OOP Fall 2016 Documentation

Release 0.1

Brian McMahan

January 19, 2017

CHAPTER 1

Course Description

Video games, phone applications, and many other kinds of programs use the Object Oriented Programming (OOP) design paradigm. Objects are discrete components defined by specific syntax in programming languages. They reflect real world distinctions between types of objects. Object Oriented Programming is more than just syntax, however. It is also a design philosophy that promotes computational thinking, efficient programming, and the reuse of code.

Building upon the Introduction to Python course, we will use Pygame and Python to simultaneously learn about objects and building video games.

The class will tour through the major OOP concepts:

- encapsulation (packaging of code)
- polymorphism (code reuse)
- inheritence (syntactically efficient code structure)
- composition (functionally efficient code structure).

While learning about these major concepts, students will learn about game engines, the need for objects in games, and how to turn their creative ideas into tangible products. By the end of the course, the students will have a working game and be able to make progress on furthering it on their own.

CHAPTER 2

How to Browse This Document

This document is intended to be a companion to the Object Oriented Programming course taught at HEROES Academy. For more information about HEROES Academy, please visit it here.

This document is still in the works. The Spring session does not begin until the second weekend in April.

Contents:

2.1 Course Information

2.1.1 What is HEROES Academy?

HEROES Academy is an intellectually stimulating environment where students' personal growth is maximized by accelerated learning and critical thinking. Our students enjoy the opportunity to study advanced topics in classrooms that move at an accelerated pace.

2.1.2 When does this course meet?

The Object Oriented Programming course will meet from 1:40 to 3:40 starting October 2nd.

2.1.3 How do I register for this course?

The list of courses are listed on the HEROES website. If you have any questions about the process, you can check out the HEROES Frequently Asked Questions.

2.1.4 What are the expectations of this course?

I expect that...

1. You will ask questions when you do not get something.
2. You will keep up with the work.
3. You will fail fast:

- Failing is good
- We learn when we fail
- We only find bugs when code fails; we rarely hunt for bugs when code is working

4. You will not copy and paste code from the internet
   - You are only cheating yourself.
   - It won't bother me if you do it, but you will not learn the material.
5. You will try the homework at least once and email me with solutions or questions by Wednesday

2.1.5 How do I contact you?

You can reach me anytime at teacher@njgifted.org

2.2 Installing Python

2.2.1 Python Distribution

There are several ways to get Python. My recommended way is the Anaconda distribution. It includes both Python and a bunch of other things packaged with it that make it super useful.

Instructions for downloading Anaconda Python:

- Click the link above.
- If you use a Mac, look at the section titled "Anaconda for OS X," and click on "MAC OS X 64-BIT GRAPHICAL INSTALLER" under the "Python 3.5" section.
- If you use a Windows computer, in the section titled "Anaconda for Windows," click either "WINDOWS 64-BIT GRAPHICAL INSTALLER" or "WINDOWS 32-BIT GRAPHICAL INSTALLER" under the "Python 3.5" section.
- On most Windows machines, you can tell if it's a 64-bit or 32-bit system by right-clicking on the Windows logo and selecting "System." The line labeled "System Type" should say either 64-bit or 32-bit. If you're having trouble with this, simply email me and I'll help you out!
- Once you click the button, an installer file will be downloaded to your computer. When it finishes downloading, run the installer file.
- Follow along with the prompts, and select "Register Anaconda as my default Python 3.5" if you're using the Windows installer.
- At the end of the installation wizard, you're done! Anaconda, and Python, are installed.

2.2.2 An Editor

There are many good editors and IDEs (Integrated Development Environments). As you're just beginning to learn how to use Python, it's a good idea to use a simplistic, lightweight development environment. PyCharm and Sublime Text are both good choices for starting out. They have nice, clean appearances, highlight your code to make it easier to read, and are easy to jump in and start coding right away.

Instructions for downloading PyCharm:

- Click the link above.
- Click "Download" under the "Community" section.
- An installer file will be downloaded to your computer. When it finishes downloading, run the installer file.
- Follow along with the installer, and select "add .py extension" if you see the option

- At the end of the installation wizard, you're done! PyCharm is now installed.

Other than those two, GitHub has an editor that is very comparable to Sublime Text. It is called Atom.

2.3 Installing PyGame

PyGame is a library that creates graphical interfaces for games. There is sometimes some difficultly in installing it, so below I have listed information to help you out.

2.3.1 Where to get it

There are a couple of good directions on the internet:

1. The main pygame repository
2. The programarcadegames website
3. Pygame Simplifed

2.3.2 Common Issues

1. **I installed Pygame, but when I use python, it says it can't find it.**
    - this is usually caused by having two versions of python installed
    - Email me and we will talk through the situation. It usually involves a couple things that need to be check to verify this is the situation.
2. **When installing Pygame, at the part where it says "Select Python Installation", it is showing no python installaion**
    - this is can be an issue sometimes with the way Python was installed.
    - I have had this happen to me with Anaconda
    - Try the following:

2.4 [Week 1] Hello PyGame

Welcome to the first week of Object Oriented Programming with Python and Pygame!

2.4.1 Summary

We went over the basics of Python to see what parts of Python would need to be practiced.

After the assessment, we started playing with PyGame, beginning with a tour of the game loop. We spent the rest of the classing playing with the basic objects of Pygame.

2.4.2 Review

We will review material today.

2.4.3 Slides

2.5 [Week 2] Introducing State

2.5.1 Summary

Today, we will cover the PyGame loop again and step through in class how to make things move. This requires understanding how to represent state.

We will practice using functions today, so that we can avoid messiness, enable reusability, and improve the readability. This will require understanding scope, encapsulation, and polymorphism (which is a fancy word for code being able to be used in multiple ways).

The first projects are also selected today. They will require the following:

1. a moving object through changing the position state
2. interaction through the event loop that changes a property of the state
3. a demonstration of reusable code with functions

2.5.2 Homework

1. **Come up with a project**
    - see slides for a couple ideas
2. Put the out of bounds checker into a function
3. Make a wall and have the objects bounce off of it

2.5.3 Slides

2.6 [Week 3] Encapsulating code

2.6.1 Summary

Now that we've started to practice mastering the state of objects (keeping track of their properties), we will work on making our code more efficient.

In-class code

1. Basic Pygame Template
2. Basic stick figure

Important Concepts

1. **Functions**
    - A code block which packages the code and provides a shortcut to executing the code
    - The code below shows how to pass information in, how to get information out

- **important**: remember that scope means what variables can be "seen" inside and outside the function

Example:

```
def hello_x(x):
    y = "hello {}".format(x)
    return y
y = hello_x("world")
print(y)
```

2. **Dictionaries**

   - A python variable type that allows you to map keys to values

Example

```
bob = dict()
bob['name'] = 'bob'
bob['species'] = 'turtle'
```

3. **Encapsulation.**

   - The packaging of code to be reused later
   - **Example: if we have multiple objects, and we want to make them bounce off walls,** then we could either write the wall bouncing code for each object, or write the code once and use a function to apply it to each object.

2.6.2 Homework

See the slides for more information. The basic gist: practice dictionaries and functions.

2.6.3 Slides

2.7 [Week 4] Our First Object

2.7.1 Class Summary

Last week we looked at how to encapsulate code. There were some hiccups so we will covered this a bit more.

Part 1: Encapsulation Exercise

On the following page, you will find a set of exercises. They are to help you understand what it means to encapsulate code and why it is useful.

Encapsulation Exercises

Part 2: Creating an Object

Objects are a way to group either variables or functionality into useful chunks. Like functions, they let us have cleaner code, and repeat ourselves less.

Objects being with "class" definitions. These definitions are like the blueprints or the recipe. Using these blueprints, we can create an object. Let's look at a simple one:

```
class Point:
    x = 0
    y = 0

p1 = Point()
p1.x = 50
p1.y = 50
```

The first 3 lines define our class. This is the blue print for our object. We create our object by using the class like a function. This is usually referred to as "construction" or "instantiation".

For the rest of this part, you will make your own objects to represent boxes.

Click here for the exercises

2.8 [Week 5] Multiple Objects

Last week we saw how to create a single object and then add extra properties to it. This time, we are going to use multiple objects.

The exercise page can be found by clicking right here!

You homework for the week is to makes ure you get through all of these exercises.

2.9 [Week 6] Designing Objects

This week we will be working on designing objects before we use them.

Check here for the exercise page

2.10 [Week 7] Object Ecosystems

Last week we talked about designing objects. This week, we will be talking about how you can make an ecosystem of objects to work together.

Topics:

1. Inheritance and how to use it
2. **Re-capping classes**
    - Click here for the class recap page
3. **Designing an ecosystem of objects in the game**
    - Click here for the exercise page

In designing the game, you should identify:

1. **an entity that will be used in many places**
    - examples include bricks in brickbreak games
    - monsters in other games
    - projectiles
    - etc

2. identify the basic functionality
3. **plan out the flow of the game**
    - what will the hero do
    - what will trigger what
    - what will be the goal
    - what will be the inputs

2.11 [Week 8] Object Practice and Projects

2.11.1 Today's Task

1. Outline your project using the rubric below
2. Using the outline, talk about your project
3. Work on your project

2.11.2 Project Rubric

1. Game Title
2. **Overall game goal**
    - What is the player trying to accomplish
3. **Objects that exist in the game**
    - List all objects that you want to use
    - List all objects currently implemented
    - **For each object, identify the common parts and specialized parts**
        - Specify which properties are inherited and which are new
    - For each object, describe when it is drawn (it can be always drawn or conditionally drawn)
4. **Interaction**
    - What interaction will the user have?
    - Specifically, what keys will be used and what effect will they have
5. What is the minimal set of things you need to have to have your game working?
6. **List three 1-step additions you could make**
    - It has to add only 1 additional piece of complexity

2.11.3 Cookbooks

You should be referencing at least some of these!

