# PyGame in a Day

We'll begin our introduction to Pygame by constructing a very simple game that involves collecting candy by moving your character around the screen. This activity was originally developed by the National Computer Science School for the Girls Programming Network, and is achievable in a very short time frame - you shouldn't take more than a couple of hours to get this done!

You will need to follow the steps in this activity, but you should do so carefully. While all the code examples are provided, they are not simply given to you, and you must place them correctly according to the instructions.

# An Introduction to Pygame

Adapted from Pygame in a Day, prepared by the Universityof Sydney GPN tutors for the 'PyGame in a Day' workshop.

## Introduction

Pygame is a set of Python modules designed for writing computer games. This activity will use Pygame to construct a very simple game.

You can download all of the resources needed for this game at:
https://drive.google.com/open?id=1sfGvpUuyFkxg2jERSndG3RrDQMfaRcDK

## Installing PyGame

To install pygame for python 3, simply run

```
pip install pygame
```

from your console/terminal. If you're on windows, the above will only work if python has been added to your PATH. If that is the case, you can visit:
https://docs.google.com/document/d/1ohA3hgD4LloJ3Eps5o7wbjzNBMOFICg970M0jGPG81o/edit?usp=sharing for information on how to add python to your path.

## Getting Started

To start off, we need to get Pygame to open a window that we will be able to play the game in. First we need to import the pygame module, so that we can use it. This will be the first line of the file.

```
import pygame
```

Then we'll need to initialise pygame.

```
pygame.init()
```

Save this in a file that we can run. Make sure you don't call it 'pygame.py', as this would cause problems! (It would try to import itself as "pygame" in our first step!) A good name might be ‘game.py’.

We can now start setting up our screen.

We need to decide the width and height we want the game window to be. We’ll save these in variables called `WIDTH` and `HEIGHT`. These particular variable names are in capitals as a convention, to show that they are constants, i.e., that the values in them won’t change. (We don’t intend the size of the game window to change whilst playing the game.)

```
WIDTH = 640
HEIGHT = 480
Screen = pygame.display.set_mode((WIDTH, HEIGHT))
```

Before we can do anything else, we need to understand some basic control flow in Python. You should already know about if statements – we’ll use these extensively in the next section.

## Task 1: Create a window that closes when you click ‘X’

Now that we’ve got a window set up, we also want to be able to close it! Since we want the window to stay open until we close it, we’ll use a while loop. We’ll set a variable called `finish` (initially set to `False`). As long as `finish` is `False`, the loop will keep running. Once we set it to `True`, the while loop will exit, and the program will finish, with the `pygame.quit()` command.

This will be the main loop (the `while finish != True:`) of our program. Note that in Python, spacing is important! Ask your teacher if you get stuck.

```
finish = False
while finish != True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True
pygame.quit()
```

This also sets up a loop that listens for all events that happen. These are things like keys being pressed or buttons being clicked.

We might also want to add the ability to close the window by pressing escape. Put this just after the other `if` statement in the main loop above. The spacing should look the same as the other `if` statement.

```
if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
    finish = True
```

## Task 2: Holding out for a Hero!

Now that we’ve got a window up and running, we need to put our player on the screen.

First, before we enter the main loop of our code, we're going to set up our hero. We'll do this by setting up a *sprite* and saving it in the variable `hero`. We can then add properties to it including which picture to load. We'll also calculate the rectangle the hero takes up. We'll add this code at the top of the file, before the main loop.

**Aside:** In computer graphics,a *sprite* is a two-dimensional image or animation that is integrated into a larger scene. (Wikipedia)

```
# Set up the hero
hero = pygame.sprite.Sprite()

# Add image and rectangle properties to the hero
hero.image = pygame.image.load('hero.png')
hero.rect = hero.image.get_rect()
```

**Note:** `'hero.png'` refers to the *hero.png* file in the resources folder discussed in the introduction. You'll need to download it and place it in the folder where you have created your *game.py* file. You could also replace this with any file you like and see what happens.

We'll also create a variable called `hero_group` to keep track of our hero. Add this before the main loop.

```
Hero_group = pygame.sprite.GroupSingle(hero)
```

Great! Now that we've got the hero set up, we need to paint it to the screen! We'll add this part of the code inside our main loop. These lines should be the last lines of the main loop - that is, the last thing we want to do is draw the hero and update the screen.

