Girls' Programming Network

@Perth 2021

Flappy Bird!

Advanced

This project was created by GPN Australia for GPN sites all around Australia!

This workbook and related materials were created by tutors at: GPN Sydney, GPN Canberra & GPN Perth

If you see any of the following tutors don't forget to thank them!!

Writers

Sasha Morrissey
Renee Noble
Courtney Ross
Joanna Elliott
Amanda Meli
Adele Scott
Sheree Pudney

Testers

Emily Aubin
Samantha Rampant
Leanne Magrath
Rowena Stewart
Manou Rosenberg

Have you heard of the game Flappy Bird, and how it went viral on the internet? It is so gloriously simple but a good challenge to play.

Let's make our own version of Flappy Bird, it can be as hard as we want!

Part 0: Getting set up!

Goal: Let's get IDLE and our file setup before we start!

Python Intro

Task 0.1: Set Up the File

Create a file where we are going to write the code for our game:

1) **Create a new folder** on the desktop with your name in it. Eg: flappy_bird_renee
2) **Download** the bg.png, bird.png and pipe.png files to your new folder.
3) In your Python 3 IDLE click ***File*** and create a ***New File***.
4) Start by saving your file and calling it ***game.py***. Make sure it's saved to the folder you created on the desktop!

Task 0.2: Making a plan!

We're going to add some comments to our file! It's our plan for making our game!
Copy this into your file:

```
# Classes
# Bird Class

# Pipes Class

# SETUP
# Pygame Setup

# Set up the bird

# Set up the pipes

# GAME TIME
# Waiting to play

# Moving Objects

# Game over

# Show the pics!
```

Task 0.3: Import pygame!

We're going to be using a python module named **Pygame** today. We need to import this into our code:

1) At the very top of the file, before all the comments, add all the Pygame features using this line:
   from pygame import *

2) We need to tell the computer to set up Pygame, on the next line write:
   init()

Note that all the computers in the lab have Pygame installed for us to use. If you are using Pygame at home, you will need to install it onto your computer before you can use it. Google 'install pygame' to get instructions on how to do this at home.

Part 1: Displaying our first image!

To start off, let's get the game working with a background displayed!

Goal: Display the screen

Task 1.1: No Screen, No Game!!

We need to make a screen to play our game on!

1) Add this line of code to your **# Pygame Setup** section:
   screen = display.set_mode((800, 600))

2) Run your code! A game screen should appear.

3) The numbers are the width and height! What happens if you change them & run again?

4) Set it back to the original numbers!

Note: *You will not be able to close the game window by pressing the 'x' in the corner. Close IDLE (not your file) to close the game window.*

Goal: Display an image as the background.

Task 1.2: Displaying the background

A black screen is pretty boring! Let's make it interesting.

A **"blit"** allows us to easily add images onto the screen (think of it as "`BLOCK .. INITIALISE`!)

1) In your **# Pygame Setup** section load your background image with this line of code:
   background_image = image.load("bg.png")

2) In the **# Show the pics** section, tell the computer which picture to use and where to put it with this line:

   screen.blit(background_image, (0, 0))

   Blit works like this ***screen.blit(img_variable, (x, y))***

3) After you blit, update the display with **display.update()**

   Run your code! What does your game screen look like now?

Part 2: The pipes!

We need some obstacles for flappy bird to avoid.

Classes

Task 2.1: Copy the Class!

To learn about classes we're going to start by copying this Pipe class and learning how to use it! Later we'll make our own class for the bird!

Copy this code into **# Pipes Class**

```
class Pipe():
    def __init__(self, x, y, direction):
        self.x = x
        self.y = y
        self.direction = direction
        self.rect = None

        # Get the correct pipe image
        up_img = image.load("pipe.png")
        if self.direction == "up":
            self.img = up_img
        else:
            self.img = transform.flip(up_img, 0, 1)

    def move(self):
        left_speed = 5
        self.x = self.x - left_speed

    def blit(self):
        self.rect = screen.blit(self.img, (self.x, self.y))
```

Check it out!