- **Simple PyGame cookbook**
    - covers pygame examples without using classes

- **PyGame with Classes cookbook**
  - covers pygame using classes
  - has a lot of functionality explained!
- **Python Cookbook**
  - This has examples of most of Python's syntax!
- **Classes cookbook**
  - This has examples of the basics of classes!
- **A giant color list!**
  - You can use this to get tons of colors into your game!

2.12 [Week 9] Project Discussions

This week will be looking at your projects to see how far you've come and what you have left. You should have a working demo!

You should fill out the rubric again, this time only with things you have done.

2.12.1 Project Rubric

1. Game Title
2. **Overall game goal**
   - What is the player trying to accomplish
3. **Objects that exist in the game**
   - List all objects that you want to use
   - List all objects currently implemented
   - **For each object, identify the common parts and specialized parts**
     - Specify which properties are inherited and which are new
   - For each object, describe when it is drawn (it can be always drawn or conditionally drawn)
4. **Interaction**
   - What interaction will the user have?
   - Specifically, what keys will be used and what effect will they have
5. What is the minimal set of things you need to have to have your game working?
6. **List three 1-step additions you could make**
   - It has to add only 1 additional piece of complexity

2.12.2 Presentation Link

You will give a presentation to your parents when we meet again in 2 weeks. You will have time at the beginning of class to finish things up, but your presentation is due to me that Friday (December 16th).

Here is the presentation template:

2.13 Exercises

2.13.1 Classes Recap

This is a set of exercises for refreshing your knowledge of classes.

The basic class

```
class Dog:
    name = ''
    age = 0

# Dog is called like a function to **instantiate** the object
fido = Dog()
# have to set the variables manually
fido.name = "Fido"
fido.age = 7
```

Class with an initial function

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# now we can accept arguments when we **instantiate** the object!
fido = Dog("Fido", 7)
```

Thinking about how self works

If you need to understand self better, try this out.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print("-2--")
        print("Inside the init function")
        print(self)

# now we can accept arguments when we **instantiate** the object!
print("-1--")
print("Before creating fido")
fido = Dog("Fido", 7)
print("-3--")
print("After creating fido")
print(fido)
```

Look at the code above and run it in PyCharm. Notice what is printed by `print(self)` inside the init function and `print(fido)` outside the class. They are both pointing at the same object! `self` is just a way of getting access to the object while inside the object.

2.13.2 Clean Environment

We are going to start keeping our constants and useful variables in a separate file.

1. Create a separate file, call it "settings.py"
2. Put in the following variables

```
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GREEN = (0, 255, 0)
RED = (255, 0, 0)
WINDOW_SIZE = (700, 500)

SPEEDX = 5
SPEEDY = 5
```

3. Anytime you have new constant variables, they should be put into here.
4. Inside your game file, you should put the following:

```
from settings import *
# if the above breaks, type:
# from .settings import *
```

2.13.3 Designing Objects

Now that we are starting to write our own objects, we will start designing our own objects!

By the end we will have the following file structure:

We will start with the game loop. Create a new file called "game.py" and start putting these into there:

The top

The top of the file is pretty standard. I have added extra imports to handle different files.

```
import pygame
from settings import *
```

The settings.py file is:

Game `__init__` function

Creating a `class` for our game loop means we can organize all of the functionality easier!

```
class Game:
    def __init__(self):

        self.walls = []
        self.hero = None
        self.done = False

        pygame.init()
        self.screen = pygame.display.set_mode(WINDOW_SIZE)
        self.clock = pygame.time.Clock()
        pygame.display.set_caption(TITLE)
```

Checkpoint questions:

1. What would instantiating this class look like?
2. What kinds of things could you add into the initial function?

```
game = Game()
game.run()
```

Game Loop

```
def run(self):
    while not self.done:
        #### EVENT CHECK SECTION
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.done = True
            ## extra stuff will go here

        ### clear the screen
        self.screen.fill(WHITE)

        ## extra stuff will go here

        #### update the display and move forward 1 frame
        pygame.display.flip()
        # --- Limit to 60 frames per second
        self.clock.tick(FPS)
```

Checkpoint questions:

1. Given that you already answered how the class could be instantiated, how would you run this function?
2. Can you predict what it will do? Try and run it now.

The Hero

Let's create the hero class. You can use the one you wrote from last week.

Put it by itself into a hero.py file and change the top of "game.py" to the following:

```
import pygame
from settings import *
from hero import *
```

Now, you should have a hero from last week! It should go into the hero file. I'm going to show the bare bones here:

```
class Hero:
    def __init__(self, x, y, w, h):
        ''' The hero constructor function '''
        self.rect = Rect(x, y, w, h)
        ## other things could/should go here

    def move_right(self, step_size=SPEEDX):
        ''' Move the hero to the right '''
        pass

    def move_left(self, step_size=SPEEDX):
        ''' Move the hero to the left '''
```

```
        pass

    def move_up(self, step_size=SPEEDY):
        ''' Move the hero up '''
        pass

    def move_down(self, step_size=SPEEDY):
        ''' Move the hero down '''
        pass

    def drift(self):
        ''' drift across the screen

        Note: the implementation should drift x and drift y separately
              After the drift in x, it should check for x collisions
              After the drift in y, it should check for y collisions
        '''
        pass

    def drift_x(self):
        ''' Handle the drift in x '''
        pass

    def drift_y(self):
        ''' Handle the drift in y '''
        pass

    def collides_with(self, other_rect):
        ''' return true if there is a collision '''
        pass

    def handle_xcollision(self, other_rect):
        ''' handle collisions going left and right '''
        pass

    def handle_ycollision(self, other_rect):
        ''' handle collisions going up and dowon '''
        pass
```

We are going to add two new functions to the `Hero` class: `update` and `draw`. I will show the functions under the class header below.

Assumption: When `update` is called, the hero will be passed a list of walls. This is so it can check for collisions.

Assumption: When `draw` is called, the hero will be passed the `screen`.

```
class Hero:
    ### all other things here

    def update(self, walls):
        ''' move and check for collisions '''
        pass

    def draw(self, screen)
        ''' draw the hero '''
        pass
```

Adding the hero into the game

Into the `Game` class, we will add a new function which will setup everything. Then, inside the main loop, we will have it run the hero's functions! This will also change how the game is instantiated and run. Updated code is below:

```
class Game:
    def __init__(self):

        self.walls = []
        self.hero = None
        self.done = False

        pygame.init()
        self.screen = pygame.display.set_mode(WINDOW_SIZE)
        self.clock = pygame.time.Clock()
        pygame.display.set_caption(TITLE)


    def setup(self):
        self.hero = Hero(___) ### fill in the underline

    def run(self):
        while not self.done:
            #### EVENT CHECK SECTION
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    self.done = True
                ## extra stuff will go here

            ### clear the screen
            self.screen.fill(WHITE)

            if self.hero is not None:
                self.hero.update(self.walls)
                self.hero.draw()

            #### update the display and move forward 1 frame
            pygame.display.flip()
            # --- Limit to 60 frames per second
            self.clock.tick(FPS)


### this changes the running to:
game = Game()
game.setup()
game.run()
```

Adding Walls

We are going to create a wall class. This will let us manage walls better. We should put this in "walls.py". I have written some code below to make this easier.

Important notes:

1. You have to write the `draw` function
2. The class is able to parse a series of strings into wall placements (see `parse_level`)

```
class Walls:
    def __init__(self):
        ''' keep track of the walls
        you could maybe pass in a COLOR here'''
        self.walls = []

    def add_wall(self, x, y, w, h):
        ''' add a single wall'''
        self.walls.append(Rect(x,y,w,h))

    def parse_level(self, level):
        '''Parse a level string into a set of walls. I've made this for you'''
        level_width = len(level[0])
        wall_width = WIDTH / level_width

        level_height = len(level)
        wall_height = HEIGHT / level_height

        for row_index in range(level_height):
            for col_index in range(level_width):
                cell = level[row_index][col_index]
                if "cell" == "W":
                    x = wall_width * col_index
                    y = wall_height * row_index
                    self.add_wall(x, y, wall_width, wall_height)

    def set_example_level(self):
        level = [
            "WWWWWWWWWWWWW",
            "W       W    W",
            "W  W    W    W",
            "W  W    W    W",
            "W  W         W",
            "WWWWWWWWWWWWW"
        ]
        self.parse_level(level)

    def draw(self, screen):
        for wall in self.walls:
            ### fill in the pygame draw code here.
```

In order to get this into the game, we have to do two things:

1. Add an import statement `from walls import *` into the game.py
2. Add this to the `setup` so that the game will make the walls
3. Add into the game loop a call which draws the walls.

Adding Keyboard Input

To get keyboard input, we need to add some extra stuff into the event loop. Specifically, the event loop should handle more complex checks. Also, optionally, we could have the HERO check for game events.

```
class Game:
    ## code was here

    def run(self):
```

```
        while not self.done:
            #### EVENT CHECK SECTION
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    self.done = True
                else:
                    self.handle_event(event)

            ### the rest of the game loop here

    def handle_event(self, event):
        ## do various checks for events here.
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                ## code here
            elif event.key == pygame.K_RIGHT:
                ## code here
            elif event.key == pygame.K_DOWN:
                ## code here
            elif event.key == pygame.K_UP:
                ## code here
```

Jumping

If you'd like to make your hero jump and land on platforms, there are a couple different things that need to happen.

1. The hero can not respond to the up/down keys anymore
2. The hero is always moving down
3. **Inside the moving down, the hero has two speeds:**
   (a) **`gravity`, which is the default speed**
      - this number show moving the hero DOWN (so, a positive number if adding to position)
   (b) **`up_energy`, which gets set to some number when a key like SPACE is pressed**
      - then, whenever the hero moves, the `up_energy` decays, for example: `up_energy = up_energy * 0.9`

2.13.4 Encapsulation Exercises

These exercises go through and make the encapsulation more and more complete for moving a square around the screen.