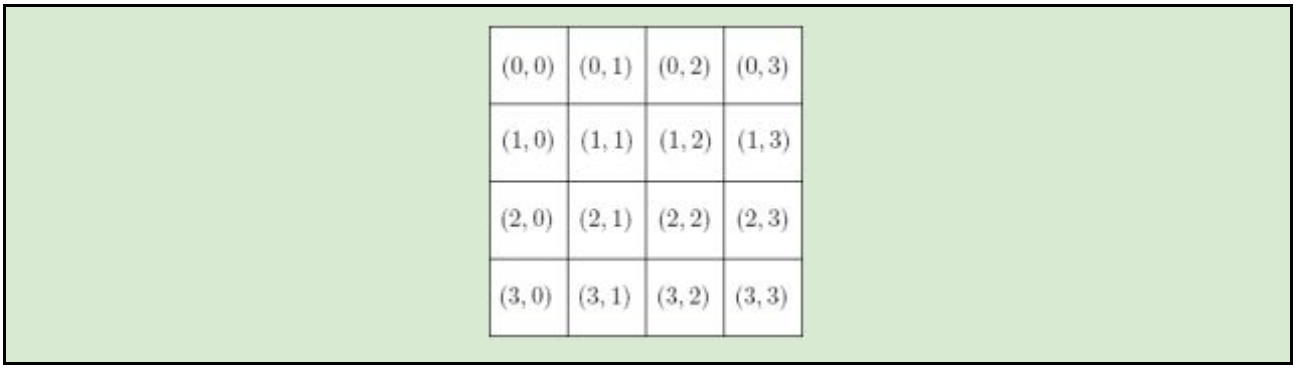
```
# add these lines at the end of the main loop
hero_group.draw(screen)
pygame.display.update()
```

Note that we need to add the above code inside the while loop, so that the hero is continually drawn to the screen.

## Task 3: Moving your Hero!

We want to be able to direct our hero around the screen. At the moment, our hero initialises in the top left corner of our screen at (0,0).

**Aside:** Graphics on the computer are just like a grid which starts at (0,0) at the top left corner of the screen. Each square in the grid represents a pixel, and can be referenced by: **(<ROW NUMBER>, <COLUMN NUMBER>)**




To move our hero around, we'll need to know how far to move them with each step. We don't want our hero to only move one pixel left and right, as it would take hundreds of key presses to move across the screen. Instead, we'll move our hero one 'tile span' each key press. That is, every time we press right, we want the hero to move on to the next grid square, based on how big the hero is.

Think of the canvas as a grid of squares that the hero can move through. To calculate how big the squares in this grid are, we'll need to know how many times the hero's image can fit on the canvas. We already know the `WIDTH` and `HEIGHT` of the display window, as we set it earlier in the code. Set the `TILE_SIZE` to be equal to `hero.rect.width`. We can then calculate the number of tiles we can have in the grid by dividing the total `WIDTH` or `HEIGHT` by the `TILE_SIZE`. This should go in the file before the main loop, but after the line starting with `hero.rect`, as you'll need this value to be available first!

```
TILE_SIZE = hero.rect.width
NUM_TILES_WIDTH = WIDTH / TILE_SIZE
NUM_TILES_HEIGHT = HEIGHT / TILE_SIZE
```

Now we've got to tell our hero to move around! Inside the main loop that listens for keypresses, we'll set up a series of cases for what to do if the up, down, left or right key is pressed. First, it will check if a key has been pressed. If it has, then we'll check what key it was, and get the hero to move appropriately. Remember again that your spacing should be consistent otherwise Python might not know what you mean. (**Hint:** make sure that this code is at the same level of indentation as the other if statements!)

```
if event.type == pygame.KEYDOWN:
    # Move the hero around the screen
    if event.key == pygame.K_UP:
        hero.rect.top -= TILE_SIZE
    elif event.key == pygame.K_DOWN:
        hero.rect.top += TILE_SIZE
    elif event.key == pygame.K_RIGHT:
        hero.rect.right += TILE_SIZE
    elif event.key == pygame.K_LEFT:
        hero.rect.right -= TILE_SIZE
```

## Leaving a Trail

If you try out the above code, you might notice that now our hero is leaving a trail of ghost images behind them. We'll need to repaint the background whenever we move them.

We're going to paint a plain black background for our character. We can do this just by filling the screen black before we redraw the hero. We'll need to add this inside the main loop, so that the

screen is continuously painted black. Make sure you add this line of code before the lines where you add the hero and update the screen. (Otherwise, you'll fill the screen black over the top of the hero!)

