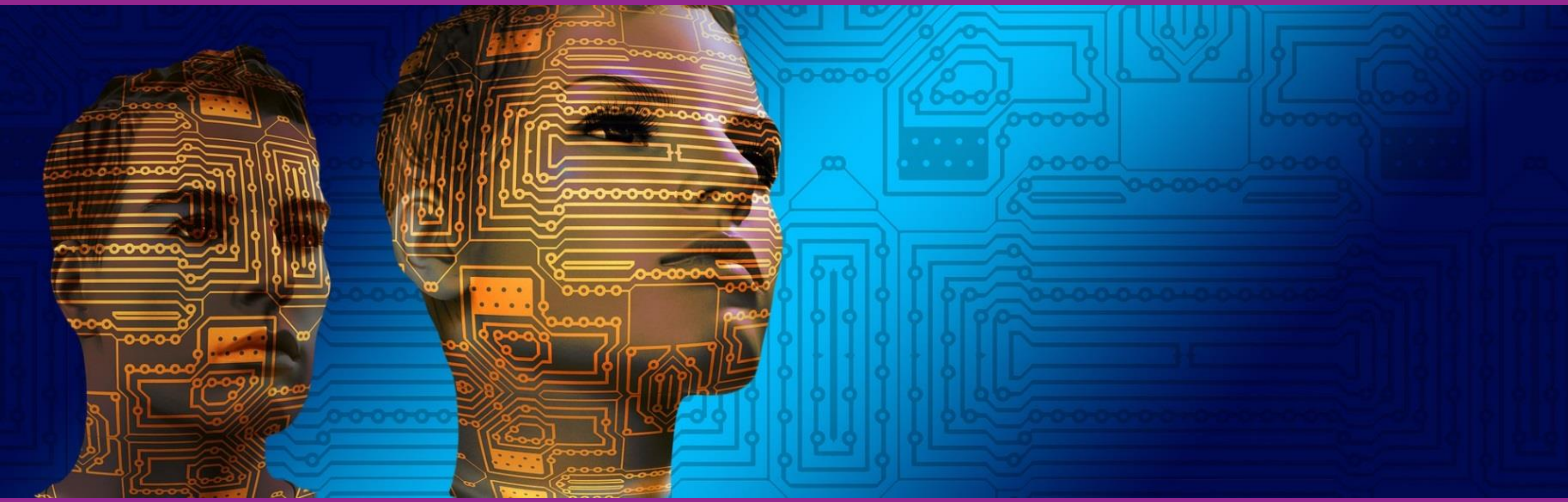
Learn Informatics

Control and Programming

Module 9

Learner Support Booklet



Version 3

Contents

Overview

This module follows on from Module 4. You will need to have completed some of the basic coding courses and have a working knowledge of Python together with an understanding of how to use an IDE and a put together algorithms.

Skills Statements

When writing code, I can

- ✓ Write a sequence of instructions
- ✓ Use the selection construct
- ✓ Use the iteration construct
- ✓ Use variables with the assignment operator
- ✓ Use Boolean expressions and operators
- ✓ Call functions and/or procedures
- ✓ Define functions and/or procedures
- ✓ Use a graphics based programming IDE
- ✓ Use a text based programming IDE
- ✓ Debug a program and correct errors

By engaging in computational thinking, I can:

- ✓ Decompose a problem
- ✓ Identify patterns to solve problems
- ✓ Use abstraction to solve a problem
- ✓ Test a hypothesis
- ✓ Understand how a flowchart works
- ✓ Produce an algorithm
- ✓ Trace through an algorithm

Required Resources and Delivery

Stage A - Pygame

PyGame Start-up

The first aspect of any programming language is to be able to create, edit and run files in your chosen language. We will be using IDLE as our design environment. Our files will end in the extension *.py*.

You will need to install Python and then the PyGame library. Here is a video which show you what to do in Windows 10. In summary:

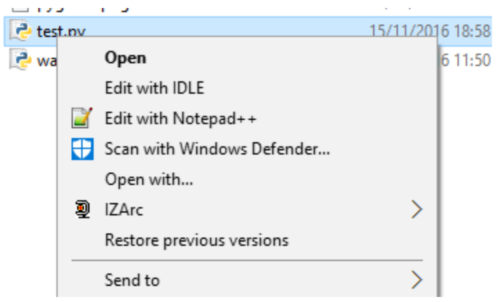
1. Download and install Python 3.4 or above (I have used Python 3.6.5) – during the install select the Path option (not done by default)
2. Then at the prompt type: **`pip install pygame`**

Note: If you get a pip out of date error message type: ***`python -m pip install --upgrade pip`***

IDLE, python and pygame should now be installed.

Step 1: Create a folder for your PyGame files and then create a new text file and call it *startUp.py*. Note that you will have changed the file's extension to **.py** and possibly select the windows show extensions option.

Step 2: Right click on it and select the **Edit with IDLE**.



Step 3: Once the file is open type the following code into the Interface.

```
import pygame
# -- Global Constants

# -- Colours
BLACK = (0,0,0)
WHITE = (255,255,255)
BLUE = (50,50,255)
YELLOW = (255,255,0)

# -- Initialise PyGame
pygame.init()

# -- Blank Screen
size = (640,480)
```

```
screen = pygame.display.set_mode(size)
# -- Title of new window/screen
pygame.display.set_caption("My Window")
# -- Exit game flag set to false
done = False
# -- Manages how fast screen refreshes
clock = pygame.time.Clock()
```

Step 4: Now press F5 to run it. You should save it first but if you haven't don't worry as you will be prompted to do that anyway! It should provide a blank screen in a new window and the Window Title should be "My Window".

Step 5: You will notice that this program has "hung" as there is no exit – so close the window to "crash" out.

Pygame Loop

Step 1: PyGame runs on one continuous loop which is executed 60 times a second. It also has an event which allows the program to quit and this is of course very useful. You can also put comments in the code using a hash tag (#) this will be vital as your programs get larger and more complex.

Step 2: Open your *startUp.py* file and type the code below into the file after the code you wrote in the previous task.

```
### -- Game Loop

while not done:

    # -- User input and controls

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            done = True

        #End If

    #Next event

    # -- Game logic goes after this comment

    # -- Screen background is BLACK

    screen.fill (BLACK)

    # -- Draw here

    pygame.draw.rect(screen, BLUE, (220,165,200,150))

    pygame.draw.circle(screen, YELLOW, (40,100),40,0)

    # -- flip display to reveal new position of objects

    pygame.display.flip()

    # - The clock ticks over

    clock.tick(60)

#End While - End of game loop

pygame.quit()
```

Step 3: Run this code and check you get a blue rectangle and a yellow circle on a black background.

Step 4: Watch the video for this task and it will explain the above code more thoroughly and how it works.

Step 5: Save you file as *template.py* and mark the above section of code using the marking engine.

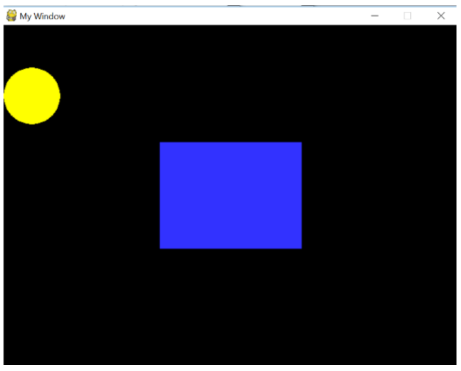
Extension Activities

When considering these activities, you find the this site useful as it contains some good tutorials.