STEP ONE

Use the template. These examples assume that you are using the basic pygame template.

Get it here

The exercises have two parts: defining the information for the square and then using that information.

Anatomy of the Pygame loop

```
##### INIT SECTION
# import pygame
# any functions you want to use should be defined right away
# create pygame variables
# create variables you want to use inside the game loop


##### WHILE LOOP SECTION
while not done:
    # check for events
    # fill the screen with white
    ##### ACTION CODE
    # do any actions that we want to do
    # this could be moving the box, etc
    ##### FINISHING CODE
    # end of while loop code, mostly the clock.tick()

#### POST WHILE LOOP SECTION
# once the code hits here, we can assume that the while loop is over and game is done
# do any last finishing code things here
# the important one is to tell pygame shut down
```

Exercise 1

Inside the INIT section:

```
origin_x = 50
origin_y = 50
square_width = 100
square_height = 100
```

Inside the ACTION CODE section:

```
# the syntax for rect is (display surface, color, rectangle_info)
# and the rectangle_info is (x, y, width, height)
pygame.draw.rect(surface, BLACK, [origin_x, origin_y, square_width, square_height])
```

Your task:

1. **Create a second rectangle and that has different starting x and y variables.**
    - In other words, create two new variables and use them to draw a new rectangle.
    - Use the same height and width as the first rectangle.

Exercise 2

Inside the INIT section:

```
box_info = {'x': 50, 'y': 50, 'width': 100, 'height': 100}
```

Inside the ACTION CODE section:

```
# the syntax for rect is (display surface, color, rectangle_info)
# and the rectangle_info is (x, y, width, height)
pygame.draw.rect(surface, BLACK, [box_info['x'], box_info['y'], box_info['width'], box_info['height']
```

Your task:

1. **Create a second rectangle that is made up of a second dictionary.**
    - It should be drawn in the exact same way as the first one, but using the second dictionary.

Exercise 3

Inside the INIT section:

```
def make_box(x, y, width, height):
    new_box_info = {'x': x, 'y': y, 'width': width, 'height': height}
    return new_box_info

box_info = make_box(50, 50, 100, 100)
```

Inside the ACTION CODE section:

```
# the syntax for rect is (display surface, color, rectangle_info)
# and the rectangle_info is (x, y, width, height)
pygame.draw.rect(surface, BLACK, [box_info['x'], box_info['y'], box_info['width'], box_info['height']
```

Your task:

1. Create a second rectangle using the function. Draw this rectangle as you did in exercise 2.

Exercise 4

Inside the INIT section:

```
def make_box(x, y, width, height):
    new_box_info = {'x': x, 'y': y, 'width': width, 'height': height}
    return new_box_info

def draw_box(surf, color, info):
    pygame.draw.rect(surf, color, [info['x'], info['y'], info['width'], info['height']])

box_info = make_box(50, 50, 100, 100)
```

Inside the ACTION CODE section:

```
# the syntax for rect is (display surface, color, rectangle_info)
# and the rectangle_info is (x, y, width, height)
draw_box(surface, BLACK, box_info)
```

Your task:

1. Create a second rectangle as you have in the past couple of exercises. Draw it in the same way.

Final Task

You will create two new functions:

1. **`make_circle`**
    - use a dictionary to represent the necessary variables for a circle
    - it needs x, y, and radius.
2. **`draw_circle` function**
    - in the same way `draw_box` is written, write a `draw_circle` function

- the syntax for drawing a circle is `pygame.draw.circle(surface_object, some_color, center_point, radius)`
- the center point is just `(x,y)` or `[x,y]`

2.13.5 Intro Object Exercises

In these exercises, you will go through making objects and using them for different things.

These examples assume that you are using the basic pygame template. If you don't have it, find it here

Anatomy of the Pygame loop

```
##### INIT SECTION
# import pygame
# any functions you want to use should be defined right away
# create pygame variables
# create variables you want to use inside the game loop


##### WHILE LOOP SECTION
while not done:
    # check for events
    # fill the screen with white
    ##### ACTION CODE
    # do any actions that we want to do
    # this could be moving the box, etc
    ##### FINISHING CODE
    # end of while loop code, mostly the clock.tick()

#### POST WHILE LOOP SECTION
# once the code hits here, we can assume that the while loop is over and game is done
# do any last finishing code things here
# the important one is to tell pygame shut down
```

Exercises

Exercise 1

Inside the INIT section:

```
# set up the class and the variables

class Box:
    x = 0
    y = 0
    w = 0
    h = 0
    speedx = 0
    speedy = 0

box_info = Box()
box_info.x = 50
box_info.y = 50
box_info.w = 100
```

```
box_info.h = 100
box_info.speedx = 10
box_info.speedy = 10
```

Similar to the other exercises, use this to make the rectangle inside the ACTION CODE section:

```
# Use the box_info object to draw!

pygame.draw.rect(surface, BLACK, [box_info.x, box_info.y, box_info.w, box_info.h])
```

Compare this code to the earlier exercises. Write the "make_box" function which uses the class instead. Also, rewrite the "draw_box" using this class.

Exercise 2

The code for getting the width and height of the screen are the following:

```
# get the screen width and height

screen = pygame.display.get_surface()
W, H = screen.get_size()
```

When testing to see if the box is beyond the sides of the screen, use the correct side:

```
# calculate special variables

right_side = box_info.x + box_info.w
left_side = box_info.x
top_side = box_info.y
bottom_side = box_info.y + box_info.h
```

Also, remember W is the width, and so is the right side of the screen. H is the height and is the bottom side of the screen. So, if the left_side is below 0, it is out of bounds. If the right side is larger than W, it is out of bounds. If the top_side is smaller than 0, it is out of bounds. Finally, if the bottom_side is larger than H, it is out of bounds.

Write the code for the update position function:

```
# Compute the new position using the box_info object

def update_position(box_info):
    ### test if the box is out of bounds
    ### if it is,
    ###        the speed should negative for
    ###        the corresponding side that is out of bounds
    ###
    ### then update the position by the speed
    ### so, the x changes by speed
    ### the y changes by speed
```

The function should be used inside the while loop to update the position before it is drawn.

Exercise 3

Let's add a function into our class so that it can draw itself.

```
# Compute the new position using the box_info object

```

```
class Box:
    x = 0
    y = 0
    w = 0
    h = 0
    speedx = 0
    speedy = 0

    def update_position(self):
        ### everything stays the same, except you can get access to the variables using "self" now
        ### test if the box is out of bounds
        ### if it is,
        ###        the speed should negative for
        ###        the corresponding side that is out of bounds
        ###
        ### then update the position by the speed
        ### so, the x changes by speed
        ### the y changes by speed

## assume we do
## box = Box()
## then, later, you can use it with
## box.update_position()
```

Exercise 4

Let's make this more interactive! For each of the following key tests, you can change some variable inside your object. For instance, you could have left and right increase or decrease the speedx. You could also have your box jump with space. Note that this last one requires thinking about gravity a bit more.

```
### inside WHILE LOOP section
for event in pygame.event.get():
    ## standard quit
    if event.type == pygame.QUIT:
        done = True
    elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_SPACE:
            print("Do something here!")
        elif event.key == pygame.K_LEFT:
            print("do something here!")
        elif event.key == pygame.K_RIGHT:
            print("do something here!")
```

Extra stuff

Class __init__ method

Using the `__init__` method lets you pass variables into the creation of the object. This is also called a "constructor" method.

```
class Box:
    def __init__(self, x, y, w, h):
        self.x = x
        self.y = y
        self.w = w
```

```
        self.h = h

box = Box(50, 50, 100, 100)
```

Gravity and Jumping 1

For fun, we are going to add gravity. Gravity is just a way of updating the y speed. Add the following code into update_position. Your Box class will need a new variable: mass.

```
# recall that
#          x += y
# is the same as
#          x = x + y

### for physics

upforce = 0 # should be some number, maybe from bouncing or jumping

gravity = 9.8 # is positive because 0 is the top and we want it to fall down
downforce = gravity * box.mass

totalforce = downforce+upforce
acceleration = totalforce / box.mass

### update speeds and locations
box.speedy += acceleration
box.y += box.speedy
```

Play with different values of gravity.

Gravity+Jumping 2

(For a great discussion of this, see this stackoverflow answer)

Gravity is a force that acts on the y direction of an object. Specifically, if your object has a speed, then it is accelerating downwards by gravity. If you jump, you are accelerating upwards.

Force is equal to mass times acceleration—. So, to get acceleration from two forces (gravity and jumping), we do . Jumping is our force upwards, gravity is our force downwards.

Use the following function to compute the acceleration. In the event loop, you could set a boolean variable which tells you whether a space bar press happened or not. Then, you can pass it to this function to enable the upward force. It would be good to have upward force be a couple times more than gravity. Also, the boolean for the space bar press should only be True once, because it's a burst of energy, not a sustained force.

```
def compute_acceleration(box, did_jump=False):
    gravity_force = 9.8 * box.mass
    downforce = gravity_force # + any other downward forces

    if did_jump:
        jump_force = somenumber * box.mass
    else:
        jump_force = 0
    upforce = jump_force # + any other upward forces; maybe bouncing

    total_force = downforce + upforce
    acceleration = total_force / box.mass # f = m*a
```

```

    return acceleration
```

Property Decorator

Using the property decorator for a class:

```
class Box:
    x = 0
    y = 0
    w = 10
    h = 10

    def print_info(self):
        pass

    @property
    def right_side(self):
        return self.x + self.w

box = Box()
box.x = 50
print(box.print_info, type(box.print_info))
print(box.right_side, type(box.right_side))
```

Up-Down triggers

Let's say you want somethign constant to be happening while a button is pressed. You could do the following:

```
### inside WHILE LOOP section
for event in pygame.event.get():
    ## standard quit
    if event.type == pygame.QUIT:
        done = True
    elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_SPACE:
            spacedown = True
    elif event.type == pygame.KEYUP:
        if event.key == pygame.K_SPACE:
            spacedown = False
```

2.13.6 Multiple Objects Exercises

In these exercises, you will go through making objects and using them for different things.

