Philipps Universität Marburg

Game Programming: From Scratch to Python

Tuesday, August 30 · 9:30 – 11:30am

Python Review

Review exercise

Finish the for-loop such that it print the value if it is an even number and less than 10, or if it is an odd number and greater than 10.

```
for i in range(1,21):
    """ your code below """
```

Rules:

1. Include at least one **if** and one **elif**
2. Use the **modulo** operator
3. Use **compound operators**

Review solution

Finish the for-loop such that it print the value if it is an even number and less than 10, or if it is an odd number and greater than 10.

```
for i in range(1,21):
    if i < 10 and i % 2 == 0:
        print(i)
    elif i >= 10 and i % 2 == 1:
        print(i)
```

Review: Spot the problem!

What's wrong with this code?

```
my_print_function(message):
    print("Message:", message)

my_print_function("Hello!")
```

Review: Spot the problem!

What's wrong with this code?

`def` was missing before the function name in the function definition

```
def my_print_function(message):
    print("Message:", message)

my_print_function("Hello!")
```

Review: Spot the problem!

What's wrong with this code?

```
score = 0
lives = 3
while lives != 0:
    if banana_position == ground_position:
        go_to_x(banana_position)
    if banana_position == player_position:
        go_to_x(banana_position)
        score = score + 1
```

Review: Spot the problem!

What's wrong with this code?

The **stopping condition** will never be reached causing an **infinite loop**

We need to subtract 1 from lives when the banana hits the ground

```
lives = lives - 1
```

```
score = 0
lives = 3
while lives != 0:
    if banana_position == ground_position:
        go_to_x(banana_position)
        lives = lives - 1
    if banana_position == player_position:
        go_to_x(banana_position)
        score = score + 1
```

Review: True or False?

```
materials = ["clay", "wood", "stone"]
materials[1] == "clay"
```

Review: True or False?

```
materials = ["clay", "wood", "stone"]
materials[1] == "clay"
```

False

Review: True or False?

```
materials = ["clay", "wood", "stone"]
materials[1][2] == materials[-1][2]
```

Review: True or False?

```
materials = ["clay", "wood", "stone"]
materials[1][2] == materials[-1][2]
```

True

materials[1] == “wo**o**d”
materials[-1] == “st**o**ne”

Review: True or False?

The code on the left and the code on the right do the same thing.

```
i = 1
while i < 21:
    if i < 10 and i % 2 == 0:
        print(i)
    elif i >= 10 and i % 2 ==
1:
        print(i)
    i = i + 1
```

```
for i in range(1,21):
    if i < 10 and i % 2 == 0:
        print(i)
    elif i >= 10 and i % 2 == 1:
        print(i)
```

Review: True or False?

The code on the left and the code on the right do the same thing.

```
i = 1
while i < 21:
    if i < 10 and i % 2 == 0:
        print(i)
    elif i >= 10 and i % 2 ==
1:
        print(i)
    i = i + 1
```

```
for i in range(1,21):
    if i < 10 and i % 2 == 0:
        print(i)
    elif i >= 10 and i % 2 == 1:
        print(i)
```

True

Review: True or False?

The code on the left and the code on the right do the same thing.

```
i = 1
while True:
    if i < 10 and i % 2 == 0:
        print(i)
    elif i >= 10 and i % 2 ==
1:
        print(i)

    if i == 21:
        break
    i = i + 1
```

```
for i in range(1,21):
    if i < 10 and i % 2 == 0:
        print(i)
    elif i >= 10 and i % 2 == 1:
        print(i)
```

Review: True or False?

The code on the left and the code on the right do the same thing.

```
i = 1
while True:
    if i < 10 and i % 2 == 0:
        print(i)
    elif i >= 10 and i % 2 ==
1:
        print(i)

    if i == 21:
        break
    i = i + 1
```

```
for i in range(1,21):
    if i < 10 and i % 2 == 0:
        print(i)
    elif i >= 10 and i % 2 == 1:
        print(i)
```

False: the code on the left will print 21

Dictionaries review

Complete the for-loop to make the dictionary map the item of the list to the item's index in the list (keys are items and values are indices).

The enumerate function can tell you the iteration (index) you are currently on. That value is stored in the index variable, and the list item is stored in the item variable.

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]
item_to_index = {}

for index, item in enumerate(grocery_list):
    """ complete the for-loop """
```

Dictionaries review

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]
item_to_index = {}

for index, item in enumerate(grocery_list):
    item_to_index[item] = index

print(item_to_index['Onions'], item_to_index["Soup"])
```

A Soft Introduction to Pygame

…and game programming

Main game loop

The “Game Loop” is the heart of the video game.

Three important things happen in the game loop:

1. Handles events (like pressing a key or clicking the mouse)
2. Updates the state of the game (like where the Pacman and the Ghosts are)
3. Draws the Screen (making the game visible to you in the window)

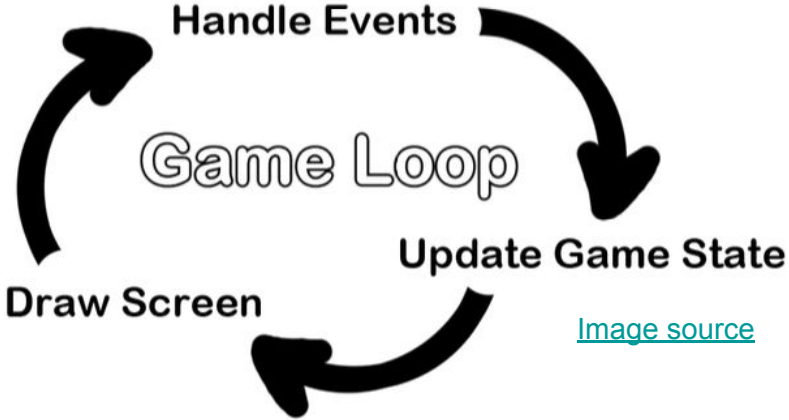


Image source

Main game loop

The “Game Loop” is the heart of the video game.

Three important things happen in the game loop:

1. Handles events (like pressing a key or clicking the mouse)
2. Updates the state of the game (like where the Pacman and the Ghosts are)
3. Draws the Screen (making the game visible to you in the window)

When you click the green flag in Scratch, the game loop begins.

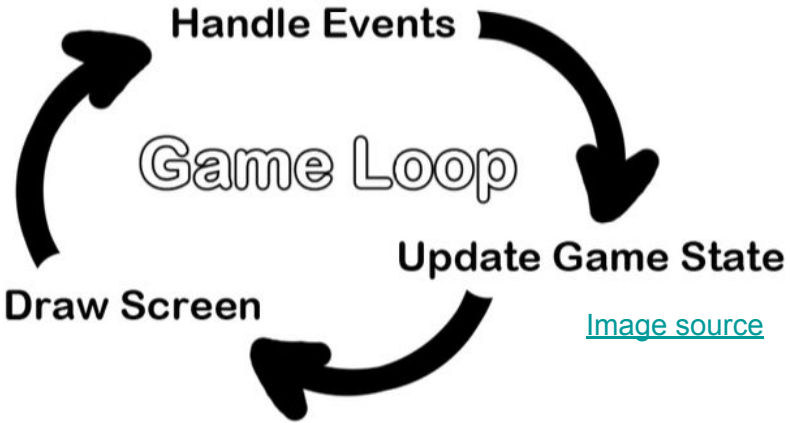
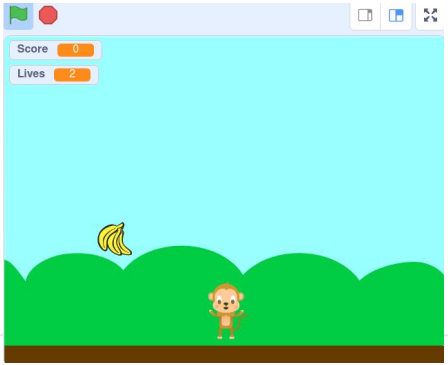


Image source



Main game loop: Pygame event example

https://www.pygame.org/docs/

Pygame is a Python library of functions and objects to help us perform those three big tasks of the Game Loop.

Here, we use Pygame to help us find out if the exit window button was clicked, so we know we write the code to quit the game and end the program.

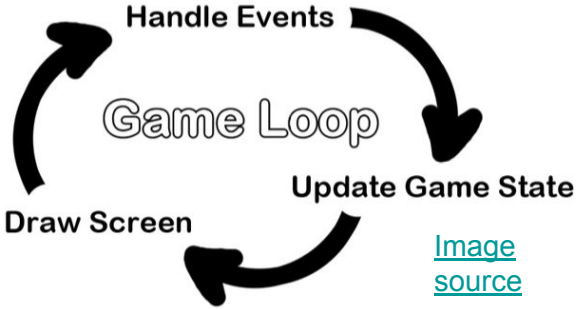


Image source

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

Main game loop

The “Game Loop” is the heart of the video game.

Three important things happen in the game loop:

1. **Handles events** - like pressing a key or clicking the mouse
2. **Updates the state of the game** - like where the Pacman and the Ghosts are
3. **Draws the Screen** - making the game visible to you in the window

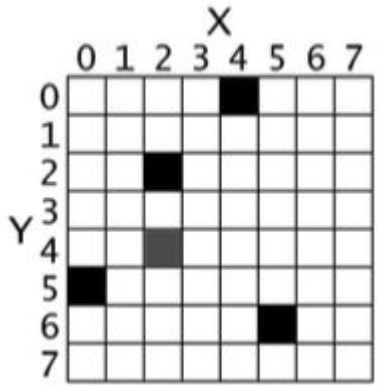
An *iteration* of the game loop is one *frame*

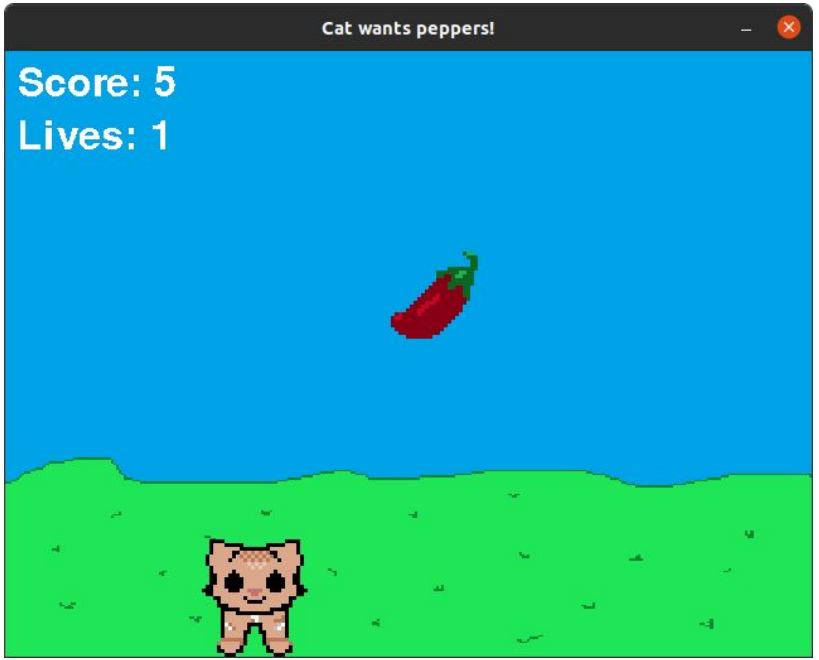
Image source

More about these three tasks and loop iterations later…

Pixel coordinates: Where the things are on the screen

In Pygame, the y-axis increases from top to bottom and the x-axis increases from left to right.