Parameters:

The drawing functions are given below. Discuss with a partner what you think the parameters of the `draw.rect` function does. For example, what does the parameter BLUE do and then what the values 220 165, 200, 150 refer to? If you are unsure change them and see what the difference is. Also note where the brackets are – why do you think they are in this format?

```
pygame.draw.rect(screen, BLUE, (220, 165, 200, 150))

pygame.draw.circle(screen, YELLOW, (40,100), 40, 0)
```



- *Try changing the colour of the shapes and removing the fill.*
- *Try making the circle bigger*
- *Can you make the rectangle into a square?*
- *Try changing the circle parameter which is currently zero to 2 and see what happens.*
- *Try putting the rectangle and the circle at different positions on the screen.*
- *Note the use of the optional highlighted parameter zero in the circle method which makes the sun filled in. It is optional in the rectangle method and leaving it out had the same effect.*

Cartesian Co-ordinates?

When you change the rectangle position co-ordinates what do you notice? *How does this differ from the way you draw co-ordinates on a graph in your Maths lessons?* When you start to move objects on the screen this will become very important!

Colours

- Define a new colour. Think about where the code for this goes and what the three values are. (Hint: RGB). This site may help you http://www.colorpicker.com/

Lines

- Use your new colour to draw a triangle on the screen using the `pygame.draw.line` and `pygame.draw.lines` methods. Find out the difference between the two.
- Now draw a triangle using the `pygame.draw.polygon` method.

House of the Rising Sun

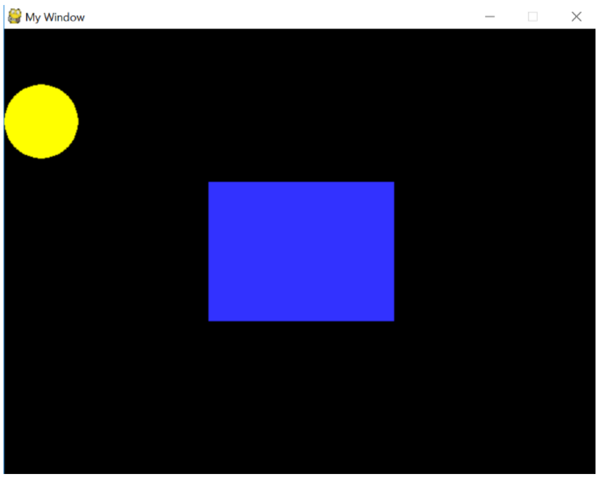
Step 1: Open *template.py* but this time draw a house using the rectangle function as below. Save your file as *house.py*

```
pygame.draw.rect(screen, BLUE, (220,165,200,150))
```

Step 2: Now draw the sun as shown below:

```
pygame.draw.circle(screen, YELLOW, (40, 100), 40, 0)
```

Step 3: The house and sun should look like the screen shot with the caption **My Window** changed to **House.**



Step 4: Clearly the sun should move across the sky rather than being static and stuck on the left-hand side. So, we need two declare variables and to introduce some game logic. Define two variables in your code after `done = False` as follows:

```
done = False

sun_x = 40

sun_y = 100
```

Step 5: These variables will define the original position of the sun. So, you need to change the circle method parameters in order to use them as follows:

```
game.draw.circle(screen, YELLOW, (sun_x, sun_y), 40, 0)
```

Step 6: In the game logic you can now increase the value of the variable `sun_x` by 5 pixels each time the game loop is executed:

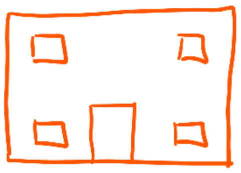
```
sun_x = sun_x + 5
```

Step 7: Test your code and make sure the sun moves in a straight line across the screen from left to right.

Step 8: Mark your code using the marking engine.

Further Activities

- Add more code so that when the sun disappears of the right-hand side of the code it "rises" back on the left-hand side. (Hint: You will need an if statement in your game logic to reset the value of sun_x)
- To improve your handling of graphics try making the house more realistic by adding windows and a door (see below). If you feel very adventurous you might want to add brickwork (use a loop and the line method) and, maybe a roof and chimney.



- Try making the background go dark blue when the sun has set, add a time lag so that it takes a while before it rises again and then make the background light blue when it the sun rises again.
- Obviously, the sun doesn't move in a straight line this but in more of an arc across the screen. For advanced mathematicians, get the sun to "rise" at co-ordinate (0, 100) and "set" at (640, 100) but arc across the screen using a quadratic curve. You will need to alter the **sun_y** value as well as the **sun_x** co-ordinate for this!

Pong

Step 1: Open the *template.py* file and save it as a file called *pong.py.* Change the window caption to "Pong" and change the rectangle so it is a 20 x 20-pixel square on the screen at co-ordinates (150, 200). You should also remove the yellow circle.

Step 2: Declare a variable before the game loop called `ball_width` and set it to 20. Now change drawing of the rectangle so that it uses the variable `ball_width.`

```
ball_width = 20
```

Step 3: Declare two variables <u>before</u> the game loop as shown below. These are the starting co-ordinates of the rectangle so change the `draw.rect` method within the loop to include these variables in the parameter list.

```
x_val = 150
y_val = 200
```

Step 4: To make the square move diagonally you will need to change both `x_val` and `y_val` each time the loop is executed. In the game logic area use the code:

```
x_val = x_val + 1
y_val = y_val + 1
```

Step 5: The direction of the ball is defined by the fact we are adding 1 to x and 1 to y each time we move round the loop. In order to control the direction we need to declare two variables `x_direction` and `y_direction` and initialise them both to 1 before the game loop. You can now alter the code above so the `x_val` and `y_val` are changed by their respective direction as below:

```
x_val = x_val + x_direction
y_val = y_val + y_direction
```

Step 6: Save your file and mark your code with the marking engine.

<u>**Next Activity (unmarked but essential!)**</u>

Step 7: Save your file as *pong_1.py*. Currently the ball will just disappear off the edge of the screen, so we need to get it to bounce off the walls when it hits them. By using selection (the IF statement) and changing the direction you can make the square "bounce" when it gets to the edge of the screen. You will need to think about the co-ordinates at the point the ball hits the edge (these will form the condition of the IF statement) and then decide which offset to change and how[1] it will change.

Step 8: Once you have it working for one edge try making it work for all four edges, so the ball just bounces around.

Step 9: Save your file still as *pong_1.py* when you have this working.