These examples assume that you are using the basic pygame template. If you don't have it, find it here

Anatomy of the Pygame loop

```
##### INIT SECTION
# import pygame
# any functions you want to use should be defined right away
# create pygame variables
# create variables you want to use inside the game loop
```

```

##### WHILE LOOP SECTION
while not done:
    # check for events
    # fill the screen with white
    ##### ACTION CODE
    # do any actions that we want to do
    # this could be moving the box, etc
    ##### FINISHING CODE
    # end of while loop code, mostly the clock.tick()

#### POST WHILE LOOP SECTION
# once the code hits here, we can assume that the while loop is over and game is done
# do any last finishing code things here
# the important one is to tell pygame shut down
```

Exercises

Exercise 0

Clean your environment (click here)!

Exercise 1

This week, we will start making the Hero of a game.

```
class Hero:
    def __init__(self, x, y, w, h):
        ''' this is a constructor function
            when you create a new Hero, it requires these 4 variables
        '''


        # Remember that classes act like factories
        # when we are inside this function, we are inside a single Hero instance
        # and we need to be able to reference the current hero
        # so, a self variable is a way of referencing the current Hero
        self.x = x
        self.y = y
        self.w = w
        self.h = h

hero1 = Hero() # this will break.
hero1 = Hero(10, 10, 50, 50)
print(hero1.x)
```

Add the following

1. **A variable into the constructor (`__init__` function) for the hero's name**
    - don't forget to "save" it to the hero using `self.name = name`

Exercise 2

This week, we will start making the Hero of a game.

```
class Hero:
    def __init__(self, x, y, w, h):
        ''' this is a constructor function
            when you create a new Hero, it requires these 4 variables
        '''

        # Remember that classes act like factories
        # when we are inside this function, we are inside a single Hero instance
        # and we need to be able to reference the current hero
        # so, a self variable is a way of referencing the current Hero
        self.rect = Rect(x, y, w, h)

    def say_hi(self):
        print("Hello, my name is {}".format(self.name))

    def move_right(self, step_size=0):
        self.rect.x += step_size

hero1 = Hero(10, 10, 50, 50)
print(hero1.x) # this will break
print(hero1.rect)
hero1.say_hi()
```

We are going to save the coordinate information into Pygame's `Rect` class. They offer some really neat functions if we do this.

Also, `Rect` has the following variables:

```
x,y
top, left, bottom, right
topleft, bottomleft, topright, bottomright
midtop, midleft, midbottom, midright
center, centerx, centery
size, width, height
w,h
```

Add the following:

1. Add the `name` code from the first exercise into this class.
2. **The three other functions that move the hero:**
    (a) `def move_left(self, step_size=0)`
    (b) `def move_down(self, step_size=0)`
    (c) `def move_up(self, step_size=0)`
3. Code that does the following:

```
hero1.move_right(100)
hero1.move_down(100)
hero1.move_left(100)
hero1.move_up(100)
```

Exercise 3

Let's make the hero move on their own!

Note: this code assumes you have done the Clean Environment exercise because it assumes the `SPEEDX` and `SPEEDY` variables.

NOTE: here we will pass in `SPEEDX` and `SPEEDY` explicitly into the move functions. However, you could (and should) change the defaults inside those functions to `SPEEDX` and `SPEEDY`.

Add the following code into the `Hero.__init__` function:

```
self.going_right = True
self.going_down = True
```

And now, a new function inside the `Hero` class:

```
def drift(self):
    if self.going_right:
        self.move_right(SPEEDX)
    else:
        self.move_left(SPEEDX)

    if self.going_down:
        self.move_down(SPEEDY)
    else:
        self.move_up(SPEEDY)
```

Add the following:

1. Inside `def drift(self)`, after the code which moves the hero, check to see if `self.rect` is outside of the screen. I have done the first one for you.

```
# this will assume WINDOW_SIZE
WIDTH = WINDOW_SIZE[0]
HEIGHT = WINDOW_SIZE[1]
# you can also do "unpacking"
# WIDTH, HEIGHT = WINDOW_SIZE

if self.rect.right > WIDTH:
    # we will now switch directions
    self.going_right = False
    # we will also set the left side to be equal to the window side
    # this means we won't go off screen and bug out
    self.rect.right = WIDTH
elif self.rect.left < 0:
    print("you should write code here!")
elif self.rect.top < 0:
    print("you should write code here!")
elif self.rect.bottom > HEIGHT:
    print("you should write code here!")
```

Exercise 4

Now, we will give our hero a wall to bump into! This will demonstrate why we use `Rect`. Check out this documentation: PyGame Rect Docs for Colliding

This code should go into the INIT SECTION part of the code:

```
# this will assume WINDOW_SIZE
WIDTH = WINDOW_SIZE[0]
HEIGHT = WINDOW_SIZE[1]
# you can also do "unpacking"
# WIDTH, HEIGHT = WINDOW_SIZE
wall1 = Rect(WIDTH // 2, 0, WIDTH // 10, HEIGHT)
```

Then, after having moved, inside the `while` loop ACTION CODE:

```
### assume hero moves here in some way
### could be calling hero1.move()

if hero1.rect.colliderect(wall1):
    print("The hero has collided with the wall!")
    print("You should be adding code here!")

    if hero1.rect.right > wall1.left:
        hero1.rect.right = wall1.left
        hero1.going_right = False
    elif hero1.rect.left < wall1.right:
        print("Add code here!")
    elif hero1.rect.bottom > wall1.top:
        print("Add code here!")
    elif hero1.rect.top < wall1.bottom:
        print("Add code here!")
```

You should finish the code inside the `if` statements.

Exercise 5

Move the above code into the `Hero` class.

```
def handle_collision(self, other_rect):
    if self.rect.colliderect(other_rect):
        print("The code is basically the same from Exercise 4!")
```

The major differences will be that `hero1` is used to refer to the hero OUTSIDE of itself, but when the code is INSIDE itself, you use the `self` variable to reference it.

What should the code look like now inside the `while` loop?

Exercise 6

Now you will add multiple walls.

```
two_thirds_height = 2 * HEIGHT//3
one_tenth_width = WIDTH // 10
one_third_width = WIDTH // 3

### Rects want x, y, w, h
### x and y are for the TOP LEFT corners.
wall1 = Rect(one_third_width, 0, one_tenth_width, two_thirds_height)
wall2 = Rect(2 * one_third_width, HEIGHT - two_thirds_height, one_tenth_width, two_thirds_height)

walls = [wall1, wall2]
```

Inside the loop:

```
for wall in walls:
    hero1.handle_collision(wall)
```

You should draw out a maze and plan the x, y, w, and h coordinates. You should be using at least 5 walls.

Bonus Exercise

If you want to add human movement to the hero, you can do the following:

```
### inside WHILE LOOP section
for event in pygame.event.get():
    ## standard quit
    if event.type == pygame.QUIT:
        done = True
    elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_LEFT:
            hero1.move_left()
```

2.13.7 Object Ecosystem Exercises

For this, you will be creating a custom, but basic pygame sprite class. You will use this as the base for all of the sprites you draw in the game. Then, you will extend it to make the Hero and some object type that will be used a lot. I am going to use it to make blocks that the Hero is going to try to get.

Basic Pygame Sprite Class

When you have a pre-defined class, you can inherit from it and get all of its functionality. This lets you implement common things once, and then reuse them in different ways.

We are going to start with blocks and then we will work our ways towards a fuller implementation.

```
class Block(pygame.sprite.Sprite):
    def __init__(self, color, width, height):
        # Call the parent class (Sprite) constructor
        super(Block, self).__init__(self)

        # Create an image of the block, and fill it with a color.
        # This could also be an image loaded from the disk.
        self.image = pygame.Surface([width, height])
        self.image.fill(color)

        # Fetch the rectangle object that has the dimensions of the image
        # Update the position of this object by setting the values of rect.x and rect.y
        self.rect = self.image.get_rect()
```

Converting Walls to use sprites

```
class Walls:
    def __init__(self):
        ''' keep track of the walls
        you could maybe pass in a COLOR here'''
        self.walls = pygame.sprite.Group()

    def add_wall(self, x, y, w, h):
        ''' add a single wall'''
        new_block = Block(BLACK, w, h)
        new_block.x = x
        new_block.y = y
        self.walls.add(new_block)
```

If you are having trouble with the game, start here

We are going to debug misunderstandings. Debugging means removing all the complex stuff and starting with simple baselines so we can know where the issue is at.

First, start with the class recap. If there is any confusion with that, then come to me.

Then, we will move onto the basic pygame setup. Go to either the code you wrote, or the code I wrote.

Using a series of English sentences, describe the flow of the PyGame game. You don't have to describe every object creation, but you should describe which functions get called and in what order. If something is created, you should describe that.

Plan your own ecosystem of objects

You can choose to start fresh now if you want, but you should make a plan with the following things:

1. **What are going to be the objects in your game**
    - Walls, hero, monsters, projectiles, etc
2. How the objects are going to interact?
3. **What is the smallest component in your game?**
    - You should plan on making this into a base class
    - Then, you can create classes that subclass it and use it as its ancestor
    - For example, you could have: `Entity`, which then is subclassed by `Monster`

If you should get to this point, you should start working on your game. Please let me know when you get here.

2.14 Cookbooks

2.14.1 Classes Cookbook

Design patterns and examples for classes! Use these to help you solve problems.

Defining a class

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Instantiating an object

```
# create the object!
fido = Dog("Fido", 7)
```

Writing a method

A method is the name of a function when it is part of a class.

