece falls one space.
```

```
    level = int(score / 10) + 1
    fallFreq = 0.27 - (level * 0.02)
    return level, fallFreq

```

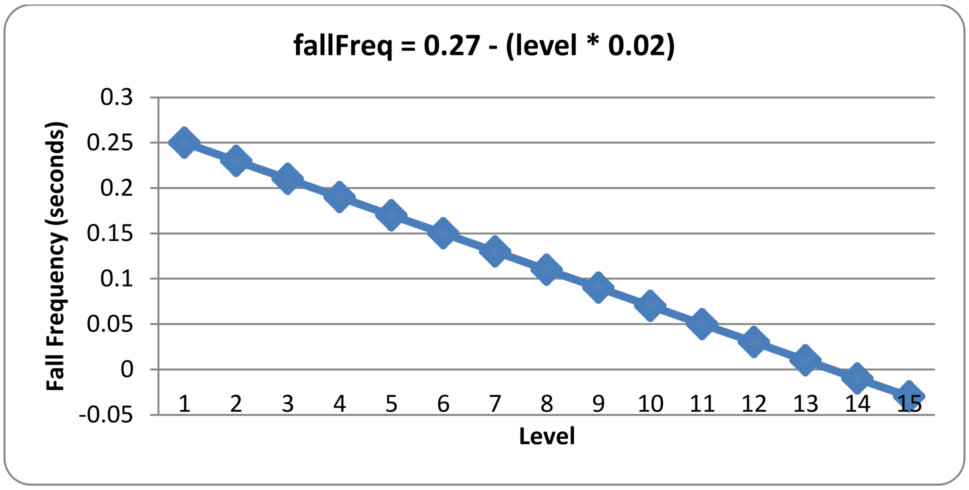
Every time the player completes a line, her score will increase by 1 point. Every ten points, the game goes up a level and the pieces start falling down faster. Both the level and the falling frequency can be calculated from the score that is passed to this function.

To calculate the level, we use the int() function to round down the score divided by 10. So if the score any number between 0 and 9, the int() call will round it down to 0. The + 1 part of the code is there because we want the first level to be level 1, not level 0. When the score reaches 10, then int(10 / 10) will evaluate to 1, and the + 1 will make the level 2. Here is a graph showing the values of level for the scores 1 to 34:



To calculate the falling frequency, we start with a base time of 0.27 (meaning that the piece will naturally fall once every 0.27 seconds). Then we multiply the level by 0.02, and subtract that from the 0.27 base time. So on level 1, we subtract 0.02 * 1 (i.e. 0.02) from 0.27 to get 0.25. On level 2, we subtract 0.02 * 2 (i.e. 0.04) to get 0.23. You can think of the level * 0.02 part of the equation as “for every level, the piece will fall 0.02 seconds faster than the previous level.”

We can also make a graph showing how fast the pieces will fall at each level of the game:



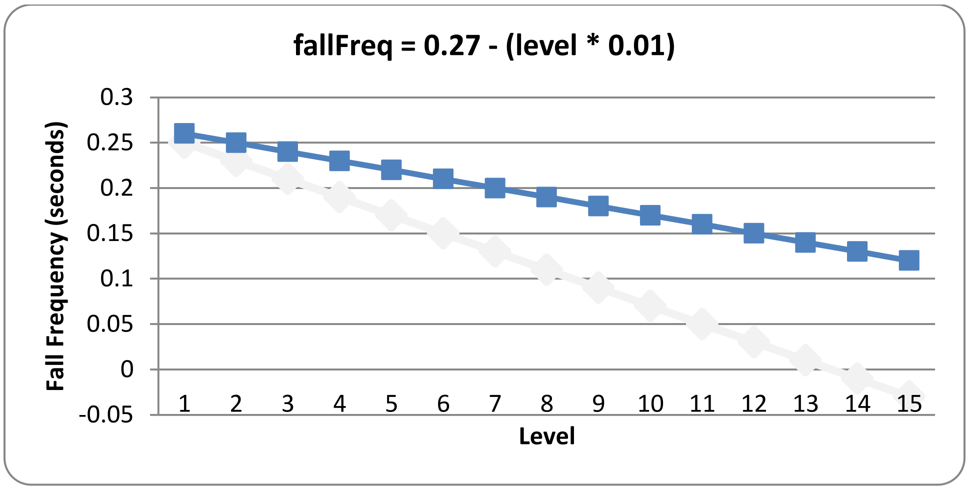
You can see that at level 14, the falling frequency will be less than 0. This won't cause any bugs with our code, because line 277 just checks that the elapsed time since the falling piece last fell one space is greater than the calculated falling frequency. So if the falling frequency is negative, then the condition on line 277 will always be True and the piece will fall on every iteration of the game loop. From level 14 and beyond, the piece cannot fall any faster.

If the FPS is set at 25, this means that at reaching level 14, the falling piece will fall 25 spaces a second. Considering that the board is only 20 spaces tall, that means the player will have less than a second to set each piece!

If you want the pieces to start (if you can see what I mean) falling faster at a slower rate, you can change the equation that the calculateLevelAndFallFreq() uses. For example, let's say line 360 was this:

```
360.     fallFreq = 0.27 - (level * 0.01)
```

In the above case, the pieces would only fall 0.01 seconds faster on each level rather than 0.02 seconds faster. The graph would look like this (the original line is also in the graph in light grey):



As you can see, with this new equation, level 14 would only be as hard as the original level 7. You can change the game to be as difficult or easy as you like by changing the equations in calculateLevelAndFallFreq().

Generating Pieces

```
363. def getNewPiece():
364.     # return a random new piece in a random rotation and color
365.     shape = random.choice(list(SHAPES.keys()))
366.     newPiece = {'shape': shape,
367.                 'rotation': random.randint(0, len(SHAPES[shape]) - 1),
368.                 'x': int(BOARDWIDTH / 2) - int(TEMPLATEWIDTH / 2),
369.                 'y': -2, # start it above the board (i.e. less than 0)
370.                 'color': random.randint(0, len(COLORS)-1)}
371.     return newPiece
372.
373.
```

The getNewPiece() function generates a random piece that is positioned at the top of the board. First, to randomly choose the shape of the piece, we create a list of all the possible shapes by calling list(SHAPES.keys()) on line 365. (The list() call is needed because technically the keys() dictionary method returns a “dict_keys” value, which must be converted to a list value before being passed to random.choice().) The random.choice() function then randomly selects an item from the list.

The piece data structures are simply a dictionary value with the keys 'shape', 'rotation', 'x', 'y', and 'color'. The value for the 'rotation' key is a random integer from 0 to one less than however many possible rotations there are for that shape.

Notice that we don’t store the list of strings value (like the ones store in the constants like S_SHAPE_TEMPLATE) in each piece data structure. Instead, we just store an index for the shape and rotation which refer to the PIECES constant.

The 'x' key’s value is always set to the middle of the board (also accounting for the width of the pieces themselves, which is found from our TEMPLATEWIDTH constant). The 'y' key’s value is always set to -2 to place it slightly above the board.

Since the COLORS constant is a tuple of the different colors, selecting a random number for 0 to the length of COLORS (subtracting one) will give us a random index value for the piece’s color.

Once all of the values in the newPiece dictionary are set, the getNewPiece() function returns newPiece.

Adding Pieces to the Board Data Structure

```
def addToBoard(board, piece):
    # fill in the board based on piece's location, shape, and rotation
    for x in range(TEMPLATEWIDTH):
        for y in range(TEMPLATEHEIGHT):
            if SHAPES[piece['shape']][piece['rotation']][y][x] != BLANK:
                board[x + piece['x']][y + piece['y']] = piece['color']


```

The board data structure is a model for the rectangular space where pieces that have previously landed are tracked. The falling piece is not marked on the board data structure, it is kept in a separate variable. What the addToBoard() function does is takes a piece data structure and adds its boxes to the board data structure. This happens after a piece has landed.

The nested for loops go through every space in the piece data structure, and if it finds a box in the space, it adds it to the board.

Creating a New Board Data Structure

```
def getBlankBoard():
    # create and return a new blank board data structure
    board = []
    for i in range(BOARDWIDTH):
        board.append([BLANK] * BOARDHEIGHT)
    return board


```

The data structure used for the board is fairly simple: it's a list of lists of values. If the value is the same as the value in BLANK, then it is an empty space. If the value is an integer, then it represents a box that is the color that the integer indexes in the COLORS constant list.

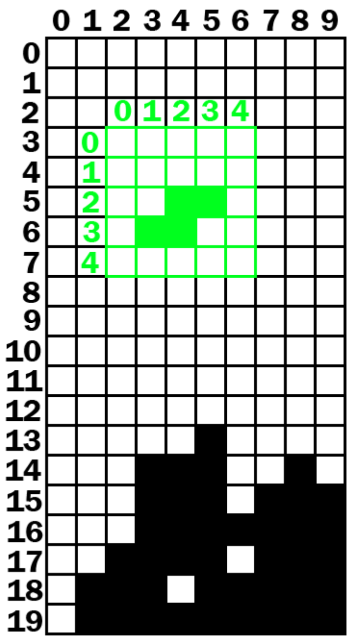
In order to create a blank board, list replication is used to create the lists of BLANK values which represents a column. This is done on line 386. One of these lists is created for each of the columns in the board (this is what the for loop on line 385 does).

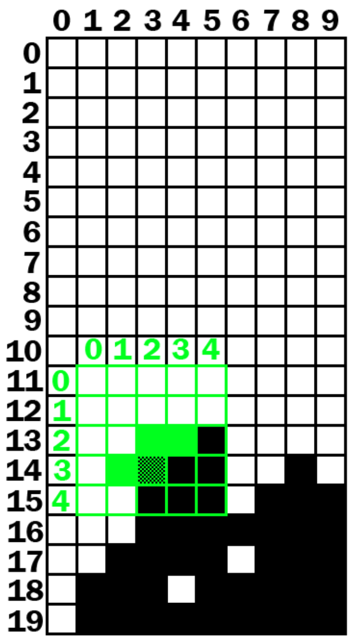
The isOnBoard() and isValidPosition() Functions