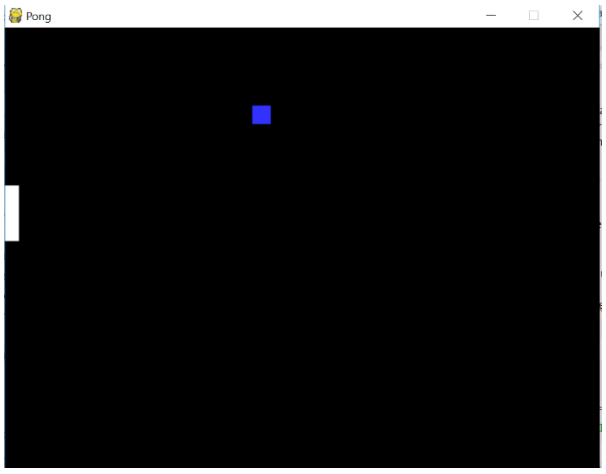
[1] A handy trick here is that multiplying a number by -1 you can change it from positive to negative. So to change the offset from +1 to -1 multiply it by -1. Doing this again reverse it back to +1 (-1*-1 = +1).

Adding a Paddle

Step 1: Open you file *pong_1.py* and save it as *pong_2.py*. The game of Pong requires a WHITE paddle on the left-hand side of the screen which is a rectangle with dimensions 15 by 60 pixels. Set two variables `padd_length` to 15 and `padd_width` to 60.

Step 2: Add two new variables `x_padd` and `y_padd` (position of the paddle) and initialise them 0 and 20 respectively.

Step 3: Now add the code in the game loop to draw the paddle and remember to use the variables you defined in steps 1 and step 2. Your screen should look as follows:



Step 3: You can add "up" and "down" user controls for the keyboard as follows:

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        done = True
    #End If
    if event.type == pygame.KEYDOWN:
       if event.key == pygame.K_UP:
          # - write logic that happens on key press here

       elif event.key == pygame.K_DOWN:
         # - write logic that happens on key press here
       #End If
    #End If
#Next event
```

This code goes in the user input section at the beginning of the loop and allows the user to press the UP key and DOWN key on the keyboard to control the paddle.

Step 3: Add logic to the above code which will move the paddle 5 pixels up and 5 pixels down when the relevant keys are pressed.

Step 4: Mark your code using the marking engine.

<u>**Next Activity (unmarked but essential!)**</u>

Save your file as *pong_3.py*. You will notice that if you press and hold down the UP key the paddle doesn’t respond it moves just 5 pixels and then is stationery. To allow for continuous movement you will need this code instead. Also add conditions so the paddle doesn’t move off the screen and save as *pong_3.py*.

```
keys = pygame.key.get_pressed()
    ## - the up key or down key has been pressed
    if keys[pygame.K_UP]:
        y_padd = y_padd - 5
    if keys[pygame.K_DOWN]:
        y_padd = y_padd + 5
```

Collisions!

Step 1: Open your file *pong_3.py* and save it as *pong_4.py*. When the ball hits the paddle it should bounce off it. Work out the conditions required to check if the ball has hit the paddle – there are three of them which will form the criteria for your if condition. Remember that y coordinates start with 0 at the top of the screen and increase as you go down and that the x and y co-ordinates of the rectangle indicate the top left-hand corner.

Step 2: If the conditions for a collision are met then the ball should bounce off the paddle just like it does if it hits the left-hand side of the screen; by changing the `x_direction`.

Step 3: Amend your code by adding further logic so that the ball will go off the screen if it gets past the paddle. If it does this, then reset the ball so that it returns to its original starting position and moves in the original direction.

Extension Activities

- Create a variable called score (initialised to 0). If the ball goes off the screen then add 1 to the score and reset it.
- Have a look on the Internet to see how to put text on the screen to show the score as the game progresses. Try here: http://programarcadegames.com/index.php?lang=en
- Think about the condition that will cause the game to quit. For example, the score gets to 10 (or you may want to count the score down from 10 and quit at zero – similar to "lives" in some arcade games.
- Increase the difficulty of the game by making the ball speed up after it has made contact with the paddle.

Sprites

Step 1: So far, we have had two objects on the screen a paddle and a ball. Look at the code you have a note how many variables we have had to define. Suppose that we wanted more than one of the same kind of object. To create 50 flakes of snow for example we would require at least 50 variables and we would need to write code to control each one. That's a lot of code and logic being repeated. So PyGame like many languages uses an Object-Oriented approach and this allows us to write a **Class.**

Step 2: Open up your *template.py* file and save it as *sprite.py*. Import two more libraries at the beginning of you code. The `random` library allows you to create random numbers and the `math` library will give you access to mathematical functions should you need them. These are imported at the very beginning of the code.

```
import pygame
import random
import math
```

Step 3: Change the title of the window to "Snow" and put the following code after the caption is set in order to define the class `Snow`. The video for this task will explain what each statement does. However, in short it creates a class `Snow` (which is a Sprite). A Sprite is an object (or surface area) on the screen and its shape is a rectangle the position of which is defined by the top left-hand corner `rect.x` and `rect.y`. PyGame has in-built functions for controlling sprites and these will be very useful for making our code readable and short. For now, just note that the class `Snow` has attributes `color`, `width` and `height`.

```
## -- Define the class snow which is a sprite
class Snow(pygame.sprite.Sprite):
    # Define the constructor for snow
    def __init__(self, color, width, height):
        # Call the sprite constructor
        super().__init__()
        # Create a sprite and fill it with colour
        self.image = pygame.Surface([width,height])
        self.image.fill(color)
        # Set the position of the sprite
        self.rect = self.image.get_rect()
        self.rect.x = random.randrange(0, 600)
        self.rect.y = random.randrange(0, 400)
    #End Procedure
#End Class
```

Step 4: So how do we use the class `Snow`. We can create as many snowflakes as we want. We put them in a group called `snow_group`. The code below will create two lists; one for the snowflakes and one for all the sprites we create in a game; at a later date we may have different classes such as enemy, player, bullet etc. SO put this code after the `done = False` statement.

```
# Create a list of the snow blocks
snow_group = pygame.sprite.Group()
```

```
# Create a list of all sprites
all_sprites_group = pygame.sprite.Group()
```

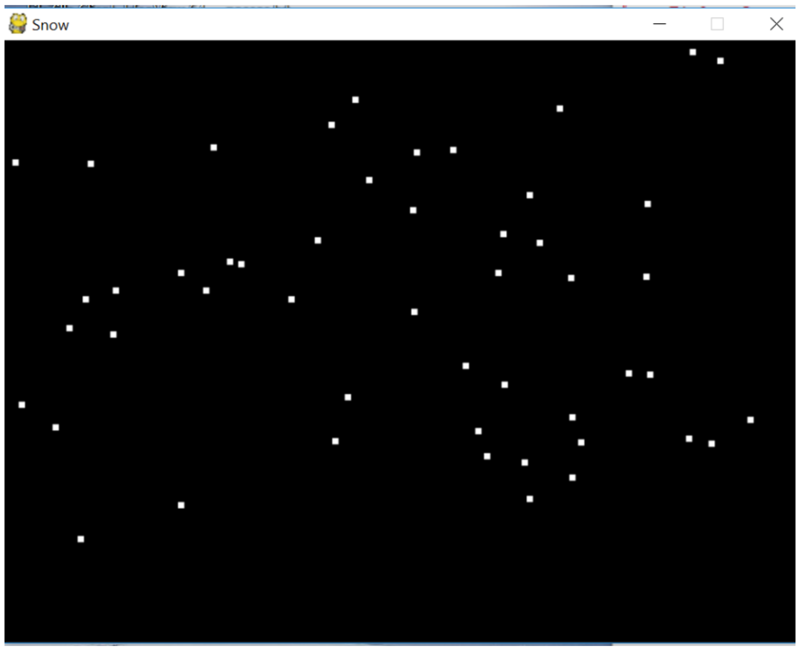
Step 5: Now we can create our snowflakes by using a loop (before the game loop) and after the sprite groups have been declared.