**Aside:** When drawing objects (such as the hero or the background) to the screen, they are drawn in the same order that the draw commands appear in your code. Objects that are drawn later are drawn on top of earlier objects. So to make the hero appear in front of the background, we draw the background first and the hero second.

```
# Paint the background black (the three values represent red, green
and blue: 0 for all of them makes it black)
screen.fill((0, 0, 0))
```

Now we don't have ghost images behind our hero.

## Task 3: Putting Candy on the Screen

Great! Now that we have our hero walking around, we need to add in the actual game! The aim of our game is for the hero to pick up lollies that have been dropped all over the screen.

Just like we made a `hero_group` to keep track of our hero, we'll also add in a candies variable. All of this code should be before the main loop, but after the line with `TILE_SIZE`.

This will make a list to store all the candies.

```
candies = pygame.sprite.OrderedUpdates()
```

First we'll go through adding one candy. We'll make a new sprite for it:

```
candy = pygame.sprite.Sprite()
```

Load up the image of it:

```
candy.image = pygame.image.load('candy.png')
```

**Note:** `'candy.png'` refers to the *candy.png* file in the resources folder discussed in the introduction. You'll need to download it and place it in the folder where you have created your *game.py* file. You could also replace this with any file you like and see what happens.

Make a rectangle property for it:

```
candy.rect = candy.image.get_rect()
```

Put it in a location - here I'll just choose to put it three tiles to the right and two tiles down:

```
candy.rect.left = 3 * TILE_SIZE
candy.rect.top = 2 * TILE_SIZE
```

And add it to our candies list:

```
candies.add(candy)
```

We'll also need to draw the candy to the screen. This will be at the end of the main loop right before the lines where we draw the hero to the screen: `hero_group.draw(screen)` and update the screen: `pygame.display.update()`

```
candies.draw(screen)
```

So the last three lines of the code should be:

```
candies.draw(screen)
hero_group.draw(screen)
pygame.display.update()
```

## Task 4: Randomising Candy Location

Now, we could hard code the location for all of the lollies we're dropping, but it would be more fun if it changed every time. We can use the random module to get random numbers that we can use as coordinates.

First we'll need to `import random`. Put this right up the top of the code, just below the line where we imported `pygame`.

```
import random
```

To generate random numbers, we'll use `random.randint()`, which takes two arguments: the minimum number and the maximum number.

Just as we calculated which tile-sized grid to put the lolly in before, we'd like it to line up nicely in the grid, so instead of choosing a random number between 0 and `WIDTH` for the x coordinate, we'll choose a random number for the grid square between 0 and (`NUM_TILES_WIDTH` - 1) or (`NUM_TILES_HEIGHT` - 1). We'll then need to convert the location back into pixels, by multiplying by `TILE_SIZE` again.

Replace the earlier lines:

```
candy.rect.left = 3 * TILE_SIZE
candy.rect.top = 3 * TILE_SIZE
```

with:

```
candy.rect.left = random.randint(0, NUM_TILES_WIDTH - 1) * TILE_SIZE
candy.rect.top = random.randint(0, NUM_TILES_HEIGHT - 1) * TILE_SIZE
```

## Task 5: More Lollies to Pick Up

Great! Now we're going to add a bunch more lollies to pick up! To do this, move all the code we added in tasks 5 and 6 into a for loop (except the "candies = "... line - that should stay the the top). Whenever we say 'into' or 'inside' in Python, that means use spaces to indent some lines (such as the lines to create candy) to move them inside the control statement. Here, that's the for statement.

Use the range() function to determine how many lollies to create. This code should still be before the main loop.

```
for i in the range(10):
    # add the candy!
```

## Task 6: Picking up the Candy

At the moment, if our hero walks over a lolly, nothing happens.

What we want is for our hero to pick up the lollies if she walks over them. To do this, we need to check if our hero rectangle intersects with any of the candy rectangles in the candy list. We'll do this in the list, just before we re-fill the background and re-draw the hero.

The `groupcollide()` function takes four arguments: the list of heroes, the list of candies, and then whether, the hero should disappear if a collision is detected, and whether the candy should disappear. Add this inside the main loop.

```
pygame.sprite.groupcollide(hero_group, candies, False, True)
```

## Task 7: Winning the Game

The way we can win our game is by picking up all the lollies. To find out whether or not you've won the game, we'll add in a `win` variable just like we set up a `finish` variable. Initially, we'll set `win` to `False`. Make sure you put this outside of the main loop, next to the `finish = False` line.

```
win = False
```

Now, when we're in the loop (while your hero is picking up lollies) we'll need to check if your hero has picked up all the lollies. We can check this really easily by seeing what the length of the list `candies` is.