```
390. def isOnBoard(x, y):
391.     return x >= 0 and x < BOARDWIDTH and y < BOARDHEIGHT
392.
393.
394. def isValidPosition(board, piece, adjX=0, adjY=0):
395.     # Return True if the piece is within the board and not colliding
396.     for x in range(TEMPLATEWIDTH):
397.         for y in range(TEMPLATEHEIGHT):
398.             isAboveBoard = y + piece['y'] + adjY < 0
399.             if isAboveBoard or
SHAPES[piece['shape']][piece['rotation']][y][x] == BLANK:

400.                 continue
401.             if not isOnBoard(x + piece['x'] + adjX, y + piece['y'] +
adjY):

402.                 return False
403.             if board[x + piece['x'] + adjX][y + piece['y'] + adjY] !=
BLANK:

404.                 return False
405.     return True
406.
```

The isOnBoard() is a simple function which checks that the XY coordinates that are passed represent valid values that exist on the board. As long as both the XY coorinates are not less the 0 or greater than or equal to the BOARDWIDTH and BOARDHEIGHT constants, then the function returns True.

The isOnBoard() function is used in the isValidPosition() function. The isValidPosition() function is given a board data structure and a piece data structure, and returns True if all the boxes in the piece are both on the board and not overlapping any boxes on the board. This is done by taking the piece's XY coordinates (which is really the coordinate of the upper right box on the 5x5 boxes for the piece) and adding the coordinate inside the piece data structure. Here's a couple pictures to help illustrate this:


The board with a falling piece.


The board with the falling piece in an invalid position.

On the left board, the falling piece's (that is, the top left corner of the falling piece's) XY coordinates are (2, 3) on the board. But the boxes inside the falling piece's coordinate system have their own coordinates. To find the "board" coordinates of these pieces, we just have to add the "board" coordinates of the falling piece's top left box and the "piece" coordinates of the boxes.

On the left board, the falling piece's boxes are at the following "piece" coordinates:

(2, 2)   (3, 2)   (1, 3)   (2, 3)

When we add the (2, 3) coordinate (the piece's coordinates on the board) to these coordinates, it looks like this:

(2 + 2, 2 + 3)   (3 + 2, 2 + 3)   (1 + 2, 3 + 3)   (2 + 2, 3 + 3)

After adding the (2, 3) coordinate the boxes are at the following "board" coordinates:

(4, 5)   (5, 5)   (3, 6)   (4, 6)

And now that we can figure out where the falling piece's boxes are as board coordinates, we can see if they overlap with the landed boxes that are on the board. The nested for loops on lines 396 and 397 go through each of the possible coordinates on the falling piece.

We want to check if a box of the falling piece is either off of the board or overlapping a box on the board. (Although one exception is if the box is above the board, which is where it could be when the falling piece just begins falling.) Line 398 creates a variable named isAboveBoard that is set to True if the box on the falling piece at the coordinates pointed to be x and y is above the board. Otherwise it is set to False.

The if statement on line 399 checks if the space on the piece is above the board or is blank. If either of those is True, then the code executes a continue statement and goes to the next space. (Note that the end of line 399 has [y][x] instead of [x][y]. This is because the coordinates in the PIECES data structure are reversed. See the previous section, "Setting Up the Piece Templates").

The if statement on line 401 checks that the piece's box is not located on the board. The if statement on line 403 checks that the board space the piece's box is located is not blank. If either of these conditions are True, then the isValidPosition() function will return False. Notice that these if statements also adjust the coordinates for the adjX and adjY parameters that were passed in to the function.

If the code goes through the nested for loop and hasn't found a reason to return False, then the position of the piece must be valid and so the function returns True on line 405.

Checking for, and Removing, Complete Lines

```
407. def isCompleteLine(board, y):
408.     # Return True if the line filled with boxes with no gaps.
409.     for x in range(BOARDWIDTH):
410.         if board[x][y] == BLANK:
411.             return False
412.     return True
413.
414.
```

The isCompleteLine does a simple check at the row specified by the y parameter. A row on the board is considered to be "complete" when every space is filled by a box. The for loop on line 409 goes through each space in the row. If a space is blank (which is caused by it having the same value as the BLANK constant), then the function return False.

```
415. def removeCompleteLines(board):
416.     # Remove any completed lines on the board, move everything above them
down, and return the number of complete lines.
```

```
417.     numLinesRemoved = 0
418.     y = BOARDHEIGHT - 1 # start y at the bottom of the board
419.     while y >= 0:
```

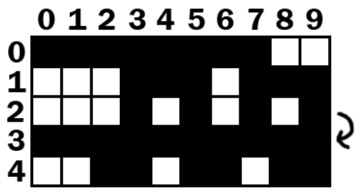
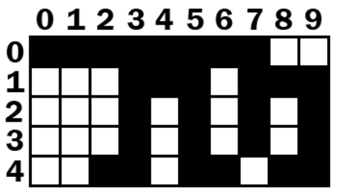
The removeCompleteLines() function will find any complete lines in the passed board data structure, remove the lines, and then shift all the boxes above that line down one row. The function will return the number of lines that were removed (which is tracked by the numLinesRemoved variable) so that this can be added to the score.

The way this function works is by running in a loop (line 417) with the y variable starting at the lowest row (which is BOARDHEIGHT – 1). Whenever the row specified by y is not complete, y will be decremented to the next highest row. The loop finally stops once y reaches -1.

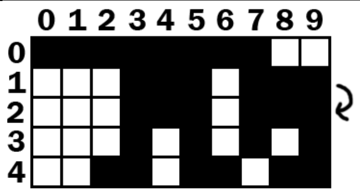
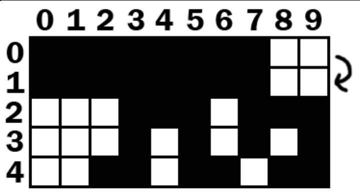
```
420.         if isCompleteLine(board, y):
421.             # Remove the line and pull boxes down by one line.
422.             for pullDownY in range(y, 0, -1):
423.                 for x in range(BOARDWIDTH):
424.                     board[x][pullDownY] = board[x][pullDownY-1]
425.             # Set very top line to blank.
426.             for x in range(BOARDWIDTH):
427.                 board[x][0] = BLANK
428.             numLinesRemoved += 1
429.             # Note on the next iteration of the loop, y is the same.
430.             # This is so that if the line that was pulled down is also
431.             # complete, it will be removed.
432.         else:
433.             y -= 1 # move on to check next row up
434.     return numLinesRemoved
435.
436.
```

The isCompleteLine() function will return True if the line that y is referring to is complete. If the line is complete, the program needs to copy the values of each row above the removed line to the next lowest line. This is what the for loop on line 422 does (which is why it’s call to the range() function begins at y, rather than 0. Also note that it uses the three argument form of range(), so that the list it returns starts at y, ends at 0, and after each iteration “increases” by -1.)

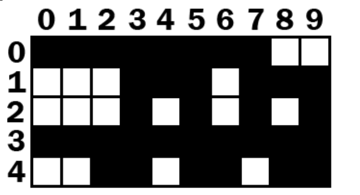
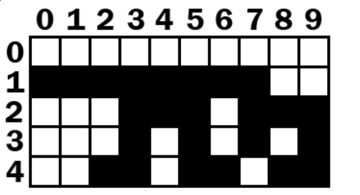
Let’s look at the following example (to save space, only the top five rows of the board are shown.) Row 3 is a complete line, which means that all the rows above it (row 2, 1, and 0) must be “pulled down”. First, row 2 is copied down to row 3. The board on the right shows what the board will look like after this is done:

This "pulling down" is really just copying the higher row's values to the row below it. After row 2 is copied to row 3, then row 1 is copied to row 2 followed by row 0 copied to row 1:

Row 0 (the row at the very top) doesn't have a row above it to copy values down. But row 0 doesn't need a row copied to it, itjust needs all the spaces set to BLANK. This is what lines 426 and 427 do. After that, the board will have changed from the board shown below on the left to the board shown below on the right:

After the complete line is removed, the execution reaches the end of the while loop that started on line 419, so the execution jumps back to the beginning of the loop. Note that at no point when the line was being removed and the rows being pulled down that the y variable changed at all. So on the next iteration, the y variable is pointing to the same row as before.

This is needed because if there were two complete lines, then the second complete line would have been pulled down and would also have to be removed. The code will then remove this complete line, and then go to the next iteration. It is only when there is not a completed line that the y variable is decremented on line 433. Once the y variable has been decremented all the way to 0, the execution will exit the while loop.

Convert from Board Coordinates to Pixel Coordinates

```
437. def convertToPixelCoords(boxx, boxy):
438.     # Convert the given xy coordinates of the board to xy
439.     # coordinates of the location on the screen.
440.     return (XMARGIN + (boxx * BOXSIZE)), (TOPMARGIN + (boxy * BOXSIZE))
441.
442.
```

This helper function converts the board's box coordinates to pixel coordinates. This function works the same way to the other "convert coordinates" functions used in the previous game programs.

Drawing a Box on the Board or Elsewhere on the Screen

```
443. def drawBox(boxx, boxy, color, pixelx=None, pixely=None):
444.     # draw a single box (each tetromino piece has four boxes)
445.     # at xy coordinates on the board. Or, if pixelx & pixely
446.     # are specified, draw to the pixel coordinates stored in
447.     # pixelx & pixely (this is used for the "Next" piece.)
448.     if color == BLANK:
449.         return
450.     if pixelx == None and pixely == None:
451.         pixelx, pixely = convertToPixelCoords(boxx, boxy)
452.     pygame.draw.rect(WINDOWSURF, COLORS[color], (pixelx + 1, pixely + 1,
BOXSIZE - 1, BOXSIZE - 1))

453.     pygame.draw.rect(WINDOWSURF, LIGHTCOLORS[color], (pixelx + 1, pixely +
1, BOXSIZE - 4, BOXSIZE - 4))