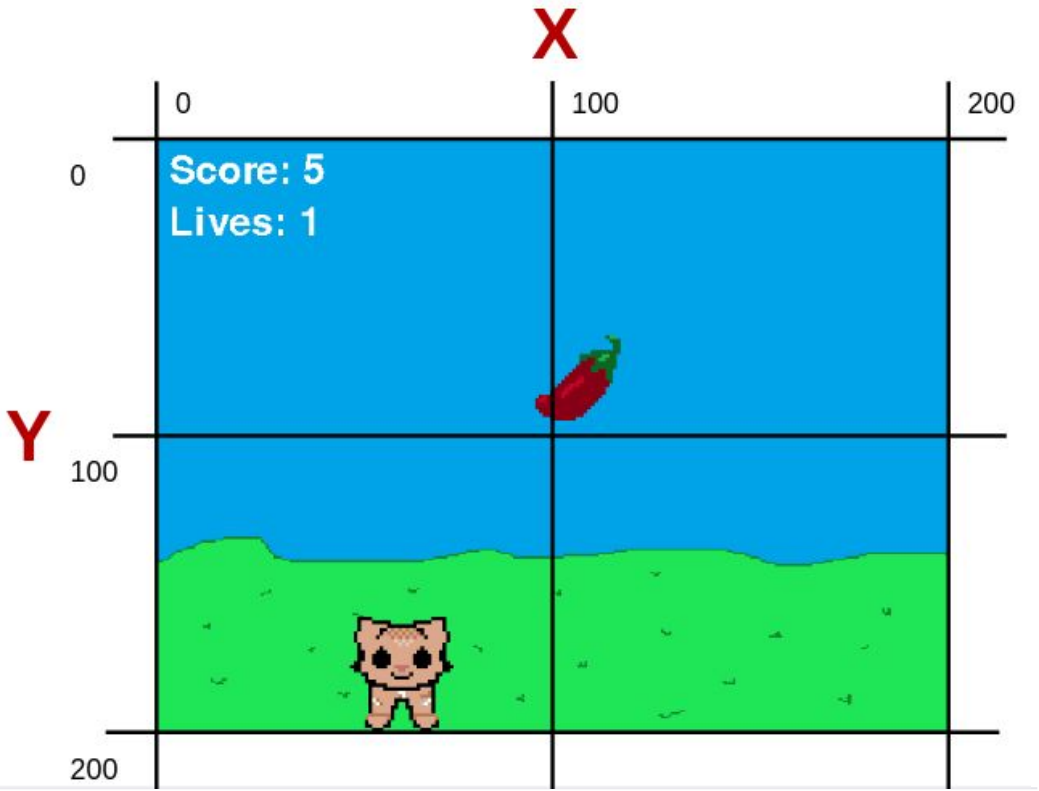




Pixel coordinates: Where the things are on the screen

Where are the following (approximately)

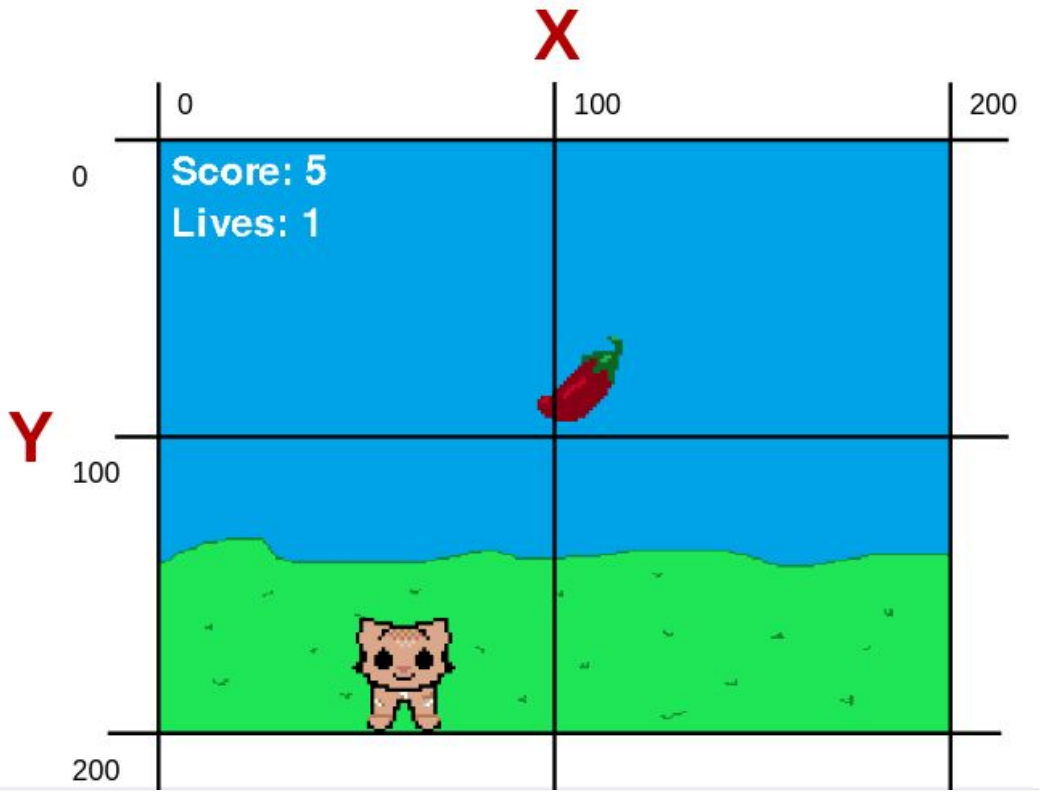
1. Score: 5
   (0 <= x < 50, 0 <= y < 20)
2. Lives : 1
3. Cat:
4. Pepper



Pixel coordinates: Where the things are on the screen

Where are the following (approximately)

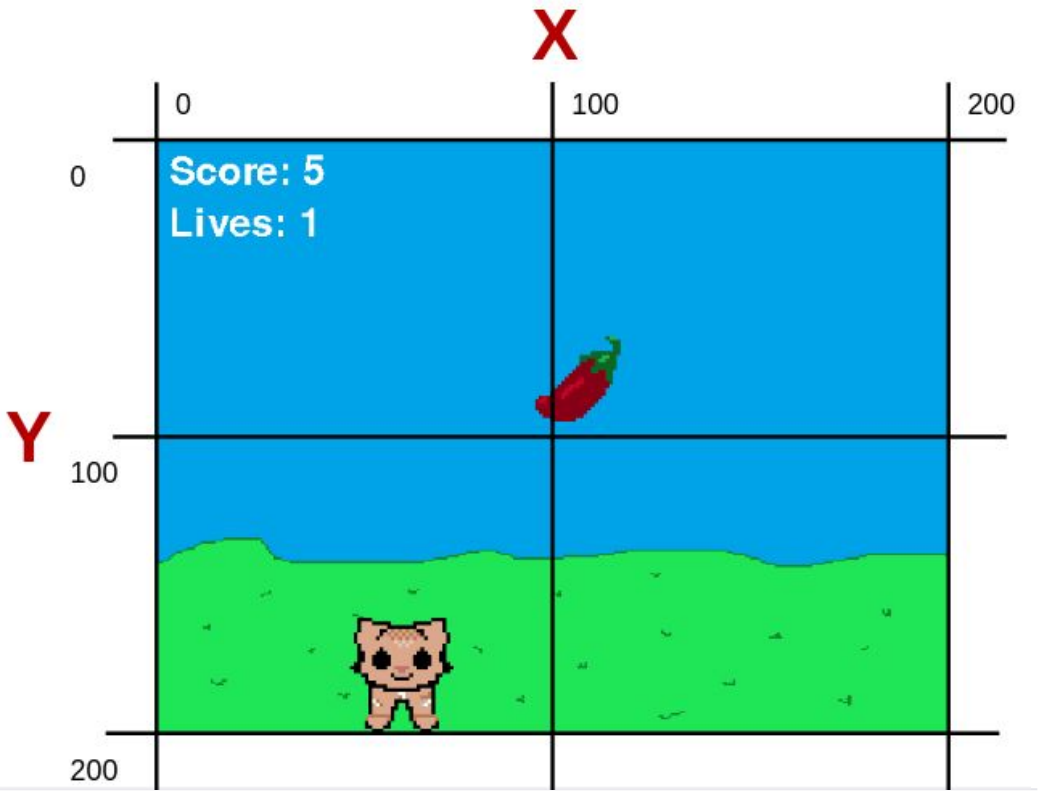
1. Score: 5
   (0 <= x < 50, 0 <= y < 20)
2. Lives : 1
   (0 <= x < 50, 10 <= y < 40)
3. Cat:
4. Pepper



Pixel coordinates: Where the things are on the screen

Where are the following (approximately)

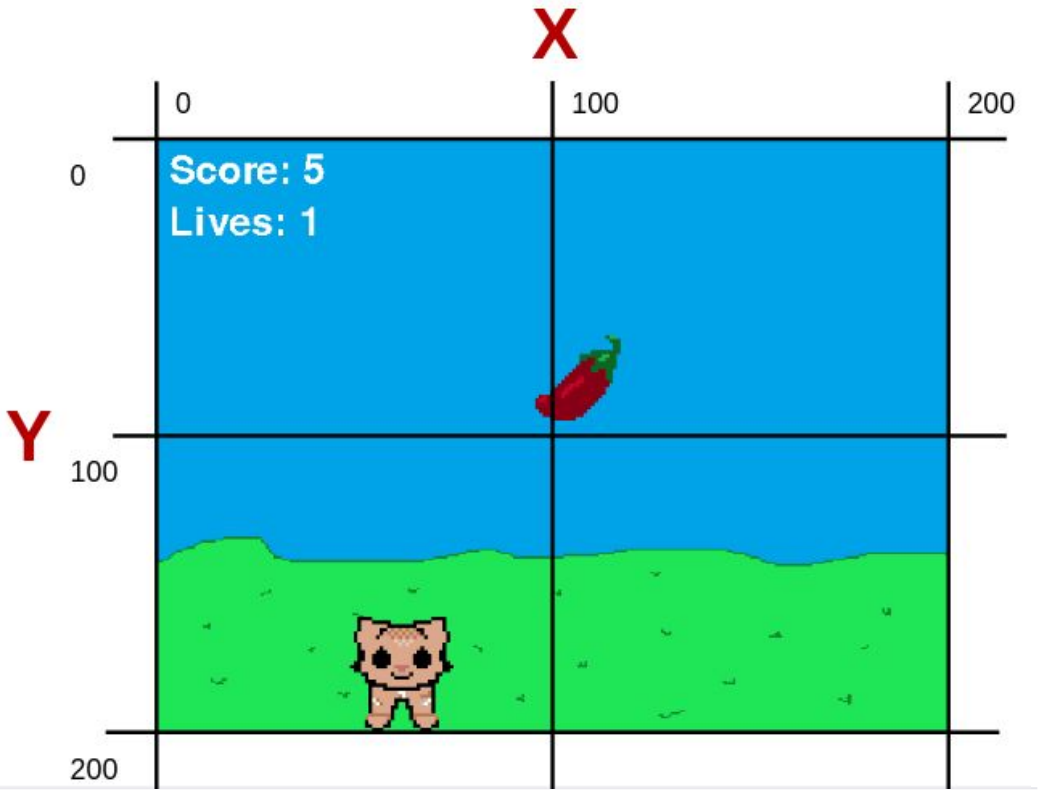
1. Score: 5
   (0 <= x < 50, 0 <= y < 20)
2. Lives : 1
   (0 <= x < 50, 10 <= y < 40)
3. Cat
   (50 <= x < 80, 150 < y < 200)
4. Pepper



Pixel coordinates: Where the things are on the screen

Where are the following (approximately)

1. Score: 5
   (0 <= x < 50, 0 <= y < 20)
2. Lives : 1
   (0 <= x < 50, 10 <= y < 40)
3. Cat
   (50 <= x < 80, 150 < y < 200)
4. Pepper
   (95 < x <= 110, 60 < y < 99)