1) Look at the **__init__** method. It takes 3 items (in addition to itself) as input; x, y, direction. It stores each of these as an attribute of the object. **When we create a pipe we need to say its x and y coordinate and if it points up or down.**

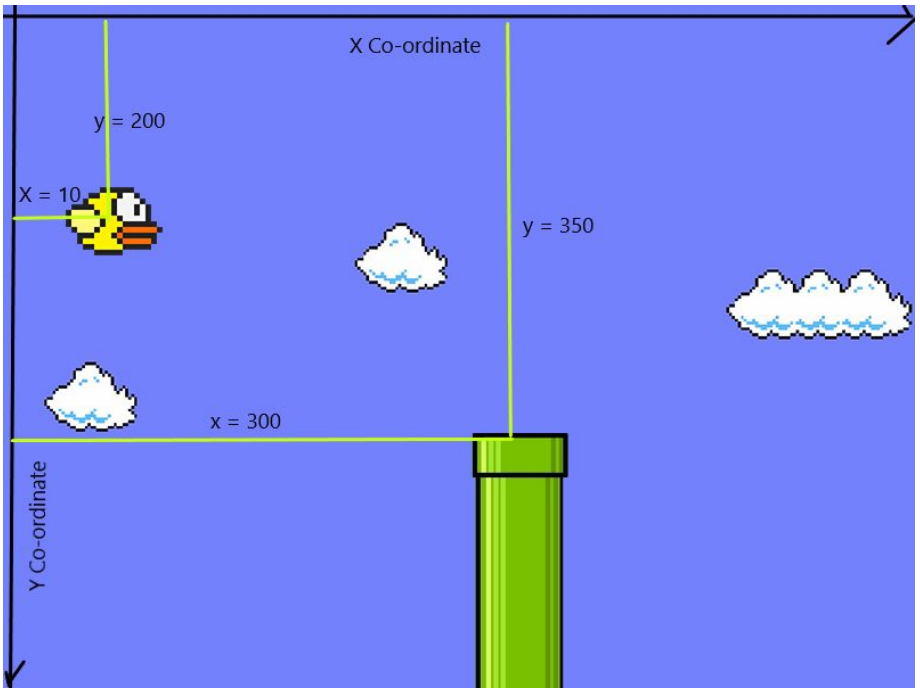
   (0, 0) are the coordinates of the top left of the game window and they increase to the right and down. So coordinates (x, y) represent x pixels to the right, and y pixels down. Have a look at the picture below to see how this works.

2) When we create the pipe it figures out what picture it needs from the direction.
   If it needs to flip the image to get a downwards pointing pipe it uses:
   down_pipe_image = transform.flip(pipe_image, 0, 1)

3) We have one more attribute called **rect**. This is where we store the results of blitting a pipe to the screen. We get back a pygame Rectangle object. We'll use this later to detect collisions

4) Check out the method called **blit**,it uses Pygame **blit** function, to make an easy way to blit any pipe. It will be nice short to write in our game logic!

5) We have another method called **move**. It updates the X coordinate to move left


X Co-ordinate
y = 200
X = 10
y = 350
x = 300
Y Co-ordinate

Goal: Let's create a list of pipes that are spread out across the screen!

Task 2.2: Starting positions!

We need to set-up our pipes! So we need to decide some information about them!

What do pipes look like??
Pipes have 3 bits of information

- **X coordinate:** the x position of the top left hand corner of the pipe
- **Y Coordinate:** the y position of the top left hand corner of the pipe
- **direction:** Whether the pipe is an "up" pipe or "down" pipe

***Example**: One pipe might be `(200, 350, "up")` where the X coordinate is 200, the Y coordinate is 350 and it points up! We group the bits of information together in brackets (call this a tuple, it's like a list)*