454.
455.
```

The drawBox() function draws a single box on the screen. The function can receive boxx and boxy parameters for board coordinates where the box should be drawn. However, if the pixelx and pixely parameters are specified, then these pixel coordinates will override the boxx and boxy parameters. The pixelx and pixely parameters are used to draw the boxes of the "Next" piece, which is not on the board.

If the pixelx and pixely parameters are not set, then the will be set to None by default when the function first begins, and then the if statement on line 450 will overwrite the None values with the return values from convertToPixelCoords(). This call gets the pixel coordinates of the board coordinates specified by boxx and boxy.

The code won’t fill the entire box’s space with color. To have a black outline in between the boxes of a piece, the left and top parameters in the pygame.draw.rect() call have + 1 added to them and a – 1 is added to the width and height parameters. In order to draw the highlighted box, first the box is drawn with the darker color on line 452. Then, a slightly smaller box is drawn on top of the darker box on line 453.

Drawing Everything to the Screen

```
def drawBoard(board):
    # draw the border around the board
    pygame.draw.rect(WINDOWSURF, BORDERCOLOR, (XMARGIN - 3, TOPMARGIN - 7, (BOARDWIDTH * BOXSIZE) + 8, (BOARDHEIGHT * BOXSIZE) + 8), 5)

    # fill the background of the board
    pygame.draw.rect(WINDOWSURF, BGCOLOR, (XMARGIN, TOPMARGIN, BOXSIZE * BOARDWIDTH, BOXSIZE * BOARDHEIGHT))

    # draw the individual boxes on the board
    for x in range(BOARDWIDTH):
        for y in range(BOARDHEIGHT):
            drawBox(x, y, board[x][y])


```

The drawBoard() function is responsible for calling the drawing functions for the board’s border and all the boxes on the board.

Draw the Score and Level Text

```
def drawStatus(score, level):
    # draw the score text
    scoreSurf = BASICFONT.render('Score: %s' % score, True, TEXTCOLOR)
    scoreRect = scoreSurf.get_rect()
    scoreRect.topleft = (WINDOWWIDTH - 150, 20)
    WINDOWSURF.blit(scoreSurf, scoreRect)

    # draw the level text
    levelSurf = BASICFONT.render('Level: %s' % level, True, TEXTCOLOR)
    levelRect = levelSurf.get_rect()
    levelRect.topleft = (WINDOWWIDTH - 150, 50)
    WINDOWSURF.blit(levelSurf, levelRect)


```

The drawStatus() function is responsible for rendering the text for the “Score:” and “Level:” information that appears in the upper right of the corner of the screen.

Draw a Piece on the Board or Elsewhere on the Screen

```
482. def drawPiece(piece, pixelx=None, pixely=None):
483.     shapeToDraw = SHAPES[piece['shape']][piece['rotation']]
484.     if pixelx == None and pixely == None:
485.         # if pixelx & pixely hasn't been specified, use the location stored in the piece data structure
486.         pixelx, pixely = convertToPixelCoords(piece['x'], piece['y'])
487.
488.     # draw each of the blocks that make up the piece
489.     for x in range(TEMPLATEWIDTH):
490.         for y in range(TEMPLATEHEIGHT):
491.             if shapeToDraw[y][x] != BLANK:
492.                 drawBox(None, None, piece['color'], pixelx + (x * BOXSIZE), pixely + (y * BOXSIZE))
493.
494.
```

The drawPiece() function will draw the boxes of a piece according to the piece data structure that is passed to it. This function will be used to draw the falling piece and the “Next” piece. Since the piece data structure will contain all of the shape, position, rotation, and color information, nothing else besides the piece data structure needs to be passed to the function.

However, the “Next” piece is not drawn on the board. In this case, we ignore the position information in the piece data structure and instead let the caller of the drawPiece() function pass in arguments for the optional pixelx and pixely parameters to specify where exactly on the window the piece should be drawn.

If no pixelx and pixely arguments are passed in, then lines 484 and 486 will overwrite those variables with the return values of convertToPixelCoords() call.

The nested for loops on line 489 and 490 will then call drawBox() for each box of the piece that needs to be drawn.

Draw the “Next” Piece

```
495. def drawNextPiece(piece):
496.     # draw the "next" text
497.     nextSurf = BASICFONT.render('Next:', True, TEXTCOLOR)
498.     nextRect = nextSurf.get_rect()
499.     nextRect.topleft = (WINDOWWIDTH - 120, 80)
```

```
500.     WINDOWSURF.blit(nextSurf, nextRect)
501.     # draw the "next" piece
502.     drawPiece(piece, pixelx=WINDOWWIDTH-120, pixely=100)
503.
504.
505. if __name__ == '__main__':
506.     main()
```

The drawNextPiece() draws the “Next” piece in the upper right corner of the screen. It does this by calling the drawPiece() function and passing in arguments for drawPiece()’s pixelx and pixely parameters.

That’s the last function. Line 505 and 506 are run after all the function definitions have been executed, and then the main() function is called to begin the main part of the program.

Summary

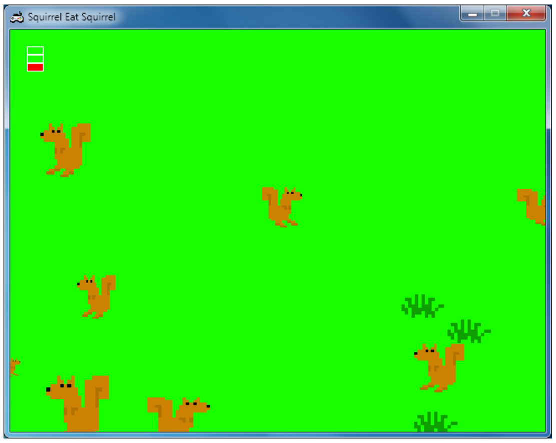
The Tetromino game (which is a clone of the more popular game, “Tetris”) is pretty easy to explain to someone in English: “Blocks fall from the top of a board, and the player moves and rotates them so that they form complete lines. The complete lines disappear (giving the player points) and the lines above them move down. The game keeps going until the blocks fill up the entire board and the player loses.”

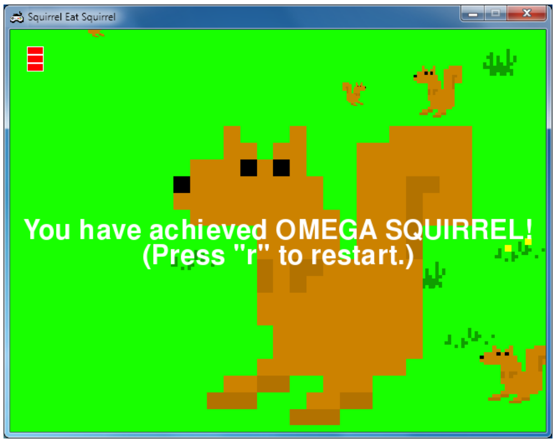
Explaining it in plain English is one thing, but we realize that when we have to tell a computer exactly what to do, there are many details we have to fill in. The original Tetris game was designed and programmed one person, Alex Pajitnov, in the Soviet Union in 1984. The game is simple, fun, and addictive. It is one of the most popular video games ever made, and has sold 100 million copies with many people creating their own clones and variations of it.

And it was all created by one person who knew how to program.

The story of Tetris’s creation and growth shows how with the right idea and some programming knowledge can create incredibly fun games. And with some practice, you will be able to turn your game ideas into real programs that might become as popular as Tetris!

Chapter 8 – Squirrel Eat Squirrel





How to Play Squirrel Eat Squirrel

Squirrel Eat Squirrel is an original game idea I came up with. The player controls a small squirrel that must hop around the screen eating smaller squirrels and avoiding larger squirrels. Each time the player’s squirrel eats a squirrel that is smaller than it, it grows larger. If the player’s squirrel gets hit by a larger squirrel, it loses a life point. The player wins when the squirrel becomes a monstrously large squirrel called the Omega Squirrel. The player loses if her squirrel gets hit three times.

I’m not really sure where I got the idea for a video game in which squirrels eat each other. I’m a little strange sometimes.

The Design of Squirrel Eat Squirrel

TODO

Source Code to Squirrel Eat Squirrel

This source code can be downloaded from http://invpy.com/squirrel.py.

The Usual Setup Code

```python
# Squirrel Eat Squirrel (a 2D Katamari Damacy clone)
# By Al Sweigart al@inventwithpython.com
# http://inventwithpython.com/pygame
# Creative Commons BY-NC-SA 3.0 US

import random, sys, time, math, pygame
from pygame.locals import *

FPS = 30 # frames per second to update the screen
WINWIDTH = 640 # width of the program's window, in pixels
WINHEIGHT = 480 # height in pixels
HALF_WINWIDTH = int(WINWIDTH / 2)
HALF_WINHEIGHT = int(WINHEIGHT / 2)

GRASSCOLOR = (24, 255, 0)
WHITE = (255, 255, 255)
RED = (255, 0, 0)

CAMERASLACK = 60     # how far from the center the squirrel moves before moving the camera
MOVERATE = 9         # how fast the player moves
BOUNCERATE = 6       # how fast the player bounces (large is slower)
BOUNCEHEIGHT = 30    # how high the player bounces
STARTSIZE = 25       # how big the player starts off
WINSIZE = 300        # how big the player needs to be to win
INVULNTIME = 2       # how long the player is invulnerable after being hit in seconds
GAMEOVERTIME = 4     # how long the "game over" text stays on the screen in seconds
MAXHEALTH = 3        # how much health the player starts with

NUMGRASS = 40        # number of grass tiles in the area
NUMSQUIRRELS = 30    # number of squirrels in the area
SQUIRRELMINSPEED = 3 # slowest squirrel speed
SQUIRRELMAXSPEED = 7 # fastest squirrel speed
DIRCHANGEFREQ = 2    # % chance of direction change per frame
LEFT = 'left'
RIGHT = 'right'

```

Describing the Data Structures

```
"""
This program has three data structures to represent the player, enemy squirrels, and grass background objects. The data structures are dictionaries with the following keys:

Keys used by all three data structures:
    'x' - the left edge coordinate of the object in the game world (not a pixel coordinate on the screen)
    'y' - the top edge coordinate of the object in the game world (not a pixel coordinate on the screen)
    'rect' - the pygame.Rect object representing where on the screen the object is located.
Player data structure keys:
    'surface' - the pygame.Surface object that stores the image of the squirrel which will be drawn to the screen.
    'facing' - either set to LEFT or RIGHT, stores which direction the player is facing.
    'size' - the width and height of the player in pixels. (The width & height are always the same.)
    'bounce' - represents at what point in a bounce the player is in. 0 means standing (no bounce), up to BOUNCERATE (the completion of the bounce)
    'health' - an integer showing how many more times the player can be hit by a larger squirrel before dying.
Enemy Squirrel data structure keys:
    'surface' - the pygame.Surface object that stores the image of the squirrel which will be drawn to the screen.
    'movex' - how many pixels per frame the squirrel moves horizontally. A negative integer is moving to the left, a positive to the right.
    'movey' - how many pixels per frame the squirrel moves vertically. A negative integer is moving up, a positive moving down.
    'width' - the width of the squirrel's image, in pixels
    'height' - the height of the squirrel's image, in pixels
    'bounce' - represents at what point in a bounce the player is in. 0 means standing (no bounce), up to BOUNCERATE (the completion of the bounce)
    'bouncerate' - how quickly the squirrel bounces. A lower number means a quicker bounce.
    'bounceheight' - how high (in pixels) the squirrel bounces
Grass data structure keys:
    'grassImage' - an integer that refers to the index of the pygame.Surface object in GRASSIMAGES used for this grass object
"""

```

The main() Function

```
def main():
    global FPSCLOCK, WINDOWSURF, BASICFONT, L_SQUIR_IMG, R_SQUIR_IMG, GRASSIMAGES

    pygame.init()
    FPSCLOCK = pygame.time.Clock()
    pygame.display.set_icon(pygame.image.load('gameicon.png'))
    WINDOWSURF = pygame.display.set_mode((WINWIDTH, WINHEIGHT))
    pygame.display.set_caption('Squirrel Eat Squirrel')
    BASICFONT = pygame.font.Font('freesansbold.ttf', 32)

    # load the image files
    L_SQUIR_IMG = pygame.image.load('squirrel.png')
    R_SQUIR_IMG = pygame.transform.flip(L_SQUIR_IMG, True, False)
    GRASSIMAGES = []
    for i in range(1, 5):
        GRASSIMAGES.append(pygame.image.load('grass%s.png' % i))

    while True:
        runGame()


```

A More Detailed Game State than Usual

```
def runGame():
    invulnerableMode = False  # if the player is invulnerable
    invulnerableStartTime = 0 # time the player became invulnerable
    gameOverMode = False      # if the player has lost
    gameOverStartTime = 0     # time the player lost
    winMode = False           # if the player has won
    hasPressedR = False       # if the player pressed "R" after winning

```

The Usual Text Creation Code

```
    # create the surfaces to hold game text
    gameOverSurf = BASICFONT.render('Game Over', True, WHITE)
    gameOverRect = gameOverSurf.get_rect()
    gameOverRect.center = (HALF_WINWIDTH, HALF_WINHEIGHT)

    winSurf = BASICFONT.render('You have achieved OMEGA SQUIRREL!', True, WHITE)
    winRect = winSurf.get_rect()
    winRect.center = (HALF_WINWIDTH, HALF_WINHEIGHT)

    winSurf2 = BASICFONT.render('(Press "r" to restart.)', True, WHITE)
    winRect2 = winSurf2.get_rect()
    winRect2.center = (HALF_WINWIDTH, HALF_WINHEIGHT + 30)

```

The Camera

```
    # screenx and screeny are where the middle of the camera is
    screenx = HALF_WINWIDTH
    screeny = HALF_WINHEIGHT

```

Keeping Track of the Location of Things in the Game World

```
    grassObjs = []    # stores all the grass objects in the game
    squirrelObjs = [] # stores all the non-player squirrel objects
    # stores the player object:
    playerObj = {'surface': pygame.transform.scale(L_SQUIR_IMG, (STARTSIZE, STARTSIZE)),
                 'facing': LEFT,
                 'size': STARTSIZE,
                 'x': HALF_WINWIDTH - int(STARTSIZE / 2),
                 'y': HALF_WINHEIGHT - int(STARTSIZE / 2),
                 'bounce':0,
                 'health': MAXHEALTH}

    moveLeft  = False
    moveRight = False
    moveUp    = False
    moveDown  = False

```

Start Off with Some Grass

```
    # start off with some random grass images on the screen
    for i in range(5):
        grassObjs.append(makeNewGrass(screenx, screeny, grassObjs))
        grassObjs[i]['x'] = random.randint(0, WINWIDTH)
        grassObjs[i]['y'] = random.randint(0, WINHEIGHT)

```

The Game Loop

```
    while True: # main game loop
```

Disable Invulnerability Check

```
        # Check if we should turn off invulnerability
        if invulnerableMode and time.time() - invulnerableStartTime > INVULNTIME:
            invulnerableMode = False

```

Moving the Enemy Squirrels

```
        # move all the squirrels
        for sObj in squirrelObjs:
            # move the squirrel, and adjust for their bounce
            sObj['x'] += sObj['movex']
            sObj['y'] += sObj['movey']
            sObj['bounce'] += 1
            if sObj['bounce'] > sObj['bouncerate']:
                sObj['bounce'] = 0 # reset bounce amount

            # random chance they change direction
            if random.randint(0, 99) < DIRCHANGEFREQ:
                sObj['movex'] = getRandomVelocity()
                sObj['movey'] = getRandomVelocity()
                if sObj['movex'] > 0:
                    imageToUse = R_SQUIR_IMG
                else:
                    imageToUse = L_SQUIR_IMG
                sObj['surface'] = pygame.transform.scale(imageToUse,
(sObj['width'], sObj['height']))

```

Removing the Far Away Grass and Squirrel Objects

```
        # go through all the objects and see if any need to be deleted.
        for i in range(len(grassObjs) - 1, -1, -1):
            if isOutOfBounds(screenx, screeny, grassObjs[i]):
                del grassObjs[i]
        for i in range(len(squirrelObjs) - 1, -1, -1):
            if isOutOfBounds(screenx, screeny, squirrelObjs[i]):
                del squirrelObjs[i]

```

Adding New Grass and Squirrel Objects

```
        # add more grass & squirrels if we don't have enough.
        while len(grassObjs) < NUMGRASS:
            grassObjs.append(makeNewGrass(screenx, screeny, grassObjs))
        while len(squirrelObjs) < NUMSQUIRRELS:
            squirrelObjs.append(makeNewSquirrel(screenx, screeny))

        # adjust screenx and screeny if beyond the "camera slack"
        centerx = playerObj['x'] + int(playerObj['size'] / 2)
        centery = playerObj['y'] + int(playerObj['size'] / 2)
        if screenx - centerx > CAMERASLACK:
            screenx = centerx + CAMERASLACK
        elif centerx - screenx > CAMERASLACK:
            screenx = centerx - CAMERASLACK
        if screeny - centery > CAMERASLACK:
            screeny = centery + CAMERASLACK
        elif centery - screeny > CAMERASLACK:
            screeny = centery - CAMERASLACK

```

Drawing the Background, Grass, Squirrels, and Health Meter

```
        # draw the green background
        WINDOWSURF.fill(GRASSCOLOR)

        # draw all the grass objects on the screen
        for gObj in grassObjs:
            if isOnScreen(screenx, screeny, gObj):
                gRect = pygame.Rect( (gObj['x'] - screenx + HALF_WINWIDTH,
                                      gObj['y'] - screeny + HALF_WINHEIGHT,
                                      gObj['width'],
                                      gObj['height']) )
                WINDOWSURF.blit(GRASSIMAGES[gObj['grassImage']], gRect)

        # draw the other squirrels
        for sObj in squirrelObjs:
            if isOnScreen(screenx, screeny, sObj):
                sObj['rect'] = pygame.Rect( (sObj['x'] - screenx + HALF_WINWIDTH,
                                             sObj['y'] - screeny + HALF_WINHEIGHT - getBounceAmount(sObj['bounce'], sObj['bouncerate'], sObj['bounceheight']),
                                             sObj['width'],
                                             sObj['height']) )
                WINDOWSURF.blit(sObj['surface'], sObj['rect'])

        # draw the player squirrel
        flashIsOn = round(time.time(), 1) * 10 % 2 == 1.0
        if not gameOverMode and (not invulnerableMode or flashIsOn):
            playerObj['rect'] = pygame.Rect( (playerObj['x'] - screenx + HALF_WINWIDTH,
                                              playerObj['y'] - screeny + HALF_WINHEIGHT,
                                              playerObj['size'],
                                              playerObj['size']) )
            playerObj['rect'].top -= getBounceAmount(playerObj['bounce'], BOUNCERATE, BOUNCEHEIGHT)
            WINDOWSURF.blit(playerObj['surface'], playerObj['rect'])

        # draw the health meter
        drawHealthMeter(playerObj['health'], MAXHEALTH)

```

The Event Handling Loop

```
        for event in pygame.event.get(): # event handling loop
            if event.type == QUIT:
                terminate()
            elif event.type == KEYDOWN:
                if event.key == K_LEFT or event.key == ord('a'):
                    moveRight = False
                    moveLeft = True
                    if playerObj['facing'] != LEFT: # change player image
                        playerObj['surface'] = pygame.transform.scale(L_SQUIR_IMG, (playerObj['size'], playerObj['size']))
                    playerObj['facing'] = LEFT
                elif event.key == K_RIGHT or event.key == ord('d'):
                    moveLeft = False
                    moveRight = True
                    if playerObj['facing'] != RIGHT: # change player image
                        playerObj['surface'] = pygame.transform.scale(R_SQUIR_IMG, (playerObj['size'], playerObj['size']))
                    playerObj['facing'] = RIGHT
```

```
                elif event.key == K_UP or event.key == ord('w'):
                    moveDown = False
                    moveUp = True
                elif event.key == K_DOWN or event.key == ord('s'):
                    moveUp = False
                    moveDown = True
                elif winMode and event.key == ord('r'):
                    hasPressedR = True
```

```
            elif event.type == KEYUP:
                # stop moving the player's squirrel
                if event.key == K_LEFT or event.key == ord('a'):
                    moveLeft = False
                elif event.key == K_RIGHT or event.key == ord('d'):
                    moveRight = False
                elif event.key == K_UP or event.key == ord('w'):
                    moveUp = False
                elif event.key == K_DOWN or event.key == ord('s'):
                    moveDown = False
```

```
                elif event.key == K_ESCAPE:
                    terminate()
```

If the key that was pressed was the Esc key, then terminate the program.

Moving the Player, and Accounting for Bounce

```
        if not gameOverMode:
            # actually move the player
            if moveLeft:
                playerObj['x'] -= MOVERATE
            if moveRight:
                playerObj['x'] += MOVERATE
            if moveUp:
                playerObj['y'] -= MOVERATE
            if moveDown:
```

```
264.                 playerObj['y'] += MOVERATE
```

The code inside the if statement on line 255 will move the player's squirrel around only if the game is not over. (This is why pressing on the arrow keys after the player's squirrel dies will have no effect.) Depending on which of the move variables is set to True, the playerObj dictionary should have its playerObj['x'] and playerObj['y'] values changed by MOVERATE. (This is why a larger value in MOVERATE makes the squirrel move faster.)