```
# Create the snowflakes
number_of_flakes = 50 # we are creating 50 snowflakes
for x in range (number_of_flakes):
    my_snow = snow(WHITE, 5, 5)  # snowflakes are white with size 5 by 5 px
    snow_group.add (my_snow) # adds the new snowflake to the group of snowflakes
    all_sprites_group.add (my_snow) # adds it to the group of all Sprites
#Next x
```

Step 6: The good news is that in order to draw all 50 snowflakes on the screen we can use the `all_sprites_group.draw` function. This one function (method) will draw ALL the snowflakes – so it really was worth creating the class and using sprites after all. This goes in the loop.

```
# - Screen background is BLACK
screen.fill (BLACK)
# -- Draw here
all_sprites_group.draw (screen)
```

Step 7: Mark the code using the marking engine. Here is a screen shot of what you should get when you run the code.



<u>Extension Activity</u>

You are not guaranteed to see all 50 snowflakes as you may find that two have been created in the same position. Can you solve this problem to make sure there are 50 snowflakes and that none are adjacent!

Let It Snow

Step 1: Obviously snow doesn't just hang in the air it falls! Open you file *sprite.py* and save it as *snow.py*. We need a way of moving our sprites. Good news! Pygame allows you to write a function for every class called `update().` Here is the update function for snow:

```
# Class update function - runs for each pass through the game loop
def update(self):
    self.rect.y = self.rect.y + self.speed
```

Step 2: You will need to add a new attribute to the class snow called `self.speed` and update the parameter list of snow so that you can set it when you create the flakes. Incidentally, making speed a parameter means you could have snow falling at different speed!

```
def __init__(self, color, width, height, speed):
    # Set speed of the sprite
    self.speed = speed
```

Step 3: Remember that snow now has an extra parameter which will need to be set for each snowflake when they are created:

```
my_snow = Snow(WHITE, 5, 5, 1)
```

Step 4: Now we can use the inbuilt `update()` function `all_sprites_group.update()` in the main loop where the logic of the program should be:

```
# -- Game logic goes in here
all_sprites_group.update()
# -- Screen background is BLACK
screen.fill(BLACK)
# -- Drawing code goes here
all_sprites_group.draw(screen)
```

The useful thing about this function is that in one call all 50 snowflakes are updated. As you will see in the next tasks this one function will update ALL objects even when there are many objects of different classes and makes life much easier than coding the movement of all objects with intertwining logic.

Extension Activity

Alter the update function of the class `Snow` so that each flake re-appears at the top of the screen once it has hit the ground so that it like the snow is continually falling. Save any changes to the file *snow_2.py*.

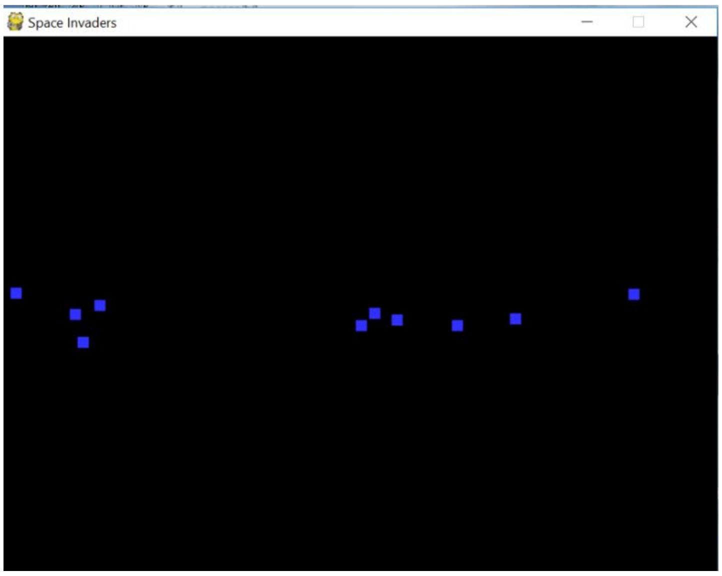
Space Invaders

One of the other useful features of using Sprites in PyGame is that it makes detecting collisions much easier. Remember how tricky it was to decide whether the paddle and the ball had collided in a previous task by working out the range of co-ordinates. Imagine how much more code you would need to decide whether 50 snowflakes had collided with themselves or another object! We will now modify the previous task so that we can make the retro game space invaders!

Step 1: Open your *snow.py* file and rename it as *invaders.py*.

Step 2: Change the code so that the class `Snow` is now the class `Invader`. Change the range of the random y co-ordinate so that it is a number between 0 and -50 (i.e. starts off the screen).

Step 3: Create a new class `Player`. This is the same as `Snow` except that it doesn't have a parameter `speed` in the constructor. It does however have the attribute speed which is set to zero within the class i.e. `self.speed = 0.` The player has starting position `(300, size[0] - height).` The update function is the same as for the class `Snow.`



Step 4: Change the `snow_group` to `invader_group.` Create 10 blue invaders (change number of snowflakes to number of invaders) which are 10 x 10 pixels and have speed 1. Be sure to put them in the invader group and the all sprite group.

Step 5: Using the class `Player` create an object `player`. This is a yellow rectangle with dimensions 10 pixels by 10 pixels. Make sure your add the player to the all sprites group.

Step 6: The player object is controlled by the user using the LEFT and RIGHT arrow keys. Here is the control code which you will see sets the "speed" of the player using the method `player_set_speed().`

```
# -- User inputs here
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        done = True
    elif event.type == pygame.KEYDOWN:  # - a key is down
        if event.key == pygame.K_LEFT: # - if the left key pressed
```

```
            player.player_set_speed(-3)  # speed set to -3
        elif event.key == pygame.K_RIGHT: # - if the right key pressed
            player.player_set_speed(3)  # speed set to 3
    elif event.type == pygame.KEYUP:  # - a key released
        if event.key == pygame.K_LEFT or event.key == pygame.K_RIGHT:
            player.player_set_speed(0)  # speed set to 0
```

Step 7: Notice that when the LEFT key is pressed the `Player` attribute `speed` is not directly changed. It is change using the method `player_set_speed(val)`. This method is defined in the class `Player` and it sets a new value for the attribute speed. This is called data encapsulation and ensures that attributes of objects are only changed using methods defined in the class.

Step 8: Define the `player_set_speed(val)` method in the `Player` class.

Step 9: Because we are calling a function and altering the attribute of an object there is no need to check that a key is held down so unlike in a previous task the movement when a key is held down is continuous and smooth – another useful side-effect of using Sprites.

Step 10: Here is some game logic which handles collisions between the player object and the invaders objects:

```
# Game logic goes in here
# -- when invader hits the player add 5 to score.
player_hit_group = pygame.sprite.spritecollide(player, invader_group, True)
```

Each time we loop through the code a group is created called `player_hit_group`. The `spritecollide()` function checks whether an object (`player`) has hit a group of objects (`invaders_group`). If it has then it adds the object to the list and because the last parameter is set to True it deletes that object from all the groups it belongs to.

Step 11: Save you file *invaders.py*. Mark your code.

Extension Activity (unmarked but compulsory)

Step 12: Save your file as *invaders_2.py*. By enhancing the code of the method `player_set_speed()` method ensure that the player cannot go off the screen to the left or the right.

Step 13: Add a new attribute to the `Player` class called `lives` which is initially set to 5. Using the `for` loop we find each object in `player_hit_group` and take a life off the number of lives the player has. Use the and you will need to display the overall number of lives in the top left-hand corner of the screen.

