Oide
Tacú leis an bhFoghlaim Ghairmiúil i measc Ceannairí Scoile agus Múinteoirí
Supporting the Professional Learning of School Leaders and Teachers

A simulation of the Kessler effect using Python

Oide
Tacú leis an bhFoghlaim Ghairmiúil i measc Ceannairí Scoile agus Múinteoirí
Supporting the Professional Learning of School Leaders and Teachers

Contents

Overview

In this step-by-step tutorial you will develop a Python program that can be used to visualise a phenomena that occurs in space known as the Kessler Effect.

The Kessler Effect describes a scenario in which collisions between objects in low Earth Orbit (LEO) could cause a cascading effect, generating even more debris and potentially making space activities more challenging.

Space debris (often referred to as 'space junk') is represented on by rectangles created using the `pygame` library. The tutorial starts off with a simple program that displays a single rectangle and progressively build up a program that simulates the Kessler effect. In the process you will learn how to control the movement of objects on the screen as well as simulate objects bouncing off the screen edges and collisions with one another. By the end, you will have a hands-on understanding of how space debris can accumulate over time, leading to an increased risk of collisions in space.

This tutorial has been developed as part of a series of teaching and learning resources developed by Oide to promote EIRSAT-1, Ireland's first ever satellite.[1] Enjoy!

Conventions

To help with navigation through this tutorial, the following conventions are used:
Italics are used to highlight important new words and phrases and `Courier New` font is used to denote Python code such as keywords, commands and variable names

```
print("Hello World")
```

The icons illustrated below are used to highlight different types of information throughout this tutorial.

Space for you to answer questions using pen and paper.

Experiment. An opportunity to 'play with the code' and see what happens.

Programming exercises. An opportunity for you to practice your Python programming skills either by yourself or with your friends.

Reflection log. A space for you to reflect on your own learning and record your thoughts.

The octocat (shown here to the left) is the GitHub integration symbol. GitHub is a source code repository. Throughout this tutorial you will notice this symbol appears along with code used. When you click on the octocat you will be directed to the source code on GitHub in your web browser.

[1] EIRSAT-1 stands for Educational Irish Research Satellite. For more information see https://www.eirsat1.ie/.

Introduction

Watch first minute of this video and answer the questions that follow,

https://www.youtube.com/watch?v=cX89BpzrAVY

What is space debris?

How big is space debris?

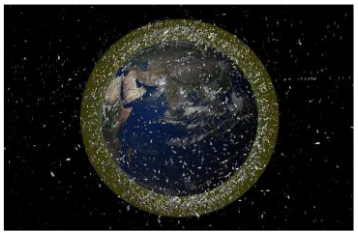
How is space debris created?

Why is there so much junk in space?

What is Low Earth Orbit?

How fast does space debris move?

What is the Kessler syndrome?



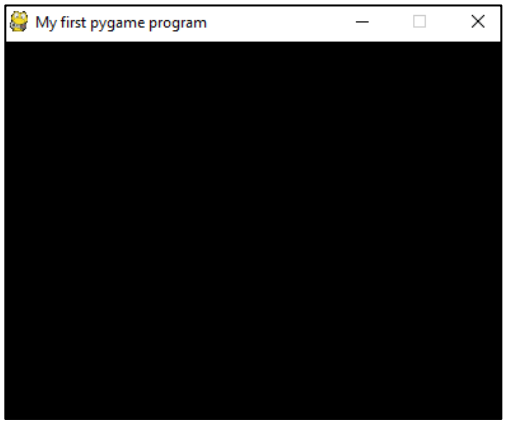
Program 1: Getting started with `pygame`

Key in the following code and save it on your computer.

```
1.  # My first pygame program
2.  import pygame, sys, random, time
3.  from pygame.locals import *
4.
5.  # start the pygame engine
6.  pygame.init()
7.
8.  # create the display window
9.  display_surface = pygame.display.set_mode((400, 300))
10.
11. # Set the title of the display window
12. pygame.display.set_caption('My first pygame program')
13.
14. # run the game loop
15. while True:
16.
17.     for event in pygame.event.get():
18.         if event.type == QUIT:
19.             pygame.quit()
20.             sys.exit()
```

Program listing 1

You will notice when you run the program it does very little - except display the window shown here to the right. Click the X on the top right corner to close.



The code is explained using comments which are displayed in *red*.

Lines 1 and 2 import the libraries necessary to run this tutorial.
Line 6 starts the `pygame` engine
Line 9 creates the `pygame` window.
Line 12 sets the title text for the `pygame` window.
Lines 15-20 make up what is called the game loop.
We will take a closer look at the game loop later.

Experiment!
Try the following and see if you can explain what is going on.

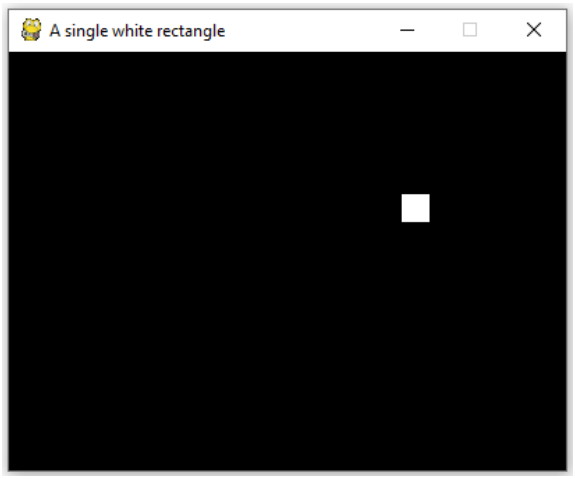
1. Change the title of the `pygame` window.
2. Change the pair of values (400, 300) used on line 9. Do this several times until you figure our what each value represents. Can you make the window appear as a square?

What does 400 represent?

What does 300 represent?

Program 2: Display a single square

In this program you will learn how to display a single white rectangle in the display area as shown here.



The code is shown below. Notice that there are 29 lines in this program whereas there were just 20 lines in the previous program.

```
1.  # Program 2. Display a single white rectangle
2.  import pygame, sys, random, time
3.  from pygame.locals import *
4.
5.  # start the pygame engine
6.  pygame.init()
7.
8.  # create the display window
9.  display_surface = pygame.display.set_mode((400, 300))
10.
11. # Set the title of the display window
12. pygame.display.set_caption('A single white rectangle')
13.
14. # create a rectangle to display
15. x = random.randint(0, 400)
16. y = random.randint(0, 300)
17. rectangle = pygame.Rect(x, y, 20, 20)
18. pygame.draw.rect(display_surface, (255,255,255), rectangle)
19.
20. # run the game loop
21. while True:
22.
23.     for event in pygame.event.get():
24.         if event.type == QUIT:
25.             pygame.quit()
26.             sys.exit()
27.
28.     # update the display surface with drawn object(s)
29.     pygame.display.update()
```

Program listing 2

What are the main differences between this and the previous listing?

The main points to notice are:

1. Changes from the original code listing are highlighted in bold. (This includes new and changed lines.)

2. The text in the title bar has changed to 'A single white rectangle'.

What caused this change?

(The text passed into the call, `pygame.display.set_caption` on line 12 is displayed on the title bar.)

3. Line 17 creates a rectangle object. It is important to understand that this line does not cause the rectangle to be displayed. Rather, it creates a representation for the rectangle in the computer's memory and stores a reference to it in the variable, `rectangle`.

Let's take a closer look at line 17:

```
17. rectangle = pygame.Rect(x, y, 20, 20)
```

What do you think the purpose of `x, y, 20, 20` are on line 17?

Explain why the rectangle is drawn in a different position each time the program is run?

The two values, `x` and `y`, represent the co-ordinates of the top-left corner of the rectangle. They tell `pygame` where to position the rectangle when it is drawn. Line 15 initialises `x` to a random integer between 0 and 400, and line 16 initialises `y` to a random integer between 0 and 300.