You always have to include `self` as a part of the method arguments.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("Bow wow!")


fido = Dog("Fido", 7)
fido.bark()
```

Using the self variable

You can access object variables through the `self` variable. Think of it like a storage system!

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("{}: Bow Wow!".format(self.name))


fido = Dog("Fido", 7)
fido.bark()

odie = Dog("Odie", 20)
odie.bark()
```

Using the property decorator

You can have complex properties that compute like methods but act like properties. Properties cannot accept arguments.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print("{}: Bow Wow!".format(self.name))

    @property
    def human_age(self):
        return self.age * 7

fido = Dog("Fido", 7)
fido.bark()
print("Fido is {} in human years".format(fido.human_age))
```

Inheriting properties and methods

You can inherit properties and methods from the ancestors! For example, the initial function below is inherited.

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class Dog(Animal):
    def bark(self):
        print("{}: Bow Wow!".format(self.name))

    @property
    def human_age(self):
        return self.age * 7

class Cat(Animal):
    def meow(self):
        print("{}: Meow!".format(self.name))

fido = Dog("Fido", 7)
fido.bark()
print("Fido is {} in human years".format(fido.human_age))
```

You can also override certain things and call the methods of the ancestor!

```
class Animal:
    def __init__(self, name, age, number_legs, animal_type):
        self.name = name
        self.age = age
        self.number_legs = number_legs
        self.animal_type = animal_type

    def make_noise(self):
        print("Rumble rumble")

class Dog(Animal):
    def __init__(self, name, age):
        super(Dog, self).__init__(name, age, 4, "dog")

    def make_noise(self):
        self.bark()

    def bark(self):
        print("{}: Bow Wow!".format(self.name))

    @property
    def human_age(self):
        return self.age * 7

class Cat(Animal):
    def __init__(self, name, age):
        super(Dog, self).__init__(name, age, 4, "cat")

    def make_noise(self):
        self.meow()

    def meow(self):
```

```
        print("{}: Meow!".format(self.name))


fido = Dog("Fido", 7)
fido.make_noise()
print("Fido is {} in human years".format(fido.human_age))

garfield = Cat("Garfield", 5, 4, "cat")
garfield.make_noise()
```

Using the classmethod decorator

There is a nice Python syntax which lets you define custom creations for your objects.

For example, if you wanted certain types of dogs, you could do this:

```
class Animal:
    def __init__(self, name, age, number_legs, animal_type):
        self.name = name
        self.age = age
        self.number_legs = number_legs
        self.animal_type = animal_type

    def make_noise(self):
        print("Rumble rumble")

class Dog(Animal):
    def __init__(self, name, age, breed):
        super(Dog, self).__init__(name, age, 4, "dog")
        self.breed = breed


fido = Dog("Fido", 5, "Labrador")
```

But you could also do this:

```
class Animal:
    def __init__(self, name, age, number_legs, animal_type):
        self.name = name
        self.age = age
        self.number_legs = number_legs
        self.animal_type = animal_type

    def make_noise(self):
        print("Rumble rumble")

class Dog(Animal):
    def __init__(self, name, age, breed):
        super(Dog, self).__init__(name, age, 4, "dog")
        self.breed = breed

    @classmethod
    def labrador(cls, name, age):
        return cls(name, age, "Labrador")

fido = Dog.labrador("Fido", 5)
```

Important parts:

1. **Instead `self`, it has `cls` as its first argument.**
    - This is a variable which points to the class being called.
2. **`@classmethod` is right above the definition of the class.**
    - It absolutely has to be exactly like this
    - No spaces in between, just sitting on top of the class definition
    - It's called a decorator.
3. **It returns `cls(name, age, "Labrador")`.**
    - This is exactly the same as `Dog("Fido", 5, "Labrador")` in this instance
    - Overall, it is letting you shortcut having to put in the labrador string.

This is a simple example, but it is useful for more complex classes

2.14.2 Colors!

Get a list of colors into your game easily.

Create a file and put the following colors into it. Then, wherever you want to use them, you can just import them!

```
# assuming file is all_colors.py

from all_colors import *
```

The colors:

```
# some colors
# gray
BLACK = (0, 0, 0)
GRAY80 = (51, 51, 51)
GRAY60 = (102, 102, 102)
GRAY40 = (153, 153, 153)
GRAY20 = (204, 204, 204)
WHITE = (255, 255, 255)

# pink
LIGHT_PINK = (255, 150, 235)
PINK = (255, 100, 255)
MAGENTA = (255, 0, 255)
CORAL = (255, 160, 160)

# red
RED = (255, 0, 0)
BRICK_RED = (175, 25, 0)
SCARLET = (255, 75, 0)

# orange
ORANGE = (255, 110, 0)
LIGHT_ORANGE = (250, 160, 0)

# yellow
GOLD = (255, 215, 0)
YELLOW = (255, 255, 0)

# green
```

```
NEON_GREEN = (160, 255, 0)
LIGHT_GREEN = (0, 255, 0)
GREEN = (10, 150, 10)
FOREST_GREEN = (10, 75, 0)

# blue
CYAN = (0, 255, 255)
PERIWINKLE = (150, 150, 210)
LIGHT_BLUE = (0, 150, 255)
BLUE = (0, 0, 255)
NAVY_BLUE = (0, 0, 100)

# violet
ROYAL_BLUE = (100, 0, 255)
VIOLET = (150, 0, 255)
LAVENDER = (225, 150, 255)

# brown
BROWN = (120, 60, 0)
BEIGE = (255, 200, 150)
```

2.14.3 Heroes Cookbook

This is a set of recipes that you should use while solving problems!

Numbers

Integers

```
# create an integer
x = 5

# convert an integer string
x = str('5')

# convert a float to an integer
## note: don't depend on this for rounding, it rounds in weird ways
x = int(5.5)

# convert a string of any number base
# for example, binary
x = int('1010101', base=2)
```

Floats

```
# create a float
x = 5.5

# convert a float string
x = float("5.5")

# convert an integer to a float
x = float(5)
```

Basic math operations

```
x = 100

# 1. Add
x = x + 5
x += 5

# 2. Subtract
x = x - 5
x -= 5

# 3. Multiply
x = x * 5
x *= 5

# 4. Divide
x = x / 5
x /= 5

# 5. Power
x = x ** 2
x **= 2
```

Advanced math operations

```
# 1. Integer Division
x = x // 5
x //= 5

# 2. Modulo
x = 84
x = x % 5
x %= 5
```

Use the math library

```
import math

x = 10

# pow is power, same as x ** 2
x = math.pow(x, 2)

# ceil rounds up and floor rounds down
x = 5.5
y = math.ceil(x) # y is 6.0
z = math.floor(x) # z in 5.0

# some other useful ones:
math.sqrt(x)
math.cos(x)
math.sin(x)
math.tan(x)

```

```
# this will give you pi:
math.pi
```

Strings

Add two strings together

```
first_name = "euclid "
space = " "
last_name = "von rabbitstein"
full_name = first_name + space + last_name
```

Repeat a string

```
message = "Repeat me!"
repeated10 = message * 10

# I like to use it for pretty printing code results
line = "-" * 12
print("   Title!   ")
print(line)
```

Index into a string

```
first_name = "Euclid"
last_name = "Von Rabbitstein"
first_initial = first_name[0]
last_initial = last_name[0]
initials = first_initial + last_initial
```

Slice a string

```
# the syntax is
#   my_string[start:stop]
# this includes the start position but goes UP TO the stop
# you can leave either empty to go to the front or end

target = "door"
last_three = target[1:]
first_three = target[:3]
middle_two = target[1:3]

# you can use negatives to slice off the end!
all_but_last = target[:-1]

pig_latin = target[1:] + target[0] + "ay"
```

String's inner functions

```
full_name = "euclid von Rabbitstein"

# all caps
full_name_uppered = full_name.upper()

# all lower
full_name_lowered = full_name.lower()

# use lower to make sure something is lower before you compare it
user_command = "Exit"
if user_command.lower() == "exit":
    print("now I can exit!")

# first letter capitalized
full_name_capitalized = full_name.capitalize()

# split into a list
full_name_list = full_name.split(" ")

# strip off any extra spaces
test_string = "   extra spaces everywhere   "
stripped_string = test_string.strip()

# replace things in a string
full_name_replaced = full_name.replace("von", "rabbiticus")

# use replace to delete things from a string!
test_string = "annoying \t tabs in \t the string"
fixed_string = test_string.replace("\t","")
```

Built-in Functions

```
print("This prints to the console/terminal!")

# notice the space at the end!
# it helps so that what you type isn't right next to the ?
name = input("What is your name? ")

# use input to get an integer
age = input("How old are you?")
# but it's still a string!
# convert it
age = int(age)

# test the length of a list or string
name_length = len(name)

# get the absolute value of a number
positive_number = abs(5 - 100)

# get the max and min of two or more numbers
num1 = 10**3
num2 = 2**5
num3 = 100003
```

```
biggest_one = max(num1, num2, num3)
smallest_one = min(num1, num2, num3)
# can do any number of variables here
#   max(num1, num2) works
#   and max(num1, num2, num3,  num4)

## max/min with a list
ages = [12, 15, 13, 10]
min_age = min(age)
max_age = max(age)

# sum over the items in a list
# more list stuff is below
ages = [12, 15, 13, 10]
sum_of_ages = sum(ages)
number_of_ages = len(ages)
average_age = sum_of_ages / number_of_ages
```

Boolean algebra

Create a literal boolean variable

```
literal_boolean = True
other_one = False
```

Create a boolean variable from comparisons

```
x = 9
y = 3
x_is_bigger = x > y # True
x_is_even = x % 2 == 0 # False
x_is_multiple_of_y = x % y == 0 # True
```

Combine two boolean variables with 'and' and 'or'

```
# example data
card_suit = "Hearts"
card_number = 7

# save the results from comparisons!
card_is_hearts = card_suit == "Hearts"
card_is_diamond = card_suit == "Diamond"
card_is_big = card_number > 8

# only 1 of them needs to be true
card_is_red = card_is_hearts or card_is_diamond

# both need to be true
card_is_good = card_is_red and card_is_big

# creates the opposite!
card_is_bad = not card_is_good
```