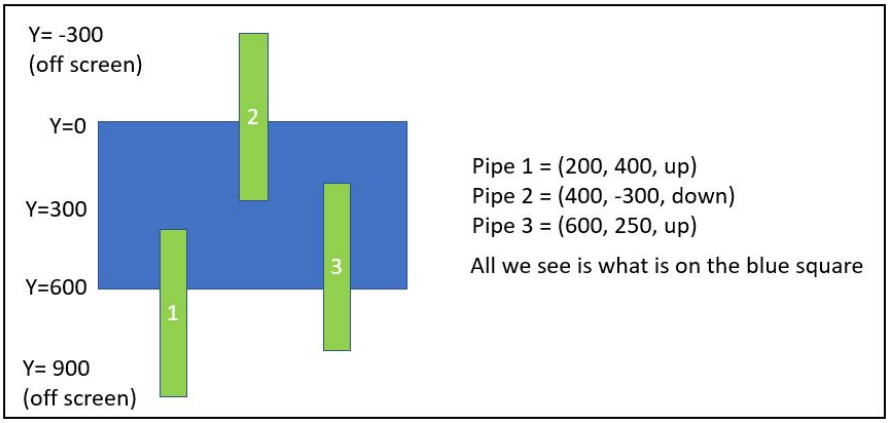
1. **Let's make 5 pipes** and put them all in a list called **`pipes_info`**.
   Copy this line of code for 2 pipes into your **# Set up the pipes** section:
   `pipes_info = [(300, 350, "up"), (600, -350, "down")]`
   Note : we'll explain the -ve Y coordinate in (3) below.

2. Two pipes isn't enough! **Add more pipes so you have 5 pipes!**
   Start by copying the pipes that are already there to extend the list!
   (don't forget to separate your pipes with commas)

3. Different pipes are better! **Let's make some changes to the pipes.**
   a. **X coordinate:** We want the space between to the left and right of each other to always be the same. It should go up by 300 each time **`300`**….**`600`**....**`900`**....**`1200`**.

   b. **Y Coordinate:** You get to choose!
   Remember that the **Y coordinate** represents how far up the screen the top of the pipe will be. At the top of our screen Y = 0 and Y increases as you move down.

   If Y > 0, top of pipe will be on screen so it will be a "up" pipe
   If Y < 0, top of pipe will be above screen so it will be a "down" pipe



   If the pipe is pointing **`"up"`** make Y between **`250`** and **`500`**
   If the pipe is pointing **`"down"`** make Y between **`-250`** and **`-500`**

   c. **Direction:** Choose **`"up"`** or **`"down"`**, make sure your Y coordinate matches (has a minus sign if it is "down")
   d. Make sure you have a mix of up and down pipes in your list.

We'll learn how to generate pipes automatically later. For now 5 is enough!

For Loops

Lists

Goal: Draw the pipes to the game window

Task 2.3: Pipe Printing!

1) In the section **# Set up the pipes**, create an empty list called pipes

   Use a **for** loop to iterate through pipes_info, break each pipe item down into:
   - X
   - Y
   - Direction

   For each of these we want to create a pipe object using **Pipe(x, y, direction)** which we can then **append** to pipes.

2) Go to the section **# Display the pics**, add another for loop in there, after the background blit , but before screen.update. Use the loop here to iterate through all the pipes and do **pipe.blit()** for every pipe in the list of pipes.

3) Run your code to see if the pipes appear. Note that at the moment you'll only see pipes that have an X coordinate less than 800 (as this was the width of the screen we defined in Task 1.1)

Goal: Create a game loop to keep updating the game

While Loops

Task 2.4: Let's Get Loopy

Let's make a game loop, it will keep the game going even after we do stuff (until the game ends)

1) In the **#GAME TIME** section, create an infinite while loop to be the game loop

2) Indent all the blits and comments below #GAME TIME so they are in your infinite loop

Since we haven't changed our pipes yet they don't move yet! That's next!

Task 2.5: Pipe Parade!

We need to make the pipes move to the left!
The **move** method is going to let us update where an individual pipe is.

1. In the **# Moving Objects** section:
   We want to move every pipe to the left every loop. Create a **for** loop to iterate through each **pipe in pipes** and use **pipe.move()** to shift it to the left.

2. Run your code!

Part 3: The Bird is the word!

Goal: Now we saw how to use a class, let's make our own!

Task 3.1: Bird Class!

Go to the **# Bird Class** section. We're going to create a class here. It will let us store information about our bird and also change the information.

1) Start your class with the line these two lines:
```
class Bird():
    def __init__(self, x, y):
```
The first line defines the class name as bird. The second line is the start of our initialisation method that runs when we make a new object.
We've told it we will provide two variables when we make a bird object; x and y, these will be our bird's starting coordinates

2) Inside the **_init_** method **store x and y as attributes** of the bird.
We can start with this line: **`self.x = x`**
Then do the same for y. Don't forget to indent inside __init__!