Experiment!
What happens if you set the co-ordinates of the top-left corner to 0, 0. Change line 17 as shown below and test your change by running the program several times until you figure out what is happening.

```
rectangle = pygame.Rect(0, 0, 20, 20)
```

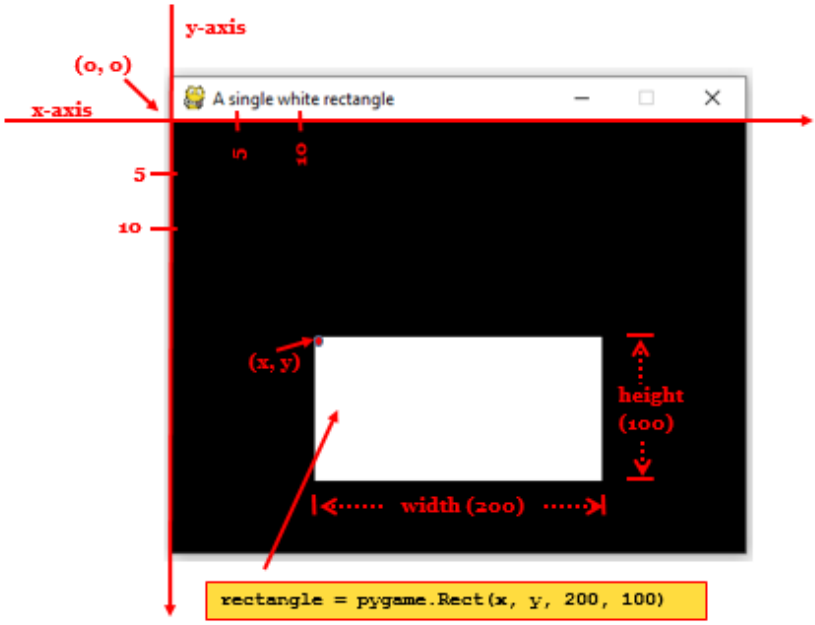
Did you notice that the rectangle no longer appears in a random position every time the program is run?

The values 20, 20 represent the *width* and *height* of the rectangle to be drawn.

Experiment!
Change the values for `width` and `height` and see what happens. You can make the changes either individually or together.
What happens if you make the `width` wider than the width of the display area? (Recall that the display area is 400 pixels wide.)
What's the largest rectangle you can make?

A note on the `pygame` co-ordinate system

The `pygame` co-ordinate system is important to understand. The diagram below highlights the key aspects of this system. By understanding the `pygame` co-ordinate system you can accurately position and control objects in your `pygame` program.



The `pygame` co-ordinate system

Notice that the origin (0, 0) is at the top-left of the window. This means that the top-left position of the screen has coordinates (0, 0), and the positive x-axis extends to the right, while the positive y-axis extends downward. The x-coordinate values increase as you move to the right from the origin and the y-coordinate values increase as you move down from the origin.

4. Line 18 tells `pygame` that it wants to draw the rectangular shape referenced by the variable, `rectangle`. Please note that nothing is drawn until the line 29 - `pygame.display.update()` - is executed.

This is done because drawing to the screen is a slow operation for the computer. You do not want to draw to the screen after each drawing function is called, but only draw the screen once after all the drawing functions have been called.

A note on colours

There are three primary colours of light: red, green and blue. By combining different amounts of these three colours you can form any other colour.

In Python, colours can be represented using tuples of three integers. The first value in the tuple is how much red is in the colour. A value of `0` means there is no red in this colour, and a value of `255` means there is a maximum amount of red in the colour. The second value is for green and the third value is for blue.

(0, 0, 0) → black
(255, 0, 0) → red
(0, 255, 0) → green
(0, 0, 255) → blue
(255, 255, 255) → white

Black is the absence of any colour. This is represented by the tuple `(0, 0, 0)`.

White is the full combination of red, green, and blue. Therefore, it can be represented by the tuple `(255, 255, 255)` for a maximum amount of red, green, and blue.

Experiment!

Declare a variable RED as shown below.

```
RED = (255,0,0)
```

Now modify line 18 of program listing 2 to the following:

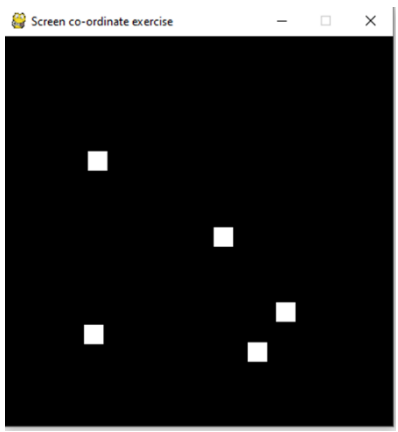
```
pygame.draw.rect(display_surface, RED, rectangle)
```

Run the program and record what happens below.

What happens if you change the word RED to BLUE?

Match the squares in the display window below to the co-ordinates. Either draw a line from the letter to the corresponding square or write in the letter beside the square

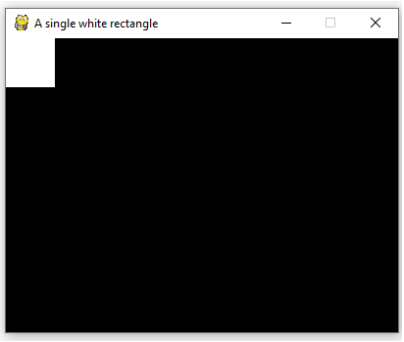
	x	y
A	52	148
B	22	225
C	134	54
D	39	378
E	329	317



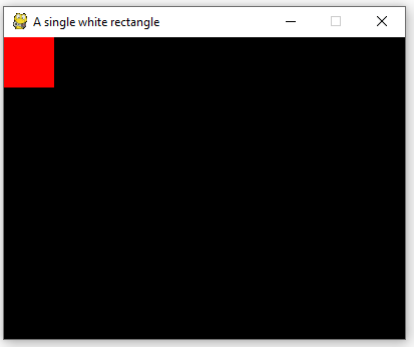
Programming Exercises 1

1. The window on the left-hand side below shows a white 50x50 rectangle on a black background. Modify the code shown in program listing 2 so that it displays a red rectangle on a black background as shown to the right-hand side.

c

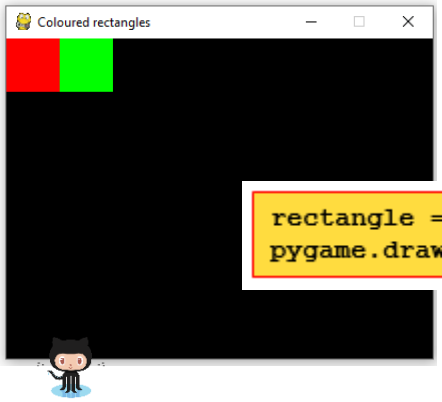


…. to this



Solution

2. Change the program so that it displays a second rectangle – this time in green – beside the red one.

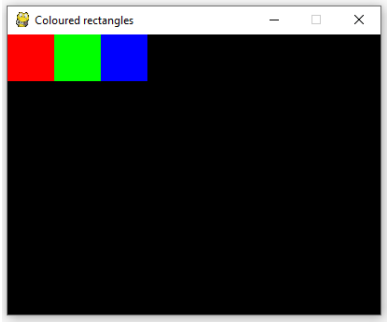


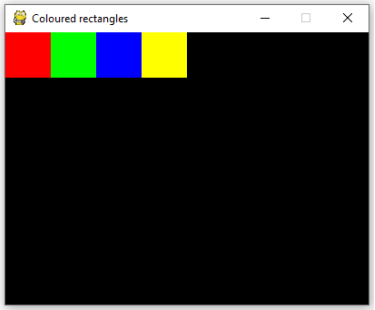
Hint: You will need to add the following lines of code at an appropriate place in the program.

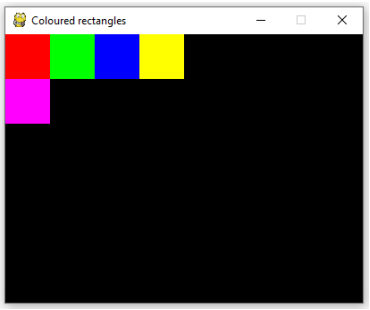
```
rectangle = pygame.Rect(50, 0, 50, 50)
pygame.draw.rect(display_surface, (0,255,0), rectangle)
```

Solution:

3. Make three further changes so that the program displays the following three windows.
(Click on the Octocat to see each solution.)







Experiment!
Replace lines 15-18 in program listing 2 with the lines below.
Run the program and record your findings in the space provided below.

```
x = 0
y = 0
for _ in range(5):
    rectangle = pygame.Rect(x, y, 50, 50)
    pygame.draw.rect(display_surface, (255,255,255), rectangle)
    x = x+50
    y = y+50
```

What pattern do you get?

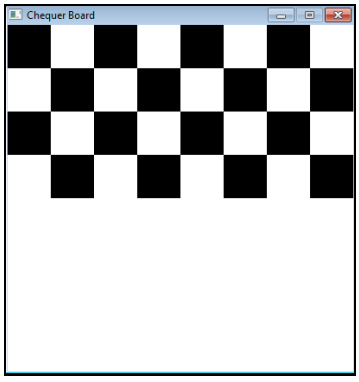
How many rectangles can you fit across the window?