If, elif, and else

Use an if to test for something

```
power_level = 1000
min_power_level = 500
max_power_level = 1000

# one thing is larger than another
if power_level > minimum_power_level:
    print("We have enough power!")

if power_level == max_power_level:
    print("You have max power!")
```

Create conditional logic

```
selected_option = 2

if selected_option == 1:
    print("Doing option 1")
elif selected_option == 2:
    print("Doing option 2")
elif selected_option == 3:
    print("doing option 3")
else:
    print("Doing the default option!")
```

Nest one if inside another if

```
name = "euclid"
animal = "bunny"

if animal == "bunny":
    if name == "euclid":
        print("Euclid is my bunny")
    elif name == "leta":
        print("Leta is my bunny")
    else:
        print("this is not my bunny..")
else:
    print("Not my animal!")
```

Lists

Create an empty list

```
new_list = list()
# or
new_list = []
```

Create a list with items

```
my_pets = ['euclid', 'leta']
```

Add onto a list

```
my_pets.append('socrates')
```

Index into a list

```
first_pet = my_pets[0]
second_pet = my_pets[1]
third_pet = my_pets[2]
```

Slice a list into a new list

```
# the syntax is
#   my_list[start:stop]
# this includes the start position but goes UP TO the stop
# you can leave either empty to go to the front or end

first_two_pets = my_pets[:2]
last_two_pets = my_pets[1:]
```

Test if a value is inside a list

```
## with any collection, you can test if an item is inside the collection
## it is with the "in" keyword

my_pets = ['euclid', 'leta']
if 'euclid' in my_pets:
    print("Euclid is a pet!")
```

Sets

Create a set or convert a list to a set

```
my_pet_list = ['euclid', 'leta']

# you can convert lists to sets using the set keyword
my_pet_set = set(my_pet_list)

# sets are like lists but you can't index into them or slice them
# they are used for fast membership testing

# you can create a new set by:
my_pet_set = set(['euclid', 'leta'])
```

Add an item to a set

```
my_new_set = set()

# instead of append, like a list, you use 'add'
my_new_set.add("Potatoes")
```

Using sets to enforce uniqueness

```
my_grocery_list = ['potatoes', 'cucumbers', 'potatoes']

# now if you want to make sure items only appear once, you can convert it to a set
# it will automatically do this for you, because items are only allowed to be in sets one time

my_grocery_set = set(my_grocery_list)
```

For Loops

Write a for loop

```
for i in range(10):
    print("do stuff here")
```

Use the for loop's loop variable

```
for i in range(10):
    new_number = i * 100
    print("The loop variable is i.  It equals {}".format(i))
    print("I used it to make a new number. That number is {}".format(new_number))
```

Use range inside a for loop

```
start = 3
stop = 10
step = 2

for i in range(stop):
    print(i)

for i in range(start, stop):
    print(i)

for i in range(start, stop, step):
    print(i)
```

Use a list inside a for loop

```
my_pets = ['euclid', 'leta']

for pet in my_pets:
    print("One of my pets: {}".format(pet))
```

Nest one for loop inside another for loop

```
for i in range(4):
    for j in range(4):
        result = i * j
        print("{} times {} is {}".format(i, j, result))
```

While Loops

Use a comparison

```
response = ""

while response != "exit":
    print("Inside the loop!")
    response = input("Please provide input: ")
```

Use a boolean variable

```
done = False

while not done:
    print("Inside the loop!")
    response = input("Please provide input: ")
    if response == "exit":
        done = True
```

Loop forever

```
while True:
    print("Don't do this!  It is a bad idea.")
```

Special Loop Commands

Skip the rest of the current cycle in the loop

```
for i in range(100):
    if i < 90:
        continue
    else:
        print("At number {}".format(i))
```

Break out of the loop entirely

```
while True:
    response = input("Give me input: ")
    if response == "exit":
        break
```

Functions

No arguments and returns nothing

```
def say_hello():
    print("hello!")
```

Takes one argument

```
def say_something(the_thing):
    print(the_thing)
```

Returns a value

```
def double(x):
    return 2*x
```

Takes two arguments

```
def exp_func(x, y):
    result = x ** y
    return result

final_number = exp_func(10, 3)
```

Takes keyword arguments

```
def say_many_times(message, n=10):
    for i in range(n):
        print(message)

say_many_times("Hi!", 2)
say_many_times("Yay!", 10)
```

Time module

Using time.time() to count how long something takes

```
import time

start = time.time()

for i in range(10000):
    continue

new_time = time.time()
total_time = new_time - start
print(total_time)
```

Using time.sleep(n) to wait for n seconds

```
import time

start = time.time()

time.sleep(10)

end = time.time()

print(start - end)
```

Random Module

Generate a random number between 0 and 1

```
import random

num = random.random()
print("the random number is {}".format(num))
```

Generate a random number between two integers

```
import random

num = random.randint(5, 100)
print("the random integer between 5 and 100 is {}".format(num))
```

Select a random item from a list

```
import random

my_pets = ['euclid', 'leta']
fav_pet = random.choice(my_pets)
print("My randomly chosen favorite pet is {}".format(fav_pet))
```

2.14.4 Cookbook

A set of common recipes and design patterns for pygame with classes

Game Loop

The main game logic can be divided into two parts:

1. Initialize the variables
2. **Run the game loop which does the following steps:**
    (a) Handle Events
    (b) Update objects
    (c) Draw

```
import pygame

class Game:

    def initialize(self):

        ## start pygame's engines
        pygame.init()

        ## get a screen
        self.screen = pygame.display.set_mode(WINDOW_SIZE)

        ## get a clock used for FPS control
        self.clock = pygame.time.Clock()

        self.example_box = pygame.Rect(0, 0, 100, 100)

    def run(self):
        ## a simple flag variable for the loop
        done = False

        ## the main game loop
        while not done:

            ### 1. Events

            ## the event loop; used to check for events that occurred since the last time around
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    done = True

            ### 2. Updates
            ## update the example box with whatever you want
            self.example_box.x += 1


            ## 3.  Drawing
            pygame.draw.rect(self.screen, BLACK, self.example_box)

            #### update the display and move forward 1 frame
            pygame.display.flip()
            # --- Limit to 60 frames per second
            self.clock.tick(FPS)
```

Basic Sprites

There are several ways to include objects, monsters, obstacles, etc in your pygame code. The best way is to define your own classes that inherit from pygame's Sprite class.

You should think of this as defining recipes for different objects in your game. In this section, there are the following recipes:

1. **A basic sprite**
    - the core components of a sprite and how to use them
2. **Adding the drawing function to the basic sprite**
    - You can put the logic for the sprites inside the class, so it makes the game logic cleaner
    - Your game shouldn't have to worry about how sprites get drawn!
3. **Colliding with one other sprite**
    - Colliding with another sprite is handled just like in the simple case
    - The trick is to correctly identify how the collision happened so you can fix it!
4. **Using Groups of sprites**
    - Group is a special pygame object that gives us extra shortcuts!
5. **Colliding with many sprites**
    - Using a Group, we can easily get the list of sprites our main sprite is colliding with
6. **Adding an image to your sprite**
    - Usually you will want to draw more than basic shapes. This will show you how!
7. **Adding event handling to your sprite**
    - If you want your sprite to do things, it should handle its own event logic!
    - This means that the game just gives the events to the sprite and the sprite does what it needs to do.
8. **Making an animated sprite**
    - This will show you how the basic animation happens

Basic Sprite

For our basic sprite, we will **subclass** pygame's sprite class. Subclassing means that we will tell python that our new class is the exact same as pygame's sprite class. Then, whatever we can specialize any parts we want.

```
class BasicSprite(pygame.sprite.Sprite):

    # by defining this function, we are overriding the parent class's function
    def __init__(self, color, width, height):

        # this is a special command which tells python to execute the parent's function
        # the pattern is
        # super(ThisClassName, self).func_to_call()
        super(BasicSprite, self).__init__()

        ### When you sublcass the sprite, you need two things

        #  1. self.image
```

```
        self.image = pygame.Surface([width, height])
        self.image.fill(color)

        #  2. self.rect

        self.rect = self.image.get_rect()

        # self.rect starts out at 0,0. if you want to change the location, you have to update these
        # this hard codes the BasicSprite to start at the coordinates 50,50
        self.rect.x = 50
        self.rect.y = 50
```

You can use this class in the same places you would before:

1. **Instantiate** (create) the object at the beginning of the game
2. **Update** the coordinates inside the game loop
3. **Draw** the coordinates inside the game loop

One of the nice features about using sprites is that we only have to draw the sprite's `self.image` property. We do this with the following:

```
class Game:

    def initialize(self):
        # other code was here

        # just remember that our screen is made here
        self.screen = pygame.display.set_mode(WINDOW_SIZE)

        self.example_object = BasicSprite(BLACK, 100, 100)

    def run(self):
        done = False

        ## the main game loop
        while not done:

            # other code was here


            ## the way to read this dot notation is:
            ## inside this Game object access (using "self") a variable called example_object
            ## inside example_object is the property "image" (which we defined just above)
            ## inside image is a function called blit
            ## blit takes two arguments:
            ##      1. the surface it should draw on, this is our screen.
            ##      2. the coordinates of where to draw it. this is the rect inside example_object
            ## overall, the syntax is:
            ##      surface_variable.blit(screen_variable, rect_variable)

            self.example_object.image.blit(self.screen, self.example_object.rect)


            ## then don't forget the rest of the code here
```

So, to summarize:

1. Subclass pygame's `Sprite` class and define the `self.image` and `self.rect`.

2. **Inside the `Game` object's initialize function, use the class to make a new object**
    - save this object to the `self` variable so we can access it later
3. **Inside the `Game` object's run function, use the saved object to draw**
    - the syntax for drawing a sprite is showing above.
    - You are calling `blit` to draw the sprite's surface onto the main surface.