If the list of `candies` has no elements in it (ie, has length of 0) we should set `win` to `True`! We want the following code in the main loop, as we want to keep checking whether or not the player has won. You could put this after the `groupcollide()` function.

```
if len(candies) == 0:
    win = True
```

Great! Now that we know whether or not the player has won! If they have, we want to display some text to the screen. Put the following code inside the main loop. Hint: if you can't see the text, maybe you're displaying it before you clear the screen - it needs to go afterwards!

```
if win:
    font = pygame.font.Font(None, 36)
    text_image = font.render("You Win!", True, (255, 255, 255))
    text_rect = text_image.get_rect(centerx=WIDTH/2, centery=100)
    screen.blit(text_image, text_rect)
```

# Finished!

Congratulations! You now have your very own working game!

# Extensions

If you want to keep working on and improving this game you can add a couple extra features, adding more lollies every 3 seconds, and adding sound.

## Adding more lollies every 3 seconds!

We want to make our game more fun, by making it harder to pick up all the lollies! To do this, we'll add an extra lolly to the screen every three seconds! First, we'll need to change the way we add lollies. We'll now use a function to add each lolly to the candies list. Outside the main loop, swap out the code we had before that adds lollies, and write this instead. It defines a function that we are calling `add_candy`, and takes in the `candies` list.

We'll keep the line `for i in range(10):` and now make it call the `add_candy` function for each iteration (i.e., 10 times).

```
def add_candy(candies):
    candy = pygame.sprite.Sprite()
    candy.image = pygame.image.load('candy.png')
    candy.rect = candy.image.get_rect()
    candy.rect.left = random.randint(0, NUM_TILES_WIDTH - 1) *
TILE_SIZE
    candy.rect.top = random.randint(0, NUM_TILES_HEIGHT - 1) *
TILE_SIZE
    candies.add(candy)

for i in range(10):
    add_candy(candies)
```

Now we'll add a timer that will add an event to the event queue every certain number of milliseconds. Here we'll set it to add a `USEREVENT` every 3 seconds. This should go before the main loop of the code.

```
pygame.time.set_timer(pygame.USEREVENT, 3000)
```

We'll now catch this event as we have been catching key presses. In the main loop, below where we are checking for up, down, left and right keypresses, add the following code:

```
if event.type == pygame.USEREVENT:
    add_candy(candies)
```

Great! We're nearly there! Now we just need to make sure that if the player has won the game, we don't keep adding lollies onto the screen! To do this, we'll check immediately before the `add_candy` line whether or not win is still `False`. If it is, we can keep adding candy.

So the beginning of our main loop should now look like this. Make sure you don't indent the new lines too much! They should be on the same level of indentation as the other tests for `event.type`.

```
win = False
finish = False

while finish != True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            finish = True
        if event.type == pygame.KEYDOWN and event.key ==
pygame.K_ESCAPE:
            finish = True
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_UP:
                hero.rect.top -= TILE_SIZE
            elif event.key == pygame.K_DOWN:
                hero.rect.top += TILE_SIZE
            elif event.key == pygame.K_RIGHT:
                hero.rect.right += TILE_SIZE
            elif event.key == pygame.K_LEFT:
                hero.rect.right -=TILE_SIZE
        if event.type == pygame.USEREVENT:
           if win == False:
               add_candy(candies)
```

## Om nom nom nom! (adding sound)

You should have a file already called nom.wav which sounds like a cookie monster eating a cookie. We're going to use this file to make the sound for when our hero eats a candy.

At the beginning of your program (near the top somewhere) we load the sound so that it's faster to use when we need it.

```
munch_sound = pygame.mixer.Sound('nom.wav')
```

Then later when you work out which candies are currently being eaten, the `groupcollide` function returns the list of candies that were eaten. If at least one candy was eaten, we play the sound!

```
collides = pygame.sprite.groupcollide(hero_group, candies, False,
True)
if len(collides) > 0:
    munch_sound.play()
```

# Where to from here?

What we've developed here is a way to create a simple, two dimensional, top down game. Here we don't have to worry about fiddly things like gravity, or collision physics. However PyGame is capable of doing all these things and more.

If you'd like to keep going with PyGame, check out:
http://programarcadegames.com/index.php?chapter=example_code

Program Arcade Games is a site that contains a number of both Python and PyGame tutorials. For our purposes they have an example code section that contains code for text based games, sprites, platforming games and more.

Definitely worth checking out to really see how far you can go with PyGame.