How many rectangles can you fit down the window?

What if you change the size of the window? What if you change the size of the shapes?

---

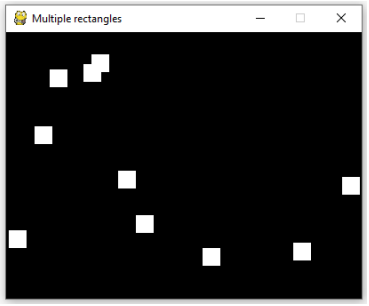
4. Can you write a program to display a chequerboard as shown below?



Program 3: Display multiple squares

In the previous lesson you learned how to create and display a single rectangle. In this lesson you will learn how to build on this in order to display multiple rectangles as shown on the right hand side below.

```
1.  # Program 3. Display multiple rectangles
2.  import pygame, sys, random, time
3.  from pygame.locals import *
4.
5.  # start the pygame engine
6.  pygame.init()
7.
8.  # create the display window
9.  display_surface = pygame.display.set_mode((400, 300))
10.
11. # Set the title of the display window
12. pygame.display.set_caption('Multiple rectangles')
13.
14. # define some colours
15. BLACK = (0,0,0)
16. WHITE = (255,255,255)
17.
18. # create the rectangles
19. for _ in range(10):
20.     x = random.randint(0, 400)
21.     y = random.randint(0, 300)
22.     rectangle = pygame.Rect(x, y, 20, 20)
23.     pygame.draw.rect(display_surface, WHITE, rectangle)
24.
25. # run the game loop
26. while True:
27.
28.     for event in pygame.event.get():
29.         if event.type == QUIT:
30.             pygame.quit()
31.             sys.exit()
32.
33.     pygame.display.update()
```



Program listing 3

The program shown in listing 3 creates and displays 10 rectangles. This effect is achieved using the `for` loop statement on line 19. The `for` loop statement on line 19 tells Python to run a loop 10 times. Note that lines 20-23 are indented. These lines make up the *loop body*.

You have already seen lines 20-22 – they are identical to lines 15-17 in program listing 2. These lines create a single rectangle using a randomly generated (x, y) co-ordinate for the top-left corner.

You have also seen line 23 – this corresponds to line 18 in program listing 2. This line 18 tells `pygame` that it wants to draw the rectangular shape referenced by `rectangle`. Please note that nothing is drawn until the line 33 - `pygame.display.update()` - is executed.

Programming Task

Can you figure out how to set the width and height of the rectangles to be random? What about drawing randomly coloured squares?

Program 4: Animation I (basic movement)

In this lesson we will learn how to give the appearance that the square is moving. This is called animation.

Animation can be achieved simply by re-drawing the block shape in different positions. We create the illusion of movement if this re-drawing is done continually i.e. within a loop such as the game loop.

The position of each shape can be accessed and modified by using the rectangle variable in our program listings. Each rectangle contains four special values (called attributes) as outlined below:

- `rectangle.left`: This is the x-coordinate of the left side of the rectangle
- `rectangle.right`: This is the x-coordinate of the right side of the rectangle
- `rectangle.top`: This is the y-coordinate of the top side of the rectangle
- `rectangle.bottom`: This is the y-coordinate of the bottom side of the rectangle

These attributes can be treated as variables. The secret to implementing animation lies in the manipulation of these variables.

Moving right

For example, if we want to give the impression something is moving rightwards we would change the value of `rectangle.right` by a positive amount.

This is illustrated on line 32 in program listing 4 shown here to the right. This line increments the value of `rectangle.right` by 1 pixel. Therefore, the next time the rectangle is drawn and the screen is updated the rectangle will appear 1 pixel to the right of its old position.

When the program is run you will see the square move from left to right across the window until it eventually disappears off the right edge of the window.

```
1.  # Program to demonstrate how to move right
2.  import pygame, sys, random, time
3.  from pygame.locals import *
4.
5.  # start the pygame engine
6.  pygame.init()
7.
8.  # create the display window
9.  display_surface = pygame.display.set_mode((400, 300))
10.
11. # Set the title of the display window
12. pygame.display.set_caption('Move right')
13.
14. # define some colours
15. BLACK = (0,0,0)
16. WHITE = (255,255,255)
17.
18. # create a rectangle to display
19. y = random.randint(0, 300)
20. rectangle = pygame.Rect(0, y, 20, 20)
21. pygame.draw.rect(display_surface, WHITE, rectangle)
22.
23. # run the game loop
24. while True:
25.
26.     for event in pygame.event.get():
27.         if event.type == QUIT:
28.             pygame.quit()
29.             sys.exit()
30.
31.     display_surface.fill(BLACK)
32.     rectangle.right = rectangle.right + 1
33.     pygame.draw.rect(display_surface, WHITE, rectangle)
34.
35.     # update the display surface with drawn object(s)
36.     pygame.display.update()
37.     time.sleep(0.01)
```

If you run the program several times you should notice that the square always starts along the left edge. Why do you think this is?

Answer.
The square always starts on the left edge because line 20 creates the initial rectangle using 0 as the value for the left co-ordinate.

Why does the square always appear to start at a different position along the left edge of the display window?

The initial vertical position of the square appears to be random because the y-coordinate is a random number between 0 and 300. This causes the square to be appear at a random position along the y-axis.

This illusion of movement is achieved by changing the right co-ordinate of the square on each iteration of the game loop. This is done on line 32 which adds 1 to the `rectangle.right` attribute. This causes the square to appear 1 pixel to the right the next time it is drawn (line 33) and the window's display area is updated (line 36).

Experiment!
Comment out line 31 and see what happens.
Run the program and record your findings in the space provided below.
Now comment out line 37.

Explain the purpose of lines 31 and 37.

Line 31:

Line 37:

- Line 31 acts like an eraser. It fills the background with black to remove any trace of the square in its original position before re-drawing it in its new position.
- Line 37 causes the program to slow down. It tells Python to 'sleep' for a millisecond on each iteration.

Programming Task 4.1

Program listing 4 shown below demonstrates how to move a square from the bottom to the top of the window. Pay particular attention to the techniques used to set the initial position of the square (lines 19 and 20) and move the square upwards by subtracting 1 from the `rectangle.top` attribute on each iteration of the loop (line 32).

Modify the code to achieve the following animations:

a) from the right edge of the window to the left (Hint: the x-coordinate of the right edge is 400.)
b) from top edge of the window to the bottom (Hint: the y-coordinate of the top edge is 0.)

```
1.  # Program to demonstrate how to move UP
2.  import pygame, sys, random, time
3.  from pygame.locals import *
4.
5.  # start the pygame engine
6.  pygame.init()
7.
8.  # create the display window
9.  display_surface = pygame.display.set_mode((400, 300))
10.
11. # Set the title of the display window
12. pygame.display.set_caption('Move right')
13.
14. # define some colours
15. BLACK = (0,0,0)
16. WHITE = (255,255,255)
17.
18. # create a rectangle to display
19. x = random.randint(0, 400)
20. rectangle = pygame.Rect(x, 300, 20, 20)
21. pygame.draw.rect(display_surface, WHITE, rectangle)
22.
23. # run the game loop
24. while True:
25.
26.     for event in pygame.event.get():
27.         if event.type == QUIT:
28.             pygame.quit()
29.             sys.exit()
30.
31.     display_surface.fill(BLACK)
32.     rectangle.top = rectangle.top - 1
33.     pygame.draw.rect(display_surface, WHITE, rectangle)
34.
35.     # update the display surface with drawn object(s)
36.     pygame.display.update()
37.     time.sleep(0.01)
```

Program listing 4 (move up)

Reflection.
What was your main challenge in completing this programming task?
Outline any limitation to the animation you can think of so far.