3) Give the bird object a picture that we'll use later to display the bird.
Set a new attribute called **`self.img`**, **`load "bird.png" just like was done by the pipes!`**

4) Methods are like special functions that have access to all of an object's information! **Write a method that helps us blit the bird using the info in bird!**

   a) **Add this line inside your Bird class:**
   `def blit(self):`.

   b) Then inside this method, use Pygame's *blit function just like the Pipe class does. When we access this with **self** here it will be accessing attributes of our bird, like* **`self.x`**, **`self.y`** and **`self.img`** !

   c) **Assign the returned value from Pygame's blit to a variable called self.rect** (We'll use this later to see if a bird hit a pipe!)

5) We wrote a bird blitter! Let's use it!
   a) Go to the **# Setup the bird** section, use the line:
   `bird = Bird(x, y)` , but **replace x and y** with the location you want your bird to start on the screen!

   b) Go to the **# Display the pics** section and use **`bird.blit()`** after you blit the background, but before you blit the pipes.

6) Later we're going to need two more attributes, in **`__init__`**. Create **self.velocity and self.gravity**. Start with **`0`** velocity & we suggest **`0.5`** for gravity!

Task 3.2: Are we playing?

Let's make a game loop, it will keep the game going even after we do stuff (until the game ends)

1) Under the **# GAME TIME line, before your infinite loop**: Create a variable called **`game_mode`** and set it to "waiting". We will use it to store three possible game states:
`"waiting"`: Waiting for the user to press any key to start the game
`"playing"`: Once we start the game, before we crash
`"game over"`: After we crash

2) The game loop has three different game modes to account for, `"waiting"`, `"playing"` and `"game over"`. Set up `if`, `elif` and `else` statements to account for these three cases inside your game loop. Put your pipe moving code from before inside the `"playing"` section. (See the comments **# Waiting to play**, **# Moving objects** and **# Game over** , these match up to these `if`/`elif`/`else` statements)

   To avoid syntax errors write `pass` inside the sections until you have filled them out

3) Ensure the contents of **# Show the pics!** are inside the game loop, but not inside any of the `if/elif/else` statements (displaying comes at the end of the while loop after these sections)

Task 3.3: Game start!

To start the game we want to detect if any key was pressed

- We can look at the last key pressedt with this line:
  `new_event = event.poll()`
- We only care about button presses, we can check for them with this line
  `if new_event.type == KEYDOWN:`

In the game loop, in your `"waiting"` game scenario we want to detect the event, see if it was a KEYDOWN event, if it was then start the game by jumping the bird!. If `"waiting"`

1. **Capture the event and check if it was a KEYDOWN using an `if` statement**

2. **Add another method to the Bird class called `jump`.**

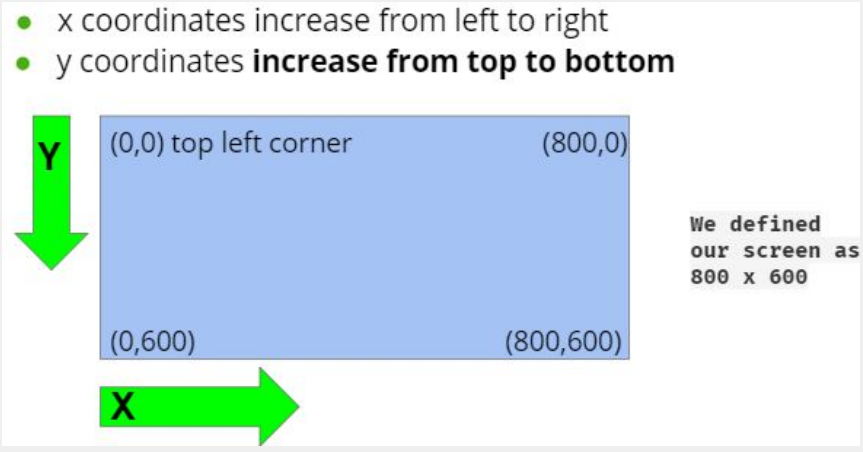
   In our game, we want flappy bird to move in jumps. When you press any key, flappy bird will jump up and then will fall down the screen until you press any key to make flappy jump again. This effect of flappy jumping up and falling is achieved by using the velocity and gravity variables.