Adding the drawing function to the basic sprite

Doing that drawing logic inside the game loop is a bit messy. Also, maybe we want to change how we draw the object based on some situation. We don't want to have the main game loop get all messy with that code.

To solve this problem, we put a `draw` function inside the `BasicSprite` class

```
class BasicSprite(pygame.sprite.Sprite):

    # by defining this function, we are overriding the parent class's function
    def __init__(self, color=BLACK, width=100, height=100):
        # notice it has default values for its paremeters!

        # this is a special command which tells python to execute the parent's function
        # the pattern is
        # super(ThisClassName, self).func_to_call()
        super(BasicSprite, self).__init__()

        ### When you sublcass the sprite, you need two things

        #  1. self.image

        self.image = pygame.Surface([width, height])
        self.image.fill(color)

        #  2. self.rect

        self.rect = self.image.get_rect()

        # self.rect starts out at 0,0. if you want to change the location, you have to update these
        # this hard codes the BasicSprite to start at the coordinates 50,50
        self.rect.x = 50
        self.rect.y = 50



    def draw(self, screen):
        # draw this object's image onto the passed in screen variable
        screen.blit(self.image, self.rect)
```

Moving a sprite

Moving a sprite is really easy! Everytime through the game loop, the sprite is drawn using its internal `rect` object, which stores the location coordinates.

To move it, we just change those coordinates before it is drawn!

We are going to have a theme with this code. Any functionality we want our `BasicSprite` to have, we will put it inside that class!

To illustrate how you can subclass and keep specializing, let's subclass our previous `BasicSprite` to make a `MovingSprite`:

```
class MovingSprite(BasicSprite):
    # MovingSprite has all the functions and properties that
    # BasicSprite has

    def move(self, dx, dy):
        ## move dx units in the x direction
        ## move dy units in the y direction

        self.rect.x += dx
        self.rect.y += dy
```

Now, let's change one more thing about this. Let's alter the `__init__` function so that the dx and dy are internal!

```
class MovingSprite(BasicSprite):
    # MovingSprite has all the functions and properties that
    # BasicSprite has
    def __init__(self, color=BLACK, width=100, height=100):
        super(MovingSprite, self).__init__(color, width, height)

        self.dx = 0
        self.dy = 0

    def move(self):
        ## move dx units in the x direction
        ## move dy units in the y direction

        self.rect.x += self.dx
        self.rect.y += self.dy
```

Colliding with one other sprite

Pygame provides several ways to handle collisions with sprite objects.

From the documentation, it says the following thing:

```
pygame.sprite.collide_rect()

Collision detection between two sprites, using rects.

collide_rect(left, right) -> bool

Tests for collision between two sprites. Uses the pygame rect colliderect function to calculate the c
Intended to be passed as a collided callback function to the *collide functions. Sprites must have a
```

Basically, this means that you can give this function two sprites and it will tell you `True` or `False`.

We are going to have a theme with this code. Any functionality we want our sprite objects to have, we will put it inside that class!

To illustrate how you can subclass and keep specializing, let's subclass our previous `BasicSprite` to make a `CollisionSprite`:

```
class CollisionSprite(BasicSprite):
    # CollisionSprite has all the functions and properties that
    # BasicSprite has, which has all of the functions BasicSprite has!


    def handle_collision(self, other_sprite, dx, dy):
        # we are going to define the logic for handling the collision with
        # one other sprite

        # there are two extra variables this function is taking.
        # they are the dx and dy. we need these so we know which direction
        # the sprite is moving!
        # Note: we want to make sure we only move x or y.
        # if we are moving both, then we don't know whether the collision
        # is from the top/bottom or from the sides.

        if dx != 0 and dy != 0:
            # this syntax is:
            #   "raise" is a way of manually throwing errors and exceptions
            #   "Exception" is the default exception
            # by doing
            #   raise Exception(some_message)
            # we are stopping the program and causing an error.
            raise Exception("ERROR: don't move both x and y at the same time; Collision checking is i


        if pygame.sprite.collide_rect(self, other_sprite):
            ## if this "if" is true, then this means a collision is happening!
            ## let's check and see which direction it is

            ## check if the sprite is moving in the x direction:
            # if dx is positive, it is moving right
            # if the right side is past the other rect's left, snap them together
            if dx > 0 and self.rect.right > other_sprite.rect.left:
                self.rect.right = other_sprite.rect.left

            # if dx is negative, it is moving up
            # if the left side is past the other rect's right, snap them together
            elif dx < 0 and self.rect.left < other_sprite.rect.right:
                self.rect.left = other_sprite.rect.right

            # if dy is positive, it is moving down
            # if the bottom is past the other rect's top, snap them together
            if dy > 0 and self.rect.bottom > other_sprite.rect.top:
                self.rect.bottom = other_sprite.rect.top

            # if dy is negative, it is moving up
            # if the top is past the other rect's bottom, snap them together
            elif dy < 0 and self.rect.top < other_sprite.rect.bottom:
                self.rect.top = other_sprite.rect.bottom




    ## Let's re-write the move function from before to handle collisions
    def move(self, other_sprite=None):
        ## we will assume that we are given access to a single other sprite
```

```
        ## as an argument to this function
        ## we will give it a default value of None though, so it's only optional

        ## move dx units in the x direction
        self.rect.x += self.dx

        if other_sprite is not None:
            # handle the x collision!
            self.handle_collision(other_sprite, self.dx, 0)

        ## move dy units in the y direction
        self.rect.y += self.dy

        if other_sprite is not None:
            # handle the y collision!
            self.handle_collision(other_sprite, 0, self.dy)
```

Using Groups of sprites

Pygame's `Group` class is really useful for storing objects. We would use it inside the `initialize` function of `Game` so store each of the sprites that we create.

```
class Game:

    def initialize(self):
        # other code was here

        # just remember that our screen is made here
        self.screen = pygame.display.set_mode(WINDOW_SIZE)

        ## use group to manage a list of basic sprites
        self.basic_sprites = pygame.sprite.Group()

        # let's create a couple basic sprites
        for i in range(5):
            # create the new sprite
            # notice no self variable
            # that's because I know I'm not saving this inside self
            # instead, I'm saving this inside self.basic_sprites
            new_sprite = BasicSprite(BLACK, 100, 100)

            # doing this to offset the sprites so we can see them
            new_sprite.rect.x += i * 50
            new_sprite.rect.y += i * 50

            # save it to self.basic_sprites
            self.basic_sprites.add(new_sprite)


    def run(self):
        done = False

        ## the main game loop
        while not done:

            # other code was here

```

```
            # because you used a group to handle the basic sprites, you
            # can shortcut the drawing of them by using group's draw function:

            self.basic_sprites.draw(self.screen)
```

Colliding with many sprites

First, we are going to add some functionality to our `CollisionSprite` to handle group collisions!

```
class GroupCollisionSprite(CollisionSprite):
    # CollisionSprite has all the functions and properties that
    # CollisionSprite has, which has all of the functions CollisionSprite has!

    def handle_group_collision(self, sprite_group, dx, dy):
        # we pass in the "sprite_group", and the movements again

        # the False here is the option to remove all sprites being collided with
        # from the group.
        # if True, sprite_group will no longer have them and they won't be drawn anymore
        # the returned object, colliding_sprites, is a list of sprites!
        colliding_sprites = pygame.sprite.spritecollide(self, sprite_group, False)

        # go through each of the sprites in this list
        for sprite in colliding_sprites:

            # use the function from CollisionSprite to handle this!

            self.handle_collision(sprite, dx, dy)



    ## Let's re-write the move function from before to handle group collisions
    def move(self, collision_group=None):
        ## we will assume that we are given access to a single other sprite
        ## as an argument to this function
        ## we will give it a default value of None though, so it's only optional

        ## move dx units in the x direction
        self.rect.x += self.dx

        # make sure it's not the default value
        if collision_group is not None:
            # handle the x collision!
            self.handle_group_collision(collision_group, self.dx, 0)

        ## move dy units in the y direction
        self.rect.y += self.dy

        # make sure it's not the default value
        if collision_group is not None:
            # handle the y collision!
            self.handle_group_collision(collision_group, 0, self.dy)
```

Now that we have `GroupCollisionSprite` which can handle colliding with a group of sprites, let's add it into `Game`.

```
class Game:

    def initialize(self):
        # other code was here

        # just remember that our screen is made here
        self.screen = pygame.display.set_mode(WINDOW_SIZE)

        ## use group to manage a list of basic sprites
        self.basic_sprites = pygame.sprite.Group()

        # let's create a couple basic sprites
        for i in range(5):
            # create the new sprite
            # notice no self variable
            # that's because I know I'm not saving this inside self
            # instead, I'm saving this inside self.basic_sprites
            new_sprite = BasicSprite(BLACK, 100, 100)

            # doing this to offset the sprites so we can see them
            new_sprite.rect.x += i * 50
            new_sprite.rect.y += i * 50

            # save it to self.basic_sprites
            self.basic_sprites.add(new_sprite)


        # it has the same __init__ function as BasicSprite
        self.hero = GroupCollisionSprite(BLACK, 100, 100)



    def run(self):
        done = False

        ## the main game loop
        while not done:

            # other code was here

            # remember the loop order:
            # Events, Updates, and then Draw

            # Updates is where collisions and movement goes
            # let's move the hero and have it handle sprite collision!
            self.hero.move(self.basic_sprites)

            # because you used a group to handle the basic sprites, you
            # can shortcut the drawing of them by using group's draw function:

            self.basic_sprites.draw(self.screen)
            self.hero.draw(self.screen)
```

Adding an image to your sprite

Adding an image is super easy! The main thing is to change how `self.image` gets defined!