```
for foo in player_hit_group:
    player.lives = player.lives - 1
```

[The observant will notice Python shortcut which allows you to directly change the attribute of a class without writing a method– in a safety critical environment this would not be ideal!]

Step 14: Use a graphic package to produce a small graphic for your invaders (or download a free one from the Internet). Then add the image to the class. This makes the game much more realistic.

Ready, Aim, Fire

Step 1: Open your *invaders_2.py* file.

Step 2: Your final task is to add a new class `Bullet`. Here are the requirements for the bullet class:

- A bullet is a 2 by 2-pixel sprite. It has two attributes: color and speed.
- The constructor method takes the parameters color and the x and y co-ordinates of a bullet when an object of this class is instantiated. It sets the value of the attribute `speed` to 2 and the `height` and `width` of the surface of the sprite to 2.
- The `Bullet` class `update()` method changes the y-coordinate of the sprite by the value of the attribute speed on each iteration around the game loop.

Step 3: Add a `bullet_count` attribute to the `Player` class which is initially set to 50. This will decrease when a bullet is fired. Use the code `player.bullet_count = player.bullet_count – 1` although it would be better to define a `Player` method to do this.

Step 4: Create a new bullet each time the up arrow is pressed by the user. The bullet should be red (you will need to define the colour RED) and is fired from the top of the player sprite. Use the `bullet_group` variable to contain all the bullets created.

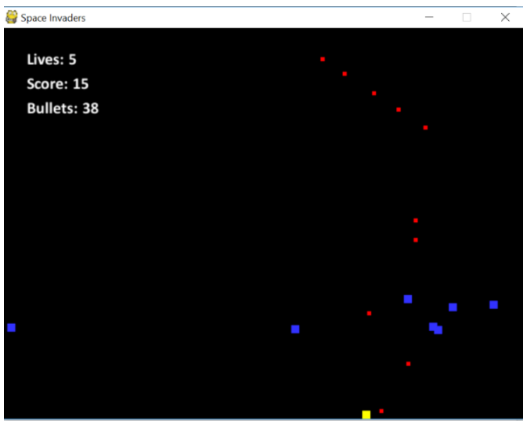
Step 5: Loop through the `bullet_group` and create a group called `bullet_hit_group` which contains those bullets that have collided with the `invader_group` – use the `spritecollide()` function you used in the previous task for this. This should delete those invaders that have been hit.

Step 6: Define a new attribute for player called `score` and initialise it to 0. Loop through the `bullet_hit_group` (you will need a loop within a loop) to add 5 onto the score each time an invader is hit.

Step 7: Mark your code. A snapshot of the game is shown below.

Extra Activities:

- You can try replacing the loop within a loop by using the `groupcollide()` function. Research how this works.
- Make sure the bullet count, score and number of lives are displayed on the screen.



Lives: 5
Score: 15
Bullets: 38

Maps

The traditional (retro) game PacMan (pictured below) requires walls! There are several ways of laying out games like this and the most common is using tiles with each tile being on abject (Sprite). To do this you will require a map.

Step 1: Create a 10 tile by 10 tile map. Below is an example where each value of 1 represents a tile on a wall and each 0 an empty space where the Pacman move to. The variable map is a list of 10 lists of values 1 and 0. You can use any letter or number to represent tiles. Here is an example map.

```
map = [[1,1,1,1,1,1,1,1,1,1],
       [1,0,0,0,0,0,0,0,0,1],
       [1,0,0,0,0,0,0,0,0,1],
       [1,1,0,1,1,1,1,1,0,1],
       [1,0,0,0,0,0,1,0,0,1],
       [1,0,1,1,1,0,1,0,0,1],
       [1,0,1,1,1,0,1,0,0,1],
       [1,0,1,1,1,0,1,0,0,1],
       [1,0,0,0,0,0,0,0,0,1],
       [1,1,1,1,1,1,1,1,1,1]]
```

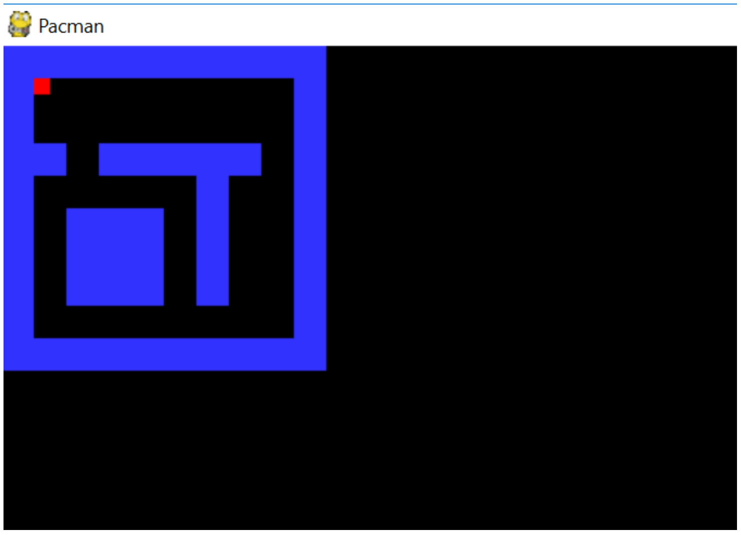
Step 2: Translate the map into graphical form by defining a sprite object called tile.

```
## -- Define the class tile which is a sprite
class tile(pygame.sprite.Sprite):
    # Define  the constructor for invader
    def __init__(self, color, width, height, x_ref, y_ref):
        # Call the sprite constructor
        super().__init__()
        # Create a sprite and fill it with colour
        self.image = pygame.Surface([width,height])
        self.image.fill(color)
        self.rect = self.image.get_rect()
        # Set the position of the player attributes
        self.rect.x = x_ref
        self.rect.y = y_ref
```

Step 3: We can combine the map and sprite class so create the walls using a loop and adding each tile to the sprite list and the wall list.