Sprites, Surfaces, and Rectangles

This is a Sprite

Sprites, Surfaces, and Rectangles

This is a Sprite

The sprite's image is represented by a pygame.Surface object

* You will understand this better later, after you become familiar with **Classes**

Sprites, Surfaces, and Rectangles

This is a Sprite

The sprite's image is represented by a pygame.Surface object

The Sprite has an imaginary rectangle surrounding its surface

The imaginary rectangle is represented by a pygame.Rect object

What you need to know about Rectangles now

The Sprite has an imaginary rectangle surrounding its surface

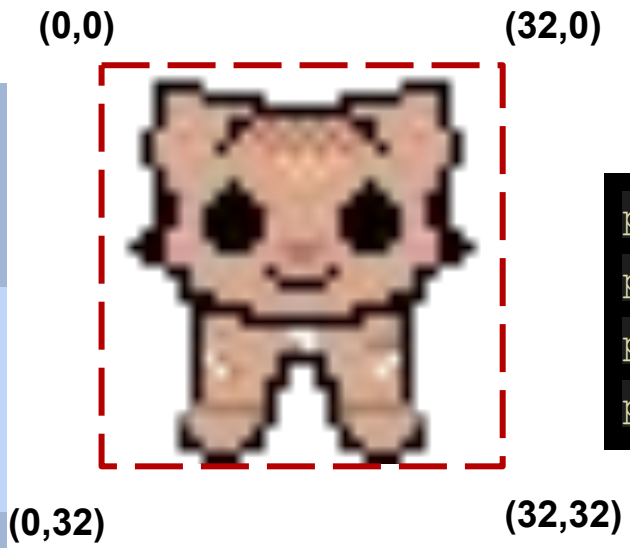
The imaginary rectangle is represented by a pygame.Rect object

We name the sprite's imaginary rectangle '**rect**'

As you can imagine, rect has a width and a height and we can get the width and height easily

```
import pygame
# upper left corner is at (0,0)
rect = pygame.Rect(0, 0, 32, 32)
print(rect.width, rect.height)
```

What you need to know about Rectangles now



The purpose of the Sprite's imaginary rectangle is so that we can keep track of the Sprite's coordinates or location on the screen.

```
print(rect.right) # x value of right side
print(rect.left) # x value of left side
print(rect.top) # y value of top side
print(rect.bottom) # y value of bottom side
```

What you need to know about Rectangles now

The purpose of the Sprite's imaginary rectangle is so that we can keep track of the Sprite's coordinates or location on the screen. **(There are many different attributes about its location)**

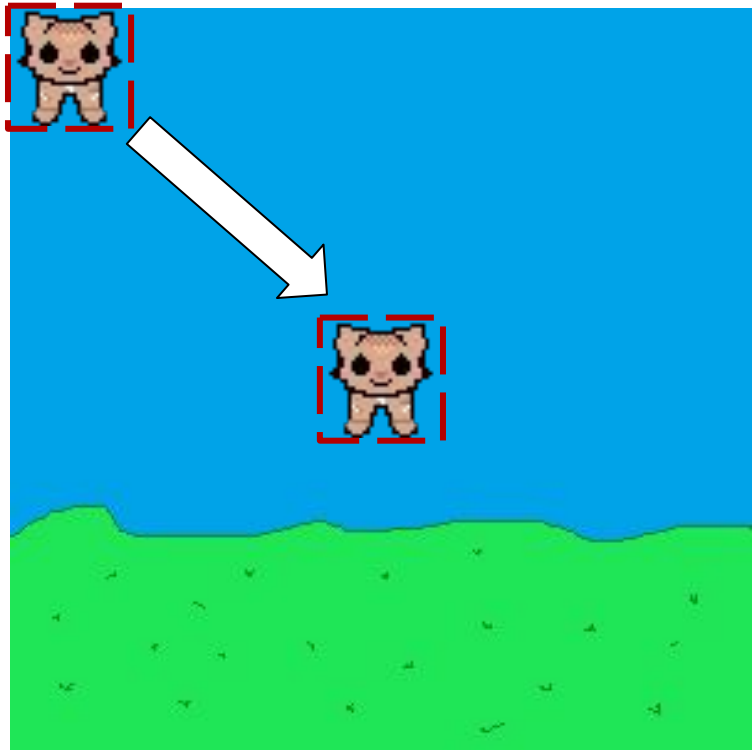
```
x,y
top, left, bottom, right
topleft, bottomleft, topright,
bottomright
midtop, midleft, midbottom, midright
center, centerx, centery
size, width, height
w,h
```

What you need to know about Rectangles now

The Sprite moves if you change any of its values about its position

```
rect.midbottom = (50, 100)
print(rect.right) # x=66
print(rect.left) # x=34
print(rect.top) # y=68
print(rect.bottom) # y=100
```

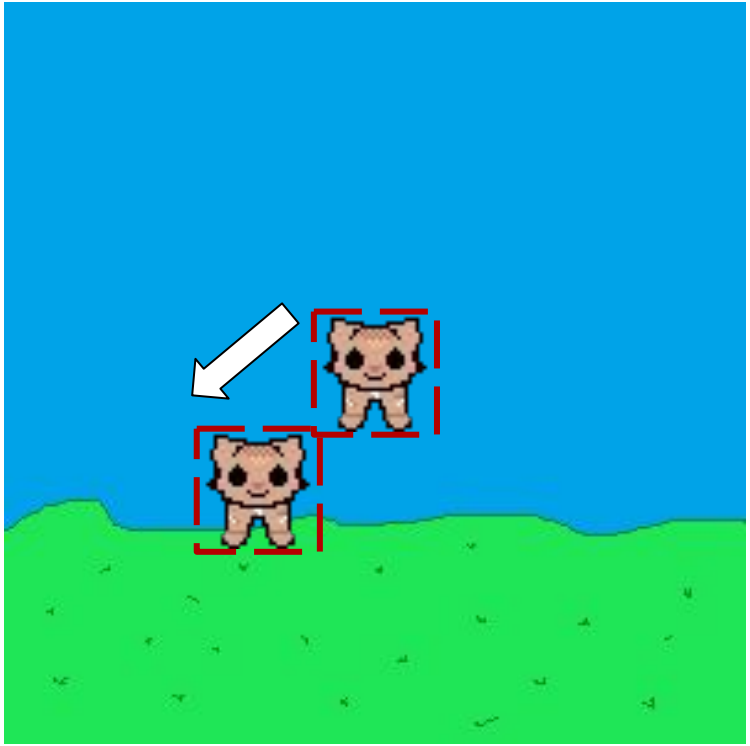
Moving the Sprite



```
rect.topleft = (0, 0)
rect.center = (50, 50)
```

Dimensions of background are not exact ;)

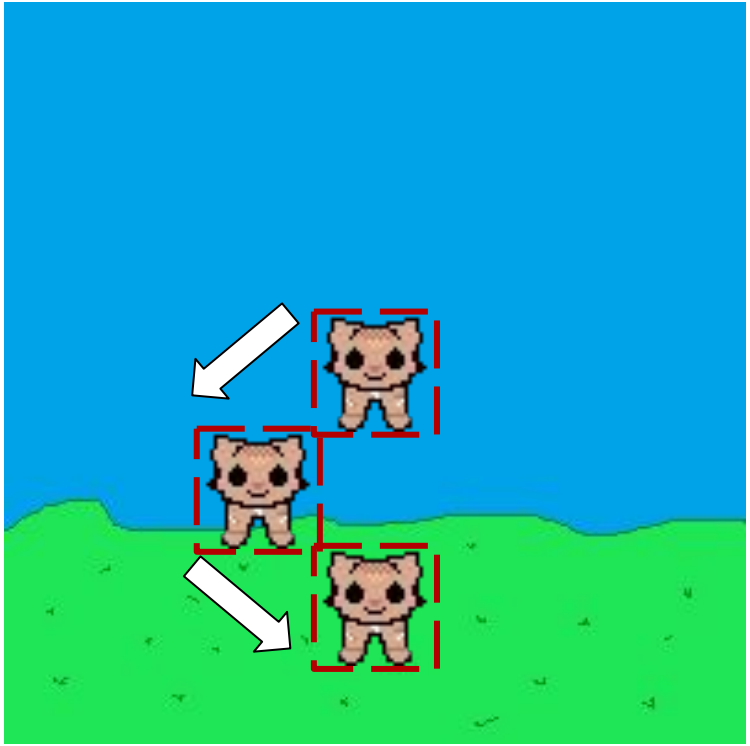
Moving the Sprite



```
rect.center = (50, 50)
rect.topright = rect.bottomleft
```

Dimensions of background are not exact ;)

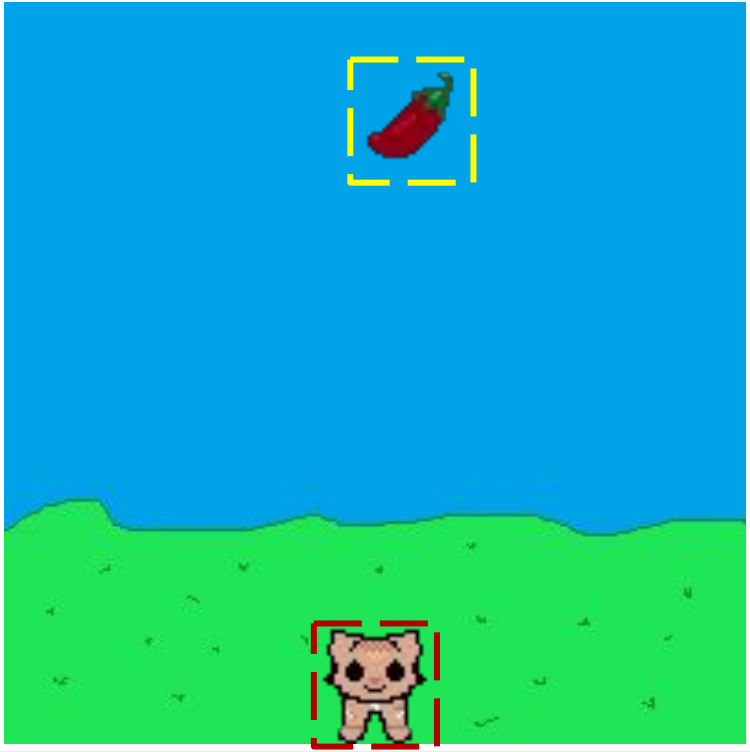
Moving the Sprite



```
rect.center = (50, 50)
rect.topright = rect.bottomleft
rect.topleft = rect.bottomright
```

Dimensions of background are not exact ;)

Collisions exercise



We have two sprites.

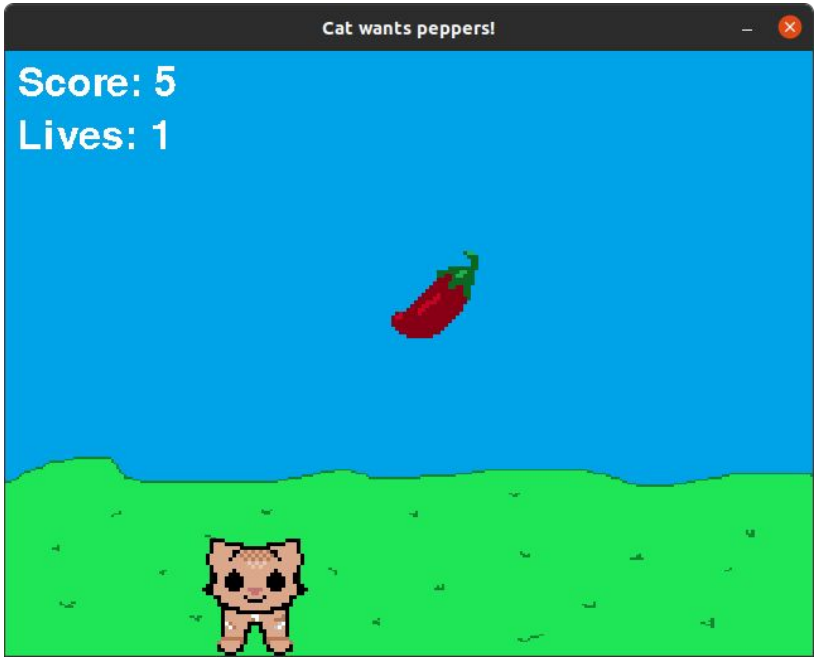
The cat's rectangle is called cat_rect and the pepper's rectangle is called pepper_rect.