Since our class, `GroupCollisionSprite` has so much functionality now, let's just subclass it and override the `__init__` function:

```
class ImageSprite(GroupCollisionSprite):

    def __init__(self, image_filename, colorkey=WHITE):

        # because all of the arguments in BasicSprite were optional, we
        # can just call the init function
        super(ImageSprite, self).__init__()

        # now, we overwrite image
        self.image = pygame.image.load(image_filename).convert()

        # Set our transparent color
        self.image.set_colorkey(colorkey)

        # refresh the rect now
        self.rect = self.image.get_rect()
```

And that's it!

If you wanted to do this without subclassing `GroupCollisionSprite`, you could just subclass `pygame.sprite.Sprite` again and define `self.image` in this way.

Adding event handling to your sprite

It's really useful to be able to handle keyboard input! In fact, if you want people to play your game, it has to be able to handle input.

There are two ways you could do this. You could add code inside `Game` which will manually update the `hero`. But we don't want `Game` to care about such things!

So, instead, we will let `Game` just give every single event to the hero!

```
class Game:




    def run(self):
        done = False

        ## the main game loop
        while not done:

            ## the event loop

            ## the event loop; used to check for events that occurred since the last time around
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    done = True
                else:
                    # if the event isn't a quitting event, give it to the hero!
                    self.hero.handle_event(event)
```

And that's it! Now, writing this code creates an expectation from python that our `hero` will have this function implemented. So, let's do that.

```
class EventHandlingSprite(ImageSprite):
    # I inherited from the ImageSprite
    # if you don't want to do this, you can replace ImageSprite with GroupCollisionSprite
    # since that was our second most advanced sprite so far

    # remember, because we are inheriting, we get all of the functionality from before!

    def handle_event(self, event):
        # there are a couple of different pygame events:
        if event.type == pygame.KEYDOWN:
            # this is a keydown event
            # this means a key is pressed

            if event.key == pygame.K_LEFT:
                self.dx = -5
            elif event.key == pygame.K_RIGHT:
                self.dx = 5
        elif event.type == pygame.KEYUP:
            # this is a keyup event
            # this means a key was let go

            if event.key == pygame.K_LEFT:
                self.dx = 0
            elif event.key == pygame.K_RIGHT:
                self.dx = 0
```

Thsi is really simple event handling. For instance, if you press two keys at once, this will have some weird results. But at least it will handle some input!

To overcome the two-keys-at-once problem, you will have to do something a bit more complicated. For instance, you could have the left key subtract 5 from `self.dx` and then use `min` to make sure it is never smaller than -5. You could also have some boolean variables that are internal to the sprite which keep track of which keys have been pressed.

Making an animated sprite

Basic Game Physics

Physics is very important to games! Since you are telling the game how each object updates, you have to use math to update the objects to match how physics works. This can sometimes be hard, but there are plenty of ways to make it easier.

In this section, there are the following recipes:

1. **Bouncing off walls**
    - If an object is moving in a direction and encounters an obstacle, it could bounce
    - Bouncing in certain ways looks and feels weird
    - So, you should bounce in a way that feels real!
2. **Gravity**
    - Instead of letting objects freely move in both x and y directions, gravity constantly affects the y!
    - You can think of this as making so that your object always wants to be moving down at 9 units at a time
3. **Jumping**

- Jumping is just the opposite of gravity
- When the jump happens, there is a force which makes the object want to move up at 9 units!
- In other words, the y speed is set to -9
- Then, every frame, the speed slowly goes back to +9.

Handling Keyboard Input

1. **Basic keyboard input**
    - handle single keys
    - do specialized things
2. **Continuous keyboard input**
    - continue to do something until key is released
    - this is basically the example in the earlier section!
3. **Advanced continuous keyboard input**
    - use extra variables to keep track of which key was pressed!

Scoreboards

1. **Drawing an extra surface that never moves**
    - In the same logic as the sprite, except that it doesn't move and is always drawn last.

Menus

1. **Use a "card" concept to draw different viewpoints**
    - A "card" is a certain way the game is
    - The standard one is your actual game
    - The menu one handles menu inputs and draws the menu
    - Inside the game loop, you check which card is active and give all event, update, and draw information to it.
    - The card then gives all up the information to its members.

    The concept of a card view is pretty simple. Instead of having the `Game` object initalize, update, and draw everything, it instead only calls the functions of a card. Then, the card calls everything.

```
import pygame

from settings import *

class Card:
    def __init__(self, objects):
        self.objects = objects

    def update(self):
        pass
```

```
        def draw(self, screen):
            self.objects.draw(screen)

    class Cards:
        def __init__(self, cards, initial_card):
            self.cards = cards
            self.current_card = self.cards[initial_card]

        def update(self):
            self.current_card.update()

        def draw(self, screen):
            self.current_card.draw(screen)

        def switch(self, card_name):
            if card_name in self.cards.keys():
                self.current_card = self.cards[card_name]


    class Game:
        def __init__(self, cards, initial_card):
            pygame.init()
            self.screen = pygame.display.set_mode(WINDOW_SIZE)
            self.clock = pygame.time.Clock()
            pygame.display.set_caption(TITLE)
            self.cards = cards

        def run(self):
            done = False

            while not done:

                card = self.cards.current_card()

                for event in pygame.event.get():

                    if event.type == pygame.QUIT:
                        done = True
                    else:
                        current_card.handle_event(event)

                if current_card.is_switching == True:
                    self.current_cardname = current_card.get_next_card()
                    current_card = self.cards[current_card]

                current_card.update_all()

                current_card.draw(self.screen)

                pygame.display.flip()
                self.clock.tick(FPS)
            pygame.quit()


    game = Game()
    cards = {'test': BasicCard()}
    cards['test'].add_text(Text('Awesome'))
```

```
    cards['alt'] = BasicCard()
    cards['alt'].add_text(Text('Sauce'))
    game.set_cards(cards, 'test')
    game.run()
```

2.14.5 Cookbook

A set of common recipes and design patterns

Game Loop

```
import pygame

## start pygame's engines
pygame.init()

## set the screen size
WINDOW_SIZE = (700, 500)

## get a screen
screen = pygame.display.set_mode(WINDOW_SIZE)

## get a clock used for FPS control
clock = pygame.time.Clock()

## a simple flag variable for the loop
done = False


## the main game loop
while not done:

    ## the event loop; used to check for events that occurred since the last time around
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True


    #### update the display and move forward 1 frame
    pygame.display.flip()
    # --- Limit to 60 frames per second
    self.clock.tick(FPS)
```

Drawing

Using Rect to draw

Rect is a useful PyGame class that is a wrapper around the standard rectangle information.

```
x = 0
y = 0
width = 100
height = 100
r1 = pygame.Rect(x, y, width, height)
```

The variable `r1` now has access to a variety of different properties

```
x,y
top, left, bottom, right
topleft, bottomleft, topright, bottomright
midtop, midleft, midbottom, midright
center, centerx, centery
size, width, height
w,h
```

You can also update `r1` using any of those variables. For example:

```
r1.center = (50,50)
r1.right = 10
r1.bottomright = 75
```

Bouncing off obstacles

Basic collision detection with screen boundaries

In the simplest case, we are testing to see if our rect is over some threshold. This happens in the case of bouncing off the edges of the screen. For this example, we assume we know the height and width of the window as well.

```
# W, H are window width and window height

if r1.right > W:
    print("Over right side")
elif r1.left < 0:
    print("over left side")

if r1.top < 0:
    print("Over top")
elif r1.bottom > H:
    print("Over bottom")
```

Changing direction based on screen boundary collision

Let's assume that the object in question is moving at some speed. In other words, the `x` and `y` properties are being updated by some variable `dx` and `dy`. Then, when the object bounces, it should flip the signs of those speeds.

```
# W, H are window width and window height
r1.x += dx

if r1.right > W or r1.left < 0:
    dx *= -1

r1.y += dy
if r1.top < 0 or r1.bottom > H:
    dy *= -1
```

Colliding with another Rect

If you wanted to collide with another Rect, there are several different ways you could it. The easiest way is to use the built in functions which test for collision. However, these functions don't tell you which parts collided. An example of why this is a problem:

- There is a collision with a Rect and an obstacle from the bottom
- The Rect's right side is technically past the obstacle's left
- But, the issue is the y-movement, not the x-movement.

The first part of the solution is to update the X and Y parts separately. With this method, one dimension is changed and checked for collisions. Then, the other is changed and checked for collisions.

The second part of the solution is to "snap" the edges of the object and the obstacle together. This just means making them line up exactly so no more collision is taking place.

The below code illustrates the Rect collision code, the separate x and y movements, and the edge snapping.

```
'''
in this example, self.rect is the rect of the object you are moving
'''


def move(self, dx, dy, other_rects):

    # move this object in the x direction
    self.rect.x += dx

    # go over each obstacle
    for other_rect in other_rects:


        # if there is a collision
        # since we moved only the x, we know it has to be this object's left or right
        if self.rect.colliderect(other_rect):

            # if dx is positive, it is moving right
            # if the right side is past the other rect's left, snap them together
            if dx > 0 and self.rect.right > other_rect.left:
                self.rect.right = other_rect.left

            # if dx is negative, it is moving left
            # if the left side is past the other rect's right, snap them together
            elif dx < 0 and self.rect.left < other_rect.right:
                self.rect.left = other_rect.right


    # move this object in the y direction
    self.rect.y += dy

    # go over each obstacle
    for other_rect in other_rects:

        # if there is a collision
        # since we moved only the y, we know it has to be this object's top or bottom
        if self.rect.colliderect(other_rect):

            # if dy is positive, it is moving down
            # if the bottom is past the other rect's top, snap them together
            if dy > 0 and self.rect.bottom > other_rect.top:
                self.rect.bottom = other_rect.top

            # if dy is negative, it is moving up
            # if the top is past the other rect's bottom, snap them together
```

```
47                elif dy < 0 and self.rect.top < other_rect.bottom:
48                    self.rect.top = other_rect.bottom
```