```
# Create a list of all sprites
all_sprites_list = pygame.sprite.Group()

# Create a list of tiles for the walls
wall_list = pygame.sprite.Group()

# Create walls on the screen (each tile is 20 x 20 so alter cords)
for y in range(10):
    for x in range (10):
        if map[x][y] == 1:
            my_wall = tile(BLUE, 20, 20, x*20, y *20)
            wall_list.add(my_wall)
            all_sprites_list.add(my_wall)
```

Step 4: Note when we create a tile that we define its colour, its dimensions and its position. The position depends on the dimensions of the tiles and in this case it is 20 pixels.



Step 5: You will also see that there is a red rectangle on the screen. This is the Pacman. It is a Sprite with class **player** (similar to a previous task) and is moved using the up, down, left and right arrows. Define the class **player** and then instantiate a variable called **pacman** as being an object of this class; set the x and y coordinates to 20 and the size of the sprtie to be 10 by 10 pies. . You should add the pacman Sprite to the **all_sprites_list** list. There is no need to define a separate pacman_list as there is only one of them!

Step 6: Write the code to handle the movement of the **pacman** (player) so the user can it up, down, left and right. Don't' worry about collisions we will deal with that tricky issue in the next task. The pacman should move by 1 pixel in the selected direction per loop. I have chosen to re-0define the **player_update_speed** function so it has a y and x speed which can be set to -1, 0 and 1 depending on the key pressed. So, LEFT is **player_update_speed (-1,0)**

Step 7: When you have done this mark your code.

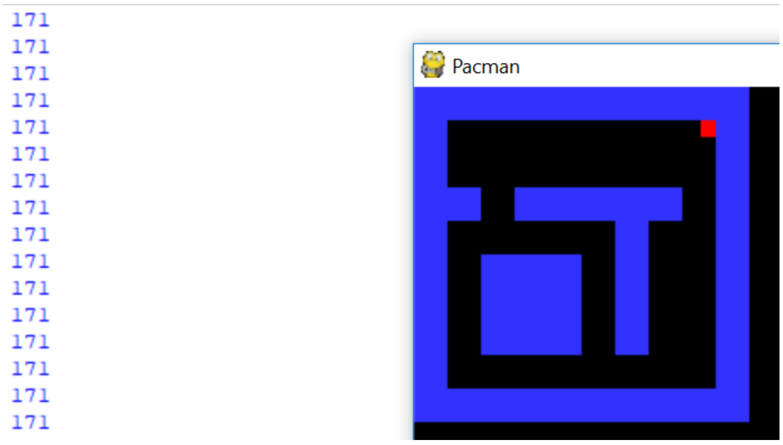
PacMan

Step 1: When Pacman hits a wall, we will need to stop him and wait for the user to indicate in which direction he should move next. There are a variety of ways of accomplishing but a good way is use the sprite_collide function. When a collision occurs the speed of x and y should both be set to 0 so Pacman is still.

Step 2: You will notice that once a collision occurs pacman stops and a user cannot get him get going again no matter what key they press! To find out what is going it is possible to a **print** statement in the code as follows:

```
# -- Check for collisions between pacman and wall tiles
    player_hit_list = pygame.sprite.spritecollide(pacman, wall_list, False)

    print (pacman.rect.x)
    for foo in player_hit_list:
        pacman.player_speed_update(0, 0)

    # Run the update function for all sprites
    all_sprites_list.update()
```

This lists the x coordinate of our pacman every time we drop through the loop. So running the code and pressing the left key moves Pacman along and he stops at the wall on the right as shown. Notice that the x coordinate of Pacman is 171 which is one pixel into the wall. This means the code above will always add Pacman to the collisions list regardless of the user input. We need to reset Pacman to his position before the collision.



```
171
171
171
171
171
171
171
171
171
171
171
171
171
171
171
171
```

Step 3: Amend you code as follows:

```
# -- Check for collisions between pacman and wall tiles
player_hit_list = pygame.sprite.spritecollide(pacman, wall_list, False)
for foo in player_hit_list:
    pacman.player_speed_update(0, 0)
    pacman.rect.x = pacman_old_x
    pacman.rect.y = pacman_old_y
# Run the update function for all sprites
pacman_old_x = pacman.rect.x
pacman_old_y = pacman.rect.y
```

```
all_sprites_list.update()
```

Notice that we keep a record of the previous coordinates of Pacman after a collision and before the update. In the spritecollide function we reset Pacman to the coordinates before the collision and our problem is solved.

Step 4: You should now have working code where Pacman moves smoothly around the screen. Mark your code.

PyGame Projects

Space Invaders

Develop the space invader game by adding different levels to the game. Each level could for example have more invaders, less bullets and differing invader speeds.

You may wish to allow the player to move up and down as well as across.

Consider adding “special” invaders which explode when they hit the bottom of the screen and fragment to destroy anything with a 1 pixel radius of impact.

Consider different types of invaders which may have more value when hit but move quicker.

PacMan

Pacman is a classic game where enemies move around the maps and try to catch the Pacman. Pacman also moves around tries to eat fruit and other objects.

At the point where computer-controlled enemies are moving around the screen you may wish to look at some theory behind choosing a path in order to move the enemy towards Pacman – Dijkstra’s A star algorithm is a good place to start!

Consider different rooms or levels.

Platform Games

Often games have platforms which objects can jump onto. This may require you to use some Physics and implement gravity so that objects can rise and fall smoothly

The platforms are simply rectangles however in some games they move which adds another level of complexity.

Other Games

There are a whole plethora of games on the Internet lots of them are retro games written for the first computers in the late 1970s and early 1980s. Whilst being visually quite simple Pygame allows you to recreate them – with a twist of your own.!

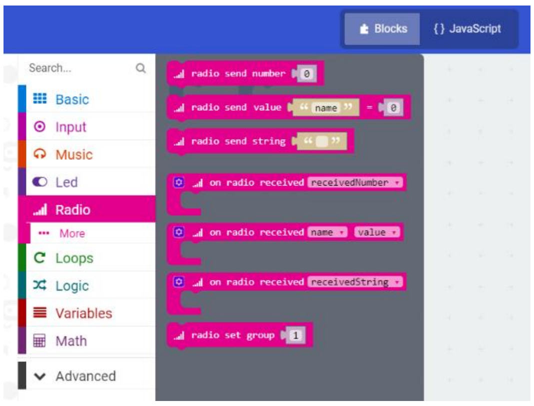
Stage B – Micro:bits Communication

Radio Messages (1)

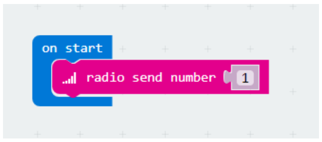
In this section we will look at connecting two Micro:bits together so that they can communicate with each other wirelessly. As a reminder open https://makecode.microbit.org/# in Google Chrome and save your .hex files regularly!

Step 1: You will need your own channel in order to communicate and your teacher will assign these to you.

Step 2: Select the "Radio" blocks from the menu on the left side



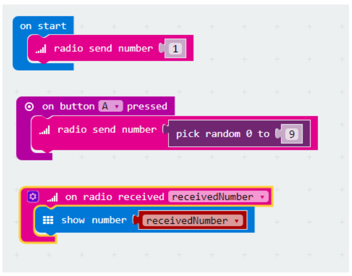
Step 3: To start the radio communication drag the "radio set group" into the "on start" block. Change the **radio send number** to the channel your teacher gave you.