Diagonal Animation

Movement from the top-left corner to the bottom right corner of the window (diagonal animation) is achieved using the code shown in program listing 4.2 below.

```
1.  # Program 4.2 - create a single moving rectangle
2.  import pygame, sys, random, time
3.  from pygame.locals import *
4.
5.  # start the pygame engine
6.  pygame.init()
7.
8.  # create the display window
9.  WINDOWWIDTH = 400
10. WINDOWHEIGHT = 400
11. display_surface = pygame.display.set_mode((WINDOWWIDTH,
                                                WINDOWHEIGHT))
12.
13. # Set the title of the display window
14. pygame.display.set_caption('Moving square')
15.
16. # define some colours
17. BLACK = (0, 0, 0)
18. WHITE = (255, 255, 255)
19.
20. # create a rectangle to display at (0,0)
21. x = 0
22. y = 0
23. rectangle = pygame.Rect(x, y, 20, 20)
24. pygame.draw.rect(display_surface, WHITE, rectangle)
25.
26. # declare movement variables
27. MOVESPEED = 1
28. move_down = True
29. move_left = False
30.
```

```
31. # run the game loop
32. while True:
33.
34.     for event in pygame.event.get():
35.         if event.type == QUIT:
36.             pygame.quit()
37.             sys.exit()
38.
39.     # Perform the animation
40.     if move_down:
41.         rectangle.bottom = rectangle.bottom + MOVESPEED
42.     else:
43.         rectangle.bottom = rectangle.bottom - MOVESPEED
44.
45.     if move_left:
46.         rectangle.left = rectangle.left - MOVESPEED
47.     else:
48.         rectangle.left = rectangle.left + MOVESPEED
49.
50.     display_surface.fill(BLACK)
51.     pygame.draw.rect(display_surface, WHITE, rectangle)
52.
53.     # update the display surface with drawn object(s)
54.     pygame.display.update()
55.     time.sleep(0.01)
```

Program listing 4.2

Notice from lines 28 and 29 that two Boolean variables – `move_down` and `move_left` are initialised to `True` and `False` respectively.

- By setting `move_down` to `True` will cause the movement of the square to be from top to bottom
- By setting `move_left` to `False` will cause the movement of the square to be from left to right

These settings combine to give the illusion that the square is moving diagonally – downward and rightward – at the same time.

The actual animation itself is implemented in lines 40-48. As before, the animation is created by repeatedly changing the position of the square.

Experiment!

Change lines 28 and 29 as follows

```
move_down = random.choice([True, False])
move_left = random.choice([True, False])
```

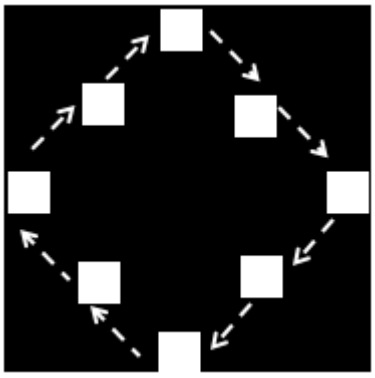
Run the program and record your findings in the space provided below.

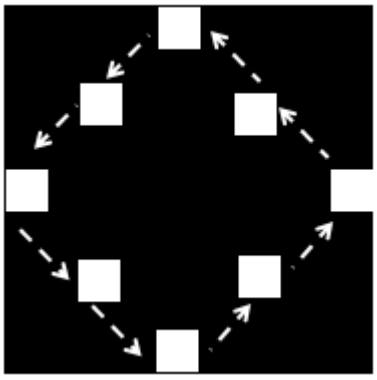
Program 5: Animation II (bouncing)

In the previous lesson we learned how by changing the position of a square we can create the illusion of animation. One obvious drawback to the way our animation behaves is that the square disappears as soon as it reaches the edge of the window.

We need to think more carefully about what we want. How exactly should the system behave? What should the shape do when it reaches the edge of the window? Should it re-appear on the opposite side or simply bounce back?

The answers to these question will define our *system requirements*. For the purpose of this lesson we define our requirements as depicted in the illustrations shown below.





The key to implementing this motion lies in controlling the values of the two Boolean variables – `move_down` and `move_left` – and knowing when the shape has reached one of the edges.

The pseudo-code to explain what happens when the square is moving vertically (i.e. either in a downward direction or an upward direction) is shown below.

If the square is moving down:
    If the square is at the bottom edge of the window:
        change the direction (i.e. set `move_down` to `False` to indicate that the square is now moving up)
    Else (the square is just moving down):
        Increase the y-coordinate of the square's bottom edge
Else (the square must be moving up):
    If the square is at the top edge of the window:
        change the direction (i.e. set `move_down` to `True` to indicate that the square is now moving down)
    Else (the square is just moving up):
        Decrease the y-coordinate of the square's top edge

This Python implementation of this pseudo-code is shown between lines 42 and 52 in program listing 5 on the next page.

Note that the code below shows the game loop only. The first 30 lines are identical to those shown in program listing 4.2.

```
31. # run the game loop
32.
33. while True:
34.
35.     for event in pygame.event.get():
36.         if event.type == QUIT:
37.             pygame.quit()
38.            sys.exit()
39.
40.     display_surface.fill(BLACK)
41.
42.     # Perform the vertical animation
43.     if move_down:
44.         if rectangle.bottom >= WINDOWHEIGHT:
45.             move_down = False
46.         else:
47.             rectangle.bottom = rectangle.bottom + MOVESPEED
48.     else:
49.         if rectangle.top <= 0:
50.             move_down = True
51.         else:
52.             rectangle.top = rectangle.top - MOVESPEED
53.
54.     # Perform the horizontal animation
55.     if move_left:
56.         if rectangle.left <= 0:
57.             move_left = False
58.         else:
59.             rectangle.left = rectangle.left - MOVESPEED
60.     else:
61.         if rectangle.right >= WINDOWWIDTH:
62.             move_left = True
63.         else:
64.             rectangle.right = rectangle.right + MOVESPEED
65.
66.     pygame.draw.rect(display_surface, WHITE, rectangle)
67.
68.     # update the display surface with drawn object(s)
69.     pygame.display.update()
70.     time.sleep(0.01)
```

Program listing 5

Lines 42-64 handle the animation. The vertical movement (as described in the pseudo-code earlier is implemented from lines 42-52), while lines 54-64 implement the horizontal animation.

The expression `rectangle.bottom >= WINDOWHEIGHT` on line 44 is used to determine if the square has reached the bottom edge of the window. Explain the purpose of the following three expressions:

```
rectangle.top <= 0
```

```
rectangle.left <= 0
```

```
rectangle.right >= WINDOWWIDTH
```

Programming Task 5.1. (Parson's problem)

The graphic below depicts 11 separate blocks of code in no particular order. When the blocks are combined into the correct order the resulting program will display 10 squares. Each square will be animated in a manner like the animation achieved earlier using program listing 5.

1.
```
# update the display surface with drawn object(s)
pygame.display.update()
time.sleep(0.01)
```

2.
```
# Set the title of the display window
pygame.display.set_caption('Moving squares that bounce')
```

3.
```
# Perform the animation on each rectangle
for rectangle in rectangles:
    # Perform the vertical animation
    if move_down:
        if rectangle.bottom >= WINDOWHEIGHT:
            move_down = False
        else:
            rectangle.bottom = rectangle.bottom + MOVESPEED
    else:
        if rectangle.top <= 0:
            move_down = True
        else:
            rectangle.top = rectangle.top - MOVESPEED

    # Perform the horizontal animation
    if move_left:
        if rectangle.left <= 0:
            move_left = False
        else:
            rectangle.left = rectangle.left - MOVESPEED
    else:
        if rectangle.right >= WINDOWWIDTH:
            move_left = True
        else:
            rectangle.right = rectangle.right + MOVESPEED

    pygame.draw.rect(display_surface, WHITE, rectangle)
```

4.
```
# create the display window
WINDOWWIDTH = 400
WINDOWHEIGHT = 400
display_surface = pygame.display.set_mode((WINDOWWIDTH,
                                           WINDOWHEIGHT))
```

5.
```
# start the pygame engine
pygame.init()
```

6.
```
# create multiple rectangles to display
rectangles = []
for _ in range(10):
    x = random.randint(0, WINDOWWIDTH)
    y = random.randint(0, WINDOWHEIGHT)
    rectangle = pygame.Rect(x, y, 20, 20)
    pygame.draw.rect(display_surface, WHITE, rectangle)
    rectangles.append(rectangle)
```

7.
```
# define some colours
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
```