We can use the sprite's rectangles to check if they are touching.

How can we do this?

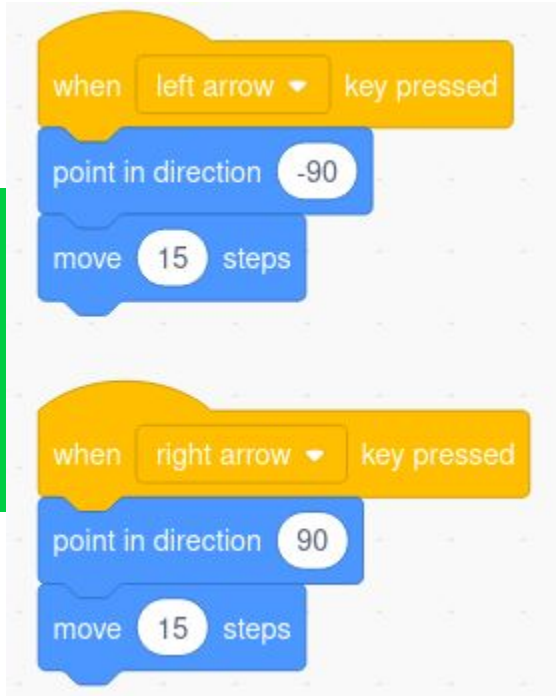
Try to work this out on a piece of paper or on the whiteboard.

Coding challenge



Previously

- We added the ability for the monkey to move right



Previously

- We added the ability for the monkey to move right
- We fixed the banana so it would fall toward the ground and not fly up

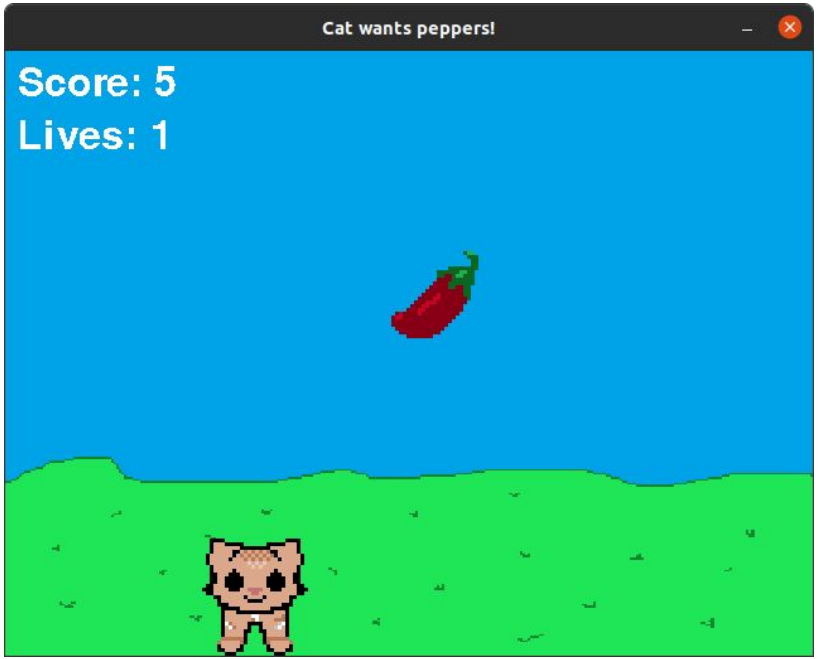
Previously

- We added the ability for the monkey to move right
- We fixed the banana so it would fall toward the ground and not fly up
- We fixed a bug that made the bananas always go to the center of the screen when the Player catches one

Bugs! Again!

Now we have some python code and we need your help with some bugs again!

The Game: The game is the same as the Monkey and Bananas game, except with a cat and hot peppers. The cat wants to catch hot peppers because she knows her owner likes them.



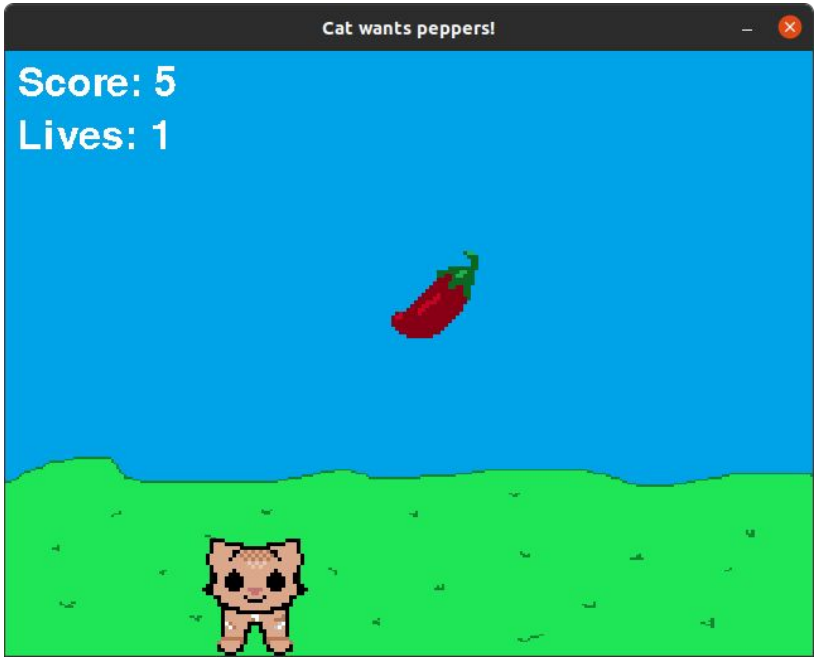
Getting started

To begin you need to download this folder from Google Drive.

Open the whole folder in VS Code.

To run the game, open a terminal in the folder and type **python Game.py**.

When you first run the game, you will notice that it is not working quite right…



The Bugs

1. The pepper flies away! Can you find a bug in the code that is making that happen? The bug is somewhere in **Pepper.py**.

2. We can't move the Player! We need your help making the player move. Help us fix this by completing the **move function** in **Player.py**.

3. The pepper always goes to the center of the screen once the cat catches it! Help us find the code that causes this to happen! The bug is somewhere in **Game.py**.

💡Tips: The bug instructions are also written in the top of each file and in the README file. Solve the bugs in order because they become increasingly more difficult. The print function can be really helpful for debugging. Use it as a tool to help you understand the variables, when certain code runs, etc. Look at the existing code (without bugs) to get ideas for your solutions. Importantly, **help each other and ask questions and HAVE FUN!**

Part 2

Philipps Universität Marburg

Pygame

Tuesday, August 30 · 10:30 – 11:30am

Main game loop

The “Game Loop” is the heart of the video game.

Three important things happen in the game loop:

1. **Handles events** (like pressing a key or clicking the mouse)
2. **Updates the state** of the game (like where the Pacman and the Ghosts are)
3. **Draws the Screen** (making the game visible to you in the window)

An *iteration* of the game loop is one *frame*

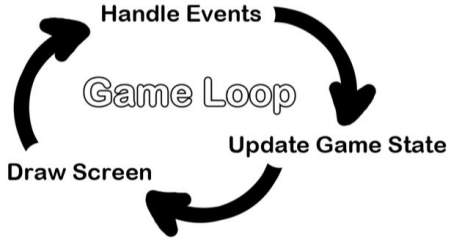
Image source

More about these three tasks and loop iterations later…now!

Basic Game Loop in Pygame

```
import sys, pygame # import the pygame library
from pygame.locals import * # provides constant variables like QUIT

pygame.init() # needed for other pygame functions to work
GAME_SCREEN = pygame.display.set_mode((640, 480)) # dimensions of display
pygame.display.set_caption('Basic game loop') # Adds title to window

while True: # Game loop
    for event in pygame.event.get(): # get events and iterate through them
        if event.type == QUIT: # if user clicks exit button
            pygame.quit() # Deactivates the pygame library
            sys.exit() # ends the whole program
    pygame.display.update()
```

Basic Game Loop in Pygame



Handle Events: Only 1 event is handled (quitting).

Update Game State: We have no game, so there is no game state to update.

Draw Screen: The screen is empty. If we want something to appear, we have to draw it on the **GAME_SCREEN** surface.

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

pygame.init()
GAME_SCREEN = pygame.display.set_mode((640, 480))
pygame.display.set_caption('Basic game loop')


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

Drawing Primitives

The first thing we changed was adding a **pygame.Rect** to correspond with the **GAME_SCREEN pygame.Surface**, so we can easily refer to locations on the screen.

Thus:

screen_rect.center refers to the (x,y) position of the centermost pixel in the whole screen.

```
pygame.init()
screen_rect = pygame.Rect(0,0,640,480)
GAME_SCREEN = pygame.display.set_mode((screen_rect.width, screen_rect.height))
pygame.display.set_caption('Drawing')

WHITE = (255, 255, 255) # rgb color values.
pygame.draw.circle(GAME_SCREEN, WHITE, screen_rect.center, 20)

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

Drawing Primitives

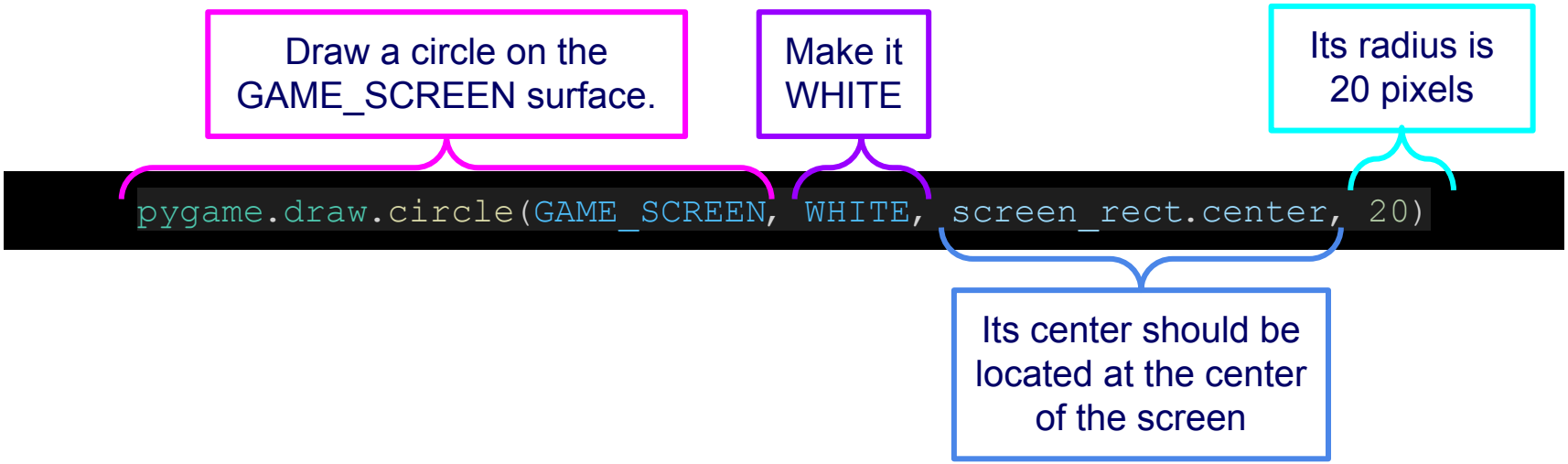
Then we make a variable **WHITE** where we store the rbg (red, blue, green) values that make the color white in the computer.

Let's just focus on that next line of code…