```
on start
    radio send number 1
```

Step 4: You should now have two "virtual" Micro:bits on the left side. Both Micro:bits will share the same code and communicate on the channel number you have been given. If you can't see two Micro:bits, press the refresh button under the single Micro:bit.

Step 5: You can now send text and numbers between Micro:bits using the "radio send" and "on radio received" blocks. Try the following example to send a random number from one Micro:bit to another when you press the "A" button. Note: "receivedNumber" can be found on the "Variables" menu.



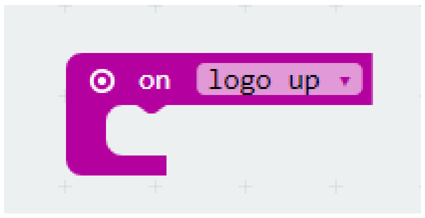
```
on start
    radio send number 1

on button A pressed
    radio send number pick random 0 to 9

on radio received receivedNumber
    show number receivedNumber
```

Step 6: Work in pairs (using the same communication channel, load the code and try to communicate!

Radio Messages (2)

Step 1: Create a program that sends the following letters to the another Micro:bit when you tilt the Micro:bit as follows:

- Tilt forwards (logo down): F
- Tilt backwards (logo up): B
- Tilt left: L
- Tilt right: R



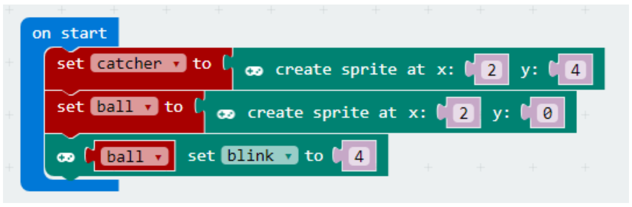
Step 2: Test your programs in pairs by directing each other around the classroom.

Step 3: Keep this code, you will need it to drive a robot later in the course.

Sprites (1)

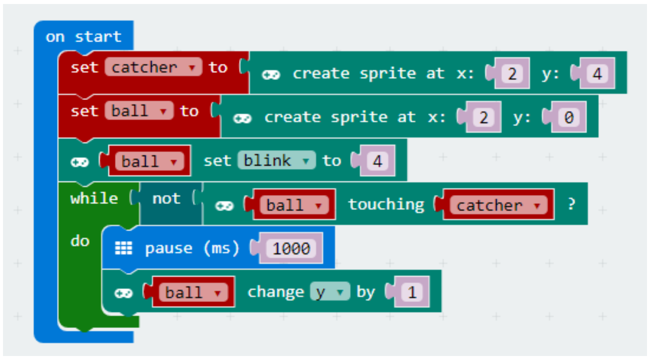
Sprites are special variables that can control the behaviour of an LED on the Micro:bit. Sprites are used to create games as you can move their position and detect whether they are touching the edge of the screen or other sprites.

Step 1: Create a Sprite called "catcher" and another one called "ball" using the code below. The last block sets the "ball" blink attribute to 4 times per second, so you can distinguish it from the catcher.



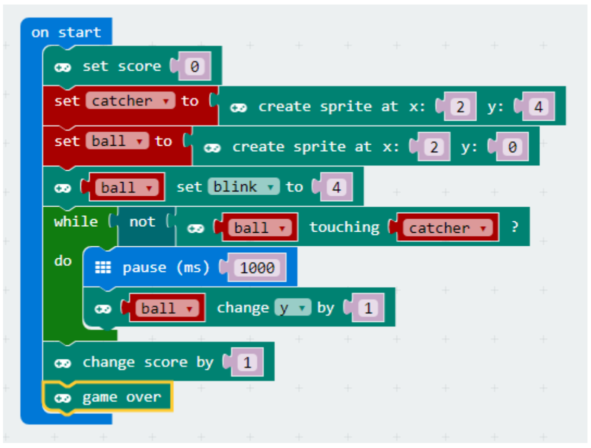
```
on start
    set catcher to create sprite at x: 2 y: 4
    set ball to create sprite at x: 2 y: 0
    ball set blink to 4
```

Step 2: Make the ball move towards the catcher by changing its "y" position by 1 every second:



```
on start
    set catcher to create sprite at x: 2 y: 4
    set ball to create sprite at x: 2 y: 0
    ball set blink to 4
    while not ball touching catcher ?
    do
        pause (ms) 1000
        ball change y by 1
```

Step 3: The game can "score" when "catcher" is touching the "ball" and you can end the game:



```
on start
    set score 0
    set catcher to create sprite at x: 2 y: 4
    set ball to create sprite at x: 2 y: 0
    ball set blink to 4
    while not ball touching catcher ?
    do
        pause (ms) 1000
        ball change y by 1
    change score by 1
    game over
```

Sprites (2)

For this task you will need to amend the code you produced in Spires (1). The following steps will help you complete the game:

Step 1: Get the “ball” to start from a random position at the top of the Micro:bit and then fall vertically down to the bottom.

Step 2: Enable the catcher to move to the left and right along the bottom row of the Mirco:bit using the “A” and “B” buttons. It shouldn’t fall off the edge.

Step 3: Display the number of times the ball has been caught (i.e. touches the catcher).

Step 4: Once the ball has been “dropped” ten times (i.e. it gets past the catcher) the game should end.

Project (Battleships)

Combining the radio task and the sprites task build a battleships game for two Micro:bits.

- Each Micro:bit should display 3 battleships (randomly placed). The design of the battle ships is up to you ... it may initially be just one LED square.
- Each Micro:bit should launch a "torpedo" at the other Micro:bit.
- You score if you hit a ship and destroy it.
- The winner is the first to destroy all three ships!

Here are some things to consider in your design:

- Do you want to regulate when torpedoes are fired i.e. will players take it in turns or is there a time limit before you can fire your next torpedo?
- Do you want to limit the number of torpedoes per player?
- Can you extend the game so the players decide where to put their battleships?

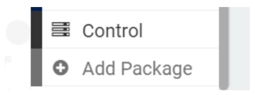
Stage C – Micro:bits Robotics

Control Setup

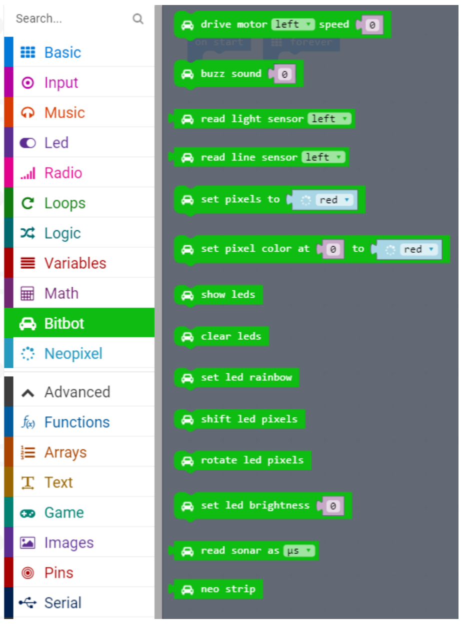
For the robotics section we will be using the Bit:Bot

To control the Bit:Bot with the Micro:bit, you will need to add the Bit:Bot package to PXT code editor.

Step 1: Select "Add package" from the bottom of the menu on the left side of the screen.

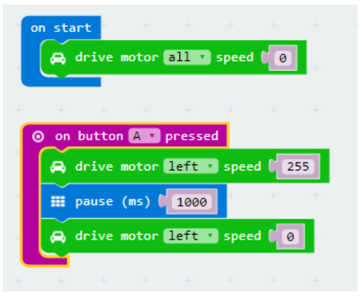


Step 2: Search for "Bitbot" (no colon) and select the Bitbot package. You will now have a new menu on the left side:



Step 3: To move one of the Bit:bot wheels we will need to use the "drive motor" block. To make sure that the Bit:Bot doesn't get stuck moving all the time, we need set the speed for a fixed amount of time. The amount the Bit:Bot moves will be controlled by the speed (-1023 to 1023) and the amount of time the motor is on.

Step 4: Try the following with your BitBot.