8.
```
# run the game loop
while True:

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

9.
```
display_surface.fill(BLACK)
```

10.
```
import pygame, sys, random, time
from pygame.locals import *
```

11.
```
# declare movement variables
MOVESPEED = 1
move_down = True
move_left = False
```

A link to the solution is provided here

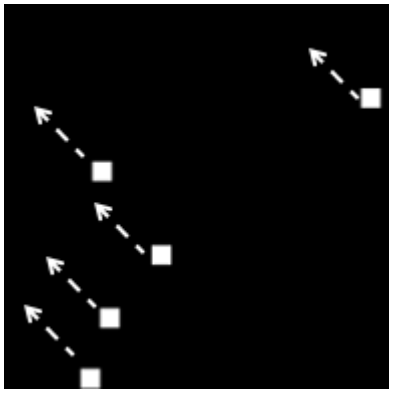
Reflection

What were the main challenges and how did you overcome them?

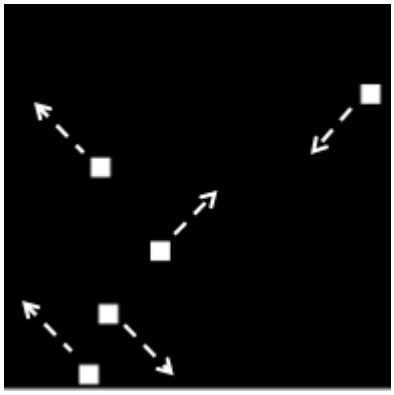
Evaluate the solution. Identify one improvement that could be made.

Program 6: Animation III (independent movement)

The solution to the previous programming task results in a program that displays 10 animated squares. However, there is one major drawback (which hopefully you identified in the evaluation) - the squares do not move independently. This gives a 'herd like' behaviour in which every square follows the same movement pattern as depicted on the left hand side below. The arrows indicate the direction of movement. The desired behaviour – all squares moving independently in different directions – is depicted on the right hand side.



Actual behaviour
Every square moves in the same direction



Desired behaviour
Squares moving independently

In this lesson you will learn how to make each square move independently. The solution makes use of a list of dictionaries. The code below shows how this list is constructed. The list is called `satellites` and the for loop is used to create 10 different elements. Each element is a dictionary called `satellite` and each satellite contains the following keys:

- `sprite`: A reference to the rectangle object.
- `h_speed`: A random integer – either 1 or 2 used to control the horizontal speed.
- `v_speed`: A random integer – either 1 or 2 used to control the vertical speed.
- `move_down`: a Boolean variable used to indicate and control the vertical movement of sprite.
- `move_left`: a Boolean variable used to indicate and control the horizontal movement of sprite

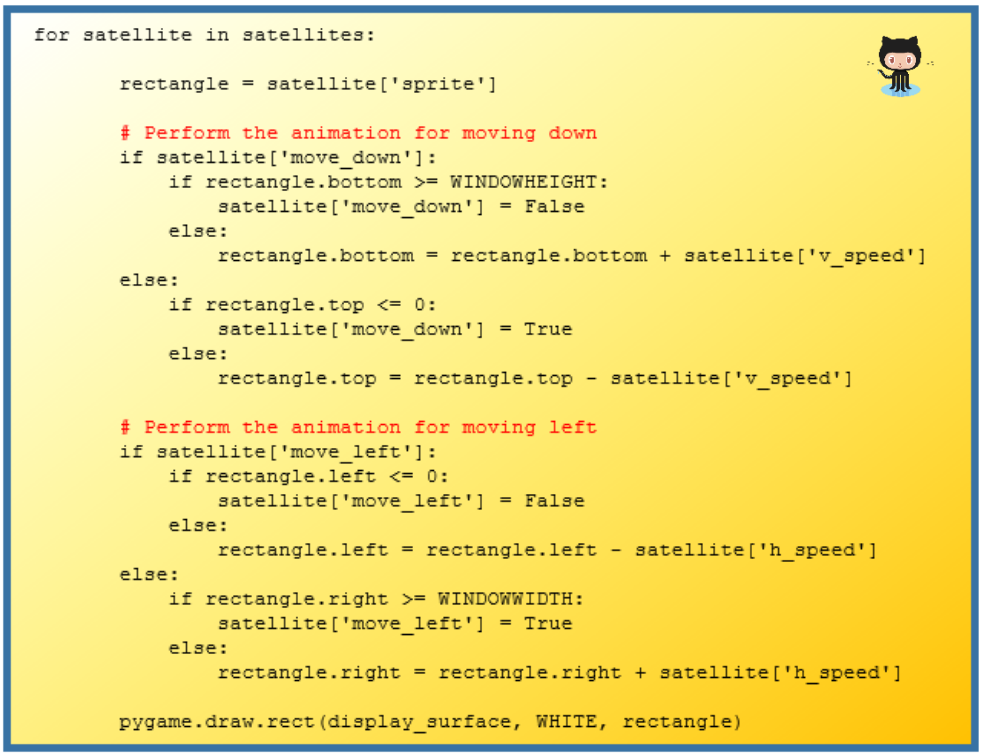
```
# create a list of dictionaries
satellites = [] # initial list is empty
for count in range(10):
    x = random.randint(0, WINDOWWIDTH)
    y = random.randint(0, WINDOWHEIGHT)

    satellite = {} # initial dictionary is empty
    satellite['sprite'] = pygame.Rect(x, y, 20, 20)
    satellite['h_speed'] = random.randint(1, 2)
    satellite['v_speed'] = random.randint(1, 2)
    satellite['move_down'] = random.choice([True,False])
    satellite['move_left'] = random.choice([True,False])

    satellites.append(satellite) # add the dictionary to the list
```

Program listing 6 (part 1)

The independent movement is made possible because the dictionary stores values for `h_speed`, `v_speed`, `move_down`, and `move_left` for each individual sprite. (This can be contrasted with

the previous implementation in which the values of `MOVESPEED`, `move_down`, and `move_left` were applied to every rectangle.)

The code below demonstrates how animation is achieved for each individual sprite.



```
for satellite in satellites:

        rectangle = satellite['sprite']

        # Perform the animation for moving down
        if satellite['move_down']:
            if rectangle.bottom >= WINDOWHEIGHT:
                satellite['move_down'] = False
            else:
                rectangle.bottom = rectangle.bottom + satellite['v_speed']
        else:
            if rectangle.top <= 0:
                satellite['move_down'] = True
            else:
                rectangle.top = rectangle.top - satellite['v_speed']

        # Perform the animation for moving left
        if satellite['move_left']:
            if rectangle.left <= 0:
                satellite['move_left'] = False
            else:
                rectangle.left = rectangle.left - satellite['h_speed']
        else:
            if rectangle.right >= WINDOWWIDTH:
                satellite['move_left'] = True
            else:
                rectangle.right = rectangle.right + satellite['h_speed']

        pygame.draw.rect(display_surface, WHITE, rectangle)
```

Program listing 6 (part 2)

Compare the animation algorithm shown in listing 6B with that of listing 5. What similarities and differences do you notice?

Similarities:

Differences:

Using pseudo-code (or otherwise) explain how vertical animation is achieved in the above code.

A more detailed explanation of program listings 6A and 6B

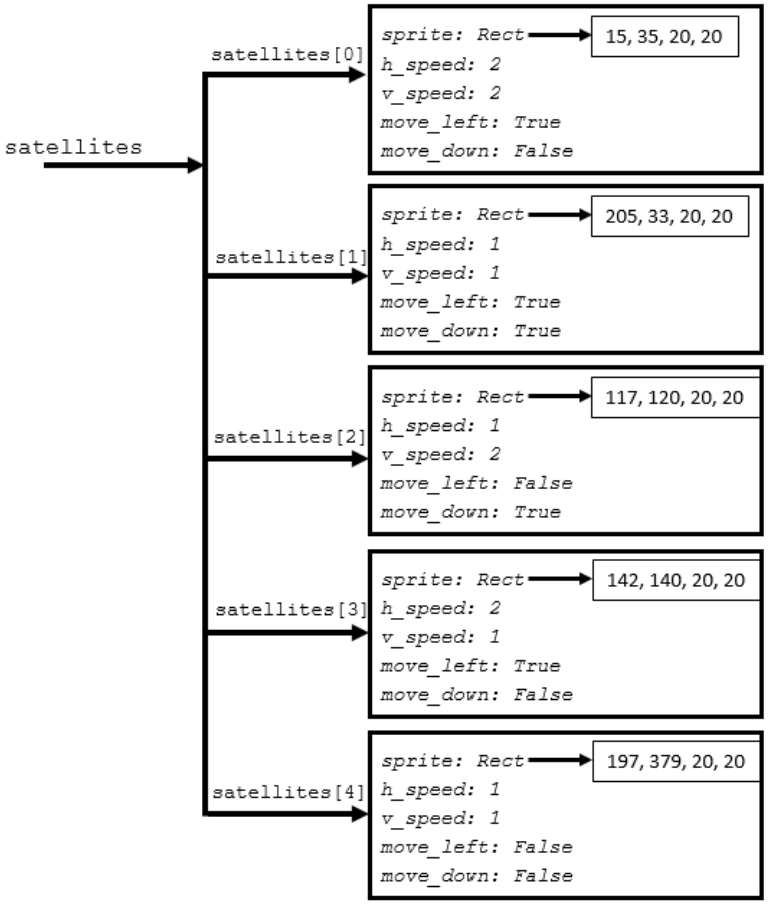
Let's examine the code in program listing 6A and 6B in a little more detail. You should focus particular attention on two variables:

1. `satellites` in listing 6A and
2. `satellite` in listing 6B.

The former is a list and the latter is an element of the list which has a dictionary datatype.

For the sake of simplicity let us say that the code in program listing 5A created 5 (and not 10) satellite objects. (To make this happen all you would need to do is change the `for` loop counter from 10 to 5.). The `for` loop in listing 5A builds up a list of satellites.

Each of the 5 satellite objects are represented in memory by a dictionary. The dictionary for each satellite stores that satellite's position on the screen as well as the vertical and horizontal direction and speed. Each dictionary is a separate element in a list that contains *all* the satellites. So, the memory representation for these 5 satellites might look as follows:



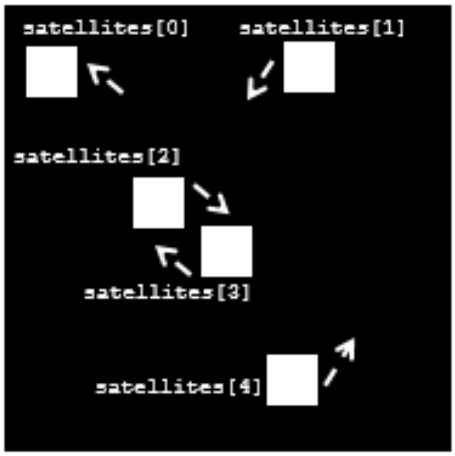
The graphic depicts a memory representation for 5 satellites. The satellites are stored as a list of dictionaries. The list is called `satellites` and each individual satellite is a dictionary stored as a separate element in this list.

Program listing 6B starts with a `for` loop:

```
for satellite in satellites:
```

This tells Python to iterate through each of the elements in the `satellites` list and, as it passes over each element i.e. `satellites[0]`, `satellites[1]`, `satellites[2]`, `satellites[3]`, and `satellites[4]`,– refer to it by the variable `satellite`. So, the variable `satellite` is used as a generic name to refer to whichever element of the `satellites` list the loop happens to be iterating over.

The image shown below depicts 5 'satellites' all moving in different directions around the display window. The 5 satellites shown here as 20x20 squares correspond to the 5 dictionaries on the previous page.



The information required to display and move each satellite object is contained in its own dictionary.

For example, the first element – `satellites[0]` - corresponds to the square on the top left hand corner in the display window. The arrows are used to indicate which direction the objects are moving in. Since the arrow beside the first object is pointing diagonally upwards and leftwards it indicates that the object is moving in an up-left direction. This information can be found in the dictionary entry where the value of `move_left` is `True` and `move_down` is `False`.

Notice also that the position of each individual satellite (square) is stored as part of the `sprite` entry in each dictionary. For example, the x-coordinate of the top-left square is 15 and the y-coordinate is 35. Each square has a width and height of 20 pixels.

Finally, the dictionary stores the horizontal and vertical speeds in `h_speed` and `v_speed` respectively. These are the amounts by which the animation algorithm adds or subtracts (depending on the direction the object is travelling) to/from the object's current position in order to achieve the illusion of movement.

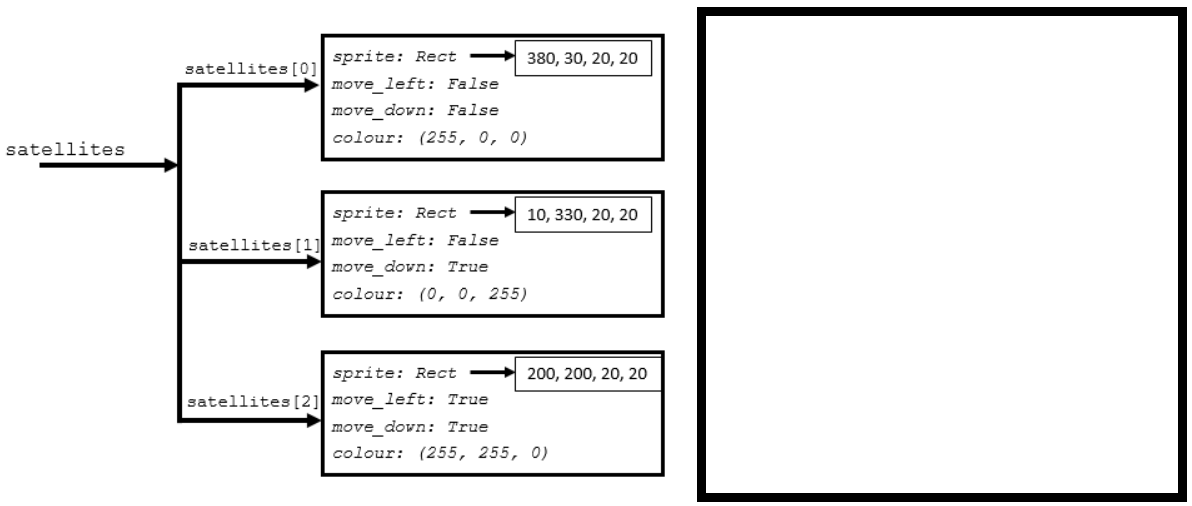
Explain how the variables `h_speed` and `v_speed` are used in program listing 6B.

`h_speed`

`v_speed`

Given the following memory representation for 3 satellites, illustrate how they would be displayed on the 400 x 400 display window to the right. Recall that the top-left corner is (0, 0)

Note: For each satellite you should draw a square at the approximate correct position, indicate its direction using a diagonal arrow and state which colour it will be displayed as.



Explain why the animation algorithm shown in listing 6B results in four types of diagonal movement only i.e. up-left, up-right, down-left and down-right.

List four other types of movement not implemented by the animation algorithm

Programming Task 6.1 (Parson's problem)

Arrange the blocks of code shown below into the correct order to produce a running program. The program should display 10 independently animated squares.

1.
```
# update the display surface with drawn object(s)
pygame.display.update()
time.sleep(0.01)
```

2.
```
# Set the title of the display window
pygame.display.set_caption('Moving squares that bounce')
```

3.
```
for satellite in satellites:

        rectangle = satellite['sprite']

        # Perform the vertical animation
        if satellite['move_down']:
            if rectangle.bottom >= WINDOWHEIGHT:
                satellite['move_down'] = False
            else:
                rectangle.bottom = rectangle.bottom + satellite['v_speed']
        else:
            if rectangle.top <= 0:
                satellite['move_down'] = True
            else:
                rectangle.top = rectangle.top - satellite['v_speed']

        # Perform the horizontal animation
        if satellite['move_left']:
            if rectangle.left <= 0:
                satellite['move_left'] = False
            else:
                rectangle.left = rectangle.left - satellite['h_speed']
        else:
            if rectangle.right >= WINDOWWIDTH:
                satellite['move_left'] = True
            else:
                rectangle.right = rectangle.right + satellite['h_speed']

        pygame.draw.rect(display_surface, WHITE, rectangle)
```

4.
```
# define some colours
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
```

5.
```
# create a list of dictionaries
satellites = [] # initial list is empty
for count in range(10):
    x = random.randint(0, WINDOWWIDTH)
    y = random.randint(0, WINDOWHEIGHT)

    satellite = {} # initial dictionary is empty
    satellite['sprite'] = pygame.Rect(x, y, 20, 20)
    satellite['h_speed'] = random.randint(1, 2)
    satellite['v_speed'] = random.randint(1, 2)
    satellite['move_down'] = random.choice([True,False])
    satellite['move_left'] = random.choice([True,False])

    satellites.append(satellite) # add the dictionary to the list
```

6.
```
# create the display window
WINDOWWIDTH = 400
WINDOWHEIGHT = 400
display_surface = pygame.display.set_mode((WINDOWWIDTH,
                                           WINDOWHEIGHT))
```

7.
```
# run the game loop
while True:

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

8.
```
display_surface.fill(BLACK)
```

9.
```
import pygame, sys, random, time
from pygame.locals import *
```

10.
```
# start the pygame engine
pygame.init()
```

A link to the solution is provided here

Reflection

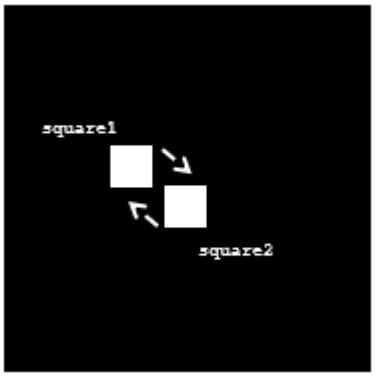
What were the main challenges and how did you overcome them?