```
pygame.init()
screen_rect = pygame.Rect(0,0,640,480)
GAME_SCREEN = pygame.display.set_mode((screen_rect.width, screen_rect.height))
pygame.display.set_caption('Drawing')

WHITE = (255, 255, 255) # rgb color values.
pygame.draw.circle(GAME_SCREEN, WHITE, screen_rect.center, 20)

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

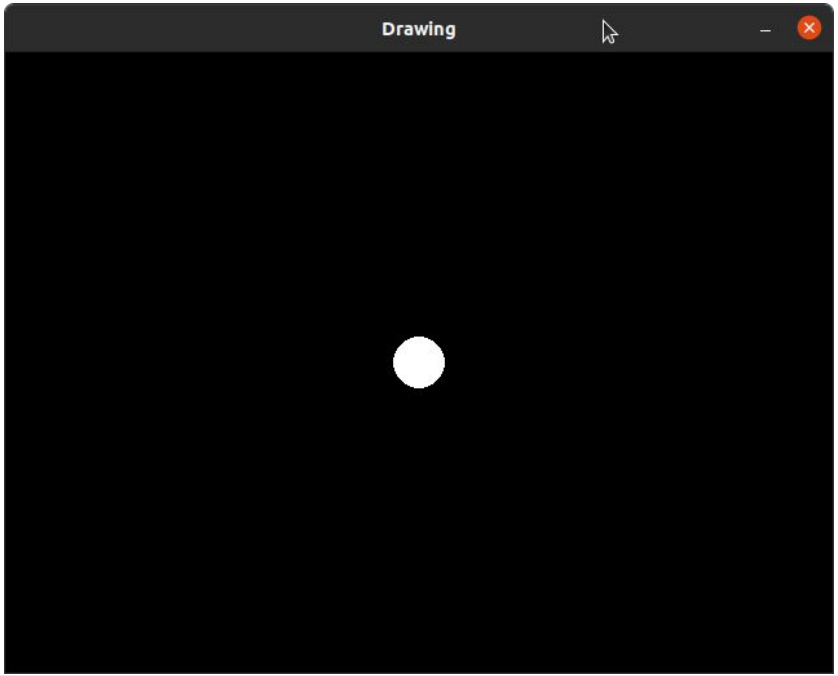
Drawing Primitives

In English, we are telling the computer:



Drawing Primitives: The Result

```
pygame.draw.circle(GAME_SCREEN, WHITE, screen_rect.center, 20)
```

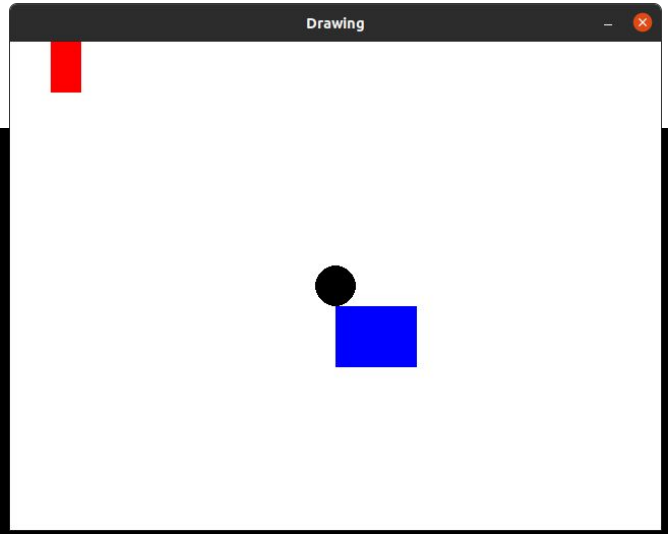


Drawing Primitives: More Shapes

```
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

GAME_SCREEN.fill(WHITE)

pygame.draw.circle(GAME_SCREEN, BLACK, screen_rect.center, 20)
pygame.draw.rect(GAME_SCREEN, RED, (screen_rect.left + 40, screen_rect.top, 30, 50))
my_rect = pygame.Rect(screen_rect.centerx, # top left x
                      screen_rect.centery + 20, # top left y
                      screen_rect.width / 8, # width of my_rect
                      screen_rect.height / 8) # height of my_rect
pygame.draw.rect(GAME_SCREEN, BLUE, my_rect)
```

Many more shapes can be drawn with pygame.draw such as rect (rectangle), ellipse, arc, polygon.

Drawing Primitives: The Result

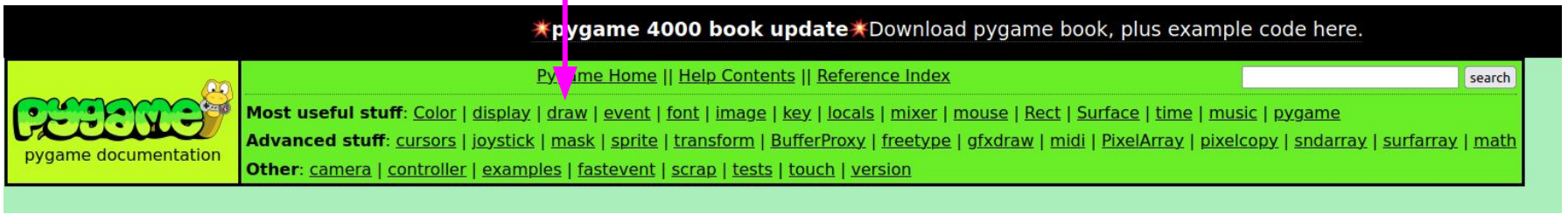



```
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)

GAME_SCREEN.fill(WHITE)

pygame.draw.circle(GAME_SCREEN, BLACK, screen_rect.center, 20)
pygame.draw.rect(GAME_SCREEN, RED, (screen_rect.left + 40, screen_rect.top, 30, 50))
my_rect = pygame.Rect(screen_rect.centerx, # top left x
                      screen_rect.centery + 20, # top left y
                      screen_rect.width / 8, # width of my_rect
                      screen_rect.height / 8) # height of my_rect
pygame.draw.rect(GAME_SCREEN, BLUE, my_rect)
```

Pygame Documentation

If you want to know the syntax for drawing other shapes, you can check the Pygame Documentation.

https://www.pygame.org/docs/



It is a very helpful reference for other things you need to know about Pygame as well ;)

Back to the game loop…

Now that you know a little bit about drawing on the screen, let's get back to our game loop.

Before we get started, we added a **clock** to help us control how fast the game loop iterates.

One iteration is one frame. So when we call clock.tick(1), we say we want the speed to be ***one frame per second*** (FPS)

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

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

GAME_SCREEN = pygame.display.set_mode((640, 480))
pygame.display.set_caption('Drawing in the loop')

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    pygame.display.update()
    clock.tick(1)
```

Update Game State & Draw Screen

We want to make a red rectangle fall from the top of the screen.

What are the parts of our game loop?

Handle events: Consider the passage of time an event
Update game state: The y-position of the rectangle should increase
Draw: Draw the rectangle in its proper position

Update Game State & Draw Screen

Handle events: Consider the passage of time an event
Update game state: The y-position of the rectangle should increase
Draw: Draw the rectangle in its proper position

Let's think of this rectangular object as a pepper :)
Before the game loop, let's define some properties of the pepper.

Understanding check! Where is the pepper's position after this code?

```
pepper_width = 80
pepper_height = 100
pepper_rect = pygame.Rect(0, 0,
pepper_width, pepper_height)
pepper_rect.centerx = screen_rect.centerx
pepper_rect.top = screen_rect.top
pepper_color = RED
pepper_speed = 20
```

Update Game State & Draw Screen

Handle events: Consider the passage of time an event
Update game state: The y-position of the rectangle should increase
Draw: Draw the rectangle in its proper position


```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    """ Draw pepper """
    pygame.draw.rect(GAME_SCREEN, pepper_color, pepper_rect)
    pygame.display.update()
    clock.tick(1)
```

Let's add the drawing part to the game loop right before pygame.display.update()

Update Game State & Draw Screen

Handle events: Consider the passage of time an event

Update game state: The y-position of the rectangle should increase

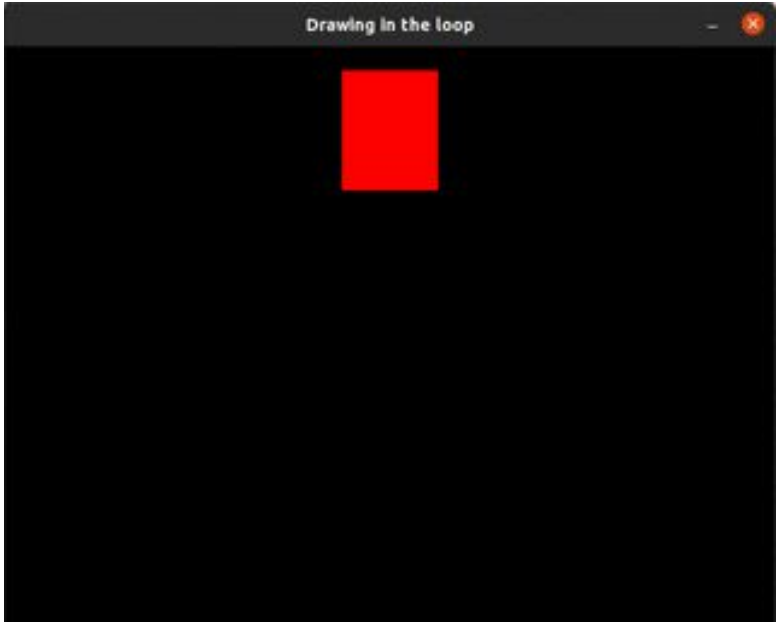
Draw: Draw the rectangle in its proper position


```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    """ Update pepper to make it fall. """
    pepper_rect.top = pepper_rect.top + pepper_speed
    """ Draw pepper """
    pygame.draw.rect(GAME_SCREEN, pepper_color, pepper_rect)
    pygame.display.update()
    clock.tick(1)
```

For the pepper to fall, we need to *update* its position. Let's add that update code.

What’s the pepper doing?

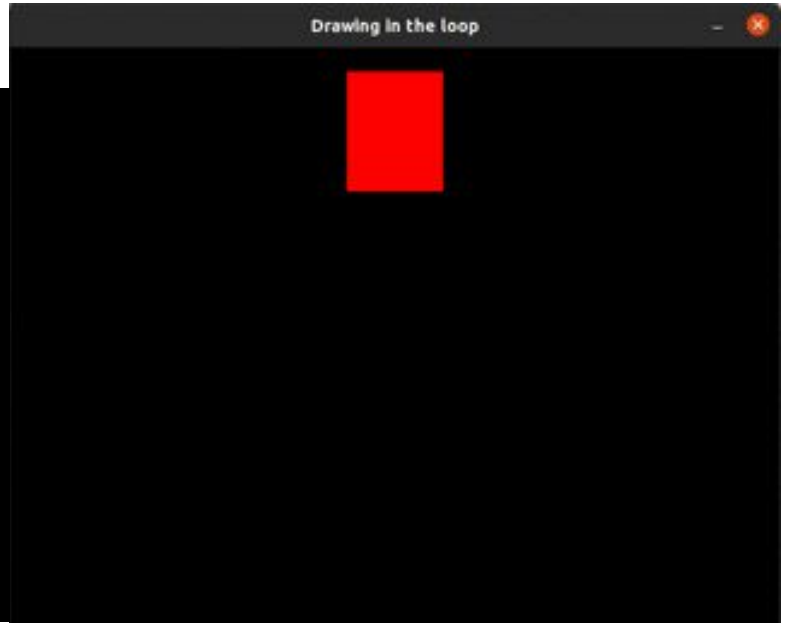
That doesn’t seem right…



What's the pepper doing?

Our code draws rectangles, but the previous drawing is left on the screen!

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    """ Update pepper to make it fall. """
    pepper_rect.top = pepper_rect.top + pepper_speed
    """ Draw pepper """
    pygame.draw.rect(GAME_SCREEN, pepper_color, pepper_rect)
    pygame.display.update()
    clock.tick(1)
```



