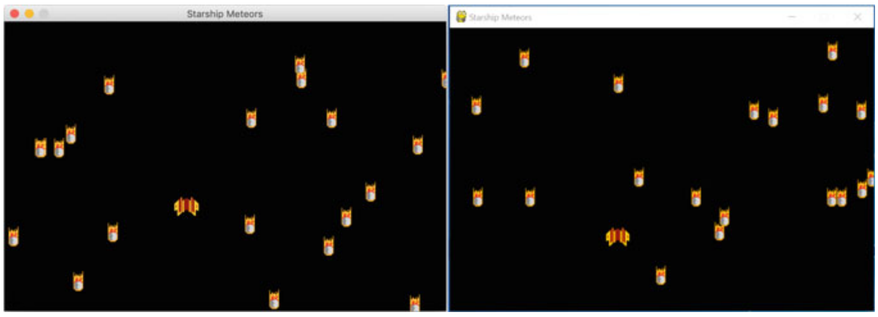
Chapter 13
StarshipMeteors pygame

13.1 Creating a Spaceship Game

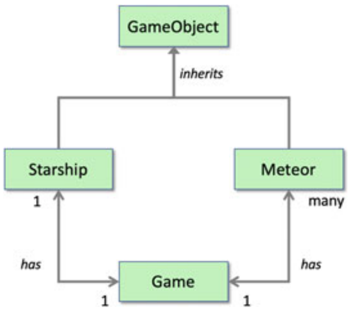
In this chapter we will create a game in which you pilot a starship through a field of meteors. The longer you play the game the larger the number of meteors you will encounter. A typical display from the game is shown below for a Apple Mac and a Windows PC:



We will implement several classes to represent the entities within the game. Using classes is not a required way to implement a game and it should be noted that many developers avoid the use of classes. However, using a class allows data associated with an object within the game to be maintained in one place; it also simplifies the creation of multiple instances of the same object (such as the meteors) within the game.

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_13

The classes and their relationships are shown below:



This diagram shows that the Starship and Meteor classes will extend a class called GameObject.

In turn it also shows that the Game has a 1:1 relationship with the Starship class. That is the Game holds a reference to one Starship and in turn the starship holds a single reference back to the Game.

In contrast the Game has a 1 to many relationship with the Meteor class. That is the Game object holds references to many Meteors and each Meteor holds a reference back to the single Game object.

13.2 The Main Game Class

The first class we will look at will be the Game class itself.

The Game class will hold the list of meteors and the starship as well as the main game playing loop.

It will also initialise the main window display (for example by setting the size and the caption of the window).

In this case we will store the display surface returned by the pygame.display.set_mode() function in an attribute of the Game object called display_surface. This is because we will need to use it later on to display the starship and the meteors.

We will also hold onto an instance of the pygame.time.Clock() class that we will use to set the frame rate each time round the main game playing while loop.