Evaluate the solution. Identify one improvement that could be made.

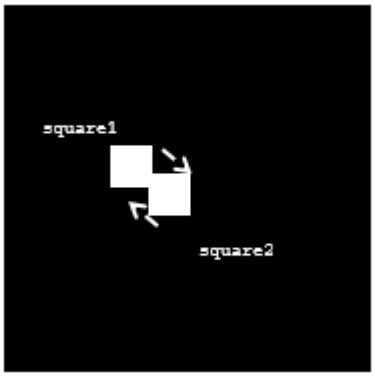
Program 7: Collision Detection I

Now that you have learned how to control the squares so that they can move independently, the possibility of collisions between squares has been introduced. What should happen when two squares crash into one another?

Before answering this question, it might be useful to consider how to detect collisions. Let's say there are two squares referenced in your code by the variables `square1` and `square2`. The graphic on the left shows that the squares are moving towards each other – they are on a collision course. The graphic on the right shown the squares colliding – the squares are overlapping.



Two squares on a collision course



Two squares collide!

When you want (your program) to detect collisions you can use the `pygame` method `colliderect`. Because the method returns `True` if the squares overlap and `False` of they do not overlap, it is included as part of an `if` statement as shown below.

```
# Detect if square1 and square2 collide
if square1.colliderect(square2):
   print("Collision detected")
```

Since `square1` colliding into `square2` can be considered the same as `square2` colliding into `square1`, the `if` statement shown below can also be used.

```
# Detect if square1 and square2 collide
if square2.colliderect(square1):
   print("Collision detected")
```

The main thing to understand is how to detect collisions.

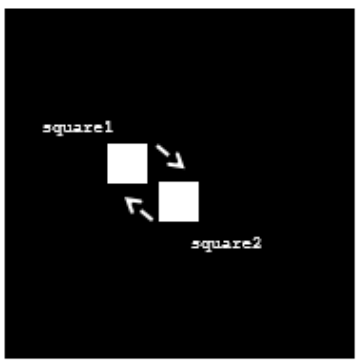
Programming Task 7.1

Given two moving objects – `circle` and `rectangle` - write a Python statement to display the line *HIT* if they collide.

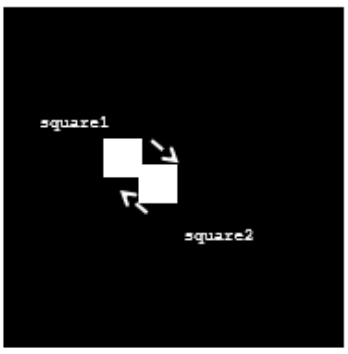
What change would you need to make to your solution so that it also displays the message MISS if the two objects do not collide?

Now that you can detect object collision we will return to our earlier question: *what should happen when two objects crash into one another?* Of course, we could do nothing i.e. ignore the collision and simply let both objects continue along their merry ways as though nothing at all happened. This is effectively what is already happening. We want to model something more realistic. One idea would be to simulate a 'bouncing effect' in which the direction in the objects are travelling is changed by the impact. Let's examine how this could be achieved.

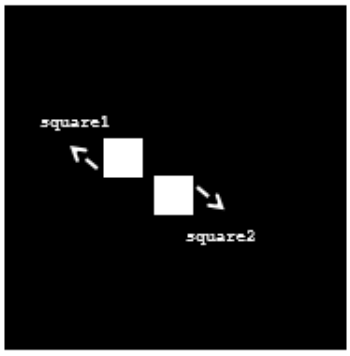
The graphic below illustrates the 'bouncing effect' we are trying to achieve.



Before collision
Squares moving towards each other



Collision



After collision
Squares have changed direction

The code snippet below shows how this 'bouncing effect' can be achieved between two objects.

```
if square1.colliderect(square2): # collision detected
   satellite['move_left'] = not satellite['move_left']
   satellite['move_down'] = not satellite['move_down']
```

As you can see from the code snippet the horizontal and vertical directions of the satellite are reversed if the code detects a collision. This is achieved by applying the `not` Boolean operator to whatever value `move_left` and `move_down` were before the collision. For example, if `move_left` was `True` before the collision its value would be changed to `False` (i.e. `not True`) as a result of the collision. Similarly, if `move_down` was `False` before the collision its value would be changed to `True` (i.e. `not False`) as a result of the collision.

By now you should understand how to achieve a bouncing effect between two objects. In order to make this work when there are multiple 'satellites' moving around the screen (as is the case here) you will need to include the code inside another `for` loop.

```
# Collision detection ...
for sat in satellites:
   # If the objects are not the same and there is overlap ...
   if sat['sprite'] != rectangle and rectangle.colliderect(sat['sprite']):
      # ... simulate a bounce effect
      satellite['move_left'] = not satellite['move_left']
      satellite['move_down'] = not satellite['move_down']

pygame.draw.rect(display_surface, WHITE, rectangle)
```

The for loop iterates across every satellite in the list of satellites checking for a collision. Notice that the condition has been modified to include the expression `sat['sprite] != rectangle`. Because the loop checks *every* statellite for a possible collision with the current satellite (rectangle) it must take action not to detect a collision with itself.

Programming Task 7.2

Insert the above code snippet at the end of the animation algorithm shown in program listing 6A to create a bouncing effect simulation with multiple objects. (You should be able to use the code resulting from programming task 6 as your base program.)

Experiment!

Replace the two lines ...

```
satellite['move_left'] = not satellite['move_left']
satellite['move_down'] = not satellite['move_down']
```

... with the following two lines:

```
satellite['move_left'] = True
satellite['move_down'] = True
```

Explain any change(s) to the bouncing effect which you notice?

The full solution to the programming task on the previous page is shown in program listing 7.

```
# Program 7 - collision detection - bounce
import pygame, sys, random, time
from pygame.locals import *

# start the pygame engine
pygame.init()

# create the display window
WINDOWWIDTH = 400
WINDOWHEIGHT = 400
display_surface = pygame.display.set_mode((WINDOWWIDTH,
                                           WINDOWHEIGHT))

# Set the title of the display window
pygame.display.set_caption('Collision detection - bounce')

# define some colours
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)

# create a list of dictionaries
satellites = []
for count in range(5):
    x = random.randint(0, WINDOWWIDTH)
    y = random.randint(0, WINDOWHEIGHT)
    print(x,y)

    satellite = {}
    satellite['sprite'] = pygame.Rect(x, y, 20, 20)
    satellite['h_speed'] = random.randint(1, 2)
    satellite['v_speed'] = random.randint(1, 2)
    satellite['move_down'] = random.choice([True,False])
    satellite['move_left'] = random.choice([True,False])

    satellites.append(satellite)
```

```
# run the game loop
while True:

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

    display_surface.fill(BLACK)

    for satellite in satellites:

        rectangle = satellite['sprite']

        # Perform the vertical animation
        if satellite['move_down']:
            if rectangle.bottom >= WINDOWHEIGHT:
                satellite['move_down'] = False
            else:
                rectangle.bottom = rectangle.bottom + satellite['v_speed']
        else:
            if rectangle.top <= 0:
                satellite['move_down'] = True
            else:
                rectangle.top = rectangle.top - satellite['v_speed']

        # Perform the horizontal animation
        if satellite['move_left']:
            if rectangle.left <= 0:
                satellite['move_left'] = False
            else:
                rectangle.left = rectangle.left - satellite['h_speed']
        else:
            if rectangle.right >= WINDOWWIDTH:
                satellite['move_left'] = True
            else:
                rectangle.right = rectangle.right + satellite['h_speed']

        # Collision detection ...
        for sat in satellites:
            # If the objects are not the same and there is overlap ...
            if sat['sprite'] != rectangle and rectangle.colliderect(sat['sprite']):

                # ... simulate a bounce effect
                satellite['move_left'] = not satellite['move_left']
                satellite['move_down'] = not satellite['move_down']

        pygame.draw.rect(display_surface, WHITE, rectangle)

    # update the display surface with drawn object(s)
    pygame.display.update()
    time.sleep(0.01)
```

Program listing 7: Collision Detection (and bounce)

Reflection

Can you think of any other effects that could be added in when the objects collide?

One possible enhancement you could make would be to simulate an object breaking up when it collides with another. This could be called *the disintegration effect*.

The disintegration effect could be achieved by halving the objects width and height every time it crashes into another object. You will achieve this effect by completing the next programming task.