What can we do…

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    """ fill screen white """
    GAME_SCREEN.fill(WHITE)
    """ Update pepper to make it fall. """
    pepper_rect.top = pepper_rect.top + pepper_speed
    """ Draw pepper """
    pygame.draw.rect(GAME_SCREEN, pepper_color, pepper_rect)
    pygame.display.update()
    clock.tick(1)
```

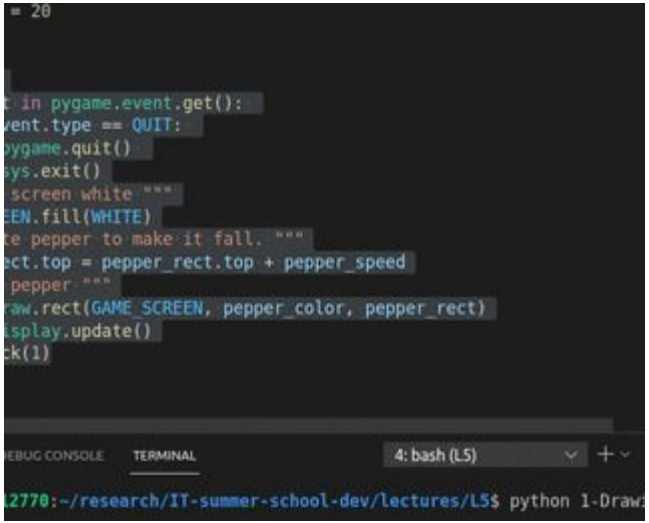
In general, how this is handled varies based on what other code we have and how we implemented it

Can be said for all of coding, not just game programming ;)

Here, in each iteration, we will fill the game screen with white, covering up anything that was previously there. And now we have a white background!

What’s the pepper doing?

That’s much better!



Let’s improve our code with Classes

🥴

```
pepper_width = 80
pepper_height = 100
pepper_rect = pygame.Rect(0, 0, pepper_width,
pepper_height)
pepper_rect.centerx = screen_rect.centerx
pepper_rect.top = screen_rect.top
pepper_color = RED
pepper_speed = 20
```

🤩

```
class Pepper:
    speed = 20
    color = RED

    def __init__(self, rect, midtop):
        self.rect = rect
        self.rect.midtop = midtop
```

```
pepper = Pepper(pygame.Rect(0, 0, 80, 100), screen_rect.midtop)
```

Let's improve our code with Classes

```
""" Update pepper to make it fall. """
pepper_rect.top = pepper_rect.top + pepper_speed
""" Draw pepper """
pygame.draw.rect(GAME_SCREEN, pepper_color, pepper_rect)
```

```
""" Update pepper to make it fall. """
pepper.rect.top = pepper.rect.top + pepper.speed
""" Draw pepper """
pygame.draw.rect(GAME_SCREEN, pepper.color, pepper.rect)
```

Now, we access the pepper object's attributes rather than using separate variables.

We can access pepper's rect, speed and color with a dot:

- pepper.rect
- pepper.speed
- pepper.color

Add an *update* method

```
class Pepper:
    speed = 20
    color = RED

    def __init__(self, rect, midtop):
        self.rect = rect
        self.rect.midtop = midtop

    def update(self):
        self.rect.top = self.rect.top +
self.speed
```

```
""" Update pepper to make it fall. """
pepper.rect.top = pepper.rect.top + pepper.speed
```

```
""" Update pepper to make it fall. """
pepper.update()
```

Add a *draw* method

```
class Pepper:
    speed = 20
    color = RED

    def __init__(self, rect, midtop):
        self.rect = rect
        self.rect.midtop = midtop

    def update(self):
        self.rect.top = self.rect.top + self.speed

    def draw(self, surface):
        pygame.draw.rect(surface, self.color, self.rect)
```

```
""" Update pepper """
pepper.update()
""" Draw pepper """
pepper.draw(GAME_SCREEN)
```

Now we can make more peppers easily :)

```
pepper = Pepper(pygame.Rect(0, 0, 80, 100), screen_rect.midtop)
pepper2 = Pepper(pygame.Rect(0, 0, 40, 50),
                 (pepper.rect.left - 100, pepper.rect.top))
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    GAME_SCREEN.fill(WHITE)
    pepper.update()
    pepper2.update()
    pepper.draw(GAME_SCREEN)
    pepper2.draw(GAME_SCREEN)
    pygame.display.update()
    clock.tick(1)
```

Break

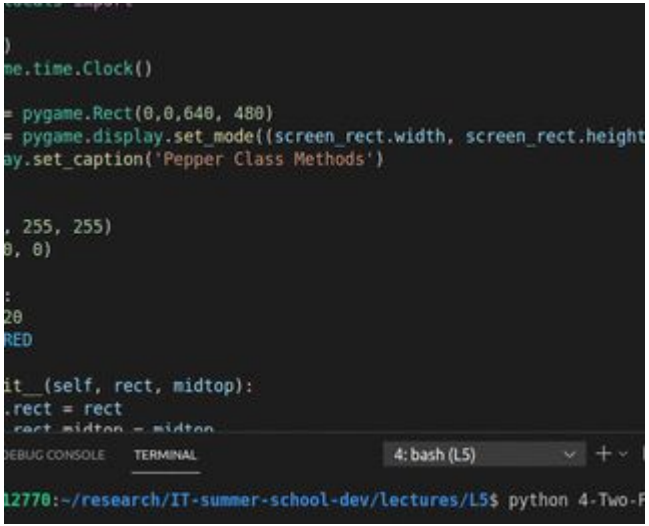
Previously…

We made a full game loop to show peppers falling from the top of the screen.

Handle events: The passage of time an event (and QUIT)

Update game state: The y-position of the rectangle should increase

Draw: Draw the rectangle in its proper position



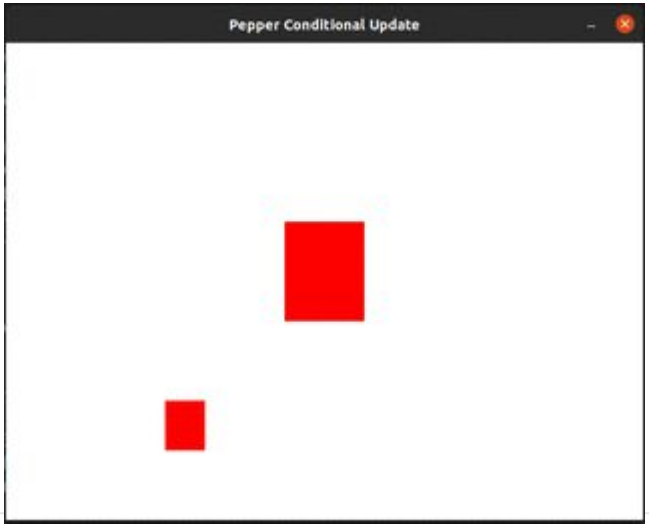
Update game state: conditional

We want the peppers to return to the top of the screen when they touch the ground.

Handle events: Pepper touches ground

Update game state: The y-position of the rectangle should be reset

Draw: Draw the rectangle in its proper position



How would you code this?

```
    pepper.update()
    pepper2.update()

    """ Code for conditional update """



    pepper.draw(GAME_SCREEN)
    pepper2.draw(GAME_SCREEN)
```

Increase the FPS to 10 to observe this at a faster rate

How would you code this? Solution

```
    pepper.update()
    pepper2.update()

    if pepper.rect.bottom >= screen_rect.bottom:
        pepper.rect.top = screen_rect.top
    if pepper2.rect.bottom >= screen_rect.bottom:
        pepper2.rect.top = screen_rect.top

    pepper.draw(GAME_SCREEN)
    pepper2.draw(GAME_SCREEN)
```

Increase the FPS to 10 to observe this at a faster rate

How would you code this? Solution

```
    pepper.update()
    pepper2.update()

    if pepper.rect.bottom >= screen_rect.bottom:
        pepper.rect.top = screen_rect.top
    if pepper2.rect.bottom >= screen_rect.bottom:
        pepper2.rect.top = screen_rect.top

    pepper.draw(GAME_SCREEN)
    pepper2.draw(GAME_SCREEN)
```

Repeat code!
What can we do?

Repeat code! What can we do? Use lists!

```
pepper = Pepper(pygame.Rect(0, 0, 80, 100), screen_rect.midtop)
pepper2 = Pepper(pygame.Rect(0, 0, 40, 50),
                 (pepper.rect.left - 100, pepper.rect.top))
pepper_list = [pepper, pepper2]
```

```
    for item in pepper_list:
        item.update()

        if item.rect.bottom >= screen_rect.bottom:
            item.rect.top = screen_rect.top

        item.draw(GAME_SCREEN)
```

```
class PepperGroup:
    def __init__(self):
        self.items = []

    def add(self, item):
        self.items.append(item)

    def update(self):
        for item in self.items:
            item.update()

    def draw(self, surface):
        for item in self.items:
            item.draw(surface)

    def touch_ground_update(self, ground_rect):
        for item in self.items:
            if item.rect.bottom >= ground_rect.bottom:
                item.rect.top = ground_rect.top
```

Even better…

Repeat code! What can we do?

```
pepper_list = PepperGroup()

pepper = Pepper(pygame.Rect(0, 0, 80, 100), screen_rect.midtop)
pepper2 = Pepper(pygame.Rect(0, 0, 40, 50),
                 (pepper.rect.left - 100, pepper.rect.top))

pepper_list.add(pepper)
pepper_list.add(pepper2)
```

Now we have a collection of peppers!

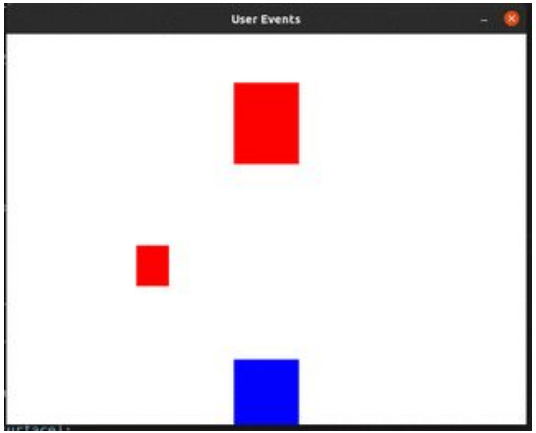
Our game loop 🥰

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    GAME_SCREEN.fill(WHITE)

    pepper_list.update()
    pepper_list.touch_ground_update(screen_rect)
    pepper_list.draw(GAME_SCREEN)

    pygame.display.update()
    clock.tick(10)
```

Now our game loop is so simple and beautiful 🥰

Update game state: Player events

We want to add a player that we can control with the keyboard. When we press the left and right arrow keys (or the a and d keys), the player moves left and right.



Handle events: Player presses left, right, ‘a’, or ‘d’ keys

Update game state: Player’s x-position is updated to move it to the desired direction

Draw: Draw the Player in its proper position

Update game state: Player events

Handle events: Player presses left, right, ‘a’, or ‘d’ keys
Update game state: Player’s x-position is updated to move it to the desired direction
Draw: Draw the Player in its proper position

Getting started, we need to add a player to the code!

Let’s make a player class :)

It’s very similar to the Pepper class, so let’s start there and make some modifications!

Player Class

```
class Player:
    speed = 10
    color = BLUE

    def __init__(self, rect, midbottom):
        self.rect = rect
        self.rect.midbottom = midbottom

    def update(self):
        return

    def draw(self, surface):
        pygame.draw.rect(surface, self.color, self.rect)
```

Let's make the player's speed 10, and the color blue.