   Inside the jump method, reset **self.velocity** to move the bird up the screen (remember to go up the screen we want to decrease the y coordinate so **self_velocity** needs to be negative (we recommend -10))

3. **If it was a KEYDOWN event:**
   a. Change the game mode to `"playing"`
   b. Give flappy a boost by using `bird.jump()`

4. Run your code! The game starts, but there's no jumping or falling yet!

Hints:
Remember that (0, 0) are the coordinates of the top left of the window and they increase to the right and down. So up is minus and down is plus.



Task 3.4: Moving bird

We haven't told flappy how to fall due to gravity and we haven't updated flappy's position based on her speed!

1. **Add another method to the Bird class called `move`.**
   This method updates flappy's velocity and position:

2. The **`move`** method is going to let us update how fast flappy is going and update the location of flappy!
   a. **Update the velocity**
      The new velocity is the old velocity + the acceleration due to gravity

   b. **Update the y coordinate**
      The new y coordinate is the old y coordinate + the new velocity

   It should look like this

```
def move(self):
    self.velocity = self.velocity + self.gravity
    self.y = self.y + self.velocity
```

3. In the **# GAME TIME** section:

   a. Check the latest event, if it was a KEYDOWN **`jump`** the bird!
      (Just like in the **`"waiting"`** section of the game!)

   b. Enact gravity and update the coordinates of the bird!
      Call **`bird.move()`**
      Make sure this is after (not in) the if statement from the previous step where you jump the bird)

Have a look at the tables below to see how the velocity and y coordinate change over time with calls to bird.move() (if a key is not pushed to start another jump). See how the **y coordinate is decreasing which will result in flappy bird moving up the screen** to mimic the 'up' part of the jump.

	jump	move	move	move	move	move	move	move
# move calls	1	2	3	4	5	6	7	8
self.gravity	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
self.velocity	-10	-9.5	-9	-8.5	-8	-7.5	-7	-6.5
self.y	**200**	**190.5**	**181.5**	**173**	**165**	**157.5**	**150.5**	**144**

After about 20 calls to bird.move() the **y coordinate starts to increase which will result in flappy bird moving down the screen** to mimic the 'down' part of the jump.

	move	move	move	move	move	move	move	move
# move calls	18	19	20	21	22	23	24	25
self.gravity	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
self.velocity	-1.5	-1	-0.5	0	0.5	1	1.5	2
self.y	**106.5**	**105.5**	**105**	**105**	**105.5**	**106.5**	**108**	**110**

How clever is that small piece of code!

Part 4: Collision Alert!

Now that we have our flappy bird, and we have all the obstacles, we need to detect when our flappy bird hits them!

Goal: Detect when the flappy bird has collided with a pipe.

Task 4.1: Crash!

Go back to your **Bird class**, we want a method that we can ask if the bird is hitting an object. We're creating a method that takes one object, it takes in a blitted rectangle!

1) **Add another method to the Bird class called `collides_with`.**
   It takes in one argument (in addition to self), called pipe.

2) We want to see if the pipes blitted rectangle (**`pipe.rect`**) hits flappy's blitted rectangle (**`self.rect`**)
   Pygame has this function **`rect_1.colliderect(rect_2)`** to test if there has been a crash! Use this inside your method, **`return`** **a boolean** from your method. **`True`** if there was a crash, **`False`** if there was not

Task 4.2: Game Over!

1) Go back to the **`for`** loop under **#Show the pics!** where you **`blit`** your pipes

2) Use an **`if`** statement to see if **`bird.colides_with(pipe)`**

3) If there's a crash set **`game_mode`** to **`"game over"`** and **`print`** out a Game Over message!

Task 4.3: Let's end this!

1) In the final section of our **`if/elif/else`** statement we can kill the game when we enter **`"game over"`** **`game_mode`**. Inside the else statement use **`break`** to exit the infinite loop

Part 5: More Pipes!

Goal: Let's automatically generate our pipes so we can have lots!

Random

Task 5.1: Starting positions!

Remember, pipes have 3 bits of information

- **X coordinate:** the x position of the top left hand corner of the pipe
- **Y Coordinate:** the y position of the top left hand corner of the pipe
- **direction:** Whether the pipe is an "up" pipe or "down" pipe

***Example**: One pipe might be* `(200, 350, "up")`
where the X coordinate is 200, the y coordinate is 350 and it points up!
We group the bits of information together in brackets (we call this a tuple, it's like a list)

Generate your pipes!