```
266.             if (moveLeft or moveRight or moveUp or moveDown) or
playerObj['bounce'] != 0:

267.                 playerObj['bounce'] += 1
268.
269.             if playerObj['bounce'] > BOUNCERATE:
270.                 playerObj['bounce'] = 0 # reset bounce amount
```

Collision Detection: Eat or Be Eaten

```
272.             # check if the player has collided with any squirrels
273.             for i in range(len(squirrelObjs)-1, -1, -1):
274.                 sqObj = squirrelObjs[i]
275.                 if 'rect' in sqObj and
playerObj['rect'].colliderect(sqObj['rect']):

276.                     # a player/squirrel collision has occurred
```

```
279.                     if sqObj['width'] * sqObj['height'] <=
playerObj['size']**2:

280.                         # player is larger and eats the squirrel
281.                         playerObj['size'] += int( (sqObj['width'] *
sqObj['height'])**0.2 ) + 1

282.                         del squirrelObjs[i]
```

```
283.                         if playerObj['facing'] == LEFT:
```

```
                            playerObj['surface'] =
pygame.transform.scale(L_SQUIR_IMG, (playerObj['size'], playerObj['size']))

                        if playerObj['facing'] == RIGHT:
                            playerObj['surface'] =
pygame.transform.scale(R_SQUIR_IMG, (playerObj['size'], playerObj['size']))
```

```
                        if playerObj['size'] > WINSIZE:
                            winMode = True # turn on "win mode"
```

```
                    elif not invulnerableMode:
                        # player is smaller and takes damage
                        invulnerableMode = True
                        invulnerableStartTime = time.time()
                        playerObj['health'] -= 1
                        if playerObj['health'] == 0:
                            gameOverMode = True # turn on "game over mode"
                            gameOverStartTime = time.time()
```

Game Over Screen

```
        else:
            # game is over, show "game over" text
            WINDOWSURF.blit(gameOverSurf, gameOverRect)
            if time.time() - gameOverStartTime > GAMEOVERTIME:
                return # end the current game

```

Winning

```
        # check if the player has won.
        if winMode:
            WINDOWSURF.blit(winSurf, winRect)
            WINDOWSURF.blit(winSurf2, winRect2)
            if hasPressedR:
                return

        pygame.display.update()
        FPSCLOCK.tick(FPS)


```

Drawing a Graphical Health Meter

```
def drawHealthMeter(currentHealth, MAXHEALTH):
    for i in range(currentHealth): # draw red health bars
        pygame.draw.rect(WINDOWSURF, RED,   (15, 5 + (10 * MAXHEALTH) - i
* 10, 20, 10))

    for i in range(MAXHEALTH): # draw the white outlines
        pygame.draw.rect(WINDOWSURF, WHITE, (15, 5 + (10 * MAXHEALTH) - i
* 10, 20, 10), 1)


```

The Same Old terminate() Function

```
def terminate():
    pygame.quit()
    sys.exit()
```

The Mathematics of the Sine Function

```
def getBounceAmount(currentBounce, bounceRate, bounceHeight):
    # Returns the number of pixels to offset based on the bounce.
    # Larger bounceRate means a slower bounce.
    # Larger bounceHeight means a higher bounce.
    # currentBounce will always be less than bounceRate
    return int(math.sin( ((currentBounce/float(bounceRate)) * math.pi) ) *
bounceHeight)

```

Backwards Compatibility with Python Version 2

The reason we call float() to convert bounceRate to a floating point number is simply so that this program will work in Python version 2. In Python version 3, the division operator will evaluate to a floating point value even if both of the operands are integers, like this:

```
>>> # Python version 3
...
>>> 10 / 5
2.0
>>> 10 / 4
2.5
>>>
```

However, in Python version 2, the / division operator will only evaluate to a floating point value if one of the operands is also a floating point value. If both operands are integers, then Python 2's division operator will evaluate to an integer value (rounding down if needed), like this:

```
>>> # Python version 2
...
>>> 10 / 5
2
>>> 10 / 4
2
>>> 10 / 5.0
2.0
>>> 10.0 / 5
2.0
>>> 10.0 / 5.0
2.0
```

But if we always convert one of the values to a floating point value with the float() function, then the division operator will evaluate to a float value no matter which version of Python runs this source code. Making these changes so that our code works with older versions of software is called **backwards compatibility**. It is important to maintain backwards compatibility, because not everyone will always be running the latest version of software and you want to ensure that the code you write works with as many computers as possible.

The getRandomVelocity() Function

```
def getRandomVelocity():
    speed = random.randint(SQUIRRELMINSPEED, SQUIRRELMAXSPEED)
    if random.randint(0, 1) == 0:
        return speed
    else:
        return -speed


```

Finding a Place to Add New Squirrels and Grass

```
def getRandomOffScreenPos(screenx, screeny, objWidth, objHeight):
    # Create the initial starting position outside of the screen's view.
    screenLeftEdge = screenx - HALF_WINWIDTH
    screenTopEdge = screeny - HALF_WINHEIGHT
    boundsRect = pygame.Rect(screenLeftEdge, screenTopEdge, WINWIDTH, WINHEIGHT)
    while True:
        x = random.randint(screenx - WINWIDTH, screenx + WINWIDTH)
        y = random.randint(screeny - WINHEIGHT, screeny + WINHEIGHT)
        objRect = pygame.Rect(x, y, objWidth, objHeight)
        if not objRect.colliderect(boundsRect):
            return x, y


```

Creating Enemy Squirrel Data Structures

```
def makeNewSquirrel(screenx, screeny):
    sq = {}
    generalSize = random.randint(5, 25)
    multiplier = random.randint(1, 3)
    sq['width']  = (generalSize + random.randint(0, 10)) * multiplier
    sq['height'] = (generalSize + random.randint(0, 10)) * multiplier
    sq['x'], sq['y'] = getRandomOffScreenPos(screenx, screeny, sq['width'], sq['height'])
    sq['movex'] = getRandomVelocity()
    sq['movey'] = getRandomVelocity()
    if sq['movex'] > 0: # squirrel is facing right
        sq['surface'] = pygame.transform.flip(pygame.transform.scale(L_SQUIR_IMG, (sq['width'], sq['height'])), True, False)
```

Flipping the Squirrel Image

```
    else: # squirrel is facing left
        sq['surface'] = pygame.transform.scale(L_SQUIR_IMG, (sq['width'], sq['height']))

    sq['bounce'] = 0
    sq['bouncerate'] = random.randint(10, 18)
    sq['bounceheight'] = random.randint(10, 50)
    return sq


```

Creating Grass Data Structures

```
def makeNewGrass(screenx, screeny, grassObjs):
    gr = {}
    gr['grassImage'] = random.randint(0, len(GRASSIMAGES) - 1)
    gr['width']  = GRASSIMAGES[0].get_width()
    gr['height'] = GRASSIMAGES[0].get_height()
    while True:
        gr['x'], gr['y'] = getRandomOffScreenPos(screenx, screeny, gr['width'], gr['height'])

        gr['rect'] = pygame.Rect( (gr['x'], gr['y'], gr['width'], gr['height']) )

        for i in range(len(grassObjs)):
            if gr['rect'].colliderect(grassObjs[i]['rect']):
                continue
        break
    return gr


```

Checking if Something is in the Camera's View

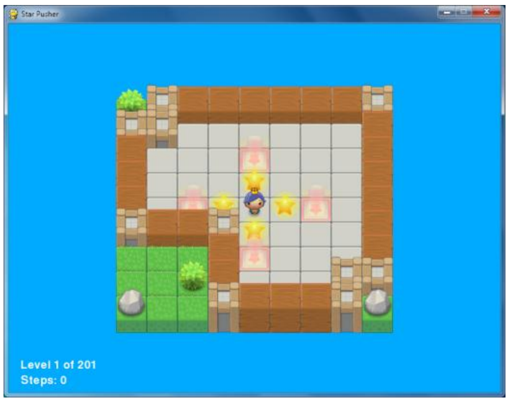
```
def isOnScreen(screenx, screeny, obj):
    screenRect = pygame.Rect(screenx - HALF_WINWIDTH, screeny - HALF_WINHEIGHT, WINWIDTH, WINHEIGHT)
    objRect = pygame.Rect(obj['x'], obj['y'], obj['width'], obj['height'])
    return screenRect.colliderect(objRect)


def isOutOfBounds(screenx, screeny, obj):
    # Return False if screenx and screeny are more than
    # a half-window length beyond the edge of the window.
    boundsLeftEdge = screenx - (HALF_WINWIDTH + WINWIDTH)
    boundsTopEdge = screeny - (HALF_WINHEIGHT + WINHEIGHT)
    boundsRect = pygame.Rect(boundsLeftEdge, boundsTopEdge, WINWIDTH * 3, WINHEIGHT * 3)
    objRect = pygame.Rect(obj['x'], obj['y'], obj['width'], obj['height'])
    return not boundsRect.colliderect(objRect)