We want the player on the bottom, so let's set the position based on midbottom.

We don't need the touch_ground_update method from the Pepper class.

The Player doesn't fall, so let's just make the update method empty for now.

Add a player to the loop

```
player = Player(pygame.Rect(0, 0, 80, 80), screen_rect.midbottom)
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    GAME_SCREEN.fill(WHITE)

    pepper_list.update()
    pepper_list.touch_ground_update(screen_rect)
    pepper_list.draw(GAME_SCREEN)
    player.draw(GAME_SCREEN)
    pygame.display.update()
    clock.tick(10)
```

First we need to make an object of the Player class, who we call “player”

Then, we can call the Player.draw method in the game loop so she shows up on the screen.

Player Class

```
class Player:
    """ Player class continued """

    def draw(self, surface):
        pygame.draw.rect(surface, self.color, self.rect)

    def move(self):
        return
```

The Player is going to need a **move** method.

Player Class

```
class Player:
    """ Player class continued """

    def draw(self, surface):
        pygame.draw.rect(surface, self.color, self.rect)

    def move(self, keystates):
        return
```

The Player is going to need a **move** method.

In order to know how and which direction to move, we need to know what keys are being pressed.

Let's add a parameter called **keystates,** to pass information about the state of the keys. The key states are either "pressed" or "not pressed." That means our information is in the form of Booleans!

Add a player to the loop

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    GAME_SCREEN.fill(WHITE)
    pepper_list.update()

    keystates = pygame.key.get_pressed()
    player.move(keystates)

    pepper_list.touch_ground_update(screen_rect)
    pepper_list.draw(GAME_SCREEN)
    player.draw(GAME_SCREEN)

    pygame.display.update()
    clock.tick(10)
```

In the game loop, we use a helpful function from the Pygame library that will tell us whether the keys are pressed or not! It returns a list of [False, True, False, …, False], where each index corresponds to a particular key.

Then, we can call the Player.move() method in the game loop, passing it the the keystates returned from pygame.key.get_pressed()

Add a player to the loop

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    GAME_SCREEN.fill(WHITE)
    pepper_list.update()

    keystates = pygame.key.get_pressed()
    player.move(keystates)

    pepper_list.touch_ground_update(screen_rect)
    pepper_list.draw(GAME_SCREEN)
    player.draw(GAME_SCREEN)

    pygame.display.update()
    clock.tick(10)
```

We won’t observe anything happen after we add these two lines because we still need to implement the Player.move method :)

Player Class

```
class Player:
    """ Player class continued """

    def draw(self, surface):
        pygame.draw.rect(surface, self.color, self.rect)

    def move(self, keystates):
        left_arrow_pressed = keystates[K_LEFT]
        right_arrow_pressed = keystates[K_RIGHT]
```

Luckily, we do not need to guess the indices of the keyboard keys in the **keystates** list.

Player Class

```
# this imports all pygame.locals
from pygame.locals import *
# Alternatively, we can specify which ones we want
from pygame.locals import K_LEFT, K_RIGHT, QUIT

class Player:
    """ Player class continued """

    def draw(self, surface):
        pygame.draw.rect(surface, self.color, self.rect)

    def move(self, keystates):
        left_arrow_pressed = keystates[K_LEFT]
        right_arrow_pressed = keystates[K_RIGHT]
```

Luckily, we do not need to guess the indices of the keyboard keys in the **keystates** list…

because these are saved in constant variables that we import from **pygame.locals** (in the top of the file)

Pygame documentation

You can find the variable names that store the indices of all the keys in the pygame documentation: https://www.pygame.org/docs/ref/key.html

It looks like that ->

```
pygame
Constant      ASCII   Description
--------------------------------
K_BACKSPACE   \b      backspace
K_TAB         \t      tab
K_CLEAR               clear
K_RETURN      \r      return
K_PAUSE               pause
K_ESCAPE      ^[      escape
K_SPACE               space
K_EXCLAIM     !       exclaim
K_QUOTEDBL    "       quotedbl
K_HASH        #       hash
K_DOLLAR      $       dollar
K_AMPERSAND   &       ampersand
K_QUOTE               quote
K_LEFTPAREN   (       left parenthesis
K_RIGHTPAREN  )       right parenthesis
K_ASTERISK    *       asterisk
K_PLUS        +       plus sign
K_COMMA       ,       comma
K_MINUS       -       minus sign
K_PERIOD      .       period
K_SLASH       /       forward slash
K_0           0       0
K_1           1       1
K_2           2       2
K_3           3       3
K_4           4       4
K_5           5       5
K_6           6       6
K_7           7       7
K_8           8       8
K_9           9       9
K_COLON       :       colon
K_SEMICOLON   ;       semicolon
K_LESS        <       less-than sign
K_EQUALS      =       equals sign
K_GREATER     >       greater-than sign
K_QUESTION    ?       question mark
K_AT          @       at
K_LEFTBRACKET [       left bracket
K_BACKSLASH   \       backslash
K_RIGHTBRACKET ]      right bracket
K_CARET       ^       caret
K_UNDERSCORE  _       underscore
K_BACKQUOTE   `       grave
K_a           a       a
K_b           b       b
K_c           c       c
K_d           d       d
K_e           e       e
K_f           f       f
K_g           g       g
K_h           h       h
K_i           i       i
```

Player Class

```
class Player:
    """ Player class continued """
    def move(self, keystates):
        if keystates[K_LEFT] or keystates[K_a]:
            print("move left")
        if keystates[K_RIGHT] or keystates[K_d]:
            print("move right")
```

Let's make a template of the conditional statements we need.

Left: We want to be able to move left with either the left arrow (right-hander friendly) or the 'a' key (left-hander friendly).

Right: We want to be able to move right with either the right arrow or the 'd' key.

Try running the code now and make sure "move left" prints in the terminal when you press the left arrow or 'a' key, and "move right" prints when you press the right arrow or 'd' key

Tip

Turn the Pepper color white so that you can focus on the Player movement without distraction!

```
class Pepper:
    speed = 20
    # color = RED
    color = WHITE
```

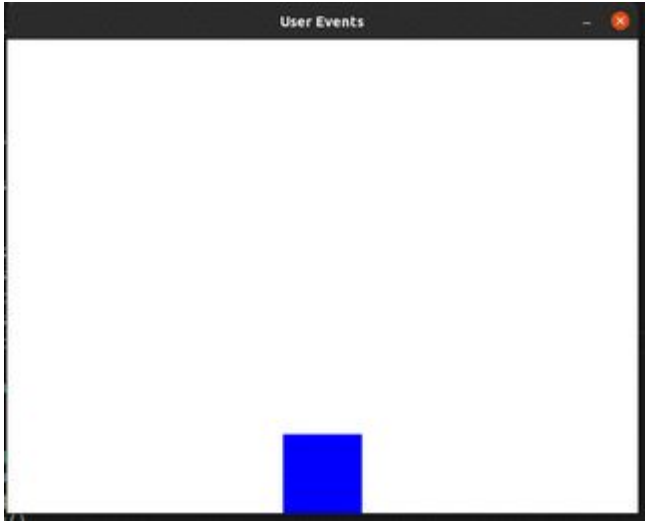
Player Class: Finish the move method

Now, implement the change of position. You definitely know how to do this! Look at how we change the position of the Pepper and think about what needs to be different for the Player.

```
class Player:
    """ Player class continued """
    def move(self, keystates):
        if keystates[K_LEFT] or keystates[K_a]:
            """ Fill in code to move left """
        if keystates[K_RIGHT] or keystates[K_d]:
            """ Fill in code to move right  """
```

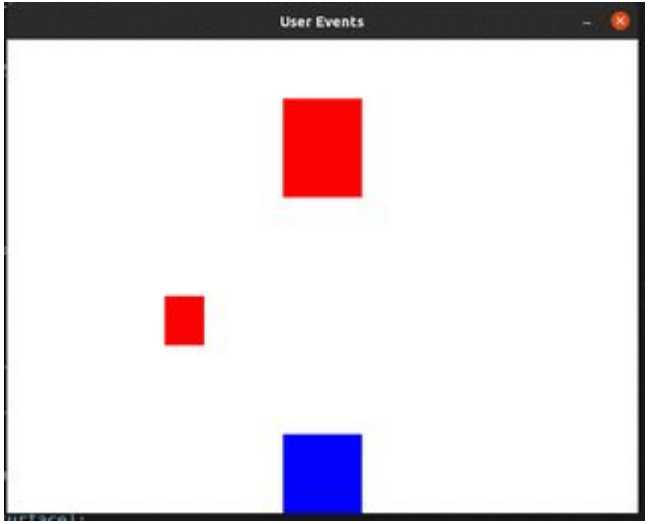
Player Class: Solution

```
class Player:
    """ Player class continued """
    def move(self, keystates):
        if keystates[K_LEFT] or keystates[K_a]:
            self.rect.x = self.rect.x - self.speed
        if keystates[K_RIGHT] or keystates[K_d]:
            self.rect.x = self.rect.x + self.speed
```



Player Class: Put our peppers back in!

```
class Pepper:
    speed = 20
    color = RED
    # color = WHITE
```

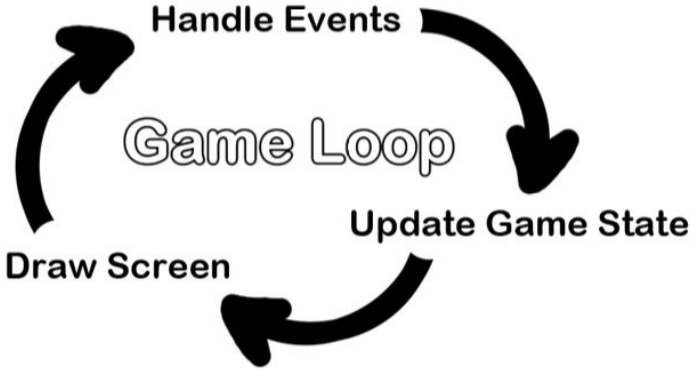


Let’s review what you’ve learned about the main game loop.

The “Game Loop” is the heart of the video game.

The Game Loop does 3 things:

1. **Handles events**
2. **Updates the state of the game**
3. **Draws the Screen**



Review: Update the game state

Event type: Time passes

We control the amount of time that passes with clock.tick(), so time passes when the Game Loop finishes an iteration.

Update: ?

What do we update and how?

Review: Update the game state

Event type: Time passes

We control the amount of time that passes with clock.tick(), so time passes when the Game Loop finishes an iteration.

Update: Pepper state

To simulate the peppers falling, we make an *update* method for the pepper class which increases the y-coordinate position of the pepper based on the pepper's speed (in pixels per frame).

Review: Update the game state

Event type: Game condition met **Update: ?**

The Pepper touches the ground. We check this with a conditional statement to find out whether the y-coordinate position of the bottom of the Pepper is greater than or equal to the bottom of the screen.

Review: Update the game state

Event type: Game condition met

The Pepper touches the ground. We check this with a conditional statement to find out whether the y-coordinate position of the bottom of the Pepper is greater than or equal to the bottom of the screen.

Update: Pepper state

The Pepper respawns at the top of the screen. To implement this, we set Pepper.rect.midtop (x,y)-coordinates to be equal to the screen's midtop (x,y)-coordinates.

Review: Update the game state

Event type: User click

The user clicks the **QUIT** X button on the game window.

Update:

Review: Update the game state

Event type: User click

The user clicks the **QUIT** button on the game window.

Update: Quit game

We disable Pygame with pygame.quit() and then we end the python program with sys.exit().

Review: Update the game state

Event type: User presses keys

Update:

The user is pressing the left or right arrows, or ‘a’ or ‘d’ keys.