The basic framework of our game is shown below; this listing provides the basic Game class and the main method that will launch the game. The game also defines three global *constants* that will be used to define the frame refresh rate and the size of the display.

```
import pygame

# Set up Global 'constants'
FRAME_REFRESH_RATE = 30

DISPLAY_WIDTH = 600
DISPLAY_HEIGHT = 400


class Game:
    """ Represents the game itself and game playing
loop """

    def __init__(self):
        print('Initialising PyGame')
        pygame.init()
        # Set up the display
        self.display_surface =
pygame.display.set_mode((DISPLAY_WIDTH, DISPLAY_HEIGHT))
        pygame.display.set_caption('Starship Meteors')
        # Used for timing within the program.
        self.clock = pygame.time.Clock()

    def play(self):
        is_running = True

        # Main game playing Loop
        while is_running:
            # Work out what the user wants to do
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    is_running = False
                elif event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_q:
                        is_running = False
            # Update the display
            pygame.display.update()

            # Defines the frame rate
            self.clock.tick(FRAME_REFRESH_RATE)

        # Let pygame shutdown gracefully
        pygame.quit()

def main():
    print('Starting Game')
    game = Game()
    game.play()
    print('Game Over')


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

The main `play()` method of the `Game` class has a loop that will continue until the user selects to quit the game. They can do this in one of two ways, either by pressing the ‘q’ key (represented by the `event.key` K_q) or by clicking on the window close button. In either case these events are picked up in the main event processing for loop within the main `while` loop method.

If the user does not want to quit the game then the display is updated (refreshed) and then the `clock.tick()` (or frame) rate is set.

When the user selects to quit the game then the main while loop is terminated (the `is_running` flag is set to `False`) and the `pygame.quit()` method is called to shut down pygame.

At the moment this not a very interactive game as it does not do anything except allow the user to quit. In the next section we will add in behaviour that will allow us to display the space ship within the display.

13.3 The GameObject Class

The `GameObject` class defines three methods:

The `load_image()` method can be used to load an image to be used to visually represent the specific type of game object. The method then uses the width and height of the image to define the width and height of the game object.

The `rect()` method returns a rectangle representing the current area used by the game object on the underlying drawing surface. This differs from the images own `rect()` which is not related to the location of the game object on the underlying surface. Rects are very useful for comparing the location of one object with another (for example when determining if a collision has occurred).

The `draw()` method draws the GameObjects’ image onto the `display_-surface` held by the game using the GameObjects current `x` and `y` coordinates. It can be overridden by subclasses if they wish to be drawn in a different way.

The code for the `GameObject` class is presented below:

```
class GameObject:

    def load_image(self, filename):
        self.image = pygame.image.load(filename).convert()
        self.width = self.image.get_width()
        self.height = self.image.get_height()

    def rect(self):
        """ Generates a rectangle representing the objects
location
        and dimensions """
        return pygame.Rect(self.x, self.y, self.width,
self.height)

    def draw(self):
        """ draw the game object at the
            current x, y coordinates """
        self.game.display_surface.blit(self.image, (self.x,
self.y))
```

The `GameObject` class is directly extended by the `Starship` class and the `Meteor` class.

Currently there are only two types of game elements, the starship and the meteors; but this could be extended in future to planets, comets, shooting stars etc.

13.4 Displaying the Starship

The human player of this game will control a starship that can be moved around the display.

The Starship will be represented by an instance of the class `Starship`. This class will extend the `GameObject` class that holds common behaviours for any type of element that is represented within the game.

The Starship class defines its own `__init__()` method that takes a reference to the game that the starship is part of. This initialisation method sets the initial starting location of the Starship as half the width of the display for the x coordinate and the display height minus 40 for the y coordinate (this gives a bit of a buffer before the end of the screen). It then uses the `load_image()` method from the `GameObject` parent class to load the image to be used to represent the `Starship`. This is held in a file called `starship.png`. For the moment we will leave the `Starship` class as it is (however we will return to this class so that we can make it into a movable object in the next section).

The current version of the `Starship` class is given below:

```
class Starship(GameObject):
    """ Represents a starship"""

    def __init__(self, game):
        self.game = game
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')
```

In the Game class we will now add a line to the `__init__()` method to initialise the Starship object. This line is:

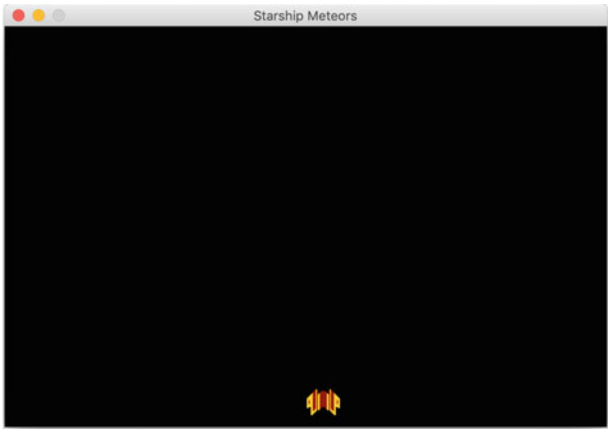
```
# Set up the starship
self.starship = Starship(self)
```

We will also add a line to the main while loop within the `play()` method just before we refresh the display. This line will call the `draw()` method on the starship object:

```
# Draw the starship
self.starship.draw()
```

This will have the effect of drawing the starship onto the windows drawing surface in the background before the display is refreshed.

When we now run this version of the StarshipMeteor game we now see the Starship in the display:



Of course at the moment the starship does not move; but we will address that in the next section.

13.5 Moving the Spaceship

We want to be able to move the Starship about within the bounds of the display screen.

To do this we need to change the starships x and y coordinates in response to the user pressing various keys.

We will use the arrow keys to move up and down the screen or to the left or right of the screen. To do this we will define *four* methods within the Starship class; these methods will move the starship up, down, left and right etc.

The updated `Starship` class is shown below:

```
class Starship(GameObject):
    """ Represents a starship"""

    def __init__(self, game):
        self.game = game
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')

    def move_right(self):
        """ moves the starship right across the screen """
        self.x = self.x + STARSHIP_SPEED
        if self.x + self.width > DISPLAY_WIDTH:
        self.x = DISPLAY_WIDTH - self.width

  def move_left(self):
      """ Move the starship left across the screen """
      self.x = self.x - STARSHIP_SPEED
      if self.x < 0:
          self.x = 0

  def move_up(self):
      """ Move the starship up the screen """
      self.y = self.y - STARSHIP_SPEED
      if self.y < 0:
          self.y = 0

  def move_down(self):
      """ Move the starship down the screen """
      self.y = self.y + STARSHIP_SPEED
      if self.y + self.height > DISPLAY_HEIGHT:
          self.y = DISPLAY_HEIGHT - self.height

  def __str__(self):
      return 'Starship(' + str(self.x) + ', ' + str(self.y)
+ ')'
```

This version of the Starship class defines the various move methods. These methods use a new *global* value `STARSHIP_SPEED` to determine how far and how fast the Starship moves. If you want to change the speed that the Starship moves then you can change this global value.

Depending upon the direction intended we will need to modify either the x or y coordinate of the Starship.

- If the starship moves to the left then the `x` coordinate is reduced by `STARSHIP_SPEED`,
- if it moves to the right then the `x` coordinate is increased by `STARSHIP_SPEED`,
- in turn if the Starship moves up the screen then the `y` coordinate is decremented by `STARSHIP_SPEED`,

- but if it moves down the screen then the y coordinate is increased by STARSHIP_SPEED.

Of course we do not want our Starship to fly off the edge of the screen and so a test must be made to see if it has reached the boundaries of the screen. Thus tests are made to see if the x or y values have gone below Zero or above the DISPLAY_WIDTH or DISPLAY_HEIGHT values. If any of these conditions are met then the x or y values are reset to an appropriate default.

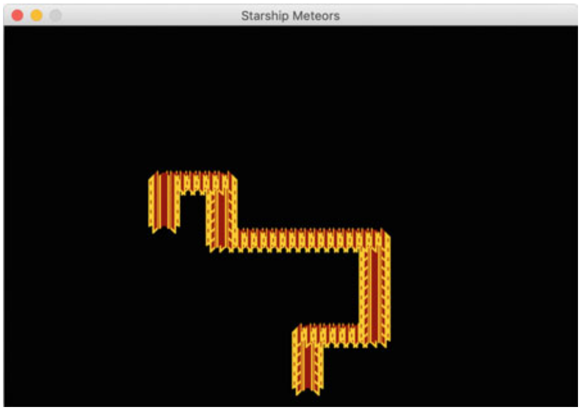
We can now use these methods with player input. This player input will indicate the direction that the player wants to move the Starship. As we are using the left, right, up and down arrow keys for this we can extend the event processing loop that we have already defined for the main game playing loop. As with the letter q, the event keys are prefixed by the letter K and an underbar, but this time the keys are named K_LEFT, K_RIGHT, K_UP and K_DOWN.

When one of these keys is pressed then we will call the appropriate move method on the starship object already held by the Game object.

The main event processing for loop is now:

```
# Work out what the user wants to do
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        is_running = False
    elif event.type == pygame.KEYDOWN:
        # Check to see which key is pressed
        if event.key == pygame.K_RIGHT:
            # Right arrow key has been pressed
            # move the player right
            self.starship.move_right()
        elif event.key == pygame.K_LEFT:
            # Left arrow has been pressed
            # move the player left
            self.starship.move_left()
        elif event.key == pygame.K_UP:
            self.starship.move_up()
        elif event.key == pygame.K_DOWN:
            self.starship.move_down()
        elif event.key == pygame.K_q:
            is_running = False
```

However, we are not quite finished. If we try and run this version of the program we will get a trail of Starships drawn across the screen; for example:



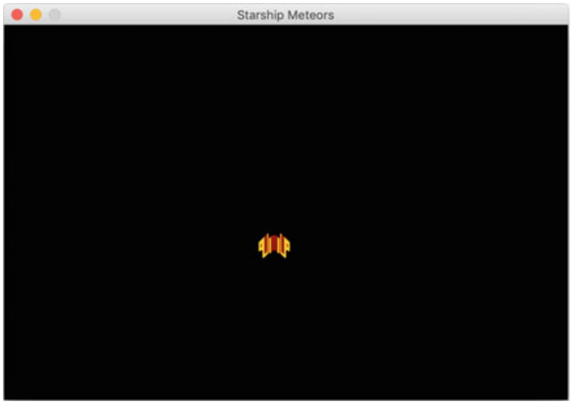
The problem is that we are redrawing the starship at a different position; but the previous image is still present.

We now have two choices one is to merely fill the whole screen with black; effectively hiding anything that has been drawn so far; or alternatively we could just draw over the area used by the previous image position. Which approach is adopted depends on the particular scenario represented by your game. As we will have a lot of meteors on the screen once we have added them; the easiest option is to overwrite everything on the screen before redrawing the starship. We will therefore add the following line:

```
# Clear the screen of current contents
self.display_surface.fill(BACKGROUND)
```

This line is added just before we draw the Starship within the main game playing `while` loop.

Now when we move the Starship the old image is removed before we draw the new image:



One point to note is that we have also defined another global value `BACKGROUND` used to hold the background colour of the game playing surface. This is set to black as shown below:

```
# Define default RGB colours
BACKGROUND = (0, 0, 0)
```

If you want to use a different background colour then change this global value.

13.6 Adding a Meteor Class

The `Meteor` class will also be a subclass of the `GameObject` class. However, it will only provide a `move_down()` method rather than the variety of move methods of the `Starship`.

It will also need to have a random starting x coordinate so that when a meteor is added to the game its starting position will vary. This random position can be generated using the `random.randint()` function using a value between 0 and the width of the drawing surface. The meteor will also start at the top of the screen so will have a different initial y coordinate to the Starship. Finally, we also want our meteors to have different speeds; this can be another random number between 1 and some specified maximum meteor speed.

To support these we need to add `random` to the modules being imported and define several new global values, for example:

```
import pygame, random

INITIAL_METEOR_Y_LOCATION = 10
MAX_METEOR_SPEED = 5
```

We can now define the `Meteor` class:

```
class Meteor(GameObject):
    """ represents a meteor in the game """

    def __init__(self, game):
        self.game = game
        self.x = random.randint(0, DISPLAY_WIDTH)
        self.y = INITIAL_METEOR_Y_LOCATION
        self.speed = random.randint(1, MAX_METEOR_SPEED)
        self.load_image('meteor.png')

    def move_down(self):
        """ Move the meteor down the screen """
        self.y = self.y + self.speed
        if self.y > DISPLAY_HEIGHT:
            self.y = 5

    def __str__(self):
        return 'Meteor(' + str(self.x) + ', ' + str(self.y) +
')'
```

The `__init__()` method for the `Meteor` class has the same steps as the Starship; the difference is that the `x` coordinate and the `speed` are randomly generated. The image used for the Meteor is also different as it is 'meteor.png'.

We have also implemented a `move_down()` method. This is essentially the same as the Starships `move_down()`.

Note that at this point we could create a subclass of `GameObject` called `MoveableGameObject` (which extends `GameObject`) and push the move operations up into that class and have the `Meteor` and `Starship` classes extend that class. However we don't really want to allow meteors to move just anywhere on the screen.

We can now add the meteors to the `Game` class. We will add a new global value to indicate the number of initial meteors in the game:

```
INITIAL_NUMBER_OF_METEORS = 8
```

Next we will initialise a new attribute for the `Game` class that will hold a list of Meteors. We will use a list here as we want to increase the number of meteors as the game progresses.

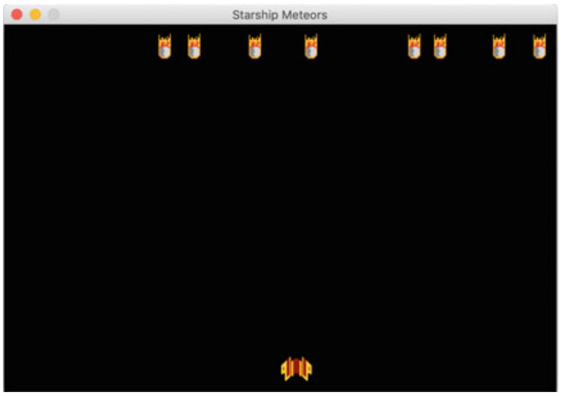
To make this process easy we will use a list comprehension which allows a `for` loop to run with the results of an expression captured by the list:

```
# Set up meteors
self.meteors = [Meteor(self) for _ in range(0,
INITIAL_NUMBER_OF_METEORS)]
```

We now have a list of meteors that need to be displayed. We thus need to update the `while` loop of the `play()` method to draw not only the starship but also all the meteors:

```
# Draw the meteors and the starship
self.starship.draw()
for meteor in self.meteors:
    meteor.draw()
```

The end result is that a set of meteor objects are created at random starting locations across the top of the screen:



13.7 Moving the Meteors

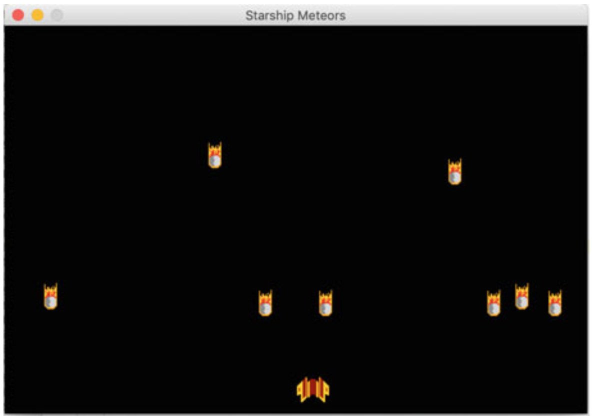
We now want to be able to move the meteors down the screen so that the Starship has some objects to avoid.

We can do this very easily as we have already implemented a `move_down()` method in the `Meteor` class. We therefore only need to add a for loop to the main game playing `while` loop that will move all the meteors. For example:

```
# Move the Meteors
for meteor in self.meteors:
    meteor.move_down()
```

This can be added after the event processing `for` loop and before the screen is refreshed/redrawn or updated.

Now when we run the game the meteors move and the player can navigate the Starship between the falling meteors.



13.8 Identifying a Collision

At the moment the game will play for ever as there is no end state and no attempt to identify if a Starship has collided with a meteor.

We can add Meteor/Starship collision detection using PyGame Rects. As mentioned in the last chapter a `Rect` is a PyGame class used to represent rectangular coordinates. It is particularly useful as the `pygame.Rect` class provides several collision detection methods that can be used to test if one rectangle (or point) is inside another rectangle. We can therefore use one of the methods to test if the rectangle around the Starship intersects with any of the rectangles around the Meteors.

The `GameObject` class already provides a method `rect()` that will return a `Rect` object representing the objects' current rectangle with respect to the drawing surface (essentially the box around the object representing its location on the screen).

Thus we can write a collision detection method for the `Game` class using the `GameObject` generated rects and the `Rect` class `colliderect()` method:

```
def _check_for_collision(self):
    """ Checks to see if any of the meteors have collided with
    the starship """
        result = False
    for meteor in self.meteors:
        if self.starship.rect().colliderect(meteor.rect()):
        result = True
        break
    return result
```

Note that we have followed the convention here of preceding the method name with an underbar indicating that this method should be considered private to the class. It should therefore never be called by anything outside of the `Game` class. This convention is defined in PEP 8 (Python Enhancement Proposal) but is not enforced by the language.

We can now use this method in the main `while` loop of the game to check for a collision:

```
# Check to see if a meteor has hit the ship
if self._check_for_collision():
     starship_collided = True
```

This code snippet also introduces a new local variable `starship_collided`. We will initially set this to `False` and is another condition under which the main game playing `while` loop will terminate:

```
is_running = True
starship_collided = False

# Main game playing Loop
while is_running and not starship_collided:
```

Thus the game playing loop will terminate if the user selects to quit or if the starship collides with a meteor.

13.9 Identifying a Win

We currently have a way to loose the game but we don't have a way to win the game! However, we want the player to be able to win the game by surviving for a specified period of time. We could represent this with a timer of some sort. However, in our case we will represent it as a specific number of cycles of the main game playing loop. If the player survives for this number of cycles then they have won. For example:

```
# See if the player has won
if cycle_count == MAX_NUMBER_OF_CYCLES:
    print('WINNER!')
    break
```

In this case a message is printed out stating that the player won and then the main game playing loop is terminated (using the `break` statement).

The `MAX_NUMBER_OF_CYCLES` global value can be set as appropriate, for example:

```
MAX_NUMBER_OF_CYCLES = 1000
```

13.10 Increasing the Number of Meteors

We could leave the game as it is at this point, as it is now possible to win or loose the game. However, there are a few things that can be easily added that will enhance the game playing experience. One of these is to increase the number of Meteors on the screen making it harder as the game progresses.

We can do this using a `NEW_METEOR_CYCLE_INTERVAL`.

```
NEW_METEOR_CYCLE_INTERVAL = 40
```

When this interval is reached we can add a new `Meteor` to the list of current Meteors; it will then be automatically drawn by the `Game` class. For example:

```
# Determine if new meteors should be added
if cycle_count % NEW_METEOR_CYCLE_INTERVAL == 0:
    self.meteors.append(Meteor(self))
```

Now every NEW_METEOR_CYCLE_INTERVAL another meteor will be added at a random x coordinate to the game.

13.11 Pausing the Game

Another feature that many games have is the ability to pause the game. This can be easily added by monitoring for a pause key (this could be the letter p represented by the event_key pygame.K_p). When this is pressed the game could be paused until the key is pressed again.

The pause operation can be implemented as a method _pause() that will consume all events until the appropriate key is pressed. For example:

```
def _pause(self):
     paused = True
    while paused:
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_p:
                    paused = False
                    break
```

In this method the outer while loop will loop until the paused local variable is set too False. This only happens when the ‘p’ key is pressed. The break after the statement setting paused to False ensures that the inner for loop is terminated allowing the outer while loop to check the value of paused and terminate.

The _pause() method can be invoked during the game playing cycle by monitoring for the ‘p’ key within the event for loop and calling the _pause() method from there:

```
elif event.key == pygame.K_p:
    self._pause()
```

Note that again we have indicated that we don’t expect the _pause() method to be called from outside the game by prefixing the method name with an underbar (‘_’).

13.12 Displaying the Game Over Message

PyGame does not come with an easy way of creating a popup dialog box to display messages such as 'You Won'; or 'You Lost' which is why we have used print statements so far. However, we could use a GUI framework such as wxPython to do this or we could display a message on the display surface to indicate whether the player has won or lost.

We can display a message on the display surface using the `pygame.font.Font` class. This can be used to create a `Font` object that can be rendered onto a surface that can be displayed onto the main display surface.

We can therefore add a method `_display_message()` to the Game class that can be used to display appropriate messages:

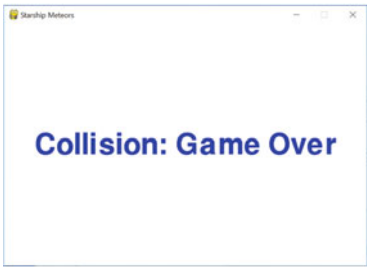
```
def _display_message(self, message):
    """ Displays a message to the user on the screen """
    print(message)
    text_font = pygame.font.Font('freesansbold.ttf', 48)
    text_surface = text_font.render(message, True, BLUE, WHITE)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center = (DISPLAY_WIDTH / 2,
DISPLAY_HEIGHT / 2)
    self.display_surface.fill(WHITE)
    self.display_surface.blit(text_surface, text_rectangle)
```

Again the leading underbar in the method name indicates that it should not be called from outside the Game class.

We can now modify the main loop such that appropriate messages are displayed to the user, for example:

```
# Check to see if a meteor has hit the ship
if self._check_for_collision():
    starship_collided = True
    self._display_message('Collision: Game Over')
```

The result of the above code being run when a collision occurs is shown below:



Collision: Game Over

13.13 The StarshipMeteors Game

The complete listing for the final version of the StarshipMeteors game is given below:

```
import pygame, random, time

FRAME_REFRESH_RATE = 30

DISPLAY_WIDTH = 600
DISPLAY_HEIGHT = 400
WHITE = (255, 255, 255)
BACKGROUND = (0, 0, 0)

INITIAL_METEOR_Y_LOCATION = 10
INITIAL_NUMBER_OF_METEORS = 8
MAX_METEOR_SPEED = 5
STARSHIP_SPEED = 10
MAX_NUMBER_OF_CYCLES = 1000
NEW_METEOR_CYCLE_INTERVAL = 40
```

```
class GameObject:

    def load_image(self, filename):
        self.image = pygame.image.load(filename).convert()
        self.width = self.image.get_width()
        self.height = self.image.get_height()

    def rect(self):
        """ Generates a rectangle representing the objects
location
         and dimensions """
        return pygame.Rect(self.x, self.y, self.width,
self.height)

    def draw(self):
        """ draw the game object at the
            current x, y coordinates """
        self.game.display_surface.blit(self.image, (self.x,
self.y))

class Starship(GameObject):
    """ Represents a starship"""

    def __init__(self, game):
        self.game = game
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')

    def move_right(self):
        """ moves the starship right across the screen """
        self.x = self.x + STARSHIP_SPEED
        if self.x + self.width > DISPLAY_WIDTH:
            self.x = DISPLAY_WIDTH - self.width

    def move_left(self):
        """ Move the starship left across the screen """
        self.x = self.x - STARSHIP_SPEED
        if self.x < 0:
            self.x = 0

    def move_up(self):
        """ Move the starship up the screen """
        self.y = self.y - STARSHIP_SPEED
        if self.y < 0:
            self.y = 0

    def move_down(self):
        """ Move the starship down the screen """
        self.y = self.y + STARSHIP_SPEED
        if self.y + self.height > DISPLAY_HEIGHT:
            self.y = DISPLAY_HEIGHT - self.height

    def __str__(self):
        return 'Starship(' + str(self.x) + ', ' + str(self.y) +
')'

class Meteor(GameObject):
    """ represents a meteor in the game """

    def __init__(self, game):
        self.game = game
        self.x = random.randint(0, DISPLAY_WIDTH)
        self.y = INITIAL_METEOR_Y_LOCATION
        self.speed = random.randint(1, MAX_METEOR_SPEED)
        self.load_image('meteor.png')

    def move_down(self):
        """ Move the meteor down the screen """
        self.y = self.y + self.speed
        if self.y > DISPLAY_HEIGHT:
            self.y = 5

    def __str__(self):
        return 'Meteor(' + str(self.x) + ', ' + str(self.y) +
')'

class Game:
    """ Represents the game itself, holds the main game playing
loop """