if __name__ == '__main__':
    main()
```

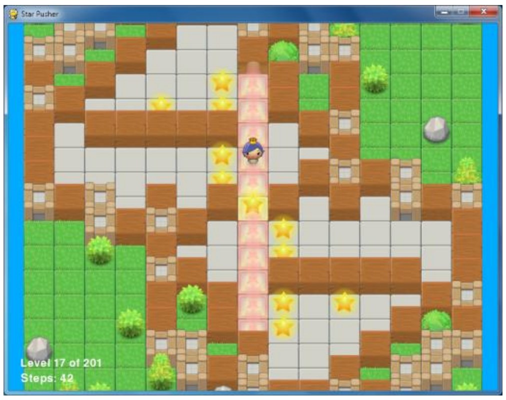
Camera Slack

Sine & Cosine

List of things to change to experiment (this should go in each chapter)

Chapter 9 – Star Pusher





How to Play Star Pusher

Star Pusher is a Sokoban or "Box Pusher" clone. The player is in a room with several stars. There are star marks on the grounds of some of the tiles in the room. The player must figure out how to push the stars on top of the tiles with star marks. The player cannot pull stars, so if a star gets pushed into a corner, the player will have to restart the level. The player cannot push a star if there is a wall or another star behind it. When all of the star-marked tiles have stars on them, the level is complete and the next level starts.

Source Code to Star Pusher

This source code can be downloaded from http://invpy.com/starpusher.py.

```
# Star Pusher (a Sokoban clone)
# By Al Sweigart al@inventwithpython.com
# http://inventwithpython.com/pygame
# Creative Commons BY-NC-SA 3.0 US

import random, sys, copy, os, pygame
from pygame.locals import *

FPS = 30 # frames per second to update the screen
WINWIDTH = 800 # width of the program's window, in pixels
WINHEIGHT = 600 # height in pixels
HALF_WINWIDTH = int(WINWIDTH / 2)
HALF_WINHEIGHT = int(WINHEIGHT / 2)

```

```
# The total width and height of each tile in pixels.
TILEWIDTH = 50
TILEHEIGHT = 85
TILEOFFSET = 45

CAM_MOVE_SPEED = 5 # how many pixels per frame the camera moves

# The percentage of outdoor tiles that have additional
# decoration on them, such as a tree or rock.
OUTSIDE_DECORATION_PCT = 20

BRIGHTBLUE = (  0, 170, 255)
WHITE      = (255, 255, 255)
BGCOLOR = BRIGHTBLUE
TEXTCOLOR = WHITE

UP = 'up'
DOWN = 'down'
LEFT = 'left'
RIGHT = 'right'
```

```
def main():
    global FPSCLOCK, WINDOWSURF, IMAGESDICT, TILEMAPPING, OUTSIDEDECOMAPPING, BASICFONT, PLAYERIMGMAPPING

    # Pygame initialization and basic set up of the global varaibles.
    pygame.init()
    FPSCLOCK = pygame.time.Clock()

    # Because the Surface object stored in WINDOWSURF was returned
    # from the pygame.display.set_mode() function, this is the
    # Surface object that is drawn to the actual computer screen
    # when pygame.display.update() is called.
    WINDOWSURF = pygame.display.set_mode((WINWIDTH, WINHEIGHT))

    pygame.display.set_caption('Star Pusher')
    BASICFONT = pygame.font.Font('freesansbold.ttf', 18)
```

```
    # A global dict value that will contain all the Pygame
    # Surface objects returned by pygame.image.load().
    IMAGESDICT = {'uncovered goal': pygame.image.load('RedSelector.png'),
                  'covered goal': pygame.image.load('Selector.png'),
                  'star': pygame.image.load('Star.png'),
                  'corner': pygame.image.load('Wall Block Tall.png'),
                  'wall': pygame.image.load('Wood Block Tall.png'),
                  'inside floor': pygame.image.load('Plain Block.png'),
                  'outside floor': pygame.image.load('Grass Block.png'),
                  'title': pygame.image.load('star_title.png'),
                  'solved': pygame.image.load('star_solved.png'),
                  'princess': pygame.image.load('Character_Princess_Girl.png'),
                  'boy': pygame.image.load('Character_Boy.png'),
                  'catgirl': pygame.image.load('Character_Cat_Girl.png'),
                  'horngirl': pygame.image.load('Character_Horn_Girl.png'),
                  'pinkgirl': pygame.image.load('Character_Pink_Girl.png'),
                  'rock': pygame.image.load('Rock.png'),
                  'short tree': pygame.image.load('Tree_Short.png'),
                  'tall tree': pygame.image.load('Tree_Tall.png'),
                  'ugly tree': pygame.image.load('Tree_Ugly.png')}

    # These dict values are global, and map the character that appears
    # in the level file to the Surface object it represents.
    TILEMAPPING = {'x': IMAGESDICT['corner'],
                   '#': IMAGESDICT['wall'],
                   'o': IMAGESDICT['inside floor'],
                   ' ': IMAGESDICT['outside floor']}
    OUTSIDEDECOMAPPING = {'1': IMAGESDICT['rock'],
                          '2': IMAGESDICT['short tree'],
                          '3': IMAGESDICT['tall tree'],
                          '4': IMAGESDICT['ugly tree']}
    # The mapping for the player uses integer keys because we do some math
    # with it later. This dict value also has a 'currentImg' key which
    # keeps track of the current image that represents the player. The
    # value of 'currentImg' acts as a key to the Surface object.
    PLAYERIMGMAPPING = {'currentImg': 0,
                        0: IMAGESDICT['princess'],
                        1: IMAGESDICT['boy'],
                        2: IMAGESDICT['catgirl'],
                        3: IMAGESDICT['horngirl'],
                        4: IMAGESDICT['pinkgirl']}
```

```
    startScreen() # show the title screen until the user presses a key

    # Read in the levels from the text file. See the readLevelsFile() for
    # details on the format of this file and how to make your own levels.
    levels = readLevelsFile('starPusherLevels.txt')
    levelNum = 0
```

```
    # The main game loop. This loop runs a single level, when the user
    # finishes that level, the next/previous level is loaded.
    while True: # main game loop
        # Run the level to actually start playing the game:
        result = runLevel(levels[levelNum], len(levels))
```

The runLevel() function handles all the action for the game. When the player has finished playing the level, runLevel() will return one of the following strings: 'solved' (because the player has finished putting all the stars on the goals), 'next' (because the player wants to skip to the next level), 'back' (because the player wants to go back to the previous level), and 'reset' (because the player wants to start playing the current level over again, probably because they pushed a star into a corner).

```
        if result in ('solved', 'next'):
            # Go to the next level.
            levelNum += 1
            if levelNum >= len(levels):
                # If there are no more levels, go back to the first one.
                levelNum = 0
        elif result == 'back':
            # Go to the previous level.
            levelNum -= 1
            if levelNum < 0:
                # If there are no previous levels, go to the last one.
                levelNum = len(levels)-1
```

If runLevel() has returned the strings 'solved' or 'next', then we need to increment levelNum by 1. If this increments levelNum beyond the number of levels there are, then levelNum is set back at 0.

The opposite is done if 'back' is returned, then levelNum is decremented by 1. If this makes it go below 0, then it is set to the last level (which is len(levels)-1).

```
        elif result == 'reset':
            pass # Do nothing. Loop re-calls runLevel() to reset the level
```

If the player has pressed the reset key, then the code does nothing. On the next iteration the loop will call runLevel() again and start the level over from the beginning. The pass statement does nothing (like a comment), but is needed because the Python interpreter expects an indented line of code after an elif statement.

We could remove lines 120 and 121 from the source code entirely, and the program will still work just the same. The reason we include it here is for program readability, so that if we make changes to the code later, we won’t forget that runLevel() can also return the string 'reset'.

```
def runLevel(levelObj, totalNumOfLevels):
    mapObj = getDecoratedMap(levelObj)
    gameStateObj = copy.deepcopy(levelObj['startState'])
    mapNeedsRedraw = True # set to True to call drawMap()
    levelSurf = BASICFONT.render('Level %s of %s' % (levelObj['levelNum'] + 1, totalNumOfLevels), 1, TEXTCOLOR)
    levelRect = levelSurf.get_rect()
    levelRect.bottomleft = (20, WINHEIGHT - 35)

    mapWidth = len(mapObj) * TILEWIDTH
    mapHeight = (len(mapObj[0]) - 1) * (TILEHEIGHT - TILEOFFSET) + TILEHEIGHT
    MAX_CAM_X_PAN = abs(HALF_WINHEIGHT - int(mapHeight / 2)) + TILEWIDTH
    MAX_CAM_Y_PAN = abs(HALF_WINWIDTH - int(mapWidth / 2)) + TILEHEIGHT

    levelIsComplete = False
    # Track how much the camera has moved:
    cameraOffsetX = 0
    cameraOffsetY = 0
    # Track if the keys to move the camera are being held down:
    cameraUp = False
    cameraDown = False
    cameraLeft = False
    cameraRight = False

    while True: # main game loop
        # Reset these variables:
        playerMoveTo = None
        keyPress = False

        for event in pygame.event.get(): # event handling loop
            if event.type == QUIT:
                # Player clicked the "X" at the corner of the window.
                terminate()

            elif event.type == KEYDOWN:
                # Handle key presses
                keyPress = True
                if event.key == K_LEFT:
                    playerMoveTo = LEFT
                elif event.key == K_RIGHT:
                    playerMoveTo = RIGHT
                elif event.key == K_UP:
                    playerMoveTo = UP
                elif event.key == K_DOWN:
                    playerMoveTo = DOWN