Review: Update the game state

Event type: User presses keys

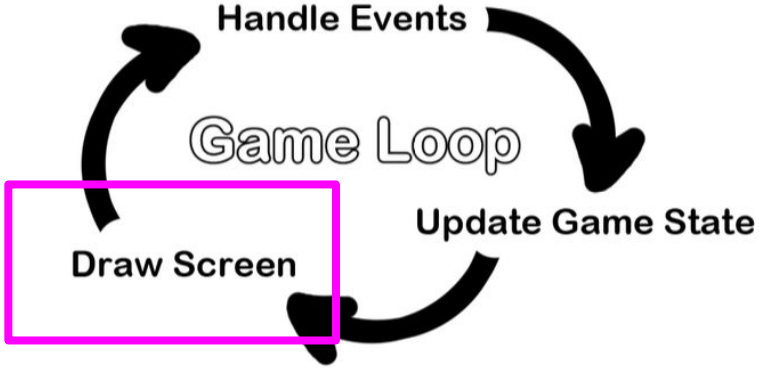
The user is pressing the left or right arrows, or ‘a’ or ‘d’ keys.

Update: Player state

The player’s position is changed, moving it either left or right. We do this by decreasing or increasing the Player.rect.x position by Player.speed (pixels per frame).

Review: Last but not least? Draw Screen!

1. The passage of time
   a. The peppers' positions always update as time passes
2. Conditions of the game state
   a. When the peppers touch the ground
3. User events
   a. Clicking the QUIT button
   b. Pressing the keys



Closing thoughts…

The logic of game programming

The key to **understanding the logic** of game programming is to ***understand the game loop***.

When you want to make your own game, you can plan the logic of your programming thinking about the 3 parts of Game Loop.

Ask yourself: What do I want to happen in the game?

Then think:

- What **event** should cause it?
- What variables and game objects need to be **updated** and how?
- What should you expect to see **drawn on the screen** based on these updates?

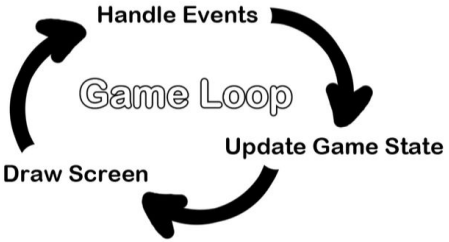
Code can look different!

```
class Player(pygame.sprite.Sprite):
    def __init__(self, image, midbottom):
        super().__init__()

        self.image = image
        self.rect =
self.image.get_rect(midbottom=midbottom)
        self.speed = 10

    def move(self, keystate):
        left_arrow_pressed = keystate[K_LEFT]
        right_arrow_pressed = keystate[K_RIGHT]

        if left_arrow_pressed:
            self.rect.x = self.rect.x - self.speed

        if right_arrow_pressed:
            self.rect.x += self.speed
```

```
class Player:
    speed = 10
    color = BLUE

    def __init__(self, rect, midbottom):
        self.rect = rect
        self.rect.midbottom = midbottom

    def draw(self, surface):
        pygame.draw.rect(surface, self.color,
self.rect)

    def move(self, keystates):
        if keystates[K_LEFT] or keystates[K_a]:
            self.rect.x = self.rect.x - self.speed
        if keystates[K_RIGHT] or keystates[K_d]:
            self.rect.x = self.rect.x + self.speed
```

Extra time?

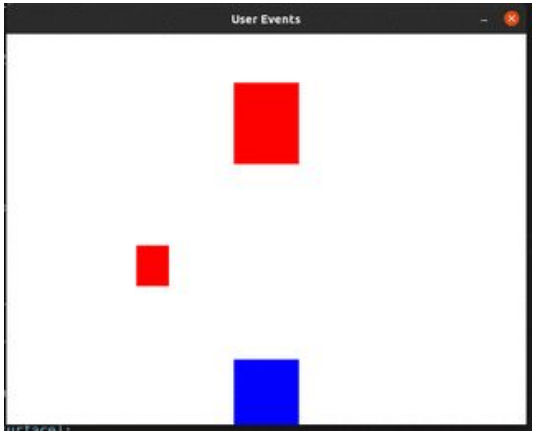
We are missing a condition! The peppers also need to respawn at the top when they touch the player. Can you solve this?



Handle events:

Update game state:

Draw:



Class inheritance

Making a Sprite base class

- What do the Player and Pepper class have in common?
  - Rect
  - Color
  - Update
  - Draw
- We can simplify the code even more by creating a parent class that Player and Pepper inherit from.

Step 1: Move commonalities to Sprite class

```
class Sprite:
    def __init__(self, rect, color):
        self.rect = rect
        self.color = color

    def update(self):
        return

    def draw(self, surface):
        pygame.draw.rect(surface, self.color, self.rect)
```

Constructor: Both Pepper and Player have a rect and a color. The rects and colors can be different, so we make them parameters that we can decide on when we create our objects.

Step 1: Move commonalities to Sprite class

```
class Sprite:
    def __init__(self, rect, color):
        self.rect = rect
        self.color = color

    def update(self):
        return

    def draw(self, surface):
        pygame.draw.rect(surface, self.color, self.rect)
```

Constructor: Pass in the desired rect and color when we create a Pepper or Player object.

Update method: Our sprites are updated in different ways, so we can leave this unimplemented and *override* it in the Pepper and Player classes

Step 1: Move commonalities to Sprite class

```
class Sprite:
    def __init__(self, rect, color):
        self.rect = rect
        self.color = color

    def update(self):
        return

    def draw(self, surface):
        pygame.draw.rect(surface, self.color, self.rect)
```

Constructor: Pass in the desired rect and color when we create a Pepper or Player object.

Update method: We will implement this individually for the child classes.

Draw method: In our game, the sprites are drawn the same way. So we can put the implementation in the base class and remove it from Pepper and Player.

Step 2: Fix child classes

```
class Pepper(Sprite):
    def __init__(self, rect, color=RED, speed=10):
        super().__init__(rect, color)
        self.speed = speed

    def update(self):
        self.rect.top = self.rect.top + self.speed

    def touch_ground_update(self, ground_rect):
        if self.rect.bottom >= ground_rect.bottom:
            self.rect.top = ground_rect.top
```

Inherit Parent: We define the Pepper class like so. Then the Pepper inherits all the attributes and methods from the Sprite class.

Step 2: Fix child classes

```
class Pepper(Sprite):
    def __init__(self, rect, color=RED, speed=10):
        super().__init__(rect, color)
        self.speed = speed

    def update(self):
        self.rect.top = self.rect.top + self.speed

    def touch_ground_update(self, ground_rect):
        if self.rect.bottom >= ground_rect.bottom:
            self.rect.top = ground_rect.top
```

Inherit Parent: Pepper inherits from Sprite class.

Constructor: We can call super().__init__() which will use the constructor from the parent Sprite class. We pass in our desired values for rect and color. We also added a speed parameter. We make default values for the color and speed, so we can change them if we want to. But in general our peppers will be red and fall at the same speed.

Step 2: Fix child classes

```
class Pepper(Sprite):
    def __init__(self, rect, color=RED, speed=10):
        super().__init__(rect, color)
        self.speed = speed

    def update(self):
        self.rect.top = self.rect.top + self.speed

    def touch_ground_update(self, ground_rect):
        if self.rect.bottom >= ground_rect.bottom:
            self.rect.top = ground_rect.top
```

Inherit Parent: Pepper inherits from Sprite class.

Constructor: Call super().__init__() to use the parent constructor. Add parameters if the child class has additional attributes to set.

Update method: Stays the same as before.

Step 2: Fix child classes

```
class Pepper(Sprite):
    def __init__(self, rect, color=RED, speed=10):
        super().__init__(rect, color)
        self.speed = speed

    def update(self):
        self.rect.top = self.rect.top + self.speed

    def touch_ground_update(self, ground_rect):
        if self.rect.bottom >= ground_rect.bottom:
            self.rect.top = ground_rect.top
```

Inherit Parent: Pepper inherits from Sprite class.

Constructor: Call super().__init__() to use the parent constructor. Add parameters if the child class has additional attributes to set.

Update method: Stays the same as before.

Additional methods: The falling behavior is unique to the Pepper class, so we leave it in the class.

Step 2: Fix child classes

```
class Player(Sprite):
    def __init__(self, rect, color=BLUE, speed=10):
        super().__init__(rect, color)
        self.speed = speed

    def move(self, keystates):
        if keystates[K_LEFT] or keystates[K_a]:
            self.rect.x = self.rect.x - self.speed
        if keystates[K_RIGHT] or keystates[K_d]:
            self.rect.x = self.rect.x + self.speed
```

Do the same with the Player class…

Inherit Parent: Player inherits from Sprite class.

Constructor: Call super().__init__() to use the parent constructor. Add parameters if the child class has additional attributes to set.

Update method: We don't need a special update method for Player.

Additional methods: The player moving when keys are pressed is unique to the Player.

Observe

```
class Sprite:
    def __init__(self, rect, color):
        print("Sprite.__init__")
        self.rect = rect
        self.color = color

    def update(self):
        print("Sprite.update")
        return

    def draw(self, surface):
        print("Sprite.draw")
        pygame.draw.rect(surface, self.color, self.rect)
```

We are going to put print statements at the top of each method, and observe what prints at the command line.

Sprite.__init__
Sprite.update
Sprite.draw

Observe

```
class Pepper(Sprite):
    def __init__(self, rect, color=RED, speed=10):
        super().__init__(rect, color)
        print("Pepper.__init__")
        self.speed = speed

    def update(self):
        print("Pepper.update")
        self.rect.top = self.rect.top + self.speed

    def touch_ground_update(self, ground_rect):
        print("Pepper.touch_ground_update")
        if self.rect.bottom >= ground_rect.bottom:
            self.rect.top = ground_rect.top
```

We are going to put print statements at the top of each method, and observe what prints at the command line.

Pepper.__init__ (after super())
Pepper.update
Pepper.touch_ground_update

Observe

```
class Player(Sprite):
    def __init__(self, rect, color=BLUE, speed=10):
        super().__init__(rect, color)
        print("Player.__init__")
        self.speed = speed

    def move(self, keystates):
        print("Player.move")
        if keystates[K_LEFT] or keystates[K_a]:
            self.rect.x = self.rect.x - self.speed
        if keystates[K_RIGHT] or keystates[K_d]:
            self.rect.x = self.rect.x + self.speed
```

We are going to put print statements at the top of each method, and observe what prints at the command line.

Player.__init__ (after super())
Player.update

Observe

```
pepper = Pepper(pygame.Rect(0, 0, 80, 100))
pepper.rect.midtop = screen_rect.midtop

player = Player(pygame.Rect(0, 0, 80, 80))
player.rect.midbottom = screen_rect.midbottom
quit()
```

Before our game loop, let's create a pepper object and a player object and then tell the program to quit.

This is what prints:

Sprite.__init__
Pepper.__init__
Sprite.__init__
Player.__init__

Observe

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    GAME_SCREEN.fill(WHITE)
    pepper.update()
    keystates = pygame.key.get_pressed()
    player.move(keystates)
    pepper.touch_ground_update(screen_rect)
    pepper.draw(GAME_SCREEN)
    player.draw(GAME_SCREEN)
    pygame.display.update()
    quit()
```

Now let's remove the quit() and run our game loop. Only using one pepper right now, no pepper_list. Quit after one iteration.

Observe

```
while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    GAME_SCREEN.fill(WHITE)
    pepper.update()
    keystates = pygame.key.get_pressed()
    player.move(keystates)
    pepper.touch_ground_update(screen_rect)
    pepper.draw(GAME_SCREEN)
    player.draw(GAME_SCREEN)
    pygame.display.update()
    quit()
```

This is what prints:

Sprite.__init__
Pepper.__init__
Sprite.__init__
Player.__init__
Pepper.update
Player.move
Pepper.touch_ground_update
Sprite.draw
Sprite.draw