    def __init__(self):
        pygame.init()
        # Set up the display
        self.display_surface =
pygame.display.set_mode((DISPLAY_WIDTH, DISPLAY_HEIGHT))
        pygame.display.set_caption('Starship Meteors')
        # Used for timing within the program.
        self.clock = pygame.time.Clock()
        # Set up the starship
        self.starship = Starship(self)
        # Set up meteors
        self.meteors = [Meteor(self) for _ in range(0,
INITIAL_NUMBER_OF_METEORS)]

    def _check_for_collision(self):
        """ Checks to see if any of the meteors have collided
with the starship """
        result = False
        for meteor in self.meteors:
            if self.starship.rect().colliderect(meteor.rect()):
                result = True
                break
        return result

    def _display_message(self, message):
        """ Displays a message to the user on the screen """
        text_font = pygame.font.Font('freesansbold.ttf', 48)
        text_surface = text_font.render(message, True, BLUE,
WHITE)
        text_rectangle = text_surface.get_rect()
        text_rectangle.center = (DISPLAY_WIDTH / 2,
DISPLAY_HEIGHT / 2)
        self.display_surface.fill(WHITE)
        self.display_surface.blit(text_surface, text_rectangle)

    def _pause(self):
        paused = True
        while paused:
            for event in pygame.event.get():
                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_p:
                        paused = False
                        break

    def play(self):
        is_running = True
        starship_collided = False
        cycle_count = 0

        # Main game playing Loop
        while is_running and not starship_collided:
            # Indicates how many times the main game loop has
been run
            cycle_count += 1

            # See if the player has won
            if cycle_count == MAX_NUMBER_OF_CYCLES:
                self._display_message('WINNER!')
                break

            # Work out what the user wants to do
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    is_running = False
                elif event.type == pygame.KEYDOWN:
                    # Check to see which key is pressed
                    if event.key == pygame.K_RIGHT:
                        # Right arrow key has been pressed
                        # move the player right
                        self.starship.move_right()
                    elif event.key == pygame.K_LEFT:
                        # Left arrow has been pressed
                        # move the player left
                        self.starship.move_left()
                    elif event.key == pygame.K_UP:
                        self.starship.move_up()
                    elif event.key == pygame.K_DOWN:
                        self.starship.move_down()
                    elif event.key == pygame.K_p:
                        self._pause()
                    elif event.key == pygame.K_q:
                        is_running = False

            # Move the Meteors
            for meteor in self.meteors:
                meteor.move_down()

            # Clear the screen of current contents
            self.display_surface.fill(BACKGROUND)

            # Draw the meteors and the starship
            self.starship.draw()
            for meteor in self.meteors:
                meteor.draw()

            # Check to see if a meteor has hit the ship
            if self._check_for_collision():
                starship_collided = True
                self._display_message('Collision: Game Over')

            # Determine if new mateors should be added
            if cycle_count % NEW_METEOR_CYCLE_INTERVAL == 0:
                self.meteors.append(Meteor(self))

            # Update the display
            pygame.display.update()

            # Defines the frame rate. The number is number of
frames per
            # second. Should be called once per frame (but only
once)
            self.clock.tick(FRAME_REFRESH_RATE)

        time.sleep(1)
        # Let pygame shutdown gracefully
        pygame.quit()

def main():
    print('Starting Game')
    game = Game()
    game.play()
    print('Game Over')
    if __name__ == '__main__':
        main()
```

13.14 Online Resources

There is a great deal of information available on PyGame including:

- https://www.pygame.org The PyGame home page.
- https://www.pygame.org/docs/tut/PygameIntro.html PyGame tutorial.
- https://www.python.org/dev/peps/pep-0008/ PEP8 Style Guide for Python Code.

13.15 Exercises

Using the example presented in this chapter add the following:

- Provide a score counter. This could be based on the number of cycles the player survives or the number of meteors that restart from the top of the screen etc.
- Add another type of `GameObject`, this could be a shooting star that moves across the screen horizontally; perhaps using an random starting y coordinate.
- Allow the game difficulty to be specified at the start. This could affect the number of initial meteors, the maximum speed of a meteor, the number of shooting stars etc.