```
on start
    drive motor all speed 0

on button A pressed
    drive motor left speed 255
    pause (ms) 1000
    drive motor left speed 0
```

Motor Control

Step 1: Using the code you looked at in the previous task. Change the “speed” and “pause” values to make the robot do a 90 degree turn.

Step 2: Add code to make the robot turn left when you press button “B”.

Step 3: Add code to make the robot move forwards approx. 20 cm when you press “A” and “B” together. Hint: You might need to run the left and right motors at different speeds to get the robot to move in a straight line depending on the surface you are on.

Extra Activities

What happens if you set the speed to minus 255? How can you use this when turning?

Make the colour LEDs on the sides of the Bit:Bot act as indicators. Here is a useful this link for details of how to do this https://makecode.microbit.org/pkg/srs/pxt-bitbot

Program a Path

Now program the robot using a loop to navigate the circuit in the classroom.

Distance Sensor (1)

The Bit:Bots are fitted with a distance sensor that works by sonar. The two cylinders on the front are a speaker (T) and microphone (R) transmitting and receive sounds which are too high pitched for humans to hear.

Step 1 Display the distance in cm to an object from the Bit:Bot on the screen. You will need to use the "read sonar" block. (note use the "pause" block to get a reading every second)

Distance Sensor (2)

You now need to control the robots movement to avoid obstacles. Think about a robot vacuum cleaner, what should happen when it reaches a wall? You might need to use some “negative” speed settings to avoid collisions!

Task

Write a program so that the Bit:Bot changes direction when it detects an obstacle less than 10 cm away. Remember to use “pause” blocks between setting motor speeds and checking the sonar.

Test your program. The sensor might give some false readings causing Bit:Bot to change direction when it doesn’t need to. How can you fix that?

Radio Controlled Motors Project

Combine the tilting radio microbit program with the controlling motors program to build a radio controlled robot the responds to the movement of a micro:bit. When you tilt the controller forwards the microbit should move forwards, tilt left to go left etc. Use the “A” and “B” buttons to start and stop the robot.

Stage D – Python Coding Tasks

In order to undertake this stage, you should have completed some of the programming stages of the core module Coding and Computational thinking or undertaken an online course in programming. The language used here is Python.

Input/output

Step 1: Write a program in Python which takes as an input the user's name (e.g. "John") and then outputs "Hello, John"

Number Sequence

Step 1: Write a program which outputs the first ten integers, one on each line. For example:

```
1
2
3
4
…
```

Times Tables

Step 1: Write a program which takes a user input between 1 and 10 and outputs the first 10 values if that numbers times table. For example, the user inputs 3 the program outputs: 3, 6, 9, … 30

Comparison of Two

Step 1: Write a program which takes in two numbers and outputs them in numerical order, highest first. E.g. 13, 23 are input and the output is 23,13.

Comparison of Three

Step 1: Write a program which takes in three integers and outputs them in numerical order, highest first. E.g. 24, 16, 30 output 16, 24, 30.

Input Validation

Step 1: Amend the Times Table program form the task above so that the user input is checked to make sure it is an integer between 1 and 10 before outputting the value. If it isn't an integer in this range, then the program asks for the input again. An input of 99 should exit the program.

Palindrome

Step 1: Write a program that takes a six-letter string (word) and outputs it in reverse. For example; input "hello" output "olleh". *You are NOT to use in-built string functions of python for this task!*

Words, Words, Words

Step 1: Write a program which takes a sentence input by the user and outputs the number of words in the sentence e.g. “The quick brown fox” outputs 4.

Seconds Anyone?

Step 1: Write a program which takes in a time as 1:30:23 (hours, minutes and seconds) and outputs the total number of seconds (5423 seconds).

Rock, Paper, Scissors

Step 1: Write a program which plays the game rock, paper, scissors. The users enters R, P or S, the computer generates 1 (rock), 2 (paper) or 3 (scissors) compares the two and displays, win. Lose or draw.

Factors:

Step 1: Write a program which takes an integer from the user and outputs all the factors or the number. E.g. user enters 48 output is 2, 3, 4, 6, 8, 12, 18, 24.

Caesar Cypher

Step 1: Write a program to perform a basic Caesar Cypher which is the most basic form of encryption. You may need to look up how this works however it moves each letter in a word or sentence along by an offset. So, for example a becomes c, f becomes h and at the end of the alphabet y becomes a. You can ignore spaces and convert capital letters to lowercase.

Hints: You will need to convert letters to and from ASCII. You will also need to be able to find the individual letters of a string – which you have done before.

Lists and Arrays

Stage E – Encryption Techniques

Caesar Cypher

Public-Private Key Encryption

Appendix 1 – Suggested Python Coding Conventions

Coding conventions are a set of guidelines that recommend programming style, practices, and methods. Dulwich students must follow these guidelines to help improve the readability of their source code and to simplify the translation into pseudocode.

Python has an extensive style guide called PEP 8 and this should be followed as closely as possible.

Variables, Subroutines, Methods, Attributes and Parameters

These should all be lowercase with words separated by an underscore.

Examples:

```
new_total = 0

def my_super_function(first_name, last_name, date_of_birth):
```

Constants

These should be all uppercase with words separated by an underscore (Note that Python does not support constants and these can only be identified by the naming convention).

Examples:

```
LIGHT_BLUE = (0, 150, 255)

QUEUE_MAX_SIZE = 20
```

Classes

These should always be in CapitalizedWords with no underscores.

Examples:

```
class Animal():

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

Pseudocode specific comments

To ensure that you develop good pseudocode habits while using Python all classwork and prep programming tasks should adhere to the following commenting style:

All coding blocks should have a suitable comment to indicate the end of the block.

Examples:

```
If game_over:

      quit = True

# end if


while not game_over:

      all_sprites_group.update()

# end while


for counter in range(10):
    print(counter)
# next
```

For python subroutine definitions indicate whether the subroutine is a function or procedure.

Examples:
```
def calc_square(num):
    return num**2
# end function


def swap_list_items(my_list, pos_1, pos_2):
    temp = my_list[pos_1}
    my_list[pos_1] = my_list[pos_2]
    my_list[pos_2] = temp
# end procedure
```

For class methods you should also indicate whether the subroutine is public or private.

Examples:
```
def calc_square(self):
    return self.num**2
# end public function


def swap_list_items(self, pos_1, pos_2):
    temp = self.my_list[pos_1}
    self.my_list[pos_1] = self.my_list[pos_2]
    self.my_list[pos_2] = temp
# end private procedure
```

Class definitions should also end with:

```
# end class
```

Python code to avoid

The following code is allowed in Python, but should be avoided as it is incompatible with most programmers understanding of pseudocode:

Multiple logical comparisons
```
if 0 < my_value <100:
```

Multiple assignments
```
my_var_1, my_var_2 = 1, 2
```

Multiple returns

```
def update_coordinates(x,y):

     return x+1, y+1

# end function

x_pos, y_pos = update_coordinates(x_pos, y_pos)
```