1. At the top of our code **import random**
2. In **# Set up the pipes** comment out your existing pipes_info list where you typed in values for 5 pipes

3. Create a variable called pipes_info that stores an empty list

4. **Create 2 variables pipe_gap and pipe_last_x**
   pipe_gap: Store the horizontal spacing between pipes in pipe gap. Set it to 300 (you can change this later to make the pipes closer together).
   pipe_last_x: to store the position of the last pipe so we can go after it. Set it to 0.

5. **We want to be able to generate as many pipes as we want, with no extra code!**
   Use a for loop with the range(n) function to loop n times, so we can make n pipes.

6. In the loop:
   a. Calculate the next pipe X coordinate, based on the gap and the last pipe.
   b. Randomly chose `"up"` or `"down"` for the direction of you pipe using
      `random.choice(["up", "down"])`
   c. Randomly chose a pipe Y coordinate using
      `random.randint(250, 500)`
      For `"up"` pipes make it positive,
      For `"down"` pipes make it negative (-250 to -500)
   d. Group up your 3 bits of information and append them to pipes_info
   e. Update last_pipe_x

List Comprehensions

CHALLENGE: Initialise the pipes with list comprehensions!

Forget the for loop, can you do this challenge just using list comprehensions?
We want to end up with something that looks like (but even longer!):

```
pipes_info = [(200, 350, “up”), (400, -350, “down”), (600, -350, “down”),
              (800, -250, “down”), (1000, 450, “up”)]
```

There are lots of ways to do it!

What a hand getting started, here’s some tips:

1) Calculate 3 lists separately, pipe_xs, pipe_ys, pipe_directions.

2) We want pipe_xs to be consistently spaced apart, so start by using range for even distribution, customise it with pipe_gap

3) You can use functions in list comprehensions, for example

```
words = [“cat”, “frog”, “horse”]
lens = [len(word) for word in words]
lens would be [3 ,4 ,5]
```

   Instead of len try using random.choice or random.randint to choose directions and y coordinates.

4) The y coordinates and directions depend on each other. “up” should be positive “down” should be negative. You can put if statements inside your list comprehension to make one list depend on the other:

```
[some_function() if direction == "up" else some_other_function()  for direction in
directions]
```

5) We can group up all our list together using zip.

```
zip([”a”, “b”, “c”], [1, 2, 3]) → [(“a”, 1), (“b”, 2), (“c”, 3)]
```

Part 6: Bonus – Better Graphics!

Task 6.1: Tilty Bird!

When a plane takes off it points up on an angle and when it's landing it points down on an angle. Flappy should point up and down too depending on which way he's going!

You can use this to tilt the image:
```
rotated_image = transform.rotate(original_image, some_angle)
```

If your angle is positive it rotates the image anticlockwise. If it's negative it will go clockwise.

Try using this in your birds blit method, before blitting your image get the transformed image.

- Try using your velocity to determine the angle.
- Try multiplying the velocity by -1 to change the direction
- Try multiplying your velocity by different numbers to exaggerate or understate the effect of the tilt until you find one you like!

Task 6.2: Flappy's got to flap!

Make Flappy's wings flap!

Download the bird_wingsup.png, bird_wingsmid.png and bird_wingsdown.png files to your desktop folder.

These 3 new files are pictures of Flappy:

1. Wings up
2. Wings middle
3. Wing down

To make Flappy flap we need to toggle through these to make an animation, the animation has 4 steps in the cycle:

1. Wings middle
2. Wings up
3. Wing middle
4. Wings down

And repeat!

There's a few ways to code it! One way is using a counter and an if statement:

1. **Create an attribute to the Bird class called self.image_counter, set it to 1.**
   We'll always set this counter to 1, 2, 3 or 4. This represents the flapping cycle

2. Load the 3 images, store them as self.img_up, self.img_mid, self.img_down (make sure you match them up correctly!)

3. In your blit method, use an if statement to check what the current image_counter is to show the right picture:
   1 → self.img_mid
   2 → self.img_up
   3 → self.img_mid
   4 → self.img_down

4. After you've blitted, update the counter to the next number. 4 will loop back around to 1.

Challenge! Can you do this with less lines of code?
Maybe try storing the images in a list, and try using the % (modulo) operator