Programming Task 7.3

Modify the collision detection algorithm of program listing 7 by including the highlighted lines from the code snippet below.

```
# Collision detection ...
for sat in satellites:
   # If the objects are not the same and there is overlap ...
   if sat['sprite'] != rectangle and rectangle.colliderect(sat['sprite']):
      # ... simulate a bounce effect
      satellite['move_left'] = not satellite['move_left']
      satellite['move_down'] = not satellite['move_down']

      # ... simulate disintegration
      rectangle.height = rectangle.height//2
      rectangle.width = rectangle.width//2

pygame.draw.rect(display_surface, WHITE, rectangle)
```

Run the code and record your observations below.

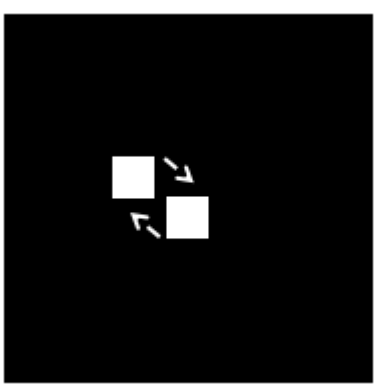
Can you suggest any enhancements that could be made?

Observations:

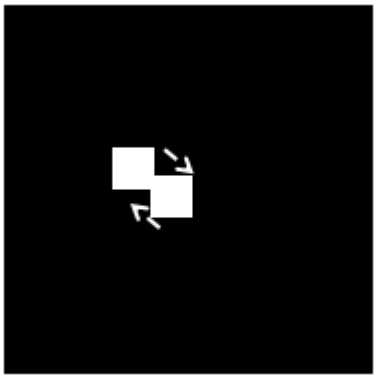
Enhancements:

So far so good! Or is it?
Did you observe from running the previous program that there were always the same number of objects on the display screen. If your initial loop created 5 objects then there would always be 5 objects. The only change that would happen when they collided was that one would get smaller.

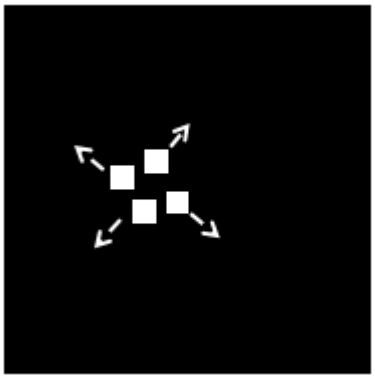
A more realistic model would be to *split* the object upon impact with another object as depicted below.



Before collision
Squares moving towards each other



Collision



After collision
Squares have changed direction

The implementation – shown below extends the collision detection algorithm

Programming Task 7.4

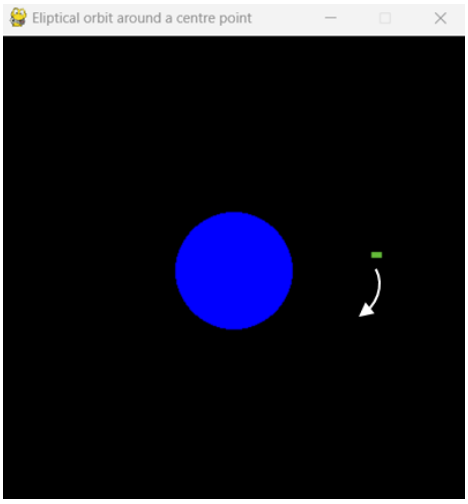
Modify the collision detection algorithm to include the lines highlighted in the code snippet below to simulate the creation of debris when two objects collide.

```
# Collision detection ...
for sat in satellites:
    # If the objects are not the same and there is overlap ...
    if sat['sprite'] != rectangle and rectangle.colliderect(sat['sprite']):
        # ... simulate a bounce effect
        satellite['move_left'] = not satellite['move_left']
        satellite['move_down'] = not satellite['move_down']

        # ... simulate disintegration
        rectangle.height = rectangle.height//2
        rectangle.width = rectangle.width//2

        # ... simulate debris
        debris = {}
        debris['sprite'] = rectangle.copy()
        debris['sprite'].x = random.randint(0, 400)
        debris['sprite'].y = random.randint(0, 400)
        debris['h_speed'] = 1
        debris['v_speed'] = 1
        debris['move_left'] = random.choice([True,False])
        debris['move_down'] = random.choice([True,False])
        satellites.append(debris)

pygame.draw.rect(display_surface, WHITE, rectangle)
```

Program 8: Circular Motion

So far, all motion has been in a straight line – horizontally, vertically, or diagonally. In this program you will learn how to make a single object move in an elliptical orbit around a fixed central object. You can think of the moving object as a satellite and the fixed central object as Earth.

The desired effect is illustrated below.



We will piece the different elements of the solution like a jigsaw.

STEP 1. The first piece of the jigsaw is the code to display Earth.

```
# draw circle in center of screen - Earth
pygame.draw.circle(display_surface, BLUE, [WINDOWWIDTH//2, WINDOWHEIGHT//2], 50)
```

This line of code draws a blue circle with a radius of 50 units at the centre of the window.
Since the window will need to be refreshed to simulate the satellite movement this code will have to be inside the main program loop.

Reflection
What causes the above code to position Earth in the centre of the window?

Reflection
What would happen if Earth was not re-drawn every time the screen was refreshed?

STEP 2. The next step is to create the satellite object itself. We already know how to do this from earlier programs, but the code is shown here again just to remind you.

```
# create a single rectangle (satellite) to display
satellite = {}
satellite['sprite'] = pygame.Rect(0, 0, random.randint(1, 10), random.randint(1, 10))
satellite['colour'] = (random.randint(0, 255), random.randint(0, 255), random.randint(0, 255))
satellite['angle'] = random.randint(0, 360)
satellite['X_ellipse'] = random.randint(100, 200) # X_ellipse is major radius of ellipsis
satellite['Y_ellipse'] = random.randint(100, 200) # Y_ellipse is minor radius of ellipsis
```

We are using a Python dictionary to store the information for the satellite. The dictionary has several items as represented by the following key-value pairs:

Key	Value
sprite	A rectangle object with random widths and heights set between 1 and 10 units.
colour	Three random values between 0 and 255. Ensures the colour of the satellite is random.
angle	A random value between 0 and 360. This value is used to calculate the position of the rectangle as it moves around the earth.
X_ellipse	This is the major radius of the ellipse.
Y_ellipse	This is the minor radius of the ellipse.

STEP 3. We now need the code to simulate the elliptical motion. This is accomplished using the formula for the circumference on a circle as illustrate in the code below. The code is placed inside the main program loop which refreshes the screen every hundredth of a second during which time the x and y position of the satellite are changed to give the illusion of movement.

```
rectangle = satellite['sprite']
degree = satellite['angle']
major_rad = satellite['X_ellipse']
minor_rad = satellite['Y_ellipse']

rectangle.x = int(math.cos(degree * 2 * math.pi/360) * major_rad) + X_center
rectangle.y = int(math.sin(degree * 2 * math.pi/360) * minor_rad) + Y_center

satellite['angle'] += 1
```

STEP 4. Now that the details of the satellite have been changed it can be displayed at its new position using the following line of code.

```
# Now display the updated satellite
pygame.draw.rect(display_surface, satellite['colour'], rectangle)
```

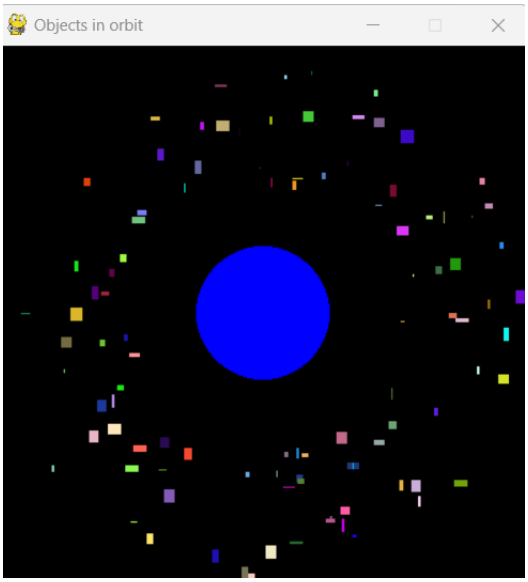
STEP 5. Putting it all together

The full solution is provided in program listing 8 (pgm8.py) here

Final Tasks

Programming Task 8.1

Modify the code to create and display 100 satellite objects all orbiting about a central point.



Programming Task 8.2

Write a program to simulate the Kessler effect. As the satellites collide into each other they will break up into smaller satellites (shown below in RED).