                # Set the camera move mode.
                elif event.key == ord('a'):
                    cameraLeft = True
                elif event.key == ord('d'):
                    cameraRight = True
                elif event.key == ord('w'):
                    cameraUp = True
                elif event.key == ord('s'):
                    cameraDown = True

                elif event.key == ord('n'):
                    return 'next'
                elif event.key == ord('b'):
                    return 'back'

                elif event.key == K_ESCAPE:
                    terminate() # Esc key quits.
                elif event.key == K_BACKSPACE:
                    return 'reset' # Reset the level.
                elif event.key == ord('p'):
                    # Change the player image to the next one.
                    PLAYERIMGMAPPING['currentImg'] += 1
                    if PLAYERIMGMAPPING['currentImg'] not in PLAYERIMGMAPPING:
                        # After the last player image, use the first one.
                        PLAYERIMGMAPPING['currentImg'] = 0
                    mapNeedsRedraw = True

            elif event.type == KEYUP:
                # Unset the camera move mode.
                if event.key == ord('a'):
                    cameraLeft = False
                elif event.key == ord('d'):
                    cameraRight = False
                elif event.key == ord('w'):
                    cameraUp = False
                elif event.key == ord('s'):
                    cameraDown = False

        if playerMoveTo != None and not levelIsComplete:
            # If the player pushed a key to move, make the move
            # (if possible) and push any stars that are pushable.
            moved = makeMove(mapObj, gameStateObj, playerMoveTo)

            if moved:
                # increment the step counter.
                gameStateObj['stepCounter'] += 1
                mapNeedsRedraw = True

            if isLevelFinished(levelObj, gameStateObj):
                # level is solved, we should show the "Solved!" image.
                levelIsComplete = True
                keyPress = False

        WINDOWSURF.fill(BGCOLOR)

```

```
        if mapNeedsRedraw:
            mapSurf = drawMap(mapObj, gameStateObj, levelObj['goals'])
            mapNeedsRedraw = False

```

```
        if cameraUp and cameraOffsetY < MAX_CAM_X_PAN:
            cameraOffsetY += CAM_MOVE_SPEED
        elif cameraDown and cameraOffsetY > -MAX_CAM_X_PAN:
            cameraOffsetY -= CAM_MOVE_SPEED
        if cameraLeft and cameraOffsetX < MAX_CAM_Y_PAN:
            cameraOffsetX += CAM_MOVE_SPEED
        elif cameraRight and cameraOffsetX > -MAX_CAM_Y_PAN:
            cameraOffsetX -= CAM_MOVE_SPEED

```

If the camera movement variables are set to True and the camera has not gone past (i.e. panned passed) the boundaries set by the MAX_CAM_X_PAN and MAX_CAM_Y_PAN, then the camera location (stored in cameraOffsetX and cameraOffsetY) should move over by CAM_MOVE_SPEED pixels.

Note that there is an if and elif statement on lines 228 and 230 for moving the camera up and down, and then a separate if and elif statement on lines 232 and 234. This way, the user can move the camera both vertically and horizontally at the same time. This wouldn't be possible if line 232 were an elif statement.

```
237.         # Adjust mapSurf's Rect object based on the camera offset.
238.         mapSurfRect = mapSurf.get_rect()
239.         mapSurfRect.center = (HALF_WINWIDTH + cameraOffsetX,
HALF_WINHEIGHT + cameraOffsetY)
240.
241.         # Draw mapSurf to the WINDOWSURF Surface object.
242.         WINDOWSURF.blit(mapSurf, mapSurfRect)
243.
244.         WINDOWSURF.blit(levelSurf, levelRect)
245.         stepSurf = BASICFONT.render('Steps: %s' %
(gameStateObj['stepCounter']), 1, TEXTCOLOR)
246.         stepRect = stepSurf.get_rect()
247.         stepRect.bottomleft = (20, WINHEIGHT - 10)
248.         WINDOWSURF.blit(stepSurf, stepRect)
249.
250.         if levelIsComplete:
251.             # is solved, show the "Solved!" image until the player
252.             # has pressed a key.
253.             solvedRect = IMAGESDICT['solved'].get_rect()
254.             solvedRect.center = (HALF_WINWIDTH, HALF_WINHEIGHT)
255.             WINDOWSURF.blit(IMAGESDICT['solved'], solvedRect)
256.
257.             if keyPress:
258.                 return 'solved'
259.
260.         pygame.display.update() # draw WINDOWSURF to the screen.
261.         FPSCLOCK.tick()
```

```
264. def isWall(mapObj, x, y):
265.     """Returns True if the (x, y) position on
266.     the map is a wall, otherwise return False."""
267.     if x < 0 or x >= len(mapObj) or y < 0 or y >= len(mapObj[x]):
268.         return False # x and y aren't actually on the map.
269.     elif mapObj[x][y] in ('#', 'x'):
270.         return True # wall is blocking
```

```
    return False
```

The isWall() function returns True if there is a wall on the map object at the XY coordinates passed to the function. Wall objects are represented as either 'x' or '#' strings in the map object.

```
def decorateMap(mapObj, startxy):
    """Makes a copy of the given map object and modifies it.
    Here is what is done to it:
        * Walls that are corners are turned into corner pieces.
        * The outside/inside floor tile distinction is made.
        * Tree/rock decorations are randomly added to the outside tiles.

    Returns the decorated map object."""

    startx, starty = startxy # Syntactic sugar

    # Copy the map object so we don't modify the original passed
    mapObjCopy = copy.deepcopy(mapObj)

```

```
    # Remove the non-wall characters from the map data
    for x in range(len(mapObjCopy)):
        for y in range(len(mapObjCopy[0])):
            if mapObjCopy[x][y] in ('$', '.', '@', '+', '*'):
                mapObjCopy[x][y] = ' '

```

```
    # Flood fill to determine inside/outside floor tiles.
    floodFill(mapObjCopy, startx, starty, ' ', 'o')

```

The floodFill() function will change all of the tiles inside the walls from ' ' characters to 'o' characters. It does this using a programming concept called recursion, which is explained in the floodFill() function section.

```
    # Convert the adjoined walls into corner tiles.
    for x in range(len(mapObjCopy)):
        for y in range(len(mapObjCopy[0])):
            if mapObjCopy[x][y] == '#':
                if (isWall(mapObjCopy, x, y-1) and isWall(mapObjCopy, x+1, y)) or \
                   (isWall(mapObjCopy, x+1, y) and isWall(mapObjCopy, x, y+1)) or \
                   (isWall(mapObjCopy, x, y+1) and isWall(mapObjCopy, x-1, y)) or \
                   (isWall(mapObjCopy, x-1, y) and isWall(mapObjCopy, x, y-1)):
                    mapObjCopy[x][y] = 'x'
            elif mapObjCopy[x][y] == ' ' and random.randint(0, 99) < OUTSIDE_DECORATION_PCT:
                mapObjCopy[x][y] = random.choice(list(OUTSIDEDECOMAPPING.keys()))

    return mapObjCopy
```

```
def getDecoratedMap(levelObj):
    """Returns a decorated map object when passed a level object."""
    return decorateMap(levelObj['mapObj'], levelObj['startState']['player'])
```

```
def isBlocked(mapObj, gameStateObj, x, y):
    """Returns True if the (x, y) position on the map is
    blocked by a wall or star, otherwise return False."""
    if isWall(mapObj, x, y):
        return True
    elif x < 0 or x >= len(mapObj) or y < 0 or y >= len(mapObj[x]):
        return True # x and y aren't actually on the map.
    elif (x, y) in gameStateObj['stars']:
        return True # a star is blocking

    return False
```

```
def makeMove(mapObj, gameStateObj, playerMoveTo):
    """Given a map and game state object, see if it is possible for the
    player to make the given move. If it is, then change the player's
    position (and the position of any pushed star). If not, do nothing.

    Returns True if the player moved, otherwise False."""

    # Make sure the player can move in the direction they want.
    playerx, playery = gameStateObj['player']

    # This variable is "syntactic sugar". Typing "stars" is more
    # readable than typing "gameStateObj['stars']" in our code.
    stars = gameStateObj['stars']

    # The code for handling each of the directions is so similar aside
    # from adding or subtracting 1 to the x/y coordinates. We can
    # simplify it by using the xOffset and yOffset variables.
    if playerMoveTo == UP:
        xOffset = 0
        yOffset = -1
    elif playerMoveTo == RIGHT:
        xOffset = 1
        yOffset = 0
    elif playerMoveTo == DOWN:
        xOffset = 0
        yOffset = 1
    elif playerMoveTo == LEFT:
        xOffset = -1
        yOffset = 0

    # See if the player can move in that direction.
    if isWall(mapObj, playerx + xOffset, playery + yOffset):
        return False
    else:
        if (playerx + xOffset, playery + yOffset) in stars:
            # There is a star in the way, see if the player can push it.
            if not isBlocked(mapObj, gameStateObj, playerx + (xOffset*2), playery + (yOffset*2)):
                # Move the star.
                ind = stars.index((playerx + xOffset, playery + yOffset))
                stars[ind] = (stars[ind][0] + xOffset, stars[ind][1] + yOffset)
            else:
                return False
        # Move the player upwards.
        gameStateObj['player'] = (playerx + xOffset, playery + yOffset)
        return True
```

```
def startScreen():
```

```
378.     """Display the start screen (which has the title and instructions)
379.     until the player presses a key. Returns None."""
380.
381.     # Position the title image.
382.     titleRect = IMAGESDICT['title'].get_rect()
383.     topCoord = 50 # topCoord tracks where to position the top of the text
384.     titleRect.top = topCoord
385.     titleRect.centerx = HALF_WINWIDTH
386.     topCoord += titleRect.height
387.
388.     # Unfortunately, Pygame's font & text system only shows one line at
389.     # a time, so we can't use strings with \n newline characters in them.
390.     # So we will use a list with each line in it.
391.     instructionText = ['Push the stars over the marks.',
392.                        'Arrow keys to move, WASD for camera control, P to
change character.',
393.                        'Backspace to reset level, Esc to quit.',
394.                        'N for next level, B to go back a level.']
395.
```

The startScreen() function needs to display a few different pieces of text down the center of the window. We will store each line as a string in the instructionText list. The title image (stored in IMAGESDICT['title'] as a pygame.image object (that was originally loaded from the star_title.png file)) will be positioned 50 pixels from the top. This is because the integer 50 was stored in the topCoord variable on line 383. The topCoord variable will track the Y axis positioning of the title image and the instructional text. (The X axis is always going to be set so that the images and text are centered, as it is on line 385 for the title image.)

On line 386, the topCoord variable is increased by whatever the height of that image is. This way we can modify the image and the start screen code won’t have to be changed.

```
396.     # Start with drawing a blank color to the entire window:
397.     WINDOWSURF.fill(BGCOLOR)
398.
399.     # Draw the title image to the window:
400.     WINDOWSURF.blit(IMAGESDICT['title'], titleRect)
401.
402.     # Position and draw the text.
403.     for i in range(len(instructionText)):
404.         instSurf = BASICFONT.render(instructionText[i], 1, TEXTCOLOR)
405.         instRect = instSurf.get_rect()
406.         topCoord += 10 # 10 pixels will go in between each line of text.
407.         instRect.top = topCoord
408.         instRect.centerx = HALF_WINWIDTH
```

```
409.         topCoord += instRect.height # Adjust for the height of the line.
410.         WINDOWSURF.blit(instSurf, instRect)
411.
```

Line 400 is where the title image is blitted to the display Surface object. The for loop starting on line 403 will render, position, and blit each instructional string in the instructionText loop. The topCoord variable will always be incremented by the size of the previous rendered text and 10 additional pixels (so that there will be a 10 pixel gap between the lines of text.)

```
412.     while True: # Main loop for the start screen.
413.         for event in pygame.event.get():
414.             if event.type == QUIT:
415.                 terminate()
416.             elif event.type == KEYDOWN:
417.                 if event.key == K_ESCAPE:
418.                     terminate()
419.                 return # user has pressed a key, so return.
420.
421.         # Display the WINDOWSURF contents to the actual screen.
422.         pygame.display.update()
423.         FPSCLOCK.tick()
```

There is a game loop in startScreen() that begins on line 412 and handles events that indicate if the program should terminate or return from the startScreen() function. Until the player does either, the loop will keep calling pygame.display.update() and FPSCLOCK.tick() to keep the start screen displayed on the screen. TODO

```
426. def relativePathToThisScript(filename):
427.     """Returns an absolute filepath for a provided filename, relative
428.     to the directory this script's file (starpusher.py) was run from."""
429.
430.     # "relativePathToThisScript" isn't much shorter than
431.     # this expression, but it is more readable.
432.     return os.path.normpath(os.path.join(os.getcwd(), filename))
```

```
435. def readLevelsFile(filename):
436.     assert os.path.exists(filename), 'Cannot find the level file: %s' %
(relativePathToThisScript(filename))
437.     fp = open(filename, 'r')
438.     # Each level must end with a blank line
439.     content = fp.readlines() + ['\r\n']
```

```

    levels = [] # Will contain a list of level objects.
    levelNum = 0

    singleMapLines = [] # contains the lines for a single level's map.
    mapObj = [] # the map object made from the data in singleMapLines
    for lineNum in range(len(content)):
        # Process each line that was in the level file.
        line = content[lineNum].rstrip('\r\n')

        if ';' in line:
            # Ignore the ; lines, they're comments in the level file.
            line = line[:line.find(';')]

        if line != '':
            # This line is part of the map.
            singleMapLines.append(line)
        elif line == '' and len(singleMapLines) > 0:
            # A blank line indicates the end of a level's map in the file.
            # Convert the text in singleMapLines into a level object.

            # Find the longest row in the map.
            maxWidth = -1
            for i in range(len(singleMapLines)):
                if len(singleMapLines[i]) > maxWidth:
                    maxWidth = len(singleMapLines[i])
            # Add spaces to the ends of the shorter rows. This
            # ensures the map will be rectangular.
            for i in range(len(singleMapLines)):
                singleMapLines[i] += ' ' * (maxWidth -
len(singleMapLines[i]))

            # Convert singleMapLines to a map object.
            for x in range(len(singleMapLines[0])):
                mapObj.append([])
            for y in range(len(singleMapLines)):
                for x in range(maxWidth):
                    mapObj[x].append(singleMapLines[y][x])

            # Loop through the spaces in the map and find the @, ., and $
            # characters for the starting game state.
            startx = None # The x and y for the player's starting position
            starty = None
            goals = [] # list of (x, y) tuples for each goal.
            stars = [] # list of (x, y) for each star's starting position.
            for x in range(maxWidth):
                for y in range(len(mapObj[x])):
                    if mapObj[x][y] in ('@', '+'):
                        # '@' is player, '+' is player & goal
                        startx = x
                        starty = y
                    if mapObj[x][y] in ('.', '+', '*'):
                        # '.' is goal, '*' is star & goal
                        goals.append((x, y))
                    if mapObj[x][y] in ('$', '*'):
                        # '$' is star
                        stars.append((x, y))

            # Basic level design sanity checks:
            assert startx != None and starty != None, 'Level %s (around line %s) in %s is missing a "@" or "+" to mark the start point.' % (levelNum+1, lineNum, relativePathToThisScript(filename))
            assert len(goals) > 0, 'Level %s (around line %s) in %s must have at least one goal.' % (levelNum+1, lineNum, relativePathToThisScript(filename))
            assert len(stars) >= len(goals), 'Level %s (around line %s) in %s is impossible to solve. It has %s goals but only %s stars.' % (levelNum+1, lineNum, relativePathToThisScript(filename), len(goals), len(stars))

            # Create level object and starting game state object.
            gameStateObj = {'player': (startx, starty),
                            'stepCounter': 0,
                            'stars': stars}
            levelObj = {'width': maxWidth,
                        'height': len(mapObj),
                        'mapObj': mapObj,
                        'goals': goals,
                        'startState': gameStateObj,
                        'levelNum': levelNum}
            levels.append(levelObj)

            # Reset the variables for reading the next map.
            singleMapLines = []
            mapObj = []
            gameStateObj = {}
            levelNum += 1
    return levels
```

```
def floodFill(mapObj, x, y, oldCharacter, newCharacter):
    """Changes any values matching oldCharacter on the map object to
    newCharacter at the (x, y) position, and does the same for the
    positions to the left, right, down, and up of (x, y), recursively."""

    # In this game, the floodfill algorithm creates the inside/outside
    # floor distinction. This is a "recursive" function.
    # For more info on the Flood Fill algorithm, see:
    #   http://en.wikipedia.org/wiki/Flood_fill
    if mapObj[x][y] == oldCharacter:
        mapObj[x][y] = newCharacter

    if x < len(mapObj) - 1 and mapObj[x+1][y] == oldCharacter:
        floodFill(mapObj, x+1, y, oldCharacter, newCharacter) # call right
    if x > 0 and mapObj[x-1][y] == oldCharacter:
        floodFill(mapObj, x-1, y, oldCharacter, newCharacter) # call left
    if y < len(mapObj[x]) - 1 and mapObj[x][y+1] == oldCharacter:
        floodFill(mapObj, x, y+1, oldCharacter, newCharacter) # call down
    if y > 0 and mapObj[x][y-1] == oldCharacter:
        floodFill(mapObj, x, y-1, oldCharacter, newCharacter) # call up
```

```
def drawMap(mapObj, gameStateObj, goals):
    """Draws the map to a Surface object, including the player and
    stars. This function does not call pygame.display.update(), nor
    does it draw the "Level" and "Steps" text in the corner."""

    # mapSurf will be the single Surface object that the tiles are drawn
    # on, so that it is easy to position the entire map on the WINDOWSURF
    # Surface object. First, the width and height must be calculated.
    mapSurfWidth = len(mapObj) * TILEWIDTH
    mapSurfHeight = (len(mapObj[0]) - 1) * (TILEHEIGHT - TILEOFFSET) + TILEHEIGHT
    mapSurf = pygame.Surface((mapSurfWidth, mapSurfHeight))
    mapSurf.fill(BGCOLOR) # start with a blank color on the surface.

    # Draw the tile sprites onto this surface.
    for x in range(len(mapObj)):
        for y in range(len(mapObj[x])):
            spaceRect = pygame.Rect((x * TILEWIDTH, y * (TILEHEIGHT - TILEOFFSET), TILEWIDTH, TILEHEIGHT))
            if mapObj[x][y] in TILEMAPPING:
                baseTile = TILEMAPPING[mapObj[x][y]]
            elif mapObj[x][y] in OUTSIDEDECOMAPPING:
                baseTile = TILEMAPPING[' ']

            # First draw the base ground/wall tile.
            mapSurf.blit(baseTile, spaceRect)

            if mapObj[x][y] in OUTSIDEDECOMAPPING:
                # Draw any tree/rock decorations that are on this tile.
                mapSurf.blit(OUTSIDEDECOMAPPING[mapObj[x][y]], spaceRect)
            elif (x, y) in gameStateObj['stars']:
                if (x, y) in goals:
                    # A goal AND star are on this space, draw goal first.
                    mapSurf.blit(IMAGESDICT['covered goal'], spaceRect)
                # Then draw the star sprite.
                mapSurf.blit(IMAGESDICT['star'], spaceRect)
            elif (x, y) in goals:
                # Draw a goal without a star on it.
                mapSurf.blit(IMAGESDICT['uncovered goal'], spaceRect)

            # Last draw the player on the board.
            if (x, y) == gameStateObj['player']:
                # Note: The value "PLAYERIMGMAPPING['currentImg']" refers
                # to a key in "PLAYERIMGMAPPING" which has the
                # specific player image we want to show.

mapSurf.blit(PLAYERIMGMAPPING[PLAYERIMGMAPPING['currentImg']], spaceRect)

    return mapSurf
```

```
def isLevelFinished(levelObj, gameStateObj):
    """Returns True if all the goals have stars in them."""
    for goal in levelObj['goals']:
        if goal not in gameStateObj['stars']:
            # Found a space with a goal but no star on it.
            return False
    return True
```

```
def terminate():
    """Shuts down the program."""

    # If you run this script from IDLE, it will freeze up if you
    # don't call pygame.quit() before exiting the program.
    pygame.quit()
    sys.exit()
```

```
if __name__ == '__main__